Model Development Guidelines for UML-RT: Conventions, Patterns and Antipatterns

Tuhin Kanti Das · Juergen Dingel

Received: date / Accepted: date

Abstract Software development guidelines are a set of rules which can help improve the quality of software. These rules are defined on the basis of experience gained by the software development community over time. This paper discusses a set of design guidelines for model-based development of complex real-time embedded software systems. To be precise, we propose *nine design conventions, three design patterns* and *thirteen antipatterns* for developing *UML-RT* models. These guidelines have been identified based on our analysis of around *100 UML-RT* models from industry and academia. Most of the guidelines are explained with the help of examples, and standard templates from the current state of the art are used for documenting the design rules.

Keywords UML-RT · Patterns · AntiPatterns · Model development guidelines

1 Introduction

Complex distributed real-time software systems are most frequently encountered in telecommunications, aerospace, defense and automatic control applications. As the complexity of real-time software is continuously increasing, the design of these systems is becoming very challenging. Model Driven Engineering (MDE) has been introduced to mitigate this problem by allowing for systems to be described at multiple levels of abstraction and providing automated support for transforming and analyzing models [19]. As an example, for the analysis and design of complex embedded software in the telecommunications industry, an architectural description language named *Real-time Object-Oriented Modeling (ROOM)* has been presented in [47]. ROOM was aligned with the Unified Modeling Language (UML) which gave rise to the *real-time profile of UML (UML-RT)* [46]. Just

Tuhin Kanti Das
School of Computing, Queen's University, Canada.
E-mail: das@cs.queensu.com

Juergen Dingel
School of Computing, Queen's University, Canada.
E-mail: dingel@cs.queensu.com

like ROOM, UML-RT supports constructs for modeling the structural and behavioral properties of a real-time system. The tool support for the development of real-time, embedded systems using Model-Driven Engineering (MDE) and UML-RT is provided by IBM Rational Rose Real-Time (RoseRT) [26], IBM RSA-RTE [24] and the open source tool PapyrusRT [15].

A software design model is an abstract representation of a system. It is the primary artifact in the MDE vision of software development, and computer-based technologies are used to transform the design models into the running systems. Developing a system involves the process of deriving a design model from requirement specifications and successively refining it until executable code can be generated from it [23]. The MDE tools mentioned above use UML-RT for compiling design models into code and link it with the runtime system.

Due to the high initial cost for developing real time software systems, preference is given to modifying existing software instead of rewriting it when major new requirements are identified. Therefore, a well-designed architecture is extremely important for developing real-time software systems which in addition to simplifying the initial construction, promotes the evolution of the system [46].

Although a large amount of research has already been conducted on the quality evaluation of software systems, there has not been much research on evaluating the quality aspects of models for real-time software systems. With the goal of investigating this area, our research focuses on quality assessment of design models for the real-time software systems. As part of this research, a set of behavioral antipatterns for UML-RT models has been introduced in our previous work [13]. This paper extends this work and presents a catalog of model development guidelines for UML-RT including a number of model-based *design conventions*, *patterns* and *antipatterns*. These guidelines are the result of our study of almost 100 UML-RT models created in industry and academia. The examples used in this document for demonstrating the proposed guidelines have been created using the IBM RSA-RTE tool.

This paper is structured in the following way: *Section 2* presents relevant background terminology for software development guidelines in general. This is followed by a brief summary of existing work on *conventions*, *patterns* and *antipatterns* in *Section 3*. The guidelines for developing UML-RT models are introduced in *Section 4*. Finally, a summary of the proposed guidelines is presented in *Section 5* along with a discussion on future plans.

2 Background

We now provide some background on relevant topics together with explanations of key terminology.

2.1 Software Quality

In the area of Software Engineering, two related but distinct notions exist whenever software quality is defined in a business context: *Software functional*

The final publication is available at Springer via [http://dx.doi.org/\[10.1007/s10270-016-0549-6\]](http://dx.doi.org/[10.1007/s10270-016-0549-6])

quality and *software structural quality*. The conformance of software to a given design, based on functional requirements or specifications is represented by software functional quality [41]. It is typically enforced and measured through software testing. On the other hand, software structural quality reflects the degree to which software is produced correctly. So, it represents how well software complies with the non-functional requirements such as reliability or maintainability that support the delivery of the functional requirements. Effectively, this is how the architecture of software adheres to the sound principles of software architecture outlined in [35].

For a piece of software to provide business value, the Consortium for IT Software Quality (CISQ) has defined five major desirable structural characteristics on the basis of ISO 9126-3 and the subsequent ISO 25000:2005 [17] quality model: Reliability, efficiency, security, maintainability and (adequate) size.

Measuring software quality is mainly motivated by two facts: risk management and cost management. In addition to causing inconveniences, historically software failures have caused human fatalities. An example of a programming error that led to multiple deaths is discussed in [30]. This enforces the requirement of regulating the development of certain types of software, particularly software embedded in medical and other devices. In addition, an application with good structural software quality is much easier to understand and change in response to evolving business needs. Therefore, the associated cost for maintaining such software is significantly less.

2.2 Definitions and Terminology

In general, every software development team maintains a well-defined set of *software development guidelines* for achieving software quality attributes. These guidelines are defined as a set of recommended best practices that must be followed to improve the productivity of development teams and to produce consistent deliverables [57]. In addition to *software development conventions*, which are standards recommended to be followed strictly, software development guidelines can also include a set of *software development principles*. Unlike a set of conventions such as *naming conventions*, the adoption of which will not have any negative consequences, a set of software development principles may be useful in some situations, but not in others. Therefore, the discussion of principles generally includes a problem context and their application usually requires the weighing of some tradeoffs. Software development principles typically include *patterns*, *antipatterns* and *smells* [9, 18, 20, 42, 57] (see Figure 1).

A *software design pattern* is a general reusable solution to a commonly occurring problem within a given context in software design. Thus the core sections that constitute a design pattern are the description of a problem followed by a solution to the problem. On the other hand, an *antipattern* consists of two solutions: a problematic solution followed by a refactored solution. The concepts of patterns and antipatterns are related to each other. The main difference is the application context. A pattern solution can be treated as a problematic solution in an antipattern if it is applied in an inappropriate context.

While the discussion of patterns and antipatterns involve identification and refactoring of a problem, *smells* are considered to be indications of weaknesses in

Fig. 1: Classification of relevant terminologies

the code or design that may slow down development or increase the risk of bugs or failures in the future. Martin Fowler defined a smell as a surface indication that usually corresponds to a deeper problem in the system [18].

For discussing patterns in this paper we have adapted the existing pattern templates from the current state of the art [14, 45]. These templates allow us to discuss the intent of a pattern, the problem it attempts to solve and the associated solution with any applicable consequences. In our discussion, for improving the readability and understandability of the proposed patterns, the description of the problem a pattern tries to solve is extended by explicitly stating any symptoms and consequences the problematic form may have.

To describe antipatterns, we use the templates proposed in [9]. The advantages of using these templates include the description of the problem in a way that can help readers recognize the problematic structure, symptoms and consequences. Moreover, these templates also allow the discussion of any alternative known solutions for improving the problematic one, in addition to the refactored solution.

3 Related Work

The body of related work is fairly large. The discussion below groups related approaches into five categories.

3.1 Coding and Design Conventions

The practice of following coding conventions can improve the maintainability of a software system significantly. In [39], a number of reasons for following coding conventions is pointed out while introducing coding conventions for the Java programming language. Common coding conventions that are followed in modern programming languages include naming conventions [38], comment conventions [36] and indent style conventions [37].

The effectiveness of coding conventions inspire the model driven development community to define and follow conventions for improving the understandability and maintainability of a particular design model. During a research visit to one of our industrial research partners, we have observed that a set of design conventions

is followed strictly by the developers of the industrial models¹. This set mainly includes naming conventions for different modeling constructs such as capsules, classes, states, transitions and protocols.

3.2 Patterns in Code

A design pattern is defined as a parameterized collaboration of model constructs for achieving a higher level purpose than each of the individual constructs [14]. In [10], the authors introduce a tool for the first time for automating the implementation of design patterns which adds a new dimension to the utility of design patterns.

The ability to recognize the similarities among problems encountered in the past and to reuse proven working solutions encourages object oriented designers to start cataloging proven solutions to recurring problems for the first time as object oriented design patterns which are commonly known as Gang of four (GoF) design patterns [20]. In this very famous catalog, the authors present three different types of design patterns: creational, structural and behavioral. Creational design patterns focus on abstracting the instantiation process. Their purpose is to make a system independent of how its objects are created, composed, and represented. On the contrary, structural design patterns are mainly concerned with how classes and objects are composed to form larger structures. Different algorithms and the assignment of responsibilities between objects are key concerns for the third category of this pattern catalog, the behavioral patterns. In addition to describing patterns of objects or classes, behavioral patterns also discuss the patterns of communication between them.

3.3 Patterns in Models

Similar to object oriented (OO) patterns, the concept of state patterns has first been proposed in [14]. In contrast to OO patterns which are focused on demonstrating optimal ways of structuring classes and objects, the state patterns try to capture general solutions to common problems of structuring statechart constructs. The patterns can then lead to efficient and effective solutions to different problems by being instantiated and specialized for specific problems. It is worth noting here that, the term “state pattern” in this context is different from a similar term used in [20] for introducing a pattern in the *GoF* patterns catalog. While the *state pattern* proposed in [20] allows an object to change its behavior at runtime when a change occurs in its internal state, the generalized term *state patterns* used in [14] represents a set of *twelve* behavioral patterns. These patterns allow for states to collaborate and are general enough to solve one or more common design problems and optimize one or more criteria. An important feature of many of the state patterns in this catalog is the use of orthogonal state components for distinguishing independent aspects of the state machine. A mini-catalog of five basic state patterns is introduced in [45]. Each of these patterns is discussed by presenting: the pattern name, problem definition, proven solution, sample code and the

¹ We do not have permission to mention the name of the industrial research partner.

consequences. In contrast to the state patterns discussed on [14], which revolve primarily around orthogonal regions, the state patterns defined in [45] focus on reusing behavior through hierarchical state nesting. Another feature which makes this pattern collection unique is that patterns are illustrated with executable code. As suggested in this paper, to be genuinely useful, a pattern must be accompanied by a specific working example that will help developers understand and evaluate the pattern and give them a good starting point for their own implementation. For avoiding complexity that can be caused by highly coupled control and service-providing aspects of a real-time system, an architectural design pattern has been introduced in [49]. Realizing the fact that a system cannot start performing its service-level functionalities before reaching an *operational* state, this pattern allows encapsulation of the system service functionality within the control functionality which improves reliability and maintainability of the system. This pattern can be applied to high-level architectures and to individual, low-level components.

Software often evolves to accommodate changes in requirements or to improve quality attributes. A set of behavior-preserving refactoring techniques for *class diagrams* and *state charts* is presented in [53]. A short description associated with each of the refactoring techniques illustrates behavior preservation. The authors also formalize the refactoring transformations using *OCL* constraints at the meta-model level of *UML*. In related work [52], some misconceptions regarding the *parameterized collaboration mechanism*, a feature in *UML* that allows developers to model commonly occurring design structures, are clarified. The authors illustrate the limitations of this mechanism while effectively modeling design patterns. Moreover, techniques to overcome the limitations are proposed.

Stateflow is an environment for modeling and simulating combinatorial and sequential decision logic based on state machines and flowcharts. To make models understandable and reusable, a set of guidelines for developing models using *Stateflow* is presented in [32]. This set includes the use of patterns and transitions in flowcharts, the placement of default transitions and the use of state machine patterns for conditions and action code in transitions. Each of these guidelines is presented with the impact it has on different quality attributes of the final model such as readability, verification, validation, workflow, code generation and simulation.

