Model Development Guidelines for UML-RT

Tuhin Kanti Das and Juergen Dingel
{das, dingel}@cs.queensu.ca

Technical Report 2016-628

School of Computing, Queen's University,
Kingston, Canada.

February 01, 2016

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 report discusses a set of design guidelines including design conventions, patterns and anti-patterns for developing real-time embedded software systems. These guidelines have been identified based on our analysis of around 100 UML-RT models from industry and academia.

Contents

List of Figures

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 [27]. 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 [66]. ROOM was aligned with the Unified Modeling Language (UML) which gave rise to the *real-time profile of UML (UML-RT)* [65]. Just 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) [35], IBM RSA-RTE [33] and the open source tool PapyrusRT [22].

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 anti-patterns for UML-RT models is introduced in our previous work [17]. This report extends this work and presents a comprehensive set of model development guidelines for UML-RT including a number of model-based *design conventions*, *patterns* and *anti-patterns*. During this research, our ultimate goal is to design, implement and evaluate tool support for the automatic detection of indicators of good or bad design based on these model development guidelines.

This report is structured in the following way: Section 2 presents relevant background terminology for software development guidelines in general. This is followed by the introduction of the design guidelines for UML-RT in Section 3. In addition to presenting model-based conventions, patterns and antipatterns, this section also discusses a number of smells regarding duplications in UML-RT models. Finally, a discussion on related work is presented in Section 4.

1.1 Motivation

In [36], the authors introduce a static analysis tool CodeSonar which can be used to perform a whole-program, inter-procedural analysis on C/C++ code. This tool checks the code for runtime exceptions and C-language library violations through a compile-time analysis, and can be used for identifying code smells related to null-pointer dereferences, unreachable code, concurrency defects in multi-threaded software, buffer overruns etc. that can result in system crashes, memory corruption, and other serious problems. This analysis tool is

recommended by the United States Food and Drug Administrator (FDA) to address the problem of defective software in medical devices [30]. In [75], a code smell detection tool named JDeodorant is introduced in the form of an Eclipse plugin. This tool can be used to automatically detect and remove type checking bad smells in Java source code. In [24], the authors share their experience of using a number of tools for code smell detection. In addition to emphasizing the helpful nature of using these tools, they point out some of the challenges for comparing different analysis tools.

Our research work is motivated by the success of these source code analysis toolkits. These toolkits have proven to be successful in analyzing industrial-strength code and are becoming part of recommended best practices. In a similar fashion, the development of tool support for analyzing models for embedded real time software systems could prove equally successful and influential. In fact, during our collaboration with an industrial research partner, Ericsson Inc., the ongoing development of a model analyzer tool has been observed which is believed to be very helpful for analyzing Ericsson models.

2 Background

2.1 MDE for Embedded Real-Time Systems

2.1.1 Design Model

A *software design model* is an abstract representation of a system. In MDE, designing a system involves the process of deriving a design model from requirement specifications from which a system can be generated more or less automatically [32].

2.1.2 Real Time Embedded Systems

Similar to most computing systems, *an embedded system* consists of hardware, software and an environment. However, the associated physical constraints do not allow the embedded systems to follow one of the central ideas that has enabled much of the progress in Computer Science: the separation of software from the technical specifics of the underlying hardware allowing software to be written in more abstract, application specific terms. In contrast, the design of embedded systems requires the integration of essential paradigms from hardware design, software design, and control theory in a consistent manner.

A *real-time embedded system* is an engineering artifact involving computation that is subject to physical constraints such as reaction constraints and execution constraints [32]. Common reaction constraints associated with a real-time embedded system include the specification of deadlines, throughput and jitter which typically originate in the behavioral requirements of the system. On the other hand, common execution constraints limit the availability of processor speeds and power; and hardware failure rates come from the hardware constraints of these systems [32].

Due to the increasingly reactive and distributed nature of the complex distributed real-time software systems, data processing centers are also migrating to the real-time domain [66]. Because of the high complexity associated with the design and realization of these systems, development typically involves large teams. Since it requires 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 because in addition to simplifying the initial construction, it also facilitates the evolution of the system [65].

The term "real-time" covers a wide spectrum of systems ranging from purely time-driven to purely event-driven. These systems can also be categorized as *soft real time systems* and *hard real time systems*. In the former class, some occasional deadline misses can be tolerated. However, it is extremely important to follow the timing requirements in the latter class as a missed deadline can lead to disastrous consequences such as loss of life or property [28].

2.1.3 MDE: An Approach to Deal with Complexity

Although modern programming languages as well as the supporting integrated development environments (IDEs) have advanced significantly, developing complex real-time embedded software systems using current code-centric technologies still requires grueling effort. One of the most important factors that makes the development of complex software systems difficult is the wide conceptual gap between the problem and implementation domains. For bridging this gap, the MDE vision of software development has been introduced where models are the primary artifact and computer-based technologies are used to analyze the design models and transform them into running systems. An important objective of MDE research is the production of technologies that protect software developers from the complexities of the underlying implementation platform. For example, the ROOM modeling language has been introduced based on three principles [66]: the key modeling concepts should to be domain specific and intuitive; support should be given for automatically generating an efficient implementation from a design model; and it should be feasible to construct and verify initially abstract executable analysis and design models which can be refined into final versions gradually.

2.1.4 UML-RT: The Real-Time Profile for UML

Object orientation is very effective for coping with software complexity. The de-facto standard object-oriented modeling language, the *Unified Modeling Language (UML)* is gaining interest among the real-time community. An important feature of UML is its extensibility mechanism through the use of stereo types, tagged values and profiles. This feature allows the UML to be extended for representing the specificities of a particular domain. In [65], based on the concepts of the field-proven ROOM modeling language, a set of structural modeling

concepts has been defined for facilitating the specification of complex software architectures for real-time systems. As these modeling constructs have a fully formal semantics, the correctness of the models created using these constructs can be verified formally. In addition, executable models can be developed using these modeling constructs for achieving early validation of high-level design and analysis models. The MDE tools mentioned above use UML-RT for compiling design models into code and link it with the runtime system. A review of the most important UML profiles for the development of real-time systems is presented in [28].

However, UML-RT does not support hard real-time constraints. In [31], for bridging the gap between a logical UML-RT model and its real-time implementation on the target platform, a schedulability analysis algorithm is modified for making it compatible with the native runtime model of the Rose-RT tool.

2.2 Definitions and Terminology

2.2.1 Software Quality

In the area of Software Engineering, two related but distinct notions exist wherever software quality is defined in a business context: *software functional quality* and *software structural quality*. The conformance of software to a given design, based on functional requirements or specifications is represented by the functional quality [83]. It is typically enforced and measured through software testing. On the other hand, the structural quality reflects the degree to which the software is produced correctly. So, it represents how well the software complies with the non-functional requirements such as, reliability or maintainability that support the delivery of the functional requirements. In a code-oriented development setting, the evaluation of software structural quality is done through the analysis of its source code. Effectively, it determines how well the software adheres to the sound principles of software architecture outlined in, e.g., [55].

2.2.2 Quality Attributes

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 [25] 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 [45]. This motivates the use of regulations and oversight for the development of safety critical software as found in, e.g., in medical and other devices. In addition, an application with good structural software quality is much easier to understand and change in response to changing business needs. Therefore, the associated costs for maintaining such a software can be expected to be significantly less.

The following section will discuss the techniques proposed in the literature for achieving software quality attributes:

2.2.3 Guidelines

Software development guidelines are defined as a set of recommended best practices that can be followed by a community to improve the productivity of existing development and to produce consistent deliverables [76]. They usually comprise a comprehensive set of standards and practices that should be followed when developing software (See Figure 1).

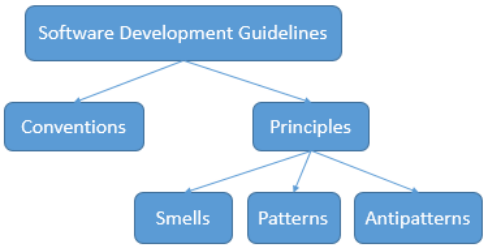

Figure 1: Classification of Relevant Terminologies

2.2.4 Conventions

Software development conventions are the standards used in a development community to improve readability and maintainability of a software system. It is particularly important in a multi-developer project where developers can read and understand each other's code more easily if a set of conventions is followed [60]. Typically conventions are recommended to be followed strictly as they are not subject to trade-offs and typically do not have any negative impact on the quality of the design. Full benefits of conventions can only be achieved when all developers follow them. Conventions are not usually enforced by compilers, but some other tools are used for ensuring their practices.

