From Java to Real-Time Java: A Model-Driven Methodology with Automated Toolchain (Invited Paper)

Wanli Chang
University of York
The United Kingdom
wanli.chang@york.ac.uk

Shuai Zhao
University of York
The United Kingdom
shuai.zhao@york.ac.uk

Ran Wei
University of York
The United Kingdom
ran.wei@york.ac.uk

Andy Wellings
University of York
The United Kingdom
andy.wellings@york.ac.uk

Alan Burns
University of York
The United Kingdom
alan.burns@york.ac.uk

Abstract

Real-time systems are receiving increasing attention with the emerging application scenarios that are safety-critical, complex in functionality, high on timing-related performance requirements, and cost-sensitive, such as autonomous vehicles. Development of real-time systems is error-prone and highly dependent on the sophisticated domain expertise, making it a costly process. There is a trend of the existing software without the real-time notion being re-developed to realise real-time features, e.g., in the big data technology. This paper utilises the principles of model-driven engineering (MDE) and proposes the first methodology that automatically converts standard time-sharing Java applications to real-time Java applications. It opens up a new research direction on development automation of real-time programming languages and inspires many research questions that can be jointly investigated by the embedded systems, programming languages as well as MDE communities.

CCS Concepts • **Computer systems organization** → **Real-time systems**; • **Software and its engineering** → **Software notations and tools**.

Keywords real-time programming languages, real-time specification for Java, model-driven engineering

LCTES '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6724-0/19/06…$15.00
https://doi.org/10.1145/3316482.3326360

ACM Reference Format:
Wanli Chang, Shuai Zhao, Ran Wei, Andy Wellings, and Alan Burns. 2019. From Java to Real-Time Java: A Model-Driven Methodology with Automated Toolchain (Invited Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3316482.3326360

1 Introduction

Real-time systems often enclose stringent temporal requirements, where a real-time application must react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment [10]. Such systems have been well practised in many fields, and their application domains keep growing with emerging scenarios [15].

Although timing requirements are categorised as non-functional requirements, they are essential to safety-related systems. In [26], the author classifies system failure modes into *random failures* and *systematic failures*, where systematic failures contribute to system hazards which could lead to incidents with catastrophic consequences. *Systematic failures* can be further classified into *functional failures* and *timing failures*. It is imperative to ensure that a safety-related system possesses correct timing requirements and at the same time, that its timing behaviour satisfies these timing requirements. Therefore, demonstrating real-time properties forms key evidence in certifying the safety of a safety-related system.

Due to the high productivity, portability and relatively low maintenance cost, the *Java* programming language has received extensive attention in the real-time and safety-critical domains [21, 45]. For instance, Java was adopted in [31] and [30] to reduce distributed computing latency in an unified could-based platform for autonomous vehicles. However, these works have been developed focusing on functionality with limited consideration of timing and safety guarantee,

especially when the complex perception functions are involved. As mandated by safety regulations, such as the ISO 26262 for automotive systems and IEC 61508 for functional safety, hard real-time constraints are essential to guarantee safety of the system (e.g., the vehicle) and its surrounding environment. Thus, there is a need to push these existing works towards the real-time regime.

There is a trend that matured Java techniques (which were developed without the notion of real-time) are re-developed to possess real-time guarantees (e.g., real-time big data systems [18] and real-time stream processing techniques [32]). The major reason is that, those simple and conservative methods (like leaving large safety margins) that were deployed in practice are losing ground, with the ever more complicated functionality, higher timing-related performance requirements and limited resources on the emerging real-time applications [3, 12–14].

Despite its popularity, standard Java cannot be directly applied to produce real-time software due to the lack of facilities such as thread scheduling, resource sharing control, memory management, etc., which are essential to achieve predictability [11] in terms of temporal behaviour. This has motivated the development of the *Real-Time Specification for Java* (RTSJ) [8]. RTSJ reserves the intrinsic advantages of Java, and provides plenty of real-time facilities to guarantee the system temporal behaviour, but at the same time is harder to be used by software engineers.