3.4 Antipatterns in Code

Since the introduction of the term design pattern in 1994, there has been an exceptional growth in the publication of design pattern literature. On the bright side, a large and growing base of reusable design solutions that can be evaluated and applied to a software development effort is available for skilled object oriented developers. However, the potential pitfalls of using design patterns became visible when many developers failed to properly evaluate the applicability of a particular design pattern in their problem context [9]. To address the challenge of using design pattern, Michael Akroyd first came up with a formal model for antipatterns based on a detailed analysis of the object-oriented literature [3].

Most of the research on antipatterns has been conducted in the context of code. To determine if the presence of *two* antipatterns, called *Blob* and *Spaghetti Code*, affect the understandability of the system during comprehension and maintenance

tasks, an empirical study has been performed in [1]. The results suggest that the combination of these *two* antipatterns can make it difficult to understand the system. In [40], antipatterns are represented as a set of rules. A methodology is introduced for detecting Enterprise JavaBeans (EJB) antipatterns which focuses on comparing the pre-defined rules with the application properties. Other research on antipatterns in code investigates the impact of antipatterns on the change- and fault-proneness of classes in object-oriented systems [28, 43] and the use of antipatterns to predict bugs [54].

3.5 Antipatterns in Models

A performance antipattern identifies a practice that badly affects performance, and it may involve static and dynamic aspects of software as well as deployment features. Some work has been done considering the detection and refactoring of performance antipatterns. The authors in [50] specify a set of 14 performance antipatterns. In subsequent work, they propose several techniques for the detection of performance antipatterns in software architectural models [11, 55]. The authors in [11] show how performance antipatterns can be defined and detected in UML models using OCL. They show with an example that the removal of a certain antipattern actually allows overcoming a specific performance problem by presenting a case study in UML annotated with the MARTE profile. For identifying performance antipatterns in architectural models and removing them, an approach is presented on the basis of rules and actions in [55]. This approach is based on the formal definition of some performance antipatterns that had not been formalized before. In [5], the authors address the problem of removing performance antipatterns detected in an architectural model. They use a Role-Based Modeling Language to represent antipattern problems and solutions. Solutions include Source Role Models (SRMs) and Target Role Models (TRMs). In their procedure, model refactoring for removing antipatterns is done by replacing an SRM with the corresponding TRM.

A catalog of 43 correctness and quality antipatterns for class diagrams is introduced in [6]. The goal of this work is to explore the educational role of antipatterns in improving modeling skills in UML class diagrams. This catalog analyzes the causes of correctness and quality problems and provides suggestions for rectifying these problems. The effectiveness of the antipatterns to improve student awareness of modeling problems in class diagrams is demonstrated by conducting two experiments.

To explore bad design choices in behavioral design of UML-RT, we present a set of *seven* state machine anti-patterns in [13]. In addition to discussing the problems associated with the antipatterns, refactoring suggestions are provided for removing the problems.

4 The Set of Guidelines

This section introduces a set of model development guidelines for UML-RT. These guidelines are outcomes of our analysis of almost 100 UML-RT models in

industry and academia. In addition to investigating a repository of anonymized student models considering the design of a simple Electronic Warfare System (EWS) in Royal Military College of Canada, inspecting a number of telecommunication models from one of our industrial partners during a research visit of six weeks helped us identify these best practices. Approximately, 80% of the models we analyzed are from the student model repository, and the rest are from the industrial partner. The average size of the models we investigated in the student model repository is as follows: 5 Capsules, 35 states, 71 transitions, 3 Protocols. The maximum depth of state nesting in these models is 4. As expected, the size and complexity of the industrial models are higher than the models in the student model repository. The maximum depth of state nesting observed in the industrial models is 6.

While investigating the UML-RT models, the inspection procedure is done manually. This is followed by several discussions of our analysis results with a number of UML-RT practitioners from industry and academia. Based on the important feedback received during these collaborations, we refine the outcomes of the model analysis.

We introduce these guidelines in the following three subsections. The first subsection introduces nine design conventions that should be followed strictly while developing UML-RT models. Four of these design conventions are suggestions for utilizing resources properly in UML-RT. This is followed by a discussion of *three* design patterns in *Section 4.2*. Finally, in *Section 4.3*, a set of *thirteen* antipatterns is introduced.

4.1 Design Conventions

As mentioned earlier, developers at our industrial partner follow naming conventions for different UML-RT elements such as capsules and protocols. These kinds of conventions are common, because they ensure consistent appearance of the development artifacts and increase the readability. In addition to the common naming conventions for different UML-RT model elements, there exist some other design conventions which should be followed strictly while developing real-time embedded software. This section discusses a set of nine design standards for UML-RT.

4.1.1 Resource Utilization

Digital systems can be classified into *two* categories: *general purpose* and *application specific* systems. In contrast to general-purpose systems which mainly include desktop computers, workstations, server systems etc., application-specific systems are designed for dedicated applications. Examples of application-specific systems can be found in process control, networking and telecommunications, home appliances and consumer-electronics devices. Application-specific systems mainly exist as integral parts of larger systems and therefore, they are treated as *embedded systems*. However, considering the widespread use of embedded systems, the authors in [58] defined embedded systems as application specific systems which are mass produced only.

An *embedded system* is a computer controlled device which has been specifically designed to perform certain tasks. A very large class of embedded systems, namely

the *real-time embedded systems*, needs to satisfy timing constraints [16]. Real-time embedded software often needs to operate under resource constraints that limit, e.g., storage capacity, internal memory, and battery power [16, 29]. This is especially true for many mobile embedded systems which have only limited computational power and energy. It is important to avoid the waste of these limited resources. This section discusses *four* conventions related to resource utilization in UML-RT.

Similar to code cleaning, model cleaning can help improve the understandability and maintainability of models, and remove instances of poor use of resources. The final *two* conventions among the *four* conventions discussed in this section are related to cleaning UML-RT models.

4.1.1.1 Cancellation of Timers after Their Use

Background: The timing services in UML-RT provide users with general-purpose timing facilities on the basis of both absolute and relative time. The timing services can be accessed by creating a port with the pre-defined Timing protocol. With the occurrence of a timeout event, the capsule instance that created the timer request would receive a message with the pre-defined message signal timeout. In order to receive the timeout message, a transition with a trigger event for the timeout signal must be defined in the state machine of the capsule. There are two ways available for creating timer requests in UML-RT: One shot timer and periodic timer. A *one shot timer* expires only once, after the specified time duration or at the specified time. On the other hand, *periodic timers* are set to timeout repeatedly after the specified duration until the timer is cancelled explicitly.

Convention: It is good practice to ensure that all timers are cancelled properly when they are not needed anymore or before going to the shutdown state of a capsule. Proper cancellation of a timer protects the capsule from receiving timeout events unexpectedly. This is beneficial in scenarios when the system timer has already expired and the timeout events of other timers are waiting in the queue to be processed. For one-shot timers, which are defined with *timer.informIn()*, the request remains valid until the timeout event is fired or the timer is cancelled. For periodic timers, which are defined with *timer.informEvery()*, the timer request remains valid until it is cancelled. If a timer is not cancelled properly, the request will keep firing even if the system is in the shutdown state. For example, a capsule can request a periodic timer which would fire once in every second with the code below:

```
Timing.Request tRequest = timer.informEvery( 1000L );
```

Afterwards, this timer can be cancelled sometime later with the following method call:

```
timer.cancelTimer(tRequest);
```

This method returns *true* if the *Timing.Request* object is valid and cancelled successfully, and *false* otherwise.

Rationale:

- Wasting resources may exacerbate *performance* problems.
- Unnecessarily executing resources may cause *errors*.

4.1.1.2. Termination of Created Capsules after Their Use

Background: A capsule can take on different roles in collaborations. *Three* types of capsule roles are supported: *fixed*, *optional*, and *plug-in*. *Fixed* capsule roles are created and destroyed automatically together with the creation and destruction of their container capsule. *Optional* capsule roles, however, can be created dynamically by the container capsule; the container creates them when needed and destroys them when they are not needed anymore. Capsule roles with *plug-in* type mainly serve as placeholders for dynamic relationships if the objects that would play the capsule roles are unknown prior to runtime. Upon receiving information regarding these objects, appropriate capsule instances can be plugged into these slots at runtime and the corresponding connections are established automatically. Therefore, *plug-in* capsules can exist independently of other capsules and their lifetime does not depend on that of the container capsules.

Convention: When we create *optional* capsules, we can control the time when to create and destroy the capsule instances using the *Frame* service ports in UML-RT. For example, assuming the availability of a capsule role called *jammerRole* for the specification of a capsule class called *Jammer* and an instance of the *Frame* class named *frame*, the following code can be used to create a new instance of the *Jammer* capsule:

```
frame.incarnate(jammerRole, java.lang.Class.forName(Jammer));
```

The created *Jammer* capsule instance can be destroyed anytime later using the following action code:

```
frame.destroy(jammerRole, 0);
```

Here, the second parameter indicates the *zero-based* index within the associated capsule role.

If there is no possibility of reusing an incarnated capsule instance, we should destroy it immediately after its use. However, if there is a possibility that after creating a capsule instance, it can be reused later, we should not destroy the capsule instance because capsule instantiation can have high runtime overhead [25]. As a capsule can contain a number of other capsules, the processing time for instantiating a capsule would depend on the number of capsule instances, ports and state machines it recursively contains. Compared to the instantiation of new capsule instances, importing a capsule instance is much faster. Therefore, if there is a possibility of reusing a capsule instance, instead of destroying it, memory should be pre-allocated for the capsule and import should be used whenever necessary.

Rationale:

- Destroying a capsule instance which will not be used again in the model help achieve better *performance*.

4.1.1.3. Removal of Unconnected Ports

Background: In UML-RT, ports are used for communicating messages among capsule instances of a design model. Ports are strongly owned by the associated capsule instance as their existence is depended on the existence of the container capsule.

Fig. 2: Unconnected ports

Convention: It is good practice to get rid of the ports that are not contributing to the functionality of the system. This kind of ports can be created for several reasons. Sometimes ports are created in the structural diagram of a UML-RT capsule, but are never used. Also, the removal of a connector from a structural diagram may leave a port unconnected (see Figure 2).

Rationale:

- Elimination of the unconnected capsule ports from the final model would protect system resources from being used unnecessarily.

4.1.1.4. Removal of Unreachable Artifacts

Background:

In the context of traditional software development, *dead code* [60] and *dead computation* [2] have been studied. Code that can never be executed at runtime are considered dead code. An example of dead code is given in Figure 3.

```

int getValue(int x) {
    x=x+1;
    return x;
    x=x+3; // Dead code
}
  
```

Fig. 3: Dead code

Similar to the concept of dead code, the modeling artifacts that will never be reached during the execution of a UML-RT model can be considered unreachable. For example, in a UML-RT model behavior, if states have no incoming transitions (see Figure 4 (a)) or incoming transitions that can never be taken (due to triggers that will never be matched by an incoming message or unsatisfiable guards, as

illustrated in Figure 4 (b)), and if the missing transitions are not added through specialization and inheritance, these states can be considered unreachable.

Fig. 4: Unreachable states (a) isolated state (b) state without any triggered transitions

If the model is still under development, the existence of such states will be likely and acceptable as they may become reachable as the model evolves. However, existence of these states in the final deployed model can only be considered as waste of modeling resources and an unnecessary complication of the model. The same is true for unreachable capsules, i.e., capsules that do not have a single port which is connected to any other capsules.

```
int x=0; // Dead computation
x=1;
y=x+2;
```

Fig. 5: Dead computation

In contrast to the concept of dead code, dead computation refers to a code segment that is executed during runtime, but the produced values do not have any effect in the code behavior (see Figure 5). In UML-RT, a similar scenario can occur if a message signal is sent from a capsule X to another capsule Y and if the signal is never used in Y to trigger any transition events. In this scenario, the sending of the message signal in capsule X can be considered dead computation as it is not actually affecting the modeling behavior.