2.2.5 Principles

In contrast to conventions, *software development principles* can be affected by many factors and are usually subject to trade-offs. Following a set of principles assists software developers to use the best practices for some particular situations, but the same principles might not be very effective in some other situations. Therefore, special care is needed for defining software development principles to ensure appropriate use. Usually, they are defined with some possible tradeoff scenarios. Principles are typically comprised of smells, patterns and antipatterns.

2.2.6 Metrics

A *metric* is the mapping of a particular characteristic of a measured entity to a numerical value. *Software metrics* are used to control the quality, size and complexity of the software development. Metrics are usually defined with statistical threshold values that divide the range of a metric value into different regions. Depending on the region a metric value is in, an informed assessment can be made about the measured entity [44].

2.2.7 Smells

In software development, a *smell* is any symptom in any phase of the development that possibly indicates a deeper problem. Martin Fowler defined a smell as a surface indication that usually corresponds to a deeper problem in the system [26]. Smells are usually not incorrect and do not prevent the software from functioning. So, instead of being treated as software bugs, they are considered to be indications of weaknesses in the code or design that may slow down the development process or increase the risk of bugs or failures in the future [78].

2.2.8 Patterns

A *software pattern* is a general reusable solution to a commonly occurring problem within a given context in software design. Patterns are formalized best practices that can be used by software developers for solving common problems when designing an application or system [82].

2.2.9 Antipatterns

An *antipattern* is used for describing a commonly occurring solution to a problem having some decidedly negative consequences. An antipattern is usually described with a good alternative solution that is documented, repeatable and proven to be effective [77].

3 Design Guidelines for UML-RT

This section introduces a set of software development guidelines for designing UML-RT models. This set includes nine design conventions, six antipatterns and three patterns. 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 of a simple Electronic Warfare System (EWS) in Royal Military College of Canada, inspecting a number of telecommunication models at Ericsson Inc. during a research visit of six weeks helped us identify these design guidelines. 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 at Ericsson

are higher than in the models of the student model repository. The maximum depth of state nesting observed in the Ericsson 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.

3.1 Design Conventions

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 will discuss a set of nine design standards for UML-RT.

3.1.1 Resource Utilization

Traditionally, digital systems are 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, consumer-electronics devices, etc. 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 in everyday human activities, the authors in [84] defined embedded systems as application specific systems which are mass produced only.

An *embedded system* is a computer controlled device which is mainly designed to perform specific tasks. A very large class of embedded systems, namely the *real-time embedded systems*, is characterized to have time constrained behavior [23]. Real-time embedded software is often subject to limited resources, concerning storage capacity, internal memory, power constraints etc. [42, 23]. This is especially true for many mobile embedded systems which are both computational and energy constrained. It is important to utilize the limited resources available within these real-time embedded systems for the proper functioning of the system. This section discusses four conventions related to resource utilization in UML-RT.

Similar to the concept of code cleaning, importance of model cleaning is significant as it improves the understandability and maintainability of the corresponding model. Model cleaning helps us achieve better performance by utilizing system resources properly. The final two conventions mentioned in this section are related to the proper cleaning of UML-RT models.

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 a good practice to ensure that all the 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 any timeout events unexpectedly. This is truly 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 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 in the capsule behavior with the following method call:

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

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

Rationale

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

2. Termination of Created Capsules after Their Use

Background: A capsule's structure is represented by the specification of the type of capsules called *capsule roles* that can exist in the capsule's collaboration. Capsule roles are strongly owned by the container capsule and consequently, their existence is depended on the existence of the container capsule.

A capsule role can fall in any of the following three types: fixed, optional and plug-in. If the type is specified as *fixed*, which is the default type, the creation and destruction of the capsule role is automatically done with that of the container capsule. In contrast, *optional* capsule roles can be created

in the container capsule behavior after the creation of the container only if their existence is necessary and can be destroyed before the container capsule terminates. On the other hand, capsule roles defined with *plug-in* type are mainly used as place holders 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 and the corresponding connections are established automatically.

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 cause runtime performance overhead [34]. 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*.

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.

Convention: It is a 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 never been used. Also, the removal of a connector from a structural diagram may leave a port unconnected (See Figure 2).

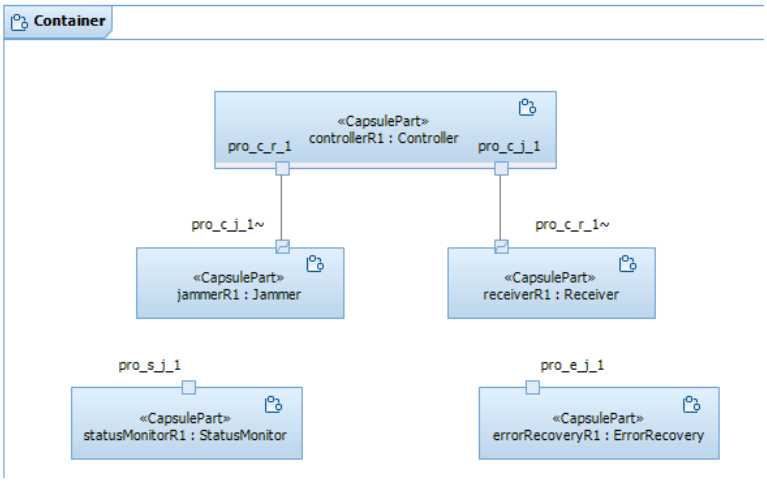

Figure 2: Unconnected Ports

Rationale:

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

4. Removal of Redundant Artifacts

Background: If we consider the concept of redundancy in code-driven development, terminologies exist in the current state of the art are *dead code* [86], *dead computation* [1] and *partial dead code* [41].

Code that can never be executed at runtime are considered dead code. An example of dead code is illustrated in Figure 3.

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

Figure 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 or transitions that can never be taken (due to triggers that will never be matched by an incoming message or unsatisfiable guards), and if the missing transitions are not added through specialization and inheritance, these states can be treated as unreachable (See Figure 4).

If the model is still under development, the existence of such states will be likely and acceptable as they can be updated as the model evolves. However,

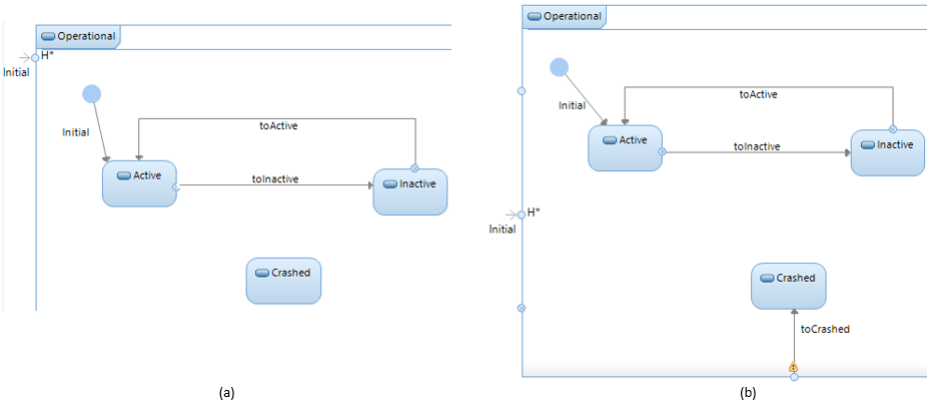

Figure 4: Unreachable States (a) Isolated State (b) State without any triggered transitions

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;
```

Figure 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.

Another form of redundancy in a UML-RT model is the existence of unused state entry 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 a good practice to remove these artifacts from the final model as they do not have any influence on the model outcomes. However, exceptions can be made if these redundant artifacts are planned to be used as the model evolves.

Rationale

- Removal of redundant 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.

3.1.2 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.

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 a 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 initiated 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 requires the 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 need a capsule to rely on 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.

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 makes these two types of self-transitions to be used alternatively with minor changes in the design of the associated state machine diagram.

Convention: It is a 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).