Compared to the generic time-sharing applications in Java, developing real-time applications using RTSJ depends highly on the expertise in the real-time systems design and requires thorough understanding of the specification. It is also error-prone due to the complexity. All of these above make development of real-time applications a costly process. Although there have been system analysis and verification techniques [35] to ensure correctness in the design phase, in terms of both logical and temporal behaviour, it remains an open and challenging problem how to eliminate human-related erroneous factors (e.g., caused by limited understanding of the real-time concepts and insufficient experience with RTSJ facilities). The safety-critical nature in many real-time systems domains amplifies the impact of such concerns.

Model-driven engineering (MDE) is a contemporary software development paradigm, which promotes *model*s as first-class artefacts. Based on models, developers are able to perform a series of model management operations in an automated manner, and eventually produce software artefacts, such as documentation and working code. This reduces the amount of time required to develop a system and thus improves the productivity of software engineers, by at least a factor of 10 in many cases [23, 25]. Adopting MDE also reduces the number of errors throughout the development process and improves consistency [51]. In addition, MDE can be applied to any domain to achieve automation, due to the concept of domain-specific modelling and the interoperability provided by model management operations, which can be executed in an automated manner.

In this paper, we apply the principles of MDE in the domain of real-time programming with Java. We propose the first methodology that is able to automatically convert existing time-sharing Java applications to real-time applications in RTSJ, through a series of model management operations. The output software is in full compliance to the RTSJ specification, with dependencies to the RTSJ runtime environment supporting scheduling, memory management, resource sharing, asynchrony, etc. This enables the developers with limited real-time background to perform temporal analysis on their non-real-time base code and convert it to source code written in RTSJ. Due to the application of MDE techniques, the productivity and consistency throughout the development. Human errors are eliminated in the automation. We describe an automated toolchain associated with the proposed methodology. All the functional blocks in the toolchain and the involved technical approaches are explained. The scientific challenges addressed and hidden issues discovered towards automatic generation of real-time applications with MDE techniques are discussed. We also point out future research directions beyond this paper.

The rest of the paper is organised as follows. Section 2 provides a review of the MDE technology and its application in the real-time systems development. Section 3 describes the real-time Java, The proposed methodology and toolchain are reported in Section 4 with detailed transformation approaches. Section 5 outlines open questions and possible research directions that are introduced by the proposed methodology. Finally, Section 6 gives the conclusion.

2 Model-Driven Engineering

Modelling is an essential part of any system engineering process. Engineers of all disciplines construct models of the systems they intend to build to capture, test and validate their system design ideas with other stakeholders before committing to a long and costly production process.

MDE is a software engineering methodology that aims to reduce the accidental complexity of software systems by promoting *model*s that focus on the essential complexity of systems, as the first-class artefacts of the software development process. In contrast to those traditional software development methodologies, where models are mainly used for communication and post-mortem documentation process, in MDE models are the main living and evolving artefacts from which concrete software development artefacts can be produced in an analysable and automated fashion.

MDE was proposed at the time when object-oriented techniques reached a point of exhaustion [7, 37]. MDE constitutes the latest paradigm shift in software engineering as it raises

the level of abstraction beyond that provided by 3rd generation programming languages. In recent studies, MDE has been shown to increase productivity by as much as a factor of 10 [23, 25], and significantly enhance important aspects of the software development process such as maintainability, consistency and traceability [33].

There are two important aspects of MDE — (i) *domain-specific modelling*, where domain experts create their own domain-specific modelling languages (DSMLs) to capture the concepts in their domain (and create instances of their DSMLs to model their systems); (ii) *model management operations*, which are programs performed on models in an automated manner to generate software engineering artefacts. Model management operations typically include, but are not limited to:

- Text-to-Model Transformation (T2M): to convert text (such as source code) into models based on parsing rules defined in the transformation;
- Model Validation: to check the well-formedness of models, as well as custom constraints against the elements in models;
- Model-to-Model Transformation (M2M): to interoperate between different modelling technologies, where one type of model is transformed into another type;
- Model-to-Text Transformation (M2T): to generate text based on the contents of the model (e.g., documentation generation and source code generation);
- Model Comparison: to compare different versions of a model to find out what is changed;
- Model Merging: to integrate models defined by different parties but share model elements.