Other forms of unreachable artifacts in a UML-RT model are unused state *entry-points* and *exit-points* in the model behavior. Entry and exit points are used for connecting transitions at different levels of a state machine hierarchy. A composite state is entered via a transition chain formed by an incoming transition to an entry point on the boundary of the composite state and a second transition originating at that entry point. In contrast, a composite state is exited if an outgoing transition from any of its substates targeting an exit point of the composite state gets triggered. The use of these modeling artifacts is encouraged as it makes the state machine more modular with the entry and the exit points on the boundary of a composite state serving as ‘openings’, connecting the inside with the outside of the state. IBM RSA RTE supports entry and exit points by, e.g.,

creating them automatically whenever a simple state with incoming or outgoing transitions is converted to a composite state. Sometimes a number of entry and exit points can be left unused accidentally in the final model (see Figure 4 (b)). This could happen if we change the target of an incoming transition of a composite state from the entry point to the edge of the composite state and the source of an outgoing transition of a composite state from an exit point to the edge of the state.

Convention: In general, it is good practice to remove these artifacts from the final model as they do not have any influence on the behavior of the model. However, exceptions can be made if these unreachable artifacts are planned to be used as the model evolves.

Rationale:

- Removal of unreachable artifacts from a model would keep the model clean. This would, in turn, improve the *understandability* of the model and ensure *better utilization* of system resources.

4.1.2 UML-RT Transitions: Best Practices

A UML-RT transition is used for illustrating the relationship between a source state and a destination state. When an object in the source state receives a specific event and certain conditions are fulfilled, the execution follows the associated outgoing transition and moves to the destination state. This section introduces two conventions regarding UML-RT transitions.

4.1.2.1. Proper Use of Initial Transitions

Background: In a state machine diagram, an *initial state* is a pseudo-state which explicitly shows the beginning of the state machine. The transition that connects the initial state with a sub-state is called the *initial transition* of the state machine. The presence of an *initial transition* in a state is optional. But, if it exists in a state, it is the first transition taken in that state. Only one initial state and only one initial transition is allowed in each state diagram. Developers are allowed to include action code in the initial transition of a state; but, other features of a transition such as guard conditions and trigger events are not allowed to be included.

Convention: It is good practice to take special care while creating an initial transition. If we have multiple capsules in a system design, during the execution of the initial transition of a capsule it could be possible that other capsules have not been created yet. This is especially true for the top level state diagram design of a capsule. For instance, if operations are invoked that have dependencies on other capsules, the operation might fail due to the absence of receiver capsules. More concretely, if a signal is sent in the initial transition and thus requires the previous creation and proper initialization of other capsules, it will get lost if the receiver capsule has not been created yet. Therefore, the use of operations that rely on the existence of other capsules should be avoided in the initial transition. However, as an exception to this scenario, a capsule can rely on its fixed contained capsules as their existence is guaranteed before the execution of the initial transition in the parent capsule. In addition, if a proper start coordination is implemented which

ensures the creation of certain capsules before the initialization of a capsule, the above mentioned restrictions do not apply.

Rationale:

- Practice of this convention prevents potential erroneous situations that can occur during the start-up process of a system.

4.1.2.2. Use of Internal Self-transitions Instead of External

Background: *Self-transitions* have identical source and target states. There are mainly two types of self-transitions available in UML-RT: *self-internal* and *self-external*. The difference between these two types lies in the execution behavior of the entry and exit code of the enclosing state. In case of external self-transitions, entry and exit code associated with the enclosing state get executed. On the other hand, entry and exit code of self-internal transitions do not get executed. This slight difference in their behavior allows these two types of self-transitions to be used interchangeably with minor changes in the design of the associated state machine diagram.

Convention: It is good practice to use internal self-transitions instead of using external ones for achieving better performance. The code generator automatically generates extra lines of code for entry and exit code if a self-external transition is used (see Figure 6).

<pre>protected void chain3_self_internal() { rtChainBegin(3, "self_internal"); rtTransitionBegin(); rtTransitionEnd(); }</pre>	<pre>protected void chain3_self_external() { rtChainBegin(3, "self_external"); rtExitState(); rtTransitionBegin(); rtTransitionEnd(); rtEnterState(3); }</pre>
(a)	(b)

Fig. 6: Generated code for a) internal self-transition and b) external self-transition

Rationale:

- As the generated code for an internal self-transition does not contain any dead computations [2], it would result in improved *performance*.

4.1.3 Conjugation of Server-side Ports in a Binary Protocol

Background: In UML-RT, instances of a *port* are used for communicating messages between capsule instances. The type of a port is defined with a *protocol role* which specifies the type of messages that can be sent to and from the port. Depending on the view of the participants of a particular communication scenario, the sets of sent and received messages are different. All the views of a communication are specified by a *protocol* instance, and instances of a *protocol role* represent each of these different views. During the creation of a port, we must specify the protocol role the port is going to represent. For being allowed to communicate with each other, two ports in a protocol must be compatible, i.e., every signal in the set of outgoing signals in one protocol role must be in the set of incoming signals of

the protocol role on the other end. Each protocol role can have additional signals for the incoming set.

There exist only two protocol roles in a *binary protocol*: *base* and *conjugate*. The incoming and outgoing sets of signals of the base role in a binary protocol are identical to the outgoing and incoming sets of the conjugate role respectively. Therefore, binary protocols can be specified using only one role: the base role; the conjugate can be derived from the base role just by inverting the incoming and outgoing sets. This inversion operation is known as *conjugation*. The MDE tools such as IBM RSA RTE and IBM Rose RT support only binary protocols.

Fig. 7: Server-side conjugation

Convention: In a client-server communication pattern, the clients should be initiating the interaction by sending requests towards the server which is the port that is being published. Conjugation of client side ports will not lead us to an erroneous state, rather it is a convention to conjugate the server side port which guarantees that the naming of ports and protocols is consistent. In addition, conjugating the server side of a connection reduces the modeling effort. By default, ports are not conjugated, and as there are usually more client ports than server ports, it requires fewer steps in a client-server pattern since only one server port needs to be conjugated (see Figure 7).

Rationale:

- The practice of server-side conjugation keeps the naming of protocols and signals consistent throughout the model.
- The readability and maintainability of the design model are improved.
- The modeling effort is reduced.

4.1.4 Avoid Unnecessary Use of Active Classes

Background: An active class in UML-RT is a *capsule*; capsules are central modeling elements in UML-RT. A capsule is used to represent independent flows of control in a system. A capsule and a passive class have some common properties such as operations and attributes. Similar to a class, a capsule can also participate in dependency, generalization and association relationships. However, capsules can have some specialized properties such as ports and capsule roles which distinguish them from passive classes. Ports and capsule roles are mainly used for enhancing modeling capabilities at the structural level. A capsule can also have state machines for modeling behavior. A passive class is an ordinary class which does not need to have its own thread of execution. On the other hand, each capsule instance has its own logical thread of control, though it may share an actual processing thread, known as a *physical thread*, with other instances.

Convention: Generally, passive classes are used to store and manipulate information in the system, whereas capsules provide coordinating behavior in the system. Therefore, some of the use cases which involve only the simple manipulation of stored information can be implemented without using capsule objects. The use of capsule objects is necessary for implementing more complex use cases which require one or more capsule objects to coordinate the behavior of other objects in the system.

Rationale:

- As function calls are much faster than signal communications, the replacement of capsule classes with passive classes, wherever appropriate, would increase the performance of the system.
- Also, passive classes require fewer resources.

4.1.5 Improving the Visualization of UML-RT Diagrams

Background: In a UML-RT design model, it is possible to decompose a large diagram into several smaller diagrams. For example, a number of class diagrams can be used instead of using a single one for showing relationships among different modeling artifacts such as capsule classes, protocols and passive classes. With the availability of hierarchical state machines in UML-RT, it is also possible to decompose the behavioral design of a UML-RT capsule at multiple levels of abstraction instead of putting all the states at a single level.

Another useful feature we have observed in UML-RT development tools such as IBM Rational Rose RT [26] and IBM RSA RTE [24] is the capability to customize the display of class diagrams without changing the actual contents of the model. To be precise, visualization may suppress model elements without actually deleting them from the model. To understand a design model properly, it is important to keep the diagrams of the design model as simple as possible.

Convention: If a large number of artifacts are placed in a single class diagram while designing a real-time embedded system, the understandability of that model would be highly affected. As pointed out in [7], if a diagram is too large, the reader may find it difficult to see what to focus on and can lose interest eventually. Therefore, it is a good idea to divide large class diagrams into several smaller diagrams. For example, a separate class diagram can be used for showing the inheritance

relationships among different UML-RT artifacts such as capsules, protocols and passive classes. Consequently, these relationships would not be mixed up with other class diagrams which are mainly used to show the communication relationships among different capsule classes. The same is true for the behavioral design of a UML-RT model where state machines can be arranged into hierarchies. A composite state would reveal its internal state machine only when it is opened.

In addition, only important relations should be shown in a class diagram. As a relation can be deleted from the visualization without actually deleting it from the model, we can often reduce the visualization complexity associated with a class diagram by hiding operations and attribute lists from the corresponding active and passive classes. While this can make the view of a class diagram inconsistent with respect to the actual model, careful use of this feature in addition to providing appropriate documentations can greatly improve the overall understandability of the model. It is also worth mentioning that, as per our observation in the above-mentioned UML-RT development tools, this feature is not available in the structure diagram and the state machine diagram of a UML-RT capsule: it is not possible to delete elements from these diagrams without changing the actual model.

Some other ways to improve visualization include minimizing crossing lines, making the lines showing relationships among different artifacts in a class diagram or in a state diagram either horizontal or vertical and creating elements of similar sizes wherever possible [7]. In addition, in all the inheritance and the containment relationships of a model, it would be a good idea to follow a convention to put parents and owner elements of the model in upward position of the diagram. Conventions like this, increase consistency and improve the readability of inheritance and containment hierarchies.

Rationale:

- Better visualization of a diagram can greatly improve the *understandability* of the model.

4.2 Patterns

We now present *three* UML-RT patterns.

4.2.1 Status Monitoring Scenario

Intent: This pattern illustrates a general reusable solution for monitoring the status of a collection of components.

Problem: In a design model, it is very common to use a system monitor which sends out status requests to different components of the system. Once the system monitor receives responses from the components, generally, it reports these responses to the system controller.

Let us consider the status monitoring scenario in an *Electronic Warfare System* which has four capsules: *Controller*, *Receiver*, *Jammer* and *Status-Monitor*. According to the requirement specification, the *Controller* capsule would request the *Status-Monitor* capsule to monitor the status information of the *Receiver* and the *Jammer* capsules periodically. After receiving the status information, the *Status-Monitor* capsule should report this information to the *Controller*.

Fig. 8: Status monitoring - wrong approach

One possible realization of this capability is illustrated in Figure 8. As we can see in this figure, we have three states in the operating level behavior of the *Status-Monitor* capsule: *Idle*, *ObtainSubsystemState* and *ReportSubsystemState*. As the name implies, *ObtainSubsystemState* is focused on retrieving statuses from all the subsystems associated with this system design: *Receiver* and *Jammer*; and *ReportSubsystemState* is focused on reporting this status information to the *Controller*. As we can see in this design choice, for retrieving statuses from these subsystems *two* other states are created inside the *ObtainSubsystemState*: *ObtainRxState* and *ObtainJxState* (see Figure 9).

Fig. 9: Obtaining statuses from Receiver and Jammer

After sending out status requests to the *Receiver* subsystem, the system waits in the *ObtainRxState* for retrieving a response from the *Receiver*. Once it gets a response from the *Receiver*, it sends out a status request to the *Jammer* and waits in the *ObtainJxState* for a response from the *Jammer*. Finally, after retrieving status information from both of the subsystems, the system goes to the *ReportSubsystemState* state to report this status information to the *Controller*. Once it receives an acknowledgement from the *Controller*, the system goes back to the *Idle* state where it waits for the next status request timer to be fired.

Fig. 10: Pattern solution for status monitoring

One major problem with this solution is the large number of extra states required. For example, if *Status-Monitor* is used for monitoring 50 system components, the use of this solution would force us to use 50 extra states unnecessarily.

In addition, although the ordering of signal communication is not important here, this particular design solution enforces some ordering in the signal communication and this enforcement is done with the use of some extra states inside the *ObtainSubsystemState*.