```
protected void chain3_self_internal() {
    rtChainBegin(3, "self_internal");
    rtTransitionBegin();
    rtTransitionEnd();
}
```

(a)

```
protected void chain3_self_external() {
    rtChainBegin(3, "self_external");
    rtExitState();
    rtTransitionBegin();
    rtTransitionEnd();
    rtEnterState(3);
}
```

(b)

Figure 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 [1], it would give us improved *performance*.

3.1.3 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 have to 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 in 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: i.e., 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 the *conjugation*. The MDE tools such as IBM RSA RTE and IBM Rose RT support only binary protocols.

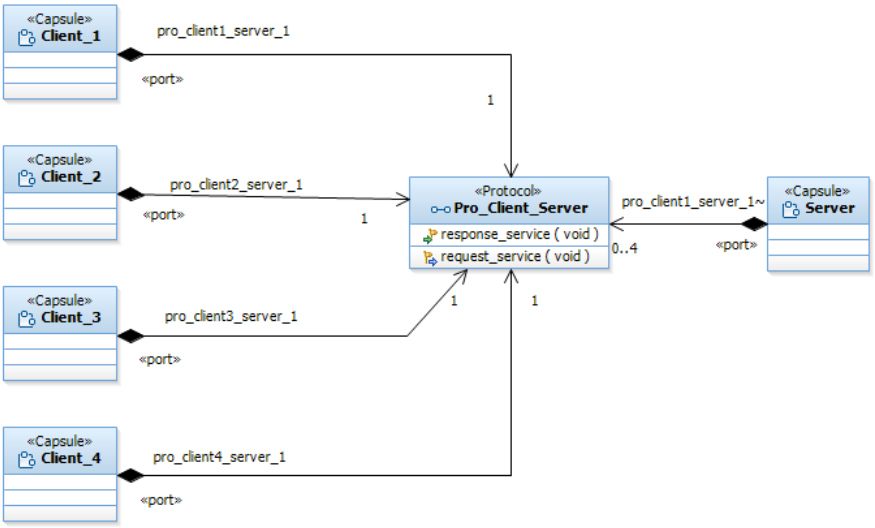

Figure 7: Server-side Conjugation

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 to this benefit, 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.
- Improvement of understandability and maintainability of the design model.
- Reduction in modeling effort.

3.1.4 Avoid Unnecessary Use of Active Classes

Background: An active class in UML-RT is a capsule which is the fundamental modeling element of embedded real-time systems. 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, capsule roles etc which distinguish them from passive classes. Ports and capsule roles of an active class in UML-RT are mainly used for enhancing the modeling capabilities of the structure of the classes. A capsule can also have state machines for modeling the behavior of the classes. 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.

3.1.5 Improvement of Visualization in 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, passive classes

etc. 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 well-known UML-RT development tools such as IBM Rational Rose RT and IBM RSA RTE is the capability of customizing the display of class diagrams without having any effects in the actual 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 it is pointed out in [10], if a diagram is too large, the audience may find it difficult to see where 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, passive classes etc. 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 we should utilize the feature of hierarchical state machines to avoid a simple state to become too complex to understand.

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 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 as much as we can, 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 [10]. 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. It would enforce better understandability of the inheritance and containment hierarchies of the corresponding model.

Rationale:

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

3.2 Antipatterns

3.2.1 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:

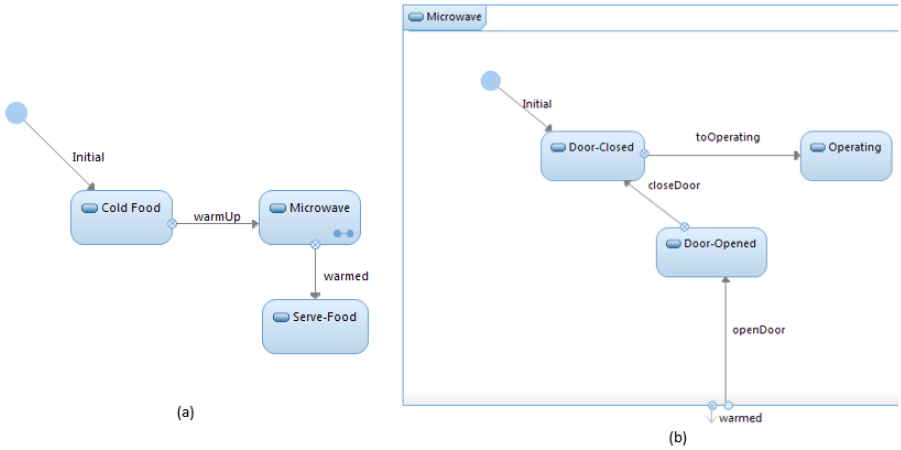

Figure 8: Misuse of Group Transition

Let us consider the scenario of a microwave oven introduced in Figure 8. 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

- Unnecessary use of group transition while a normal transition can better

serve the purpose.

Consequences

- Negatively affect 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:

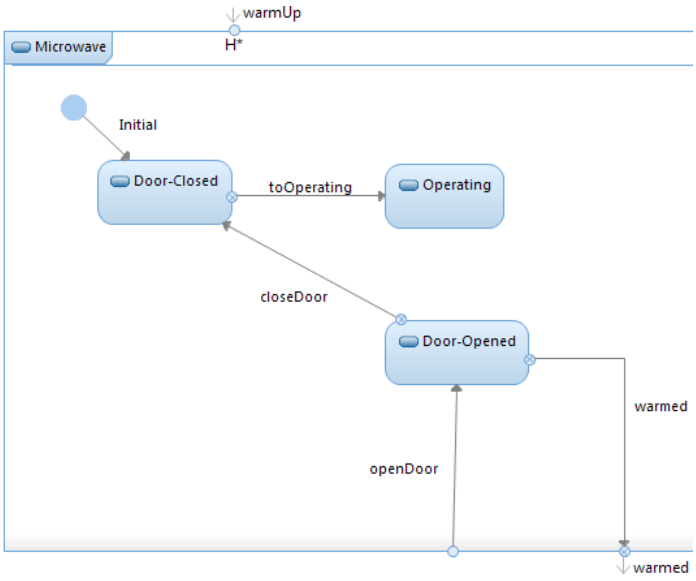

Figure 9: Refactor Solution for the Wrong Use of Group Transition

The *understandability* issue of the design solution in Figure 8 can be eliminated by replacing the group transition with a normal transition from the *Door-Opened* state. As we can see in Figure 9, 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.

3.2.2 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 10, a one shot timer can be used for implementing the functionality of a periodic timer as well.

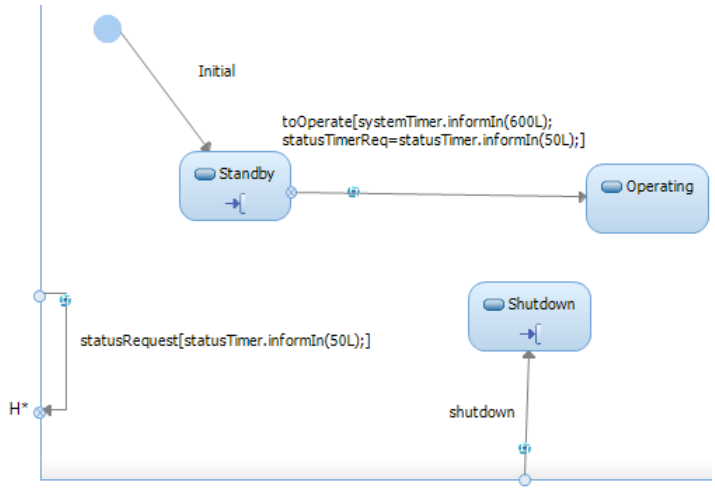

Figure 10: Use of a one-shot timer as a periodic timer

Figure 10 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 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 10. 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.

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

Figure 11: Comparison of One-shot and Periodic Timers: Timeout interval is 50 milliseconds

As we can see in Figure 11, 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 Figure 11, 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 12 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*.

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:

- Design of periodic timers using one-shot timers.

Consequences:

- Compromising the real-time aspects of the system behavior would result in a system with degraded quality.

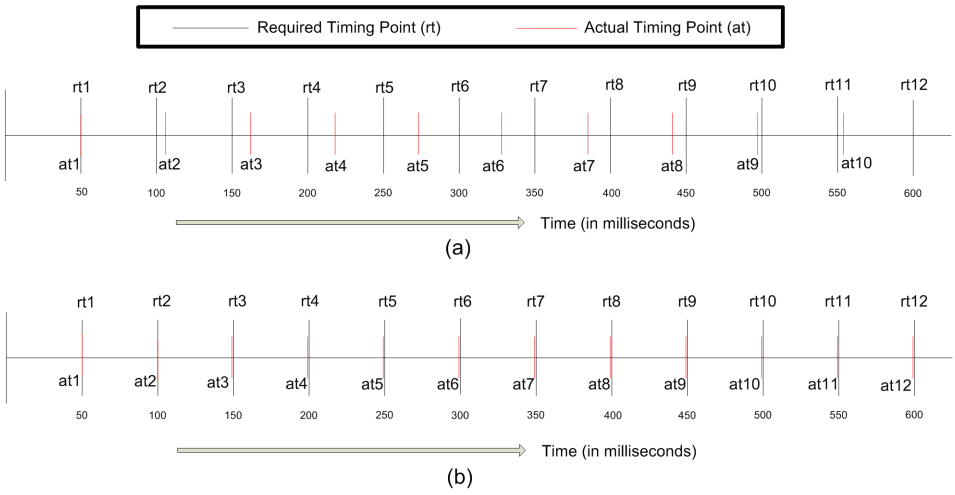

Figure 12: For a timeout interval of 50 milliseconds, the required and the actual time taken by (a) One-shot timer (b) Periodic timer

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 Figure 11 and 12).

3.2.3 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 for defining the associated interface and event pair that will cause the transition to be fired; action code for, e.g., performing operation calls, create and destroy other objects and send signals to other objects; and a guard condition which will be evaluated once the transition is triggered.

A guard condition is a Boolean expression optionally associated with a transition which decides if the transition would be taken or not once it gets triggered. If a guard is not specified, the default evaluation result would be *true* and the transition can be taken immediately after getting triggered. However, if a guard

condition is set, it must evaluate to *true* in order for the transition to be fired.

General Form: Let us consider the scenario of Figure 13 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*. Once the system receives user input for going to any particular operating mode while it is in the *Door-Closed* state, the transition associated with a valid guard condition will be executed for taking the system to the corresponding operational state. From the model visualization of this design choice, it is impossible to understand the execution behavior that would take the system from the *Door-Closed* state to the other states.

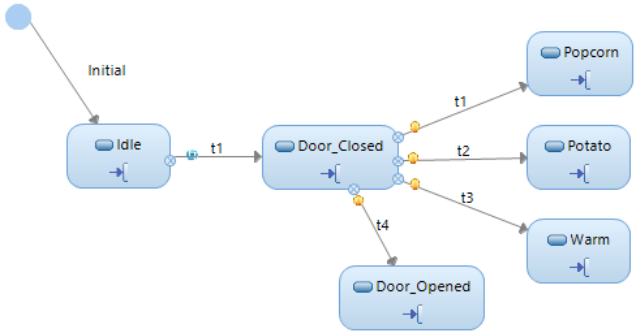

Figure 13: Misuse of Guard Conditions

Symptoms and Consequences:

- Use of guard conditions in multiple outgoing transitions from a state.
- Reduces the *understandability* of the corresponding design model.

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 14, due to the use of the choice-point *CP*, it is quite apparent that the execution behavior would depend on user-input while the microwave is in the *Door-Closed* state.