MDE has been applied to a variety of domains, with proven benefits. In [28] MDE is applied to transform model query languages to MySQL queries to reduce the effort and error rates in manually creating MySQL queries. In [51], MDE is applied to automatically generate fully functional graphical editors for UML profiles. In [5], MDE is applied to transform natural languages to database query languages to form complex query using simple natural language grammars.

Developing real-time systems via a model-based approach is not novel in the community [24, 46]. The idea proposed in this paper is partially inspired by them. None of these works study the migration from standard Java to real-time Java. In addition, many of the past efforts rely on the notion of model-driven architecture, which is an outdated MDE practice and has a lack of tool support. By applying MDE techniques, as previously described, Real-Time system developers can benefit from the productivity gain from MDE, as well as the consistency and maintainability through automation provided by MDE.

3 Real-Time Specification for Java

RTSJ, originally developed as Java Special Request 1 under the Java Community Process in 2001 [1], has been well-practised in a wide range of application domains, including automotive, manufacturing control, avionics and information systems [22, 43, 46, 47]. For instance, RTSJ has been applied to the auto-pilot system of an unmanned aerial vehicle, which is the first Java-based system that satisfies all Boeing's operational requirements and flew in tests [1]. *Jcoap*, realised by RTSJ, provides real-time communications for IoT systems [29]. In [17], RTSJ has been applied in a real-time big data processing systems with FPGA-based hardware acceleration. In industry, *JamaicaCAR* developed by both Acis and Perrone Robotics[2] provides a lightweight application framework for car headunits and in-vehicle information systems. In addition, Acis and CLAAS[3] present solutions (namely *Jamaica-IoT*) for digital factory and manufacturing, which enables deployment and operation of data analytics and control logic at the network's edge.

The RTSJ is designed to support both hard and soft real-time applications. This specification consists of two major components — (i) extensions from the Java programming language; and (ii) modifications on the semantics of the standard Java Virtual Machines (JVM) [8]. This section briefly reviews the programming specification of RTSJ, together with its reference implementations as well as the supporting Virtual Machines (VM). Detailed descriptions of each RTSJ facility and the application examples can be found in [10] and [48].

3.1 Programming Specification

In total, there are seven extensions from the standard Java language that are provided in the package `javax.realtime`, including task scheduling and dispatching, memory management, shared resource control, asynchronous event handling, etc.

One major facility provided in RTSJ is `javax.RealtimeThread`, which takes a set of scheduling-related parameters (e.g., priority, period and deadline) specifying a real-time thread's release, execution and timing properties. Three types of threads are derived from this entity: periodic, sporadic and aperiodic, depending on the input release parameter. In addition, a set of asynchronous event handlers are provided to allow user-defined actions in the cases of deadline miss or budge overrun. By default, the real-time threads are scheduled by a preemptive fixed-priority scheduler, but user-defined scheduling and dispatching policies are also possible.

Another important extension the real-time memory management model. In RTSJ, a set of memory management

[1] https://jcp.org/en/jsr/detail?id=1
[2] https://www.perronerobotics.com
[3] https://www.claas.ca