Symptoms:

- The number of extra states needed for implementing the status monitoring mechanism following the approaches illustrated in this solution is equal to number of system components intended to be monitored.
- The order in which requests are sent and responses are received is unnecessarily specific and restrictive.
- Wait time for getting a response is unbounded.

Consequences:

- Use of extra state elements and unnecessary ordering in signal communication would have a negative impact on system *performance*.
- Unbounded wait time for getting responses from other capsules would increase the level of *coupling* in the design model. Consequently, the *robustness* of the system could be compromised.

Solution: The main problem associated with the problematic solution is the enforcement of signal ordering even if it is not needed. Therefore, the refactored solution to this problem, which is depicted by Figure 10, focuses on eliminating this issue by using a self-transition.

Figure 10 illustrates the operating level behavior of the *Status-Monitor* capsule. As we can see in this figure, the *Status-Monitor* capsule sends out status requests to all the components from the *reqStatus* transition. Now, once the system is in the *ObtainSubsystemState* state, it waits there for a certain amount of time which is controlled by a timer to retrieve status information from all the associated components. Once the timer fires, the system transitions to the *ReportSubsystemState* state where it reports the status information to the *Controller*. If status information from any of the component is missing, the *Controller* would take actions accordingly for fixing the problem.

Consequences:

- The refactored solution increases *simplicity* as we can send out status requests to all the components from a single state.
- The refactored solution eliminates the problem introduced by unnecessary ordering in signal communication by sending all the status requests from a single state.
- One *trade-off* associated with the pattern solution is the necessity for some extra implementation in the *Controller* side for resolving potential issues regarding missing responses.
- Another *trade-off* is the requirement for selecting an appropriate value for the extra timer used in the refactored solution. This would include possibly some extra maintenance effort.

4.2.2 Proper Use of System Timers

Intent: This pattern discusses the proper use of a system timer for controlling the operating period of a system.

Problem: Let us consider a system which is intended to run for *60* seconds as per the requirement specifications. One possible design of this system is depicted in Figure 11 which represents the top level behavior of the *System Controller*. In this design choice *two* timers are used in the system design: *startup* timer and *shutdown* timer. The *Startup* timer gives the *System Controller* some time (*5 seconds*) for starting up other components before going to the *Operating* state, whereas the *shutdown* timer controls the actual system operating time.

Fig. 11: Wrong placement of system timer

In the design of Figure 11, both of these timers are placed in the action code of the *Initial* transition. Consequently, once the *startup* timer fires, the system reaches the *Operating* state and instead of having an operating period of *60* seconds, which is part of the requirements, it actually runs for *55* seconds and thus violates the requirement specifications.

Symptoms:

- System timers that control the system runtime are instantiated before the system actually starts operating.

Consequences:

- This design choice may cause the violation of timing requirements.

Fig. 12: Proper placement of system timer

Solution:

The problems associated with the problematic solution can be eliminated by moving the instantiation of the *shutdown* timer from the action code of the *Initial* transition to the action code of the *toOperate* transition (see Figure 12).

Consequences:

- The system will follow the timing requirement specifications and would actually run for the time period it is supposed to be run.

4.2.3 Separation of Responsibilities

Intent: This pattern represents an approach to distribute responsibilities among multiple ports, which in turn allows the protocols to be cohesive and thin.

Problem: It is a common practice to define multiple roles in a single port of a capsule. Let us consider the example of Figure 13 where the *Controller* is communicating with *two* other capsules using a single port *controllerR1*. Consequently, this *Controller* port has dual responsibilities which may affect the size and structure of the associated protocol. In addition, the potentially low cohesive nature of the protocol roles may make the model difficult to understand, maintain, test and reuse.

Fig. 13: Multiple roles in a single port

Solution: This problem can be resolved by decomposing and distributing the responsibilities among different ports (see Figure 14).

Consequences:

- Distribution of responsibilities makes a model easier to *understand* and *maintain*.
- Also, it allows the associated protocols to be *cohesive* and *thin*.

Fig. 14: Distribution of responsibilities among different ports

- Tradeoff: The *Controller* capsule would have to listen to multiple ports now.

4.3 Antipatterns

Proper documentation of an antipattern describes a general form; the primary causes which led to the general form; symptoms that describe the way of recognizing the general form; the consequences of the general form; and a refactored solution demonstrating how the code or the model can be changed to resolve the problem expressed by the antipattern. In [9], the authors present a catalog of 40 antipatterns from three major viewpoints: the software developer, the software architect, and the software manager. They introduce a number of templates for antipattern description. The choice of a specific template for a new antipattern depends on the needs of the audience and the resources available to the author of the antipattern. The use of these templates ensures the inclusion of answers to important questions while describing an antipattern. For example, the full antipattern template introduced in [9] gives the flexibility to antipattern authors to include a number of optional sections in addition to two core sections: general form of the antipattern and the refactored solution. It allows the background discussion of an antipattern to be written up under a separate heading. Antipatterns are not always wrong and there are certain scenarios when using antipattern solutions is not a bad idea. The full template allows the antipattern writers to mention these exceptional scenarios under the heading of “Known Exceptions”. In addition, if there are known alternative solutions available to the antipattern apart from the refactored solution, they can be discussed as “Variations”.

This section presents a set of 13 antipatterns in UML-RT. The first three entries in this set introduce antipatterns in UML-RT’s decision-making processes and demonstrate corresponding refactored solutions. There exist two types of decision-making processes in UML-RT state machines: *static conditional branches* and *dynamic conditional branches*. They differ with respect to when a guard is evaluated. The purpose of these branches is to direct the behavior of a state machine at the branching locations according to the specification. This is followed by the introduction of three antipatterns for transitions in UML-RT state machines. There are three types of transitions available in state machine diagrams: *external*, *internal* and *local*. The difference among these transition types lies in the execution behavior for the entry and exit code of the corresponding state. Then we introduce two antipatterns regarding the inheritance feature in UML-RT. The rest of

our antipatterns are related to signal communication, timer characteristics, guard conditions, state elements and modeling scope in UML-RT.

4.3.1 Guards in Junction Points are Affected by Action Code in Incoming Transition

Background: In state machine diagrams, dynamic conditional branches are represented by *choice-points*. A choice-point is a vertex that can have one or more incoming transitions as well as one or more outgoing transitions. The decision of following a particular outgoing transition is made dynamically, after taking the incoming transition. On the other hand, *junction-points* represent static conditional branches in state machine diagrams. Similar to a choice-point construct, a junction-point is a vertex that allows one or more incoming transitions and one or more outgoing transitions. However, in contrast to a choice-point, the decision of taking a particular outgoing branch from a junction-point is made statically, before taking the incoming transition. So, the difference between static and dynamic branches in state machine diagrams is whether the guards of outgoing transitions are evaluated before (static) or after (dynamic) the incoming transition is taken. Static conditional branches allow a transition path to be determined before the execution of the incoming transition effect, whereas their dynamic counterparts do not determine the transition path until the incoming transition effect has been executed [33].

For example, in Figure 15, static conditional branching is used. When the trigger for *t1* transition is fired, the guard conditions of *t1* and *t2*, if there are any, will be evaluated before executing any action code in *t1* and *t2*. Now, since $X=1$ and $Y=1$ in the *Begin* state and no guard condition is specified for *t1* transition, only the guard condition for transition *t2* will be checked which evaluates to true. So, the action code of *t1* and *t2* will be executed and the execution will move to the *End* state.

Fig. 15: Static conditional branching (Junction-point)

In contrast, in Figure 16, as dynamic conditional branching is used and the trigger for *t1* transition is fired, since no guard condition is defined for *t1* transition, the action code of *t1* will be executed and the execution will come to the choice-point *CP*. Now, with $Y=2$, the guard condition for *t2* will evaluate to false, and the execution will take the *Else* transition for going to *Default* state.

General Form: Let us consider the scenario of a vending machine in Figure 17. In this example, initially the value of the variable *drinkType* is assigned “none” in *StateA*. Once the user requests a drink, the *input* transition is triggered where the *drinkType* variable is supposed to be updated with the requested drink-type “X”.

Now, as a junction-point is used here which represents static conditional branches, the choice will be made solely by evaluating the guard conditions associated with

Fig. 16: Dynamic conditional branching (Choice-point)

the transitions before executing the action code of *input* transition. So, the value of the input will be ignored and the machine will always transition to the *Error* state.

Fig. 17: Incorrect use of junction points: the *drinkType* variable is updated in the incoming transition (*input*) and is affecting the guard conditions of the outgoing transitions

Symptoms:

- At least one variable used in a guard of a junction point is updated in the action code of the incoming transition.

Consequences:

- Due to the subtle, potentially confusing semantic difference between choice-points and junction-points, their use can have some adverse effects on understandability and may lead to incorrect models, especially for novice UML-RT users.

Refactored Solution: This antipattern solution can be resolved by replacing the static conditional branching with dynamic conditional branching. If we incorporate a choice-point in place of the junction-point used in the antipattern solution, the action code of *input* transition will get executed before evaluating the guards of outgoing transitions, and consequently, the execution would take the expected path depending on the user input. Thus, the refactored solution would eliminate the possibility of any potential puzzlement in the design model.

Fig. 18: Overlapping conditions in guards of code-driven and model-driven development

Known Exceptions: If the action code of the transition to the junction point does not affect its guard, we can use junction-points. Otherwise, choice-points should be used.

4.3.2 Overlapping Conditionals

Background: Choice-points allow for dynamic conditional branches. The purpose of using a choice-point in state machine is similar to that of using an if-else block or a switch statement in code. It is important to use the guard conditions associated with the outgoing transitions of a choice-point properly.

General Form: Let us consider the scenario in Figure 18(a) which shows an if-else block used in classical code-oriented programming with overlapping conditions in code. Although it is not considered good practice to implement an if-else block in this way, the execution behavior in this code segment is clearly defined. The conditions in the if-else block will be checked in the sequence of their order of appearance in the code and consequently, the code block associated with the textually first condition evaluating to true will be executed.

Let us consider a similar scenario in the context of model-driven development in Figure 18(b). Assuming that method *getX()* in the action code of the *toCP* transition returns 6, all guards on outgoing transitions evaluate to true except the guard in transition *t5* and the *False* guard. Now, given this scenario, what is the execution behavior of this model going to be?

According to our observations in the RSA-RTE tool, in *CP1*, the outgoing transition that is created first among the set of the outgoing transitions with guards evaluating to true will get executed. Consequently, without knowing in which order the branches have been created, it is impossible to predict the execution behavior of this model.

Symptoms:

- The conjunction of at least two guards of a choice point is satisfiable.

Consequences:

- Overlapping conditions in choice-points affect the understandability and maintainability of the model.

Fig. 19: Breaking overlapping conditions

- This antipattern solution can lead the system to an erroneous state. For example, in Figure 18(b), let us assume that the order of creation of the outgoing transitions in CP1 is as follows: $t3$, $t4$, $t1$, $t2$, $t5$, $False$. Since the transition $t3$ is created first among the outgoing transitions having true guard conditions, the execution will follow the $t3$ transition for going to the $State_X_GT_1$ state. Now, if the $t3$ transition is removed from the model accidentally and reintroduced again, surprisingly, the execution behavior will not remain the same. In this case, instead of taking the $t3$ transition, the execution will take the $t4$ transition for going to the $State_X_GT_3$ state. This unexpected change in the model behavior can easily lead the system to an erroneous state.

Refactored Solution: This antipattern solution can be resolved by introducing a chain of choice-points. As we can see in Figure 19, the difficulty in understanding the execution behavior of the antipattern solution is eliminated in the refactored solution. As a result, assuming $x=6$, it can be easily concluded that the execution will take the outgoing transition $t2$ of the choice-point CP_2 as it appears first in the chain of choice-points having guard condition that evaluates to true. The refactored solution protects the design from exhibiting unexpected behavior, which improves the consistency, understandability and maintainability of the model significantly. The use of extra choice-points reduces the performance of the model. However, considering the improvement in the quality attributes mentioned above, the cost in performance is acceptable.