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 a choice to use a choice-point instead of guard conditions, we should use choice-point as it improves the *understandability* of the design model.

3.2.4 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

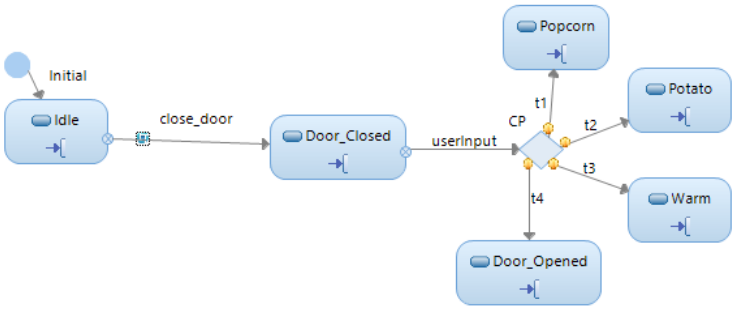

Figure 14: Replacement of Guards with a Choice-point

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 [11, 47, 4]. 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.

1. Hidden Super States

General Form: Let us consider the scenario of Figure 15. 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*.

In the design of Figure 15, 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.

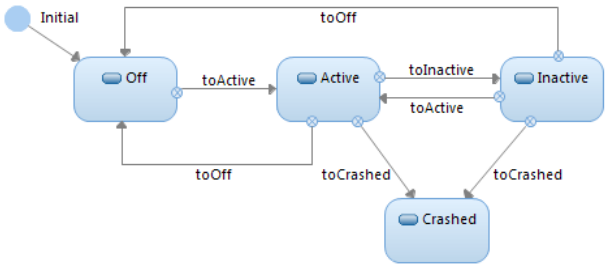

Figure 15: Wrong Use of State Abstraction

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:

- Presence of duplication 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.
- Little or no use of state nesting.
- 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.

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

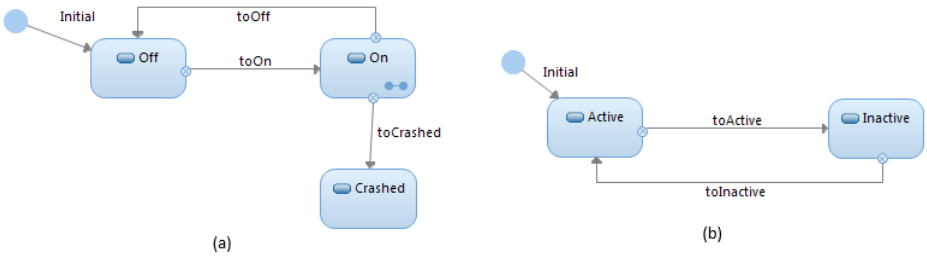

Figure 16: Proper Use of State Abstraction

16, 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.

2. Hidden Sub States

General Form: Let us consider the behavioral design choice of an adaptive cruise control software presented in Figure 17.

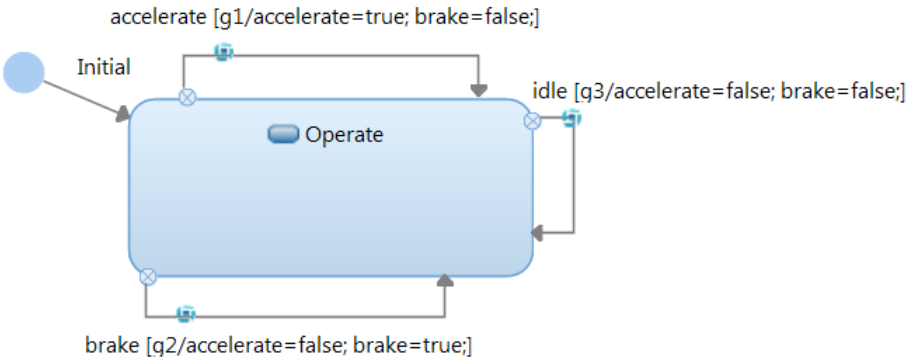

Figure 17: Existence of Hidden States in a Model

To be precise, 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 = (Speed $<$ 80 km and Distance $>$ 10 m)
Brake: g2 = (Speed $>$ 120 km or Distance $<$ 3 m)
Idle: g3 = $\neg$(g1 $\vee$ g2)

Now, as we can see in Figure 17, 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.

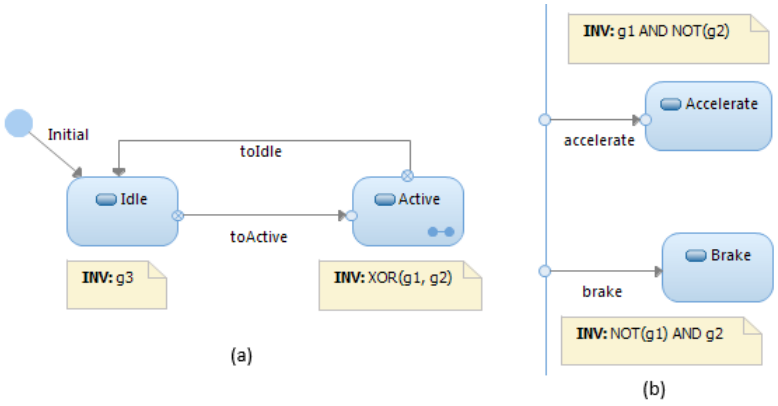

Figure 18: Refactoring Hidden States

Refactored Solution: The antipattern solution can be improved by making the hidden states visible. As we can see in Figure 18, 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.

3.2.5 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, capsule roles etc. 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 19. 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.

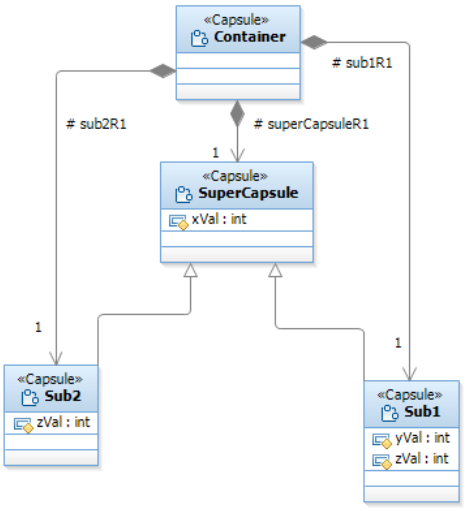

Figure 19: Inheritance in UML-RT

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.

Symptoms and Consequences

- Promotion and demotion of elements of the same type with the same name in different places of the same inheritance hierarchy.
- Affects 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.

3.2.6 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 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* [69]. 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 [16, 63]. Considering the lack of understandability in code where global variables are used, the authors in [85] 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 20 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:

- Existence of capsules unnecessarily defined with larger scopes.

Consequences

- Real time aspects would be affected as travelling through multiple ports would require extra time.
- Unnecessary use of system resources.

Known Exception: Although it is a 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

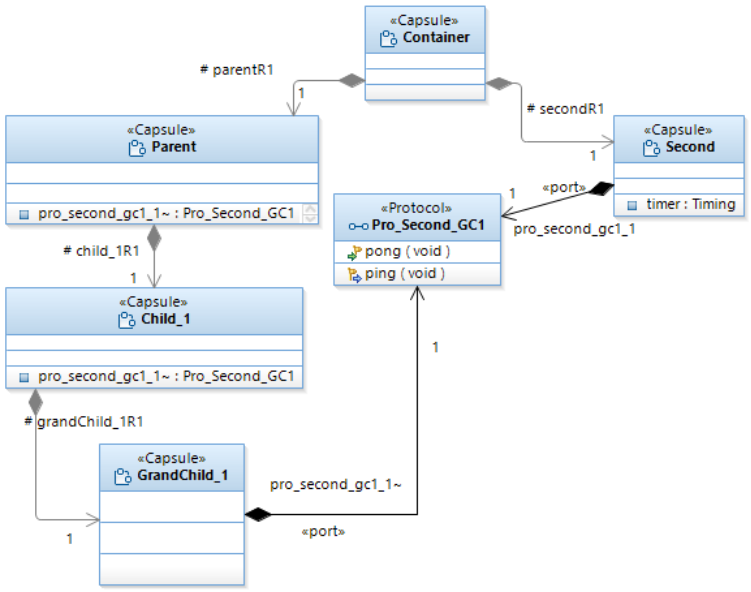

Figure 20: Global Capsule Scope in UML-RT

the global scope simplifies the code [16]. 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.

Refactor 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 21).

Consequences:

- Improves *understandability*.
- *Performance* would be improved.

3.3 Patterns

3.3.1 Status Monitoring Scenario

Intent: This pattern illustrates a general reusable solution for monitoring statuses of a set of systems.

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 *State-Monitor*.

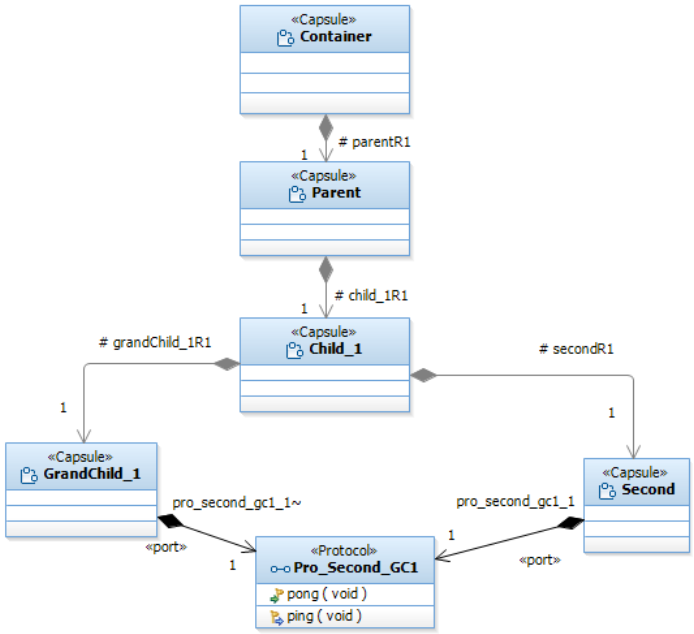

Figure 21: Refactored Capsule Scope

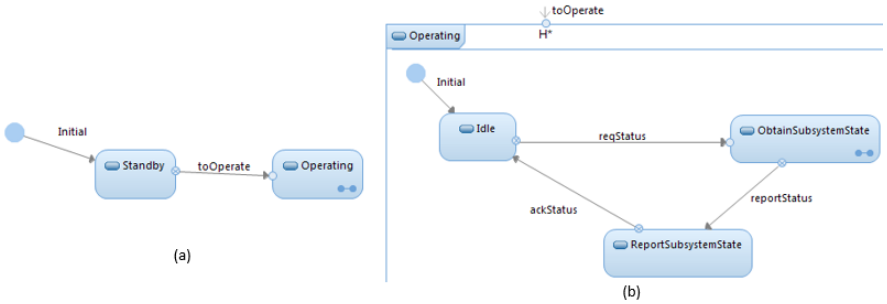

Figure 22: State Monitoring - Wrong approach

According to the requirement specification, the *Controller* capsule would request the *State-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*.

One possible realization of this capability is illustrated in Figure 22. As we can see in this figure, we have three states in the top level behavior of the *State-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 23).

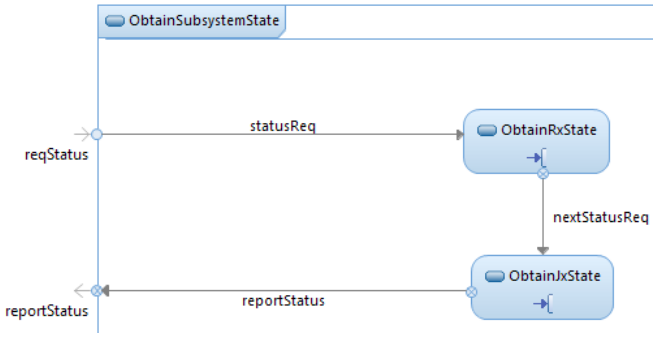

Figure 23: Obtaining Statuses from Receiver and Jammer

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 response from the *Receiver*, it sends out a status request to the *Jammer* and waits in the *ObtainJxState* for getting response from the *Jammer*. Finally, after retrieving status information from both of the subsystems, the system goes to the *ReportSubsystem* state for reporting this status information to the *Controller* and once it gets 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.

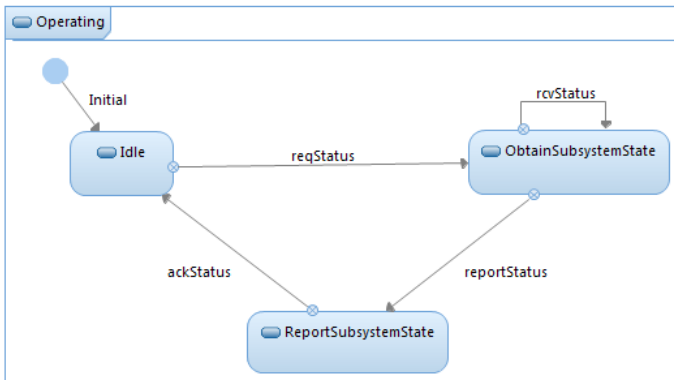

Figure 24: Pattern Solution for State Monitoring

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 24, focuses on eliminating this issue by using a self-transition.

Figure 24 illustrates the top 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.

3.3.2 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 25 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.

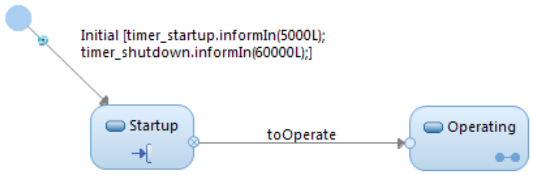

Figure 25: Wrong Placement of System Timer

In the design of Figure 25, 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:

- Instantiation of system timers that control the system runtime before the system actually starts operating.

Consequences:

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

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 26).

Consequences:

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

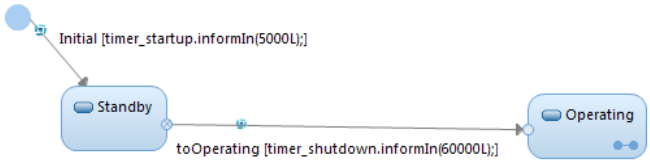

Figure 26: Proper Placement of System Timer

3.3.3 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 27 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.

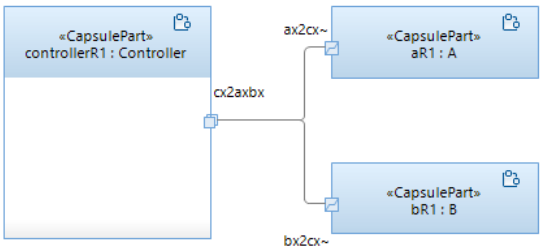

Figure 27: Multiple Roles in a Single Port

Solution: This problem can be resolved by decomposing and distributing the responsibilities among different ports (See Figure 28).

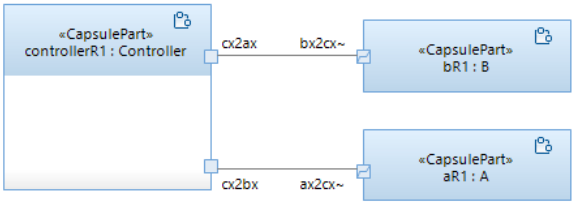

Figure 28: Distribution of Responsibilities among Different Ports

Consequences:

- Makes the model easier to *understand* and *maintain*.

- Allows the associated protocols to be *cohesive* and *thin*.
- Tradeoff: The *Controller* capsule would have to listen to multiple ports now.

3.4 Clones in UML-RT Models

This section will discuss several examples of duplication smell in UML-RT and refactor suggestions for eliminating the redundancies. First, we will discuss duplications in UML-RT behavioral diagrams. This will be followed by the discussion of duplications in UML-RT class diagrams. The examples shown here in this section are inspired by the anonymous student model repository we analyzed.