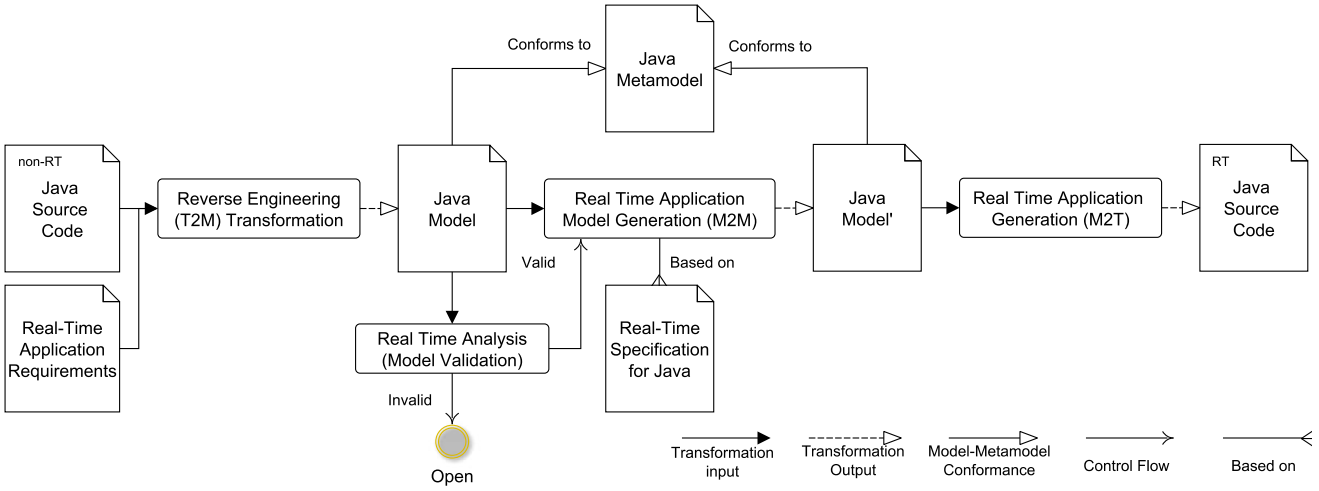

Figure 1. Time-sharing applications to real-time applications migration

facilities are provided in RTSJ (e.g.,`ImmortalMemory` and `ScopedMemory`) to allow the construction of self-defined memory models. However, RTSJ imposes a set of memory accessing rules that restrict memory-accessing behaviours to prevent *dangling reference* (i.e., references that point to objects in reclaimed memory blocks). With memory management model defined, the standard Java garbage collector is no longer required so that its unpredictable interference is avoided during run-time. Later, a real-time garbage collector is supported by JamaicaVM (see Section 3.2), which allows the use of Heap memory and eases the development of RTSJ applications by avoiding building complex memory models.

In the presence of shared objects, RTSJ provides several resource sharing policies like priority Inheritance [40] and Priority Ceiling Protocol (PCP) [36]. Among these protocols, the PCP yields the minimised blocking time (i.e., one critical section only) and guarantee dead-lock free resource accesses. In addition, asynchrony is well handled via a set of asynchronous event handling facilities. Finally, a set of time-related facilities (e.g., real-time system clock and `HighResolutionTime` with granularity of nanoseconds) are supported.

3.2 RTSJ Implementations and VMs

This specification was firstly implemented by TimeSys[4]. This implementation (i.e., a RTSJ-compliant VM and a RTSJ reference implementation) supports all versions of Linux. Later, Aicas GmbH[5] provided a different RTSJ implementation in their JamaicaVM, supporting a wide range of Real-time operating systems, such as Linux, VxWorks and QNX. In addition, there also existed other virtual machines that are compliant with RTSJ, e.g., jRate[6], OVM [4] and Aero JVM [7].

Among these VMs, JamaicaVM provides hard real-time guarantees and is the mostly adopted VM for executing RTSJ applications. Currently, JamaicaVM supports RTSJ V1.0.2 and is working towards RTSJ 2.0 implementation based on Java 8. In particular, a real-time Garbage Collector (GC) is supported by JamaicaVM [41]. This GC executes each time when threads issues requests to allocate an object in a preemptable fashion, and will not interrupt application threads. In JamaicaVM manual[8], an analytical approach for measuring worst-case execution time in the presence of the real-time garbage collector are provided.

In total, thirty-eight priority levels are supported by this VM, where priority levels 11-38 are designated for real-time threads (through the class `RealtimeThread`) and priority levels 1-10 belong to the standard Java. That is, JamaicaVM also compliant with the standard non real-time threads. However, in this work, we assume that each thread in the given application will be mapped to a real-time thread, and each real-time thread has a unique priority.

3.3 Targeted RTSJ Run-Time Environment

As the first attempt of this methodology, we assume a simple but widely applied real-time system model. The current version of proposed methodology aims transferring standard Java applications in uniprocessor systems to real-time applications. In this work, we consider those threads can be transferred to either periodic or aperiodic real-time threads,

[4] https://www.timesys.com
[5] https://www.aicas.com/cms/en
[6] http://jrate.sourceforge.net/
[7] http://www.aero-project.org
[8] https://www.aicas.com/cms/sites/default/files/JamaicaVM-8.2-manual-web.pdf