Known Exceptions: If no overlapping conditions exist in the guards of outgoing branches from a choice-point, no refactoring is required as the model with a single UML-RT choice-point executes faster since it contains fewer model elements.

4.3.3 Choice-point with Non-exhaustive Guards

General Form: Let us consider the example shown in Figure 20. As we can see, there is no *Else* guard associated with the choice-point CP .

Fig. 20: An ill-formed choice-point

Now, the question is what will happen if the event trigger for *toCP* transition arrives while the execution is in the *Begin* state?

As the choice-point model element represents dynamic conditional branch, after evaluating the associated guard condition for *toCP* transition to true, the corresponding action code of *toCP* transition will get executed and the execution will arrive in the choice-point *CP*. So, before arriving in *CP*, the value of *Y* will be updated and consequently, it makes the guard condition of the outgoing transition success false. This causes the execution to get stuck in the *CP* element. Now, unlike a state model construct, a choice-point element cannot handle any signal events.

We note that, if a junction-point is used in a similar situation like this, due to its static conditional semantics, deadlock cannot occur. If no valid path is found after evaluating all the available outgoing transitions from a junction-point, the execution will go back to the state it was in earlier.

Fig. 21: A well-formed choice-point

Symptoms:

- The choice point does not contain an *Else* guard.

Consequences:

- This antipattern could cause deadlock.

Refactored Solution: The problem created by this antipattern can be avoided by introducing an *Else* guard with every choice-point construct used in the model design. As we can see in Figure 21, due to the presence of the *Else* guard, the possibility of a deadlock is eliminated and the execution will move to the *Default* state.

4.3.4 Inappropriate Use of Self-transitions

Background: Self-transitions are available in all three forms of transitions in UML-RT state machines. The difference between different types of self-transition is in the execution behavior of the entry and exit code for the enclosing state.

General Form: A key benefit of MDE is the ease of understanding a particular design model through the visualization of interactions among modeling constructs. If the signals can only be received in a certain order, then this should be reflected in the state machine. It makes the state machine easier to understand, it likely improves the correlation with product documentation and errors are likely to be detected earlier.

Let us consider the design of a *Controller* capsule of an Electronic Warfare System in Figure 22. This capsule is responsible for coordinating message communication between two subsystems: *Receiver* and *Jammer*. When the scan timer fires, the *Controller* capsule sends a scan request to the *Receiver* capsule from the *scanReq* self-transition asking it to search for radar emitters in an electromagnetic spectrum. If an emitter is found, the *Receiver* sends it back to the *Controller* with the *emitterFound* signal and the *Controller* decides if it is a threat or a friendly emitter. If a threat is found, the *Controller* capsule sends it to the *Jammer* for jamming it. There are two modes of operation available in a *Jammer*: *jamming* and *look-through (LT)*. While the *Jammer* is in *jamming* mode, the *Receiver* must have to be in *blanking* mode, otherwise it can be damaged by the *Jammer*. On the other hand, the *Receiver* can be in the *scanning* mode only if the *Jammer* is in *LT* mode. These requirements are implemented in this antipattern solution using two self-transitions *toJam* and *toLT*. In addition to these signals, the *Controller* also uses self-transitions for requesting subsystems for status information and built in test (BIT) in regular intervals.

Fig. 22: Excessive use of self-transitions

Since all the communications in the *Operating* state are implemented using self-transitions, it is difficult to see the order of message handling in this behavioral diagram. For example, from this design it is not possible to predict whether the *emitterFound* signal can arrive at the *Controller* capsule before sending the *scanReq* signal.

Symptoms:

- Self-transitions are used excessively in the behavioral design of a model compared to the use of normal transitions.

Consequences:

- As the order of signal interaction is not clear, this antipattern directly affects understandability and maintainability of the design model.

Refactored Solution: This antipattern can be refactored by enforcing some order in the behavioral design. For example, Figure 23 demonstrates a design of the *Controller* capsule where it is easy to follow the order of execution of the message signals. This design improves the understandability of the model by introducing a set of new model elements inside the *Operating* state. For example, in contrast to the antipattern solution, this design gives us the ability to predict that the

Fig. 23: Replacement of some self-transitions using extra model elements

emitterFound signal can only be received in the *Controller* capsule after sending the *scanReq* signal.

Known Exceptions: There are situations when we might want to stay in the same state after handling a signal because of the design requirements. In these cases, the use of self-transitions is very handy. For example, in Figure 23, the use of self-transitions to request the subsystems for status information and BIT allows us to stay in the same state after responding to the events. In addition, self-transitions are also helpful in handling any unexpected signals [45].

4.3.5 Misuse of Local Transitions

Background: The origin of a *local transition* is the internal border of a state in a behavioral diagram. Local transitions can be used as self-transitions as well and therefore they are drawn using solid lines for distinguishing them from internal transitions. During the execution of a local transition, the exit and entry code of the source state do not execute. For example, in Figure 24, if the active state is a sub-state of the *Operate* state during the execution of the *toOperate* transition, the execution order would be as follows:

1. Exit code of the sub-state of the *Operate* state
2. Exit code of the *Operate* state
3. Effect of the *toOperate* transition
4. Entry code of the *Operate* state
5. Entry code of the sub-state of the *Operate* state

Fig. 24: Example of a local transition

General Form: Let us consider the design of a traffic-controller which can have three different types of traffic light: *Red*, *Green*, and *Yellow*. According to the requirement specifications, the activation of these traffic lights should follow this circular order: *Green*, *Yellow*, *Red*, *Green* ...

In Figure 25, a design choice for the top-level behavior of the traffic-controller capsule is shown. In this design choice, all the transitions are implemented as local transitions. Consequently, this machine does not rule out the possibility that the *Red* state becomes active right after the *Green* state.

Fig. 25: Abuse of local transitions

Symptoms:

- Local transitions are dominant over other types of transitions in a behavioral design.

Consequences:

- Since this antipattern solution allows all possible signal orderings, it decreases understandability and maintainability.
- Also, it may allow for executions violating the requirements and causing erroneous behavior.

Refactored Solution: The antipattern solution can be improved by replacing the local transitions with external transitions. As shown in Figure 26, the order of signal communication among different states is explicit due to the use of external transitions. The ease of following the order of communication makes this design choice much more understandable and one can easily rule out the possibility of going to the *Red* state directly from the *Green* state.

Fig. 26: Replacement of local transitions using external transitions

Known Exceptions: Suppose the requirements necessitate a large number of states and the transition to a single, specific state on arrival of a certain trigger, regardless of the currently active state. In these scenarios, for simplifying our design model we can draw a local transition from the internal border of the enclosing state to the sub-state where we want to go. Therefore, in this case, the use of a local transition can save us from drawing transitions to the target state from all other states in the corresponding state diagram level.

4.3.6 Incorrect Use of Group Transitions

Background: Group transitions are defined as transitions that originate from composite states. They can be considered as common transitions from all the sub-states within the composite state. Therefore, with the use of a group transition, any common behavior involving equivalent transitions from every sub-state of a composite state can be represented by a single transition originating from the containing hierarchical state.

General Form:

Fig. 27: Misuse of group transition

Let us consider the scenario of a microwave oven introduced in Figure 27. In this figure, with the triggering of the event *warmUp*, we can go to the composite state *Microwave* from the *Cold-Food* state. Inside *Microwave*, three sub-states are introduced to represent different states of the microwave oven such as *Door-Closed*, *Door-Opened* and *Operating*. Once the food has been warmed up, a group transition from the *Microwave* state named *warmed* can take us to the *Serve-Food* state.

Now, in this design choice, according to the requirement specification, the system can go to the *Serve-Food* state only when the microwave oven is in the *Door-Opened* state. The design solution introduced in this figure would not cause the system to fail, rather misuse of group transition in this figure would allow the possibility of the system to go to the *Serve-Food* state directly from any of the sub-states of the enclosing *Microwave* state. Therefore, the understandability of this design choice is reduced and it can be considered to have degraded quality.

Symptoms:

- A group transition is used unnecessarily while a normal transition can better serve the purpose.

Consequences:

- If a group transition is used incorrectly, it negatively affects the *understandability* of the associated model.

- The antipattern solution allows for erroneous behavior as it does not eliminate the possibility of going to the *Serve-Food* state directly from the *Door-Closed* and the *Operating* states.

Refactored Solution: The *understandability* issue of the design solution in Figure 27 can be eliminated by replacing the group transition with a normal transition from the *Door-Opened* state. As we can see in Figure 28, an outgoing transition *warmed* is introduced from the *Door-opened* state which is responsible for taking the system from the *Microwave* state to the *Serve-Food* state.

Fig. 28: Refactored solution for the wrong use of group transition

4.3.7 Absence of Inheritance

Background: In object-oriented software development, inheritance allows the derivation of structural and behavioral properties from existing elements. The main purpose of using inheritance is to reuse properties that are common among a set of model elements and to implement an appropriate abstraction mechanism for dealing with complexity by allowing details to be introduced gradually. It is important to use inheritance properly as it can improve the understandability and maintainability of the design of a software model significantly. In UML-RT, inheritance is available in three forms: Structural inheritance, behavioral inheritance and protocol inheritance [48].

General Form: Let us consider the design of a simple real-time system where a set of sub-capsules are inheriting the top level behavior from a base capsule. In Figure 29, two different versions of behavioral design of the base capsule are shown. According to the requirement specifications, the top level behaviors of the sub-capsules are identical, i.e., the corresponding states, transitions, events that trigger the transitions are the same in the top level behavior of all the sub-capsules.

Now, given these specifications, in Figure 29, the design choice on the left is obviously a poor one as it allows the subsystems to inherit only some isolated states from the base-capsule. Consequently, the sub-capsules need to implement

Fig. 29: Base-capsule behavior

the remaining top level behavior by themselves, even though the behavior is similar in each of the sub-capsules. The design choice on the right is the better one compared to the design choice on the left. It allows the sub-capsules to inherit the transitions from the base-capsule. But, on the other hand, it does not allow the sub-capsules to inherit the trigger events which are also common among the sub-capsules.

Now, let us consider another example which illustrates the absence of inheritance in protocol design. Two protocols used in the design of a radar-jammer system are shown in Figure 30(a). As we can see, although these two protocols have three incoming signals and four outgoing signals in common, protocol inheritance is not utilized here.

Symptoms:

- There exist common model constructs among multiple capsules in the same abstraction level which are not inherited.

Consequences:

- Lack of inheritance affects readability and maintainability of the design model.
- This antipattern makes the model more error-prone as a future change in the design requirement involving common elements that have not been inherited requires the designers to make changes in each of the sub-capsules separately.

Refactored Solution: This antipattern can be improved by utilizing the inheritance feature more fully. As mentioned in [48], abstract classes in UML-RT are recommended to be complete and executable. It provides us the opportunity to obtain early feedback on its potential. In addition, an executable abstract representation can be used as a basis for discussing system requirements with customers and providing the development team members with a mean to have a high level understanding of the final implementation. Therefore, in this particular example, as the requirement specifications suggest identical top-level behavior for the sub-capsules, it would be considered as a good example of inheritance if all the common elements including states, transitions, trigger events and action code are inherited from the base capsule.

The refactored solution for the antipattern regarding protocol-inheritance is demonstrated in Figure 30(b). As we can see, a base protocol is created here which offers all the common incoming and outgoing signals to the derived protocols.

Fig. 30: (a) Absence of inheritance in protocol-design (b) Utilization of protocol inheritance

Now the derived protocols only need to define the signals specific to their own implementation.

In addition to reusing the existing model elements, these refactored solutions enable us to make global changes in the derived constructs by changing the base modeling construct. This makes the design easier to understand and maintain.

Known Exceptions: There are certain scenarios in which it is preferable not to inherit all the possible model elements. For example, the design choice on the right of Figure 29 can be considered a better solution if it is possible that the requirements will change in the future in such a way that the sub-capsules need to use a different set of event triggers. In such cases, if we stick to this design choice, it would greatly improve the extensibility of the design model.