1. Duplication in Behavioral Diagram: Example 1

Background: Let us consider the design of an *Electronic Warfare System* which has three capsules: *Controller*, *Receiver* and *Jammer*. According to the design requirement, the *Controller* capsule should perform a *Built-In Test (BIT)* on the two other systems on a regular interval.

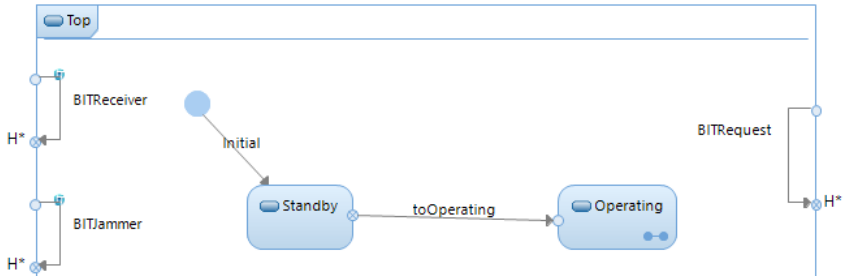

Figure 29: Duplication in Self-transitions

General Form: An example of the *Controller* capsule behavior is illustrated in Figure 29. The *Controller* capsule sends out the *BITRequest* signals to the *Receiver* and the *Jammer* capsules using a self-transition. Once it receives responses from these capsules, the *BITReceiver* and the *BITJammer* self-transitions will be triggered respectively. Action code associated with the *BITReceiver* and the *BITJammer* transitions assigns the status information to a local variable. Both transitions are structurally identical.

Consequences:

- Duplicated modeling elements can lead to increased maintenance efforts.

Refactor Solution: The unnecessary duplication in this example can be eliminated in two ways. We can either replace the two self-transitions, *BITReceiver* and *BITJammer* with a single self-transition which will have both of the triggers on its interface. Alternatively, we can create a function that will contain

the code common to both of the transitions and the function can be called in the action code of both *BITReceiver* and *BITJammer* transitions.

1. Duplication in Behavioral Diagram: Example 2

Background: This example also involves the behavioral diagram of a *Controller* capsule in the design of an *Electronic Warfare System*. Upon getting instruction from the *Controller*, the *Receiver* capsule would scan the environment for emitters and if it detects an emitter, it would send a message to the *Controller* capsule. The *Controller* would then evaluate the emitter and if it considers the emitter to be a threat, it would assign a *Jammer* to jam the emitter. Before the *Jammer* starts jamming, the *Controller* would have to make sure that the *Receiver* is in *not-scanning* mode to prevent damage. In a jamming period, the *Jammer* can switch between *jamming* and *look-through* mode so that the *Receiver* can start scanning for new emitters while the *Jammer* is in *look-through* mode.

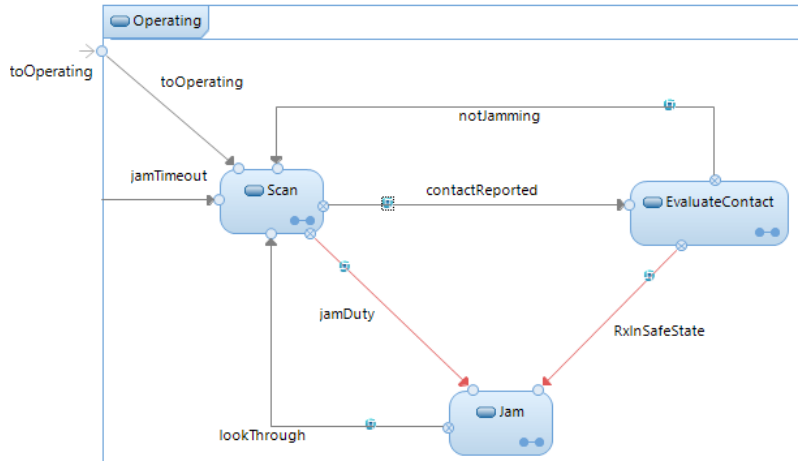

Figure 30: Duplication in State Diagram

General Form: In Figure 30, the *Controller* sends out a scan request signal to the *Receiver* from the *Scan* state. When the *Receiver* reports an emitter, the execution follows the *contactReported* transition for going to the *EvaluateContact* state. After evaluating the emitter if the *Controller* finds a threat, in the action code of *RxInSafeState* it makes sure that the *Receiver* goes to the *not-scanning* mode and then travels to the *Jam* state. In a jam cycle, the execution can switch between the *Jam* and the *Scan* state and before travelling from *Scan* to *Jam*, it makes sure again that the *Receiver* transitions to the *not-scanning* mode in the action code of *jamDuty* transition.

Now, duplication exists in this design model as the two incoming transitions in the *Jam* state are triggered by the same event and they have identical action code as well.

Refactor Solution: One way to eliminate the duplication present in Figure

30 is to replace the incoming transitions to the *Jam* state having identical event and action code with a single transition from the border of the enclosing state. But, doing so would change the model behavior as the refactored model would be able to handle the *goToJam* event while it is already in the *Jam* state. So, the best way to remove duplication in this model is to create a function using the identical action code from both of the incoming transitions to the *Jam* state and make the function call whenever necessary.

2. Duplication in Class Diagrams

Duplicated artifacts can be found in class diagrams if the feature of inheritance is not utilized fully while designing models in UML-RT. An example of this scenario is illustrated in Figure 31.

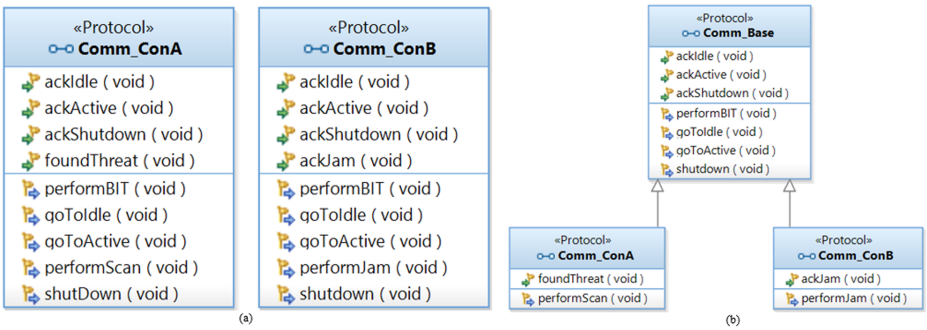

Figure 31: Duplication in UML-RT Protocols

Figure 31(a) illustrates two protocols which have some common in- and out-signals. This duplication can be eliminated by putting the common signals in another protocol and letting it be inherited by the protocols which had duplications among them (See Figure 31(b)).

4 Related Work

4.1 Coding and Design Conventions

The practice of following coding conventions can improve the maintainability of a software system significantly. In [52], 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 [81], comment conventions [79], indent style conventions [80], etc.

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 Ericsson, we have observed that a set of design conventions is followed strictly by the developers of Ericsson models. This set mainly includes naming conventions

for different modeling constructs such as capsules, classes, states, transitions, protocols etc.

4.2 Patterns in Models

In a fashion similar to object oriented (OO) patterns, the concept of state patterns began to appear in the literature in [21]. On the contrary to the 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 primitive statechart constructs. The patterns can then lead to efficient and effective solutions to different scenarios by being instantiated and specialized for specific problems. In [21], the author presents a set of twelve behavioral patterns which include the collaboration of states that 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 [64]. Each of these patterns is discussed with five components: the pattern name, problem definition, proven solution, sample code and the consequences. In contrast to the state patterns discussed on [21], which revolve primarily around orthogonal regions, the state patterns defined in [64] 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 it is suggested in this paper, to be genuinely useful, a pattern must be accompanied by a specific working example that will help the developers to truly comprehend 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 [67]. 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 from the highest architectural level to the lowest-level individual components.