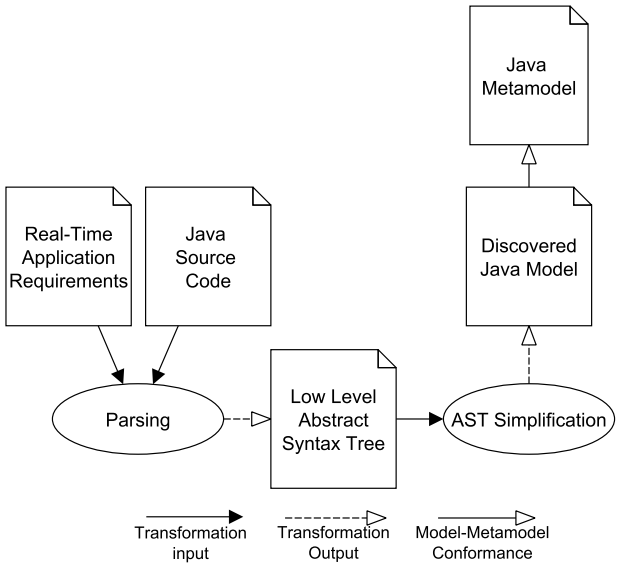

Figure 2. Discovering Java a model from Java source code

with their release parameters pre-defined in application requirements. Fixed-priority preemptive scheduling policy is enforced for coordinating executions of real-time threads. Threads can access to shared objects, but they must do so with the PCP applied, which is an optimal resource sharing solution in uniprocessor systems (i.e., deadlock-free and minimised blocking time) [16]. JamaicaVM v8.5 (with RTSJ v1.0.2 and Java 1.5) is applied as the underlying virtual machine. Finally, Memory management is handled by the real-time GC provided by JamaicaVM.

4 Proposed Methodology

The structure of our proposed methodology is shown in Figure 1. The first step in our approach is the reverse-engineering of the Java programs into models. In order to do this, we use a Text-to-Model transformation to convert the source code of standard Java applications (i.e. without real-time properties) into Java models. In addition to the Java source code, we also take a list of real-time application requirements that provide necessary information (e.g. timing and priority parameters for each thread, scheduling policies, etc.) for building a real-time system.

With the two inputs, a Java model that conforms to the Java metamodel is produced. With the Java model, there is a need to perform a model validation to check if the given application is capable to satisfy all the temporal requirement after being transformed to a real-time application. If the model validation passes (the response time of each real-time thread is equal to or less than its deadline), it means that the to be transformed application is schedulable. We then perform a model-to-model transformation to transform the Java model to the target Java model (named Java Model′) which uses RTSJ Java constructs. This transformation is a endogenous transformation - that the target model also conforms to the Java Metamodel (for RTSJ does not introduce new language syntax in Java). The transformation is derived based on our knowledge in RTSJ and our defined mappings from standard Java classes to RTSJ classes. The target Java Model′ is then used as an input for a model-to-text transformation, which is responsible to transform Java models back to Java source code.

With the proposed approach, applications developed originally in standard Java can be automatically converted to real-time applications based on RTSJ, and are directly executable on JamaicaVM. The whole conversion process is conducted without intervention of software developers, and hence, eliminates human-related erroneous factors. In addition, the proposed methodology removes the need of expertise in the real-time systems design and necessary knowledge of any targeted real-time programming specification. Consequently, the cost for real-time systems development can be significantly reduced with the high productivity brought by MDE. In the following sections, we will discuss the transformations involved in the approach individually.

4.1 Reverse Engineering Transformation (T2M)

The reverse engineering transformation is the very first transformation in the tool chain. Reverse engineering transformation is also normally referred to as *Model Discovery*, in the sense that a model is *discovered* from the source code. There are a number of available tools and approaches that are capable of performing this task. For example, JaMopp [20], Spoon [34] and MoDisco [9] are all feasible tools to perform reverse engineering from Java. It is to be noted that in this step, there is a strict requirement for model discovery in our proposed approach, that there shall be no information loss during the model discovery. This is typically due to the fact that the discovered model will be analysed, changed and then transformed back to the source code. If there is any information loss, the eventual transformed source code is not complete.