4.3.8 Pitfalls of Using Promote/Demote Operations in UML-RT Inheritance

Background: UML-RT supports features of promoting and demoting protocol signals, capsule and protocol state machine elements and capsule structure elements such as ports and capsule roles. As an example, demoting a port from a capsule would remove it from the generalizing capsule class structure and would move it into each of its sub-capsule classes. Consequently, the port becomes part of the sub-capsule structure and would no longer be inherited by the sub-capsules from the super-capsule class. In contrast, promoting a state in a sub-capsule behavior moves it into the super-capsule state machine and as a result, it is inherited by all the sub-capsules of the super-capsule.

General Form: Let us consider the example of Figure 31. Here, we have a super capsule which is inherited by two sub-capsule classes: *Sub1* and *Sub2*. *Sub1* has *two* integer attributes defined within its definition: **yVal** (=10) and **zVal** (=20), whereas the only attribute defined in *Sub2* is **zVal** (=11). Now, if we promote the **zVal** attribute of *Sub1*, it will become an attribute of the *SuperCapsule*. As a result, as *Sub2* is inheriting from the *SuperCapsule*, it will have two instances of **zVal** and thus the understandability of this design model will be affected.

In addition, if we try to promote **zVal** attribute from the *Sub2* capsule, the system will allow us to do that. But as there is already a variable defined with the same name in *SuperCapsule*, it will throw errors during compilation. Therefore, special care should be taken while using promote or demote operations in the UML-RT inheritance hierarchy.

Fig. 31: Inheritance in UML-RT

Symptoms:

- Elements of the same type with the same name are promoted and demoted in different places of the same inheritance hierarchy.

Consequences:

- This can affect the *understandability* of the design model.
- If we end up with two identical elements inside the same artifact, the model will not be well formed anymore.

4.3.9 Incorrect Use of Asynchronous Communication

Background: There are two forms of communication available while designing a real-time software system: synchronous and asynchronous communication. There exists a tradeoff between concurrency and complexity while choosing a particular form of communication.

General Form: Let us consider the design of the top-level behavior of a Controller capsule in Figure 32. Suppose, according to the design requirement, two subsystem capsules *Sender* and *Receiver* need to start executing before the *Operating* state becomes active. Due to the use of asynchronous communication in this antipattern, the delivery of the start signals to the subsystems is not guaranteed and no mechanism is implemented here for the Controller capsule to receive acknowledgement signals from the subsystems. Therefore, there exists a possibility that the Controller capsule would move to the *Operating* state without having started the subsystem capsules and thus the design requirement would be violated.

Consequences:

- Since no assumptions can be made in UML-RT about when an asynchronous message is delivered, their use can lead to unexpected race conditions - such as bugs.

Fig. 32: Inappropriate use of asynchronous communication

Fig. 33: Implementation of signal acknowledgement through asynchronous communication

Refactored Solution: One way to avoid the dangers of asynchrony is to use synchronous communication. In Figure 32, if we replace the asynchronous *send()* with synchronous *invoke()* signals, the execution will be blocked in this transition until the Controller hears back from both of the *Sender* and the *Receiver* capsules.

Consequences of the Synchronous Solution:

- It will require a fewer number of model elements for implementing the signal acknowledgement mechanism. Therefore, this refactored solution keeps the design simple compared to the asynchronous solution for this antipattern described below.
- The execution of all other capsules running in the same physical thread will be blocked until any reply is received by the sender of the synchronous signal. Therefore, the use of synchronous signals creates the possibility for the model to deadlock. One way to reduce this possibility is to send the signals again after waiting for a certain period of time, if no response has arrived.

Variations: If it is important that the sender of an asynchronous message is informed of successful delivery, acknowledgement messages can be used. Let us consider the example of asynchronous communication in Figure 33. In this example, a composite state is created in between the *Idle* and the *Operating* state which ensures that the subsystem capsules have started before the Controller reaches the *Operating* state.

Inside the *Starting* state, at first a start request is sent to the *Sender* from the *Start_Sender* state asynchronously. The execution will wait in this state until it receives an acknowledgement signal from the *Sender* capsule. Once it receives the acknowledgement, it will move to the *Start_Receiver* state and will follow the same approach for starting the *Receiver*. After getting reply from the *Receiver* capsule, the execution will go to the *Operating* state.

Consequences of the Asynchronous Solution:

- Since all other capsules in the same physical thread are not blocked because of using asynchronous communication, it does not increase the potential for deadlocks.
- A high number of extra model elements is required for implementing signal acknowledgement compared to the synchronous solution.

Considering two different solutions for solving this antipattern, an obvious question is which solution is preferable? The answer to this question entirely depends on the preference of the designer between concurrency and complexity. Asynchronous communication is the preferred form of communication if designers want to minimize concurrency in the model. In addition, due to the blocking characteristic of synchronous communication, if it is used in the behavior of a capsule, the real-time aspects and response time for all other capsules within the same physical thread might be highly affected. However, if a simple design solution with fewer model elements is preferred for incorporating signal acknowledgement, then synchronous communication may still be a good choice.

4.3.10 Incorrect Use of One-shot Timers

Background: There are two kinds of timers available in UML-RT: *One shot timer* and *periodic timer*. The difference between these two types of timers lies on their expiration behavior. One shot timers can expire only once, at any absolute time, or once the specified time duration is over. In contrast, after starting up a periodic timer, it times out repeatedly at the end of the specified period until being cancelled explicitly.

General Form: One shot timers are designed to be used only once, for triggering an event for a single time. However, as we can see in Figure 34, a one shot timer can be used for implementing the functionality of a periodic timer as well.

Fig. 34: Use of a one-shot timer as a periodic timer

Figure 34 illustrates the top-level behavior of a *Controller* capsule which is responsible for sending out status requests to other capsules of the system: *Receiver* and *Jammer*. In this design choice, at first, a one shot timer named *statusTimer* is set in the *toOperate* transition. Once this timer expires after the specified time

Table 1: Comparison of one-shot and periodic timers: timeout interval is 50 milliseconds

Step#	One-Shot Timer (milliseconds)	Periodic Timer (milliseconds)
1	50.338593	50.491826
2	55.546359	49.206204
3	56.264136	49.34177
4	56.045999	49.531488
5	54.813607	50.828782
6	55.006396	49.754233
7	57.178924	49.441621
8	56.427737	49.877126
9	55.927714	50.014228
10	56.708088	50.701281
11	-	49.695858
12	-	50.329875
Total Time	554.257553	599.214292
Average	55.4257553	49.93

period, the self-transition ‘statusRequest’ will fire and the *Controller* will send out status requests to the other capsules. However, before sending out the status requests, the one shot timer *statusTimer* is set once again in the first line of the action code of the *statusRequest* transition. The intention is to get the *statusTimer* fired periodically for requesting statuses from the other capsules of the system on a regular basis until the system transitions to the *Shutdown* state. With a high-level look, everything seems quite fine with this design and one may wonder, what could be possibly wrong with this design choice that makes it an antipattern solution!

When a one-shot timer is used for serving the purpose of a periodic timer, every time the timer expires, it must have to be reset before being employed again. So, in the case of repeated timeouts, the amount of extra time that would be needed to process each timeout and request a new timer could cause slight delay in the intended timing of the associated behavior.

For exploring the effects of using one-shot timers as periodic timers we ran an experiment using the model behavior depicted in Figure 34. Both one-shot and periodic timers are used in two different phases of the experiment with the intention to be fired on every *50 milliseconds*. The total system time is set to *600 milliseconds* and the system will shutdown once the system timer fires. The number of times the timers can fire within the system timer period is *12*. The experiment is done using the Rational RSA-RTE tool on a Windows 8.0 machine having Intel Quad Core 2.66 GHZ CPU and 4.00 GB of RAM.

As we can see in Table 1, the average time period for firing the one-shot timer is approximately *55.43 milliseconds*. In contrast, the average time period for firing the periodic timer is very close to *50 milliseconds*. This is because if a one-shot timer is used for implementing the functionality of a periodic timer, it takes around *6 milliseconds* on average for the timer to be created again after its expiration. In Table 1, a row labeled with *Step#n* shows the actual time elapsed between the *(n-1)-th* and the *n-th* timer expiration events for both types of timers. The corresponding timeline diagram is shown in Figure 35 where rt_i and at_i represent the required timing point and the actual timing point during the receipt of the *i-th* timer expiration event. As we can see in these figures, due to the extra time taken

to create the one-shot timer after its expiration, two required timer expiration events have been missed within a total system time of *600 milliseconds*.

Fig. 35: For a timeout interval of 50 milliseconds, the required and the actual time taken by (a) one-shot timer (b) periodic timer

This deviation of actual timing from the timing requirement specification could force any soft real time systems to have degraded quality. However, its influence can be extremely significant in the design of a hard real time system where it is very important to maintain real-time aspects of the system behavior strictly for avoiding any potential system failures or health hazards.

Symptoms:

- A periodic timer is designed using a one-shot timer.

Consequences:

- Use of a one-shot timer for achieving the functionality of a periodic timer would compromise the real-time aspects of the system behavior. Consequently, it would result in a system with degraded quality.

Known Exceptions: The effects of this timing requirement violation is remarkably noticeable when the timer is required to be fired repeatedly on a relatively smaller timeout period. However, if the required timeout interval is significantly higher than the time it takes to reset a one-shot timer (around 6 milliseconds), the degradation in model quality would not be that much noticeable.

Refactored Solution: The problem introduced in the antipattern solution can be resolved by the use of a periodic timer in UML-RT. A periodic timer is designed to be fired repeatedly after a specified time duration until it has been cancelled explicitly. Therefore, in contrast to one-shot timers, periodic timers are not required to be reset after each expiration. Consequently, using periodic timers instead of one-shot timers for firing an event periodically would improve the timing accuracy we would get (see Table 1 and Figure 35).

4.3.11 Misuse of Guard Conditions

Background: The relationship between a source and a target state is specified using a transition. A transition can have *three* different parts: event triggers, action code and a guard condition. Each of the event triggers associated with a transition represents a pair consisting of an incoming message signal and an associated message interface (i.e., ports), or of a timeout event of a system timer and the relevant timing port. A block of action code that can be injected in a transition allows us to perform various activities such as making operation calls, creating and destroying other objects and sending signals to other objects while the transition is active. In contrast, a guard condition is a Boolean expression that can be optionally associated with a transition. If specified, the guard condition must evaluate to *true* in order for the transition to be fired. However, if no guard is specified, the default evaluation result would be *true* and the transition can be taken immediately after getting triggered.

General Form: Let us consider the scenario of Figure 36 which illustrates the design of a microwave oven that takes user input for going to any of the following modes of operation: *Popcorn*, *Potato*, *Warm*, *Door_Opened*. The system receives user selection for the preferred operating mode while it is in the *Door_Closed* state. Then, once the user hits *Start*, depending on the received user selection, an outgoing transition of the *Door_Closed* state with a satisfied guard condition will be executed and the microwave will travel to the corresponding operating state.

Fig. 36: Misuse of guard conditions

Symptoms:

- There exist multiple outgoing transitions of a state with the same event trigger, but with distinct guard conditions.

Consequences:

- In the design choice of Figure 36, it is not clear if the event triggers associated with the outgoing transitions of the *Door_Closed* state are the same or not. Therefore, the *understandability* of the design model is affected.

- Also, we can not tell by looking at Figure 36, where exactly the *inp* variable gets bound. The system may receive a value for the *inp* variable in the *Door_Closed* state, or maybe the value is set in any of the earlier states or transitions.

Refactored Solution: The problem associated with the antipattern solution can be refactored by employing a choice-point in the design model. As we can see in Figure 37, due to the use of the choice-point *CP*, it is apparent that the transition from the *Door_Closed* state to any of the *four* operating states is caused by the same event trigger. The binding of the *inp* variable in the *userInput* transition can easily be noticed as well.