Stateflow is an environment for modeling and simulating combinatorial and sequential decision logic based on state machines and flowcharts. For achieving understandable presentation, and reusable and readable models, a set of guidelines for developing models using Stateflow is presented in [48]. This set includes, but is not limited to, guidelines for creating states and transitions in state machines, use of patterns for flowcharts, transitions in flowcharts, placement of default transitions, state machine patterns for conditions and transition actions etc. Each of these guidelines is presented with the impact it has on different properties of the final model such as readability, verification, validation, workflow, code generation and simulation.

4.3 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 [68] specify a set of 14 performance antipatterns. In subsequent work, they propose several techniques for the detection of performance antipatterns in software architectural models [15, 74]. The authors in [15] 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 [74]. This approach is based on the formal definition of some performance antipatterns that had not been formalized before. In [6], 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.

For exploring the educational role of anti-patterns in improving modeling skills in UML class diagrams, a catalog of 43 correctness and quality anti-patterns for class diagrams is introduced in [7]. This catalog analyzes the causes of correctness and quality problems and provides suggestions for rectifying these problems. On the basis of conducting two experiments the authors conclude that learning the anti-patterns helps students improve their awareness of modeling problems in class diagrams.

With the intention of exploring bad design choices in the behavioral design of UML-RT, we present a set of seven state machine anti-patterns in [17]. In addition to discussing the associated problems, each of these anti-patterns is introduced with a refactor solution for improving the problematic solution.

4.4 Clone Detection: the Current State-of-Art

4.4.1 Clone Detection Research in Code Driven Development

The process of updating a software system for improving the internal structure without altering its external behavior is called *software refactoring*. With the intention of demonstrating refactoring in a controlled and efficient manner, *Martin Fowler* introduces *22* code smells in [26]. The number one smell he has in this list is *Duplicated Code*. If the same code structure exists in two or more places, we can improve the code structure by finding a way of unifying them. The main problem associated with duplication appears during software maintenance: if a change needs to be made in one place, most probably the change needs to be made in all the other duplicated places and it is very easy to forget to make changes in one of those places, thus possibly introducing errors.

Reasons of Clone Detection: Reuse of code, logic, design or an entire system is the primary reason for code duplication in [61]. The forms of reuse include utilizing existing implementation by just copying and pasting with minor or without any modification [40], copying existing implementation considering that it would be changed significantly in future (Forking) [39]. In addition, clones can be introduced in a system while merging two similar systems into a single one. Clones can also be incorporated in a system due to the unwillingness of a development community to write new code as the existing code is already well tested [13], language limitations [8, 57], time constrained software development etc. Sometimes clones can be created accidentally as well if multiple developers are involved in designing similar functionalities or due to the cognitive limitations of developers [61].

Consequences of Code Duplication: Negative impact of duplication includes the propagation of bugs by cloning an error-prone code segment [37, 46], the introduction of errors in the newer system by adapting existing implementation wrongly [9, 38], the incorporation of inappropriate inheritance structure or abstraction [53], an increased maintenance effort and risk of error-prone design [53, 51] and the unnecessary use of system resources [61].

Code Clone Classification: According to the current state of the art [61, 43, 73], code clones can be categorized into four types:

- **Type I (Exact Clone):** Identical code blocks ignoring any variations in comments and white-spaces.
- **Type II (Renamed Clone):** Code duplication with consistent changes to identifiers, literals, types, layouts and comments.
- **Type III (Parameterized Clone):** Duplication with the allowance of more changes (i.e., addition or removal of code fragments). These kinds of clones are also called near-miss code clones.
- **Type IV (Semantic Clone):** Multiple code fragments with significant syntactic variations having similar pre- and post-conditions.

As we can see, this categorization is made with the increasing level of variations between compared code blocks. The effort required to detect these clones also follows this order.

Importance of Scope in Clone Definition: Redundancy is more like coupled text, changing one copy would suggest us to make the same changes in other copies as well. However, if the duplicated code segments do not have the same scope, it could be possible that they are not coupled and changing code in one section would not suggest making the changes in the other segment as well [43].

4.4.2 Clone Detection Research in Model Driven Development

At present, the importance of model driven development in many industrial development activities of many organizations is significant, and similar to the classical code-oriented development, the size of the models can become quite large [71, 72]. Based on the experiences of dealing with large scale domain models, it is concluded that the concept of clones in model driven development appears in almost the same way it appears in source code [19] and therefore, the importance of detecting clones in MDD is similarly valuable [58].

Approaches to model clone detection to date mainly utilized graph-based techniques [20, 59, 2]. Models are represented in these techniques as nodes and edges and variations of subgraph matching techniques are used for finding clones in the model. Although these techniques are natural and efficient for exact matching in visual models, their success in near-miss clone detection is not remarkable [59]. In addition, these techniques suffer from decreased performance when applied to graphs with many cycles, which can appear in data-flow and behavioral models. For avoiding these consequences of using graph-based techniques, recent research has used text-based code clone detection techniques to identify model clones.