Our proposed reverse engineering transformation is presented in Figure 2. The Java source code and the real-time application requirements (we assume here that this would be a Java class with static fields) are firstly parsed into an Abstract Syntax Tree (AST), which is a very low-level representation model of the Java source code. The problem with ASTs is that they are difficult to navigate and analyse. Therefore, an AST simplification is performed to produce the discovered Java model that conforms to the Java Metamodel. The AST simplification is a reversible procedure, so that even if the discovered Java model is changed, the inverse of the AST simplification is still able to produce an AST that preserves all the original information (with changes applied to the AST).

4.2 Model Validation

With the discovered Java model, a model validation is performed to check whether the given application can meet its timing constraints defined in its real-time application requirements. The validation process first checks whether the given requirements are consist with the input Java source code (e.g., whether threads' ids in the application are consistent with the ones given by the requirement). The real-time application requirements provide full thread releasing and scheduling information for each thread that needs to be transferred to real-time threads.

Then (assuming threads defined in the requirements are consist with the source code), an analysis is performed to verify the timing properties of each real-time thread via the Response Time Analysis (RTA) [2] . For a given task τ_i in the targeted system, the worst-case response time R_i is then calculated by adding the task worst-case execution time C_i, the blocking time B_i and the interference time due to preemptions from higher-priority tasks I_i:

$$\begin{aligned} R_i &= C_i + B_i + I_i \\ &= C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \end{aligned} \tag{1}$$

where $\lceil \cdot \rceil$ denotes the ceiling function and $hp(i)$ returns the set of tasks that have a higher priorities than τ_i's priority (denoted as $Pri(\tau_i)$ in the following equations).

In the presence of shared resources, C_i is further extended to include the time τ_i spends on executing each shared resources, as shown in (2).

$$C_i = WCET_i + \sum_{r^k \in F(\tau_i)} N_i^k c^k \tag{2}$$

where $WCET_i$ denotes the worst-case execution time of τ_i without accessing any shared resources, $F(\tau_i)$ gives a set of resources accessed by τ_i, c^k gives the worst-case execution time of a given resource k and N_i^k returns the number of access τ_i can issue to resource k in one release. Note, as described in [42], the overheads cost by the real-time GC for allocating objects are included into the worst-case execution time (notation $WCET$) of each thread, which should be predefined in the application requirements based on memory usage of each thread (i.e., keyword *new*).

The blocking time B_i indicates the time period that task τ_i is prevented from executing due to either the non-preemptive sections from the underlying operating system or a low-priority thread that accesses a shared resource with a ceiling priority higher than $Pri(\tau_i)$, as give in equation (3).

$$B_i = max\{\hat{c_i}, \hat{b}\} \tag{3}$$

in which $\hat{c_i}$ denotes blocking due to resource accessing and $\hat{b}$ gives the longest non-preemptive section period in the underlying operating system. Finally, $\hat{c_i}$ is computed by equation (4) with PCP assumed. Note, the value of $\hat{b}$ depends on the operating system and underlying hardware, and should be measured and reported in the input application requirements.

$$\hat{c_i} = \max\{c^k | N_{lp}^k > 0 \wedge Pri(r^k) \geq Pri(\tau_i)\} \tag{4}$$

This equation finds all resources are accessed by tasks with a lower priority but have a higher ceiling priority than $Pri(\tau_i)$, and gets the longest critical section among these resources as the worst-case blocking time for τ_i.

With the above analysis, the worst-case response time for each release of the application threads is bounded. If the system validation yields a schedulable system with given threads' scheduling parameters in the requirement, the proposed methodology processed to the next step, where it transfers the standard Java model to the real-time Java model based on the pre-generated metamodel.

However, the current version of model validation heavily depends on system requirement for pre-measured computation cost of each thread and shared resource. In the future, mature worst-case execution time measuring techniques can be integrated into the proposed toolchain for a higher degree of automation.

4.3 RTSJ Model Transformation

After the RTA (model validation) passes, in the next step, a model-to-model transformation (i.e., block *M2M* in Figure 1) is performed to migrate the standard Java model to RTSJ Java model. It is to be noted that this transformation is endogenous in the sense that RTSJ does not introduce new Java abstract syntax, therefore the both the source model and the target model conform to the same Java Metamodel. This migration is performed based on a set of transformation rules, which specify the mapping from standard Java facilities to RTSJ facilities provided in package `javax.realtime`. For the interest of brevity, below we elaborate on two major Java to RTSJ transformations (i.e., threads and synchronisation) and then briefly describe the transformation rules to resolve RTSJ run-time environment dependencies.