Fig. 37: Replacement of guards with a choice-point

Known Exceptions: Guard conditions are very helpful if we want to check the validity of a condition before executing a transition once the event trigger associated with the transition is fired. However, when we have multiple outgoing transitions from a state with an identical event trigger and the execution behavior depends solely on the associated guard conditions, it is better to incorporate a choice-point as it improves the *understandability* of the design model.

4.3.12 Hidden States

UML-RT state machine diagram supports two types of states: *simple* and *composite*. A *simple* state is defined as a state that does not contain any other states inside. In contrast, a *composite* state is defined as a state composed of other states, called *sub-states*. Support of hierarchical state machines in UML-RT allows us to model complex state machine behavior by describing a system at multiple levels of abstraction.

The main advantage of using model-driven development over code-driven development is the potential for significant improvement in understandability. For example, a state represents a specific stage of the life-time of an object where it is ready to handle specific events [4,8,31]. The use of states in the behavioral diagram

of UML-RT models allows us to capture the states an object is passing through and how its potential for interacting with other objects changes as a result.

A common problem regarding the behavioral design of a system is the *absence of state elements* where it is appropriate. This problem appears in two forms: *absence of appropriate super-states* and *absence of enough sub-states*. The following two subsections will discuss these two forms of hidden states in a UML-RT design model.

4.3.12.1 Hidden Super States

General Form: Let us consider the scenario of Figure 38. This figure illustrates the high level behavior of a *personal computer (PC)*. A *PC* can be in the *Off* state which represents the scenario when the *PC* is turned off. Once the *PC* is turned on, it can be in any of the following states: *Active*, *Inactive* and *Crashed*.

Fig. 38: Wrong use of state abstraction

In the design of Figure 38, all the four states associated with the behavior of the *PC* are placed in the same abstraction level. With the triggering of the *toActive* event, we can move to the *Active* state. Once the system is in the *Active* state, after a certain period of inactivity, the *toInactive* event will be fired and the system will be in the *Inactive* state. The system can go to the *Crashed* state if the system is crashed while it is in either the *Active* or the *Inactive* state. The system can also come back to the *Off* state if the *toOff* transition is fired while the system is in any of the *Active* or the *Inactive* states.

We note that the model contains duplicated model elements. For example, the events that take the system from the *Active* and the *Inactive* states to the *Crashed* state are identical. The same is true for the events that take the system back to the *Off* state from the *Active* and the *Inactive* states. Presence of these redundancies gives us the hint that the design can be improved by removing redundancy and explicitly showing the relationship between UML states by grouping them.

Symptoms:

- Duplication is present in the behavioral design of a UML-RT model. The duplication typically involves outgoing transitions from multiple states in the same level of abstraction having identical event triggers, guards and action code.
- State nesting is not used at all, or used to a smaller extent.
- If being in a state has to do with a certain property being true and the system behaving in a certain way, then affected states would share properties and behavior.

Consequences:

- Unnecessary duplication can make the system *hard to maintain* as any future changes in the duplicated elements would require to make the changes separately in all the elements.
- Wrong use of state abstraction would make the system *harder to understand* because the fact that a group of states share invariants and the ability to respond to the same set of messages in the same way, is not captured explicitly.

Fig. 39: Proper use of state abstraction

Refactored Solution: The existence of identical outgoing behaviors from both of the *Active* and the *Inactive* states gives us the indication that these two states actually belong to the same phase in the lifecycle of the object: the phase in which the system is ‘on’. Consequently, the antipattern solution can be improved by nesting these two states inside a composite state. In Figure 39, a new composite state *On* is created in the top level behavior of *PC* which is composed of *Active* and *Inactive* states. Now, instead of having redundant outgoing transitions in the state diagram, two outgoing group transitions from the *On* state are used for taking the system to the *Off* and the *Crashed* state.

Consequences:

- The enforcement of the refactored solution would greatly improve the *maintainability* and the *understandability* of the design model.
- Unnecessary redundancy is eliminated from the design model.

4.3.12.2. Hidden Sub States

General Form: Let us consider the behavioral design choice of an adaptive cruise control software presented in Figure 40. This design choice is responsible for activating different states of a cruise control such as *accelerate*, *brake* and *idle* depending on the value of the following two variables: *speed* and *distance*. The conditions for activating these states are as follows:

Accelerate: $g1 = (\text{Speed} < 80 \text{ km and Distance} > 10 \text{ m})$

Brake: $g2 = (\text{Speed} > 120 \text{ km or Distance} < 3 \text{ m})$

Idle: $g3 = \neg(g1 \vee g2)$

Fig. 40: Existence of hidden states in a model

Now, as we can see in Figure 40, instead of using the actual state elements, all the above mentioned states are represented using self-transitions and flag variables. Consequently, some crucial properties of the design remain hidden in this design choice, i.e., the opportunity to explicitly show circumstances where state changes occur and what properties the attribute values must satisfy in each state is lost.

Symptoms:

- Attributes are used to encode important phases during the execution of an object.

Consequences:

- The design becomes more complex because the determination if the object is in a hidden substate may require the use of complex constraints on attributes. Together with the fact that important information about the object and its behavior is hidden, this complexity impacts the *understandability* and *maintainability* of the model negatively.

Refactored Solution: The antipattern solution can be improved by making the hidden states visible. As we can see in Figure 41, the understandability issue of the antipattern solution is resolved by representing the hidden states explicitly. The attached note to each of the state elements having state invariants illustrates the crucial system properties represented by the state elements. More precisely, the explicit interactions among these states make this design choice easier to understand and maintain.

4.3.13 Inappropriate Modeling Scope

Background: In code-driven development, the term *scope* mainly refers to the code region in which declared entities are visible. For instance, the scope of

Fig. 41: Refactoring hidden states

a variable defines the part of a computer program where the variable is visible. Variables can have any of the following two scopes: *global* and *local* [51]. While declaring a variable once with the global scope makes it accessible in the entire program, a local variable can be accessed only in the block where it is declared.

If global scope is used in code, the code can become hard to understand because the corresponding variables can be used and updated anywhere in the program. In addition, the use of global scope makes the code error-prone as global variables are accessible to all threads of execution and consequently, the code will not remain thread-safe [12,44]. Considering the lack of understandability in code where global variables are used, the authors in [59] suggested the removal of global variables from all higher level programming languages. Similar to this convention, it is a good design choice to avoid unnecessary use of global scope in model-driven development as well.

General Form: Let us consider the example of Figure 42 where two capsules *Second* and *GrandChild_1* are communicating with each other using the protocol *Pro_Second_GC1*. The *Second* capsule is contained by the *Container* capsule and the only capsule it is communicating with is the *GrandChild_1*. On the other hand, the *GrandChild_1* is located three levels deep from the *Container* capsule. To be precise, the *Container* capsule contains the *Parent* capsule which is the owner of the capsule *Child_1* that contains *GrandChild_1*. Consequently, for the communication to happen between *Second* and *GrandChild_1*, the message signals will have to go through ports in three different levels before reaching the destination.

Symptoms:

- There exist capsules that are unnecessarily defined with larger scopes.

Consequences:

- Real time aspects can be affected as travelling through multiple ports requires extra time.
- Since the associated signals need to travel through some extra ports, additional system resources can be used unnecessarily.

Fig. 42: Global capsule scope in UML-RT

Known Exceptions: Although it is good practice to avoid using global variables in code-driven development, in situations where global variables represent facilities that truly need to be available in the entire program, use of the global scope simplifies the code [12]. Similarly, in the above example, if the *Second* capsule needs to communicate with other capsules as per design requirement, it would be better to have the antipattern solution.

Refactored Solution: The antipattern solution can be refactored by changing the scope of the *Second* capsule. This can be done by changing the owner of this capsule (see Figure 43).

Consequences:

- The refactored solution improves the *understandability* and *performance* of the design model.

5 Conclusion

According to some statistics, almost 33% of all software projects are cancelled, 66% suffer from cost overruns exceeding 200% and the failure rate among all software projects is more than 80% [9]. One of the main reasons for massive software failure rate is poor software design quality [34]. Following a set of software development guidelines can play an effective role in achieving and maintaining good software quality. It is especially important in the development of real-time embedded software systems where the software complexity can be extremely high.

Fig. 43: Refactored capsule scope

Although a lot of work has been carried out on assessing the quality of different kinds of models including UML models, no prior work focuses on finding good and bad design choices in UML-RT models. The UML-RT profile uses some terminology and features which are different from the UML standard [21, 46] and these differences make most of the guidelines already proposed in the literature inapplicable for UML-RT development. For addressing this research issue, based on analyzing a set of UML-RT models from industry and academia, this paper introduces a set of design conventions, patterns and antipatterns for developing model-based real-time embedded software systems. For improving the understandability and enhancing the clarification of our discussion, most of the guidelines are illustrated with examples.

5.1 Summary of the Proposed Guidelines

The UML-RT models we analyzed come with a wide range of design decisions. Some of the interesting facts noticed during the study include recurring misuses of some UML-RT features; these misuses then gave rise to some of the presented antipatterns. For example, our study reveals a common tendency of many developers to overuse self-transitions or local transitions. While working solutions can be achieved this way, the resulting system can become unnecessarily complex and difficult to understand and maintain.

Existence of these commonly occurring problems motivated us to create this catalog of design guidelines. The proposed guidelines are mostly applicable for low-level structural and behavioral modeling in UML-RT. While most of these guidelines are UML-RT specific, some of them are relevant for UML development

Table 2: Summary of conventions

Name	Short Description	Category	Quality Attribute (s)	Relevant Modeling Element (s)
Cancellation of Timers after Their Use	Timers should be cancelled if they are not intended to be used again	Behavior	Performance	Timers, Action Code
Termination of Created Capsules after Their Use	Dynamically created capsule instances should be destroyed after serving their purpose	Structure	Performance	Optional Capsule Roles
Removal of Unconnected Ports	If a port is not connected with anything else, it should be cleaned from capsule structure	Structure	Performance	Ports
Removal of Unreachable Artifacts	Unreachable artifacts, i.e., unreachable states, capsules etc. should be removed	Structure, Behavior	Performance, Understandability	Capsules, States, Ports
Proper Use of Initial Transitions	Operations having dependencies on external capsules should be avoided in initial transition of the top-level state of a capsule	Behavior	Reliability	Initial Transitions
Use of Internal Self-transitions instead of External	External self-transitions generate methods for entry and exit actions even if they are empty	Behavior	Performance	Self-transitions
Conjugation of Server-side Ports in a Binary Protocol	Minimize the user effort as the number of clients are usually greater than the number of servers	Structure	Maintainability, Readability, Consistency	Protocol Roles
Avoid Unnecessary Use of Active Classes	If a passive class can serve the purpose, it should be used instead of using a capsule	Structure	Performance	Capsules
Improving the Visualization of UML-RT Diagrams	Use of a large diagram showing different types of relationships should be avoided	Structure, Behavior	Understandability, Maintainability	Class Diagrams, State Machine Diagrams

as well. For example, the antipattern “Hidden States” is applicable for any UML development scenarios supporting Hierarchical State Machines (HSM) [45].

Some of the guidelines we proposed are grouped together based on their similarities with respect to the affected quality attributes or the modeling elements they are concerned about. For example, *four* conventions related to resource utilization in UML-RT are presented in *Section 4.1.1* and *two* conventions related to UML-RT transitions are presented in *Section 4.1.2*.

Table 2, 3 and 4 summarizes the proposed conventions, patterns and antipatterns respectively. In the tables, each row belongs to a guideline; the row contains

Table 3: Summary of patterns

Name	Short Description	Category	Quality Attribute (s)	Relevant Modeling Element (s)
Status Monitoring Scenario	A general reusable solution for collecting unordered status information from a number of capsules and report them to a controller capsule	Behavior	Simplicity, Performance, Robustness	States, Transitions, Action Code
Proper Use of System Timers	Pattern for allowing a system to properly follow the timing requirement specification	Behavior	Correctness	Timers, Action Code
Separation of Responsibilities	Distribute responsibilities among multiple capsule ports	Structure	Understandability, Maintainability	Ports, Protocols

the *name* of the guideline, a *short description*, the *modeling elements* involved, and the *quality attributes* affected.

5.2 Future Work

The next step of this research is the development of tool support for the proposed guidelines. This tool support is motivated by the success of different code-oriented analyzers [22, 27, 56] and would allow developers to detect the presence of bad design choices automatically and refactor them accordingly. In addition, once we are done with the development of the prototype tool, we would try to validate the correctness, performance and usability of this application by applying it in industrial environment.

Acknowledgements We would like to thank *Dr. Ron Smith* from *Royal Military College of Canada* for giving us access to a student repository of UML-RT models. We would like to express our gratitude to *Bran Selic* from *Malina Software Corp.* for his valuable feedback on our work. We would also like to thank our industrial research partner *Ericsson Inc.* for their great support. In addition, this work is supported in part by NSERC, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp.

References

1. M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension. In *Proceedings of the 15th European Conference on Software Maintenance and Re-engineering (CSMR '11)*, pages 181–190, Washington, DC, USA, 2011. IEEE Computer Society.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
3. M. Akroyd. Antipatterns session notes. San Francisco, USA, 1996. Object World West.

Table 4: Summary of antipatterns

Name	Short Description	Category	Quality Attribute (s)	Relevant Modeling Element (s)
Guards in Junction Points are Affected by Action Code in Incoming Transition	Outgoing guards of a junction point are evaluated before executing action code of the incoming transitions	Behavior	Correctness, Understandability	Junction Point, Action Code, Guard Conditions
Overlapping Conditionals	Outgoing guards in a model element should be mutually exclusive, otherwise we can run into an undesirable situation	Behavior	Maintainability, Correctness, Understandability	Choice Points, Guard Conditions
Choice-point with Non-exhaustive Guards	Can result in ill-formed model behavior	Behavior	Integrity	Choice Points, Guard Conditions
Inappropriate Use of Self-transitions	Self-transitions do not express the system behavior explicitly and can possibly hide important system states	Behavior	Maintainability, Understandability	Self-Transitions
Misuse of Local Transitions	Excess use of local transition makes the system hard to understand and maintain	Behavior	Maintainability, Understandability	Local Transitions
Incorrect Use of Group Transitions	If group transitions are not used properly, preciseness of the model behavior can be affected	Behavior	Understandability, Correctness	Group Transitions
Absence of Inheritance	Illustrates different forms of inheritance in UML-RT and cases where the inheritance features are not utilized	Structure, Behavior	Reusability, Maintainability	Capsule Inheritance, Protocol Inheritance, State Machine Inheritance
Pitfalls of Using Promote/Demote Operations in UML-RT Inheritance	Depicts scenarios where using these features of UML-RT can be problematic	Structure, Behavior	Understandability, Integrity	Capsule Inheritance, Protocol Inheritance, State Machine Inheritance
Incorrect Use of Asynchronous Communication	Illustrates scenarios where unprotected form of asynchronous communication should be avoided	Behavior	Reliability, Simplicity	Protocols, States, Transitions, Action Code
Incorrect Use of One-shot Timers	Use of one-shot timers for achieving periodic functionality can give us erroneous behavior	Behavior	Correctness	Timers, Action Code
Misuse of Guard Conditions	Inappropriate use of guards makes the system hard to understand	Behavior	Understandability	States, Transitions, Guard Conditions
Hidden States	Illustrates the appearance of hidden super- and sub-state in UML-RT state machine	Behavior	Maintainability, Understandability	States, Transitions
Inappropriate Modeling Scope	Use of appropriate scope keeps the model communication simple	Structure	Performance	Class Diagrams, Capsule Structure Diagrams

4. S. W. Ambler. UML 2 state machine diagrams: An agile introduction. <http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm/>. [Online; accessed 20-January-2016].
5. D. Arcelli, V. Cortellessa, and C. Trubiani. Antipattern-based model refactoring for software performance improvement. In *(QoSA '12)*, pages 33–42. ACM, 2012.
6. M. Balaban, A. Maraee, A. Sturm, and P. Jelnov. A pattern-based approach for improving model quality. *Software and Systems Modeling (SoSyM)*, 14(4):1527–1555, 2015.
7. G. Bellekens. UML best practice: 5 rules for better UML diagrams. <http://bellekens.com/2012/02/21/uml-best-practice-5-rules-for-better-uml-diagrams/>. [Online; accessed 20-January-2016].
8. R. B'far. *Mobile Computing Principles: Designing and Developing Mobile Applications with UML and XML*. Cambridge University Press, New York, NY, USA, 2004.
9. W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.
10. F. J. Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu. Automatic code generation from design patterns. *IBM Syst. J.*, 35(2):151–171, May 1996.
11. V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani. Digging into UML models to remove performance antipatterns. In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems, QUOVADIS '10*, pages 9–16, New York, NY, USA, 2010. ACM.
12. Cunningham and Cunningham Inc. Global variables are bad. <http://c2.com/cgi/wiki?GlobalVariablesAreBad>. [Online; accessed 20-January-2016].
13. T. K. Das and J. Dingel. State machine antipatterns for UML-RT. In *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 54–63, 2015.
14. B. P. Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.
15. Eclipse. Papyrus for real time (Papyrus-RT). <https://projects.eclipse.org/proposals/papyrus-real-time-papyrus-rt>. [Online; accessed 20-January-2016].
16. M. D. P. Emilio. *Embedded Systems Design for High-Speed Data Acquisition and Control*. Springer Publishing Company, Incorporated, 2014.
17. International Organization for Standardization. ISO/IEC 25000:2005, Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – guide to SQuaRE. http://www.iso.org/iso/catalogue_detail.htm?csnumber=35683. [Online; accessed 20-January-2016].
18. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
19. R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering(FOSE '07)*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.
20. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
21. S. Gopinath. Real-time UML to XMI conversion. Master's thesis, KTH Computer Science and Communication, Stockholm, Sweden, 2006.
22. GrammaTech. FDA recommends static analysis for medical devices. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.194.293&rep=rep1&type=pdf>, 2010. [Online; accessed 20-January-2016].
23. T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *Proceedings of the 14th international conference on Formal Methods (FM'06)*, pages 1–15. Springer-Verlag Berlin, Heidelberg, 2006.
24. IBM. Modeling real-time applications in RSARTE. https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W0c4a14ff363e_436c_9962_2254bb5cbc60/page/Modeling%20Real-Time%20Applications%20in%20RSARTE. [Online; accessed 20-January-2016].
25. IBM. Rational Rose Real Time (RT) documentation: Capsule instances and capsule behavior. ftp://ftp.software.ibm.com/software/rational/docs/v2003/win_solutions/rational_rosert/rosert_java_ref_guide.pdf. [Online; accessed 20-January-2016].

26. IBM. Rational Rose RealTime. <ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/rosert.html>. [Online; accessed 20-January-2016].
27. R. P. Jetley, P. L. Jones, and P. Anderson. Static analysis of medical device software using codesonar. In *Proceedings of the 2008 Workshop on Static Analysis, SAW '08*, pages 22–29, New York, NY, USA, 2008. ACM.
28. F. Khomh, M. D. Penta, Y. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.
29. H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.
30. Nancy Leveson. Medical devices: The therac-25. <http://sunnyday.mit.edu/papers/therac.pdf>. [Online; accessed 20-January-2016].
31. Sparx Systems Pty Ltd. UML 2 state machine diagram. http://www.sparxsystems.com/resources/uml2_tutorial/uml2_statediagram.html/. [Online; accessed 20-January-2016].
32. MathWorks Automotive Advisory Board (MAAB). Control algorithm modeling guidelines using MATLAB Simulink and Stateflow version 2.0. http://www.idsc.ethz.ch/Courses/embedded_control_systems/Exercises/Maab_styleguide_v_2_0.pdf/, 2007. [Online; accessed 20-January-2016].
33. Adam Neal. RSARTE cheatsheet. https://www.ibm.com/developerworks/community/wikis/form/anonymous/api/wiki/b7da455c-5c51-4706-91c9-dcca9923c303/page/a7287c71-8e14-429d-80df-a0051a8b44b7/attachment/2eb38295-77c0-48e5-86e2-5903e4b4c224/media/cheatsheet._adam.pdf. [Online; accessed 20-January-2016].
34. E. E. Ogheneovo. Software dysfunction: Why do software fail? *Journal of Computer and Communications*, 2:25–35, 2014.
35. Object Management Group (OMG). How to deliver resilient, secure, efficient, and easily changed it systems in line with CISQ recommendations. http://www.omg.org/CISQ_compliant_IT_Systemsv.4-3.pdf. [Online; accessed 20-January-2016].
36. Oracle. Comments. <http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-141999.html#385>. [Online; accessed 13-June-2016].
37. Oracle. Indentation. <http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html>. [Online; accessed 13-June-2016].
38. Oracle. Naming conventions. <http://www.oracle.com/technetwork/java/codeconventions-135099.html>. [Online; accessed 13-June-2016].
39. Oracle. Code conventions for the Java programming language. <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>, 1999. [Online; accessed 20-January-2016].
40. T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7:55–90, 2008.
41. R. S. Pressman. *Software Engineering: A Practitioner's Approach (2nd Ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1986.
42. Kepler Project. Software development guidelines. <https://kepler-project.org/developers/reference/software-development-guidelines>. [Online; accessed 20-January-2016].
43. D. Romano, P. Raila, M. Pinzger, and F. Khomh. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In *(WCRE '12)*, pages 437–446. IEEE Computer Society, 2012.
44. Michael Safyan. Avoid global variables, environment variables, and singletons. <https://sites.google.com/site/michaelsafyan/software-engineering/avoid-global-variables-environment-variables-and-singletons>. [Online; accessed 20-January-2016].
45. M. Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes Newton, MA, USA, 2008.
46. B. Selic. Using UML for modeling complex real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98)*, pages 250–260, London, UK, 1998. Springer-Verlag.
47. B. Selic, G. Gullekson, J. McGee, and I. Engelberg. ROOM: an object-oriented methodology for developing real-time systems. In *Computer-Aided Software Engineering, 1992. Proceedings., Fifth International Workshop*, pages 230–240, July 1992.
48. B. Selic, G. Gullekson, and P. T. Ward. *Real-time Object-oriented Modeling*. John Wiley & Sons, Inc., 1994.

49. Bran Selic. An architectural pattern for real-time control software. In *Pattern Languages of Program Design 2*, pages 4–6. Addison-Wesley, 1996.
50. C. U. Smith and L. G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *CMG Conference*, pages 717–725, 2011.
51. Thunderstone Software. Variable scope: Global vs. local. https://www.thunderstone.com/site/vortexman/variable_scope_global_vs_local.html. [Online; accessed 20-January-2016].
52. G. Sunyé, A. L. Guennec, and J. Jézéquel. *Design Patterns Application in UML*, pages 44–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
53. G. Sunyé, D. Pollet, Y. L. Traon, and J. Jézéquel. Refactoring UML models. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, London, UK, UK, 2001. Springer-Verlag.
54. S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan. Predicting bugs using antipatterns. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*, pages 270–279, Washington, DC, USA, 2013. IEEE Computer Society.
55. C. Trubiani and A. Koziolk. Detection and solution of software performance antipatterns in Palladio architectural models. In *(ICPE '11)*, pages 19–30. ACM, 2011.
56. N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering, CSMR '08*, pages 329–331, Washington, DC, USA, 2008. IEEE Computer Society.
57. WEBster. Software development guidelines. <http://www.literateprogramming.com/sdg.pdf>. [Online; accessed 20-January-2016].
58. S. Wong, S. Vassiliadis, S. Vassiliadis, and S. Cotofana. Embedded processors: Characteristics and trends. Technical report, in *Proceedings of the 2001 ASCI Conference*, 2004.
59. W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, February 1973.
60. H. Xi. Dead code elimination through dependent types. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages, PADL '99*, pages 228–242, London, UK, UK, 1998. Springer-Verlag.