In [62], a parser-based language-sensitive code clone detection tool NICAD is introduced on the basis of the TXL parser [14] and text-comparing syntactic fragments. This tool allows for unexpected differences in near-miss clones up to a given difference threshold. The NICAD tool is adapted for introducing SIMONE (SIMulink clONE detector) [3] to identify structurally meaningful near-miss subsystem clones in Simulink [49] models. The authors in [3] use the textual representation of Simulink models as the text input to SIMONE. Thereafter, they utilize grammar inference techniques for deriving a formal TXL grammar from a large set of example Simulink models in the public domain. The authors in [18] extended the approach used in SIMONE and applied clone detection techniques to the textual representation of Stateflow [50] models.

In [70], the authors discuss the possibility of detecting pattern and antipattern instances in models using model clone detection. One of their immediate plans is to realize Simulink antipattern detection by constructing Simulink antipattern representations. Thereafter, they would like to configure and execute the Simulink MCD tool [3, 18] for evaluating both their proposed process and the results. In [5], with the goal of identifying patterns of interactions in the run-time behavior of web applications and other interactive systems, the authors present an approach for detecting near-miss interaction clones in reverse-engineered UML sequence diagrams. They work on the level of XMI [56], the standard interchange serialization for UML, for using the near-miss code clone detector NICAD.

Based on analysis of practical scenarios, a formal definition of model clones along with the specification of a clone detection algorithm for UML domain models is given in [72]. Following the code-clone categorization [61, 43, 73], the authors in [72] also propose a classification for model clones. The detection of clones in UML models can become a difficult task due to the inconsistency that

can be created between different views of the same model. For example, most of the popular UML development tools support a containment tree view of the model elements along with other model diagram views. Removal of a model element from a diagram view usually does not remove it from the actual model, and thus inconsistencies can occur in a UML model [72].

Importance of Tool-Specific Analysis: In contrast to programs in traditional programming languages, which can be easily transferred from one tool to another in the form of text files, using model data from one tool in another can be a difficult task due to the tight coupling of models with the development tool [72]. Therefore, the tool specific representation of the model should also be taken into consideration [72].

5 Conclusion

According to some statistics, nearly one-third of all software projects are cancelled, two-thirds of software projects suffer from cost overruns in excess of 200% and more than 80% of all software projects are considered failures [12]. One of the prominent reasons for the massive software failure rate is poor software design quality [54]. 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 is extremely high. Although a good number of work has been done on investigating the quality of different kinds of models including the 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 standards [65, 29] and these differences make most of the proposed guidelines in the current state of art regarding the UML standards 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, the most of these guidelines are described with examples.

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 [30, 75, 36] and would allow developers to detect the presence of bad design choices automatically and refactor them accordingly.

Acknowledgment

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 valueable feedback on our work. We would also like to thank our industrial research

partner Ericsson Inc. for giving us access to some of their models. 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] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Bakr Al-Batran, Bernhard Schätz, and Benjamin Hummel. Semantic Clone Detection for Model-based Development of Embedded Systems. In *MODELS '11*, pages 258–272, 2011.

[3] M.H. Alalfi, J.R. Cordy, T.R. Dean, M. Stephan, and A. Stevenson. Models are code too: Near-miss clone detection for Simulink models. In *ICSM '12*, pages 295–304, Sept 2012.

[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] Elizabeth P. Antony, Manar H. Alalfi, and James R. Cordy. An approach to clone detection in behavioural models. In *WCRE '13*, pages 472–476. IEEE Computer Society, 2013.

[6] D. Arcelli, V. Cortellessa, and C. Trubiani. Antipattern-based model refactoring for software performance improvement. In *(QoSA '12)*, pages 33–42. ACM, 2012.

[7] 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.

[8] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: A study of clones in the STL and some general implications. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 451–459, New York, NY, USA, 2005. ACM.

[9] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, Washington, DC, USA, 1998. IEEE Computer Society.

[10] 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].

[11] R. B'far. *Mobile Computing Principles: Designing and Developing Mobile Applications with UML and XML.* Cambridge University Press, New York, NY, USA, 2004.

[12] 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.

[13] J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 196–205, Washington, DC, USA, 2003. IEEE Computer Society.

[14] James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, August 2006.

[15] 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.

[16] Cunningham and Cunningham Inc. Global Variables Are Bad. `http://c2.com/cgi/wiki?GlobalVariablesAreBad`. [Online; accessed 20-January-2016].

[17] 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.

[18] T. R. Dean, J. Chen, and M. H. Alalfi. Clone detection in Matlab Stateflow models. *ECEASST*, 63, 2014.

[19] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 57–64, New York, NY, USA, 2010. ACM.

[20] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *ICSE '08*, pages 603–612, 2008.

[21] 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.

[22] Eclipse. Papyrus for Real Time (Papyrus-RT). `https://projects.eclipse.org/proposals/papyrus-real-time-papyrus-rt`. [Online; accessed 20-January-2016].

[23] M. D. P. Emilio. *Embedded Systems Design for High-Speed Data Acquisition and Control*. Springer Publishing Company, Incorporated, 2014.

[24] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello. An experience report on using code smells detection tools. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 450–457, Washington, DC, USA, 2011. IEEE Computer Society.