4.3.1 Standard Threads to RTSJ Threads

One major difference between standard Java and RTSJ is the schedulable entities (i.e., threads), where Java uses `java.lang.Thread` while RTSJ applications relies on `javax.realtime.RealtimeThread`. Figure 3 shows an example transformation rule (named *Thread2RealtimeThread*), which transforms a standard Java thread into a real-time thread.

On the left side of the figure is the source model of the transformation. The source model contains a number of standard Java threads that are extracted from the input source code. The transformation rule maps each standard Java thread to a real-time thread in RTSJ, as seen in the target model. In the source model, each thread is associated with a explicitly defined `java.lang.Runnable` object, which contains all functionality implementations that should be executed by

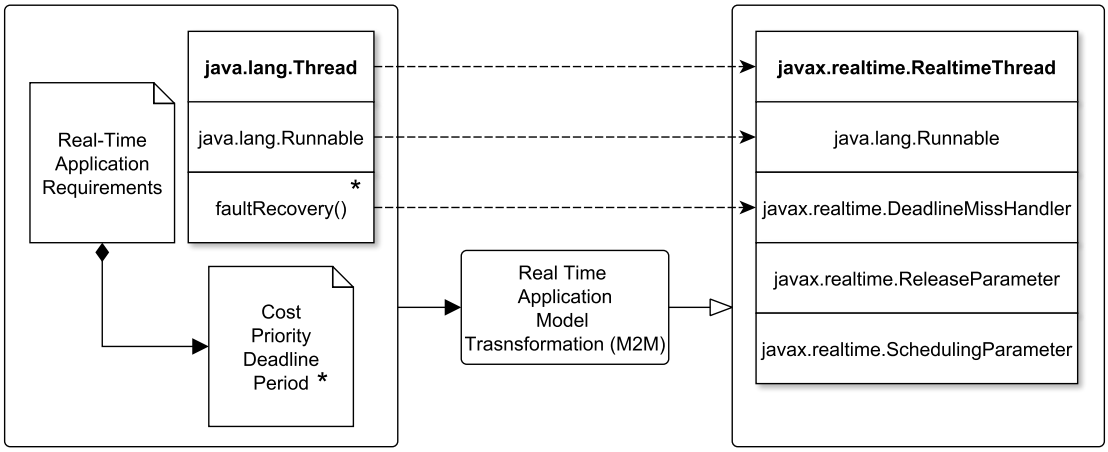

Figure 3. Example transformation rule *ead2RealtimeThread*

this thread. This Runnable objective will be passed directly into the `run()` method[9] of the real-time thread constructed during this transformation phase. In addition, each standard Java thread may also define an optional `faultRecovery()` method, which contains recovery operations to be performed in the situation that the thread misses its deadline. The transformation rule transforms the code in the `faultRecovery()` method into RTSJ dedicated handlers, in this instance, the transformation creates a `DeadlineMissHandler` based on `javax.realtime.AsyncEventHandler`. If such a method is not provided, an immediately system shutdown is triggered in case of any deadline misses.

Then, the transformation rule creates RTSJ parameters such as ReleaseParameter and SchedulingParameter to apply timing constraints. As defined in the previous section, the source model also contains a set of real-time application requirements, which is embedded in the form of a Java class with static fields. In this real-time application requirements, a set release and scheduling parameters specifying the execution eligibility and temporal proprieties of each thread should be defined by the users, as shown in Figure 3. Note that a period parameter is mandated for periodic threads, but should be omitted by those aperiodic threads to facilitate the identification of various activation pattern of real-time threads. Listing 1 provides a simple example of real-time application requirements for a periodic thread and an aperiodic thread.

```
// Periodic thread PT1
public static final String PT1_id = "PT1";
public static final int PT1_Period = 250;
public static final int PT1_Cost = 200;
public static final int PT1_Deadline = 250;
public static final int PT1_Priority = 20;

// Aperiodic