[25] 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].

[26] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[27] 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.

[28] A. Gherbi and F. Khendek. UML profiles for real-time systems and their applications. *JOURNAL OF OBJECT TECHNOLOGY*, 5:149–169, 2006.

[29] S. Gopinath. Real-Time UML to XMI Conversion. Master's thesis, KTH Computer Science and Communication, Stockholm, Sweden, 2006.

[30] 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].

[31] Z. Gu and K. G. Shin. Synthesis of real-time implementation from UML-RT models. In *2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES 04)*, 2004.

[32] 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.

[33] 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].

[34] 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].

[35] IBM. Rational Rose RealTime. `ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/rosert.html`. [Online; accessed 20-January-2016].

[36] 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.

[37] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993.

[38] J. H. Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '96, pages 16–. IBM Press, 1996.

[39] C. J. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, December 2008.

[40] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE '04, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[41] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. *SIGPLAN Not.*, 29(6):147–158, June 1994.

[42] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[43] R. Koschke. Survey of research on software clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[44] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2005.

[45] Nancy Leveson. Medical Devices: The Therac-25. `http://sunnyday.mit.edu/papers/therac.pdf`. [Online; accessed 20-January-2016].

[46] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, March 2006.

[47] Sparx Systems Pty Ltd. UML 2 State Machine Diagram. http://www.sparxsystems.com/resources/uml2_tutorial/uml2_statediagram.html/. [Online; accessed 20-January-2016].

[48] 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].

[49] MathWorks. SIMULINK: Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/. [Online; accessed 20-January-2016].

[50] MathWorks. Stateflow: Model and simulate decision logic using state machines and flow charts. http://www.mathworks.com/help/stateflow/. [Online; accessed 20-January-2016].

[51] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 244–, Washington, DC, USA, 1996. IEEE Computer Society.

[52] Sun Microsystems. Code Conventions for the Java Programming Language. http://www.oracle.com/technetwork/java/codeconvtoc-136057.html, 1999. [Online; accessed 20-January-2016].

[53] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 87–94, Washington, DC, USA, 2002. IEEE Computer Society.

[54] E. E. Ogheneovo. Software dysfunction: Why do software fail? *Journal of Computer and Communications*, 2:25–35, 2014.

[55] 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].

[56] Object Management Group (OMG). XML Metadata Interchange (XMI). http://www.omg.org/spec/XMI/. [Online; accessed 20-January-2016].

[57] J.F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th International Workshop on Program Comprehension*, pages 49–56, 1999.

[58] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering*,

ICSE '09, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[59] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE '09*, pages 276–286, 2009.

[60] Kepler Project. Software Development Guidelines. `https://kepler-project.org/developers/reference/software-development-guidelines`. [Online; accessed 20-January-2016].

[61] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queen's University, 2007.

[62] C.K. Roy and J.R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC '08*, pages 172–181, 2008.

[63] 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].

[64] M. Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes Newton, MA, USA, 2008.

[65] 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.

[66] 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.

[67] Bran Selic. An architectural pattern for real-time control software. In *Pattern Languages of Program Design 2*, pages 4–6. Addison-Wesley, 1996.

[68] 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.

[69] Thunderstone Software. Variable Scope: Global vs. Local. `https://www.thunderstone.com/site/vortexman/variable_scope_global_vs_local.html`. [Online; accessed 20-January-2016].

[70] Matthew Stephan and James R Cordy. Identifying instances of model design patterns and antipatterns using model clone detection. In *MISE '15*, pages 48–53. IEEE, 2015.

[71] H. Störrle. Large scale modeling efforts: A survey on challenges and best practices. In *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*, SE'07, pages 382–389, Anaheim, CA, USA, 2007. ACTA Press.

[72] H. Störrle. Structuring very large domain models: Experiences from industrial mdsd projects. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 49–54, New York, NY, USA, 2010. ACM.

[73] R. Tiarks, R. Koschke, and R. Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 67–76, Sept 2009.

[74] C. Trubiani and A. Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. In *(ICPE '11)*, pages 19–30. ACM, 2011.

[75] 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.

[76] WEBster. Software Development Guidelines. `http://www.literateprogramming.com/sdg.pdf`. [Online; accessed 20-January-2016].

[77] Wikipedia. Anti-pattern. `http://en.wikipedia.org/wiki/Anti-pattern`. [Online; accessed 20-January-2016].

[78] Wikipedia. Code smell. `http://en.wikipedia.org/wiki/Code_smell`. [Online; accessed 20-January-2016].

[79] Wikipedia. Comment (computer programming). `http://en.wikipedia.org/wiki/Comment_(computer_programming)`. [Online; accessed 20-January-2016].

[80] Wikipedia. Indent style. `http://en.wikipedia.org/wiki/Indent_style`. [Online; accessed 20-January-2016].

[81] Wikipedia. Naming convention (programming). `http://en.wikipedia.org/wiki/Naming_convention_(programming)`. [Online; accessed 20-January-2016].

[82] Wikipedia. Software design pattern. `http://en.wikipedia.org/wiki/Software_design_pattern`. [Online; accessed 20-January-2016].

[83] Wikipedia. Software quality. `http://en.wikipedia.org/wiki/Software_quality`. [Online; accessed 20-January-2016].

[84] S. Wong, S. Vassiliadis, S. Vassiliadis, and S. Cotofana. Embedded processors: Characteristics and trends. Technical report, in Proceedings of the 2001 ASCI Conference, 2004.

[85] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, February 1973.

[86] 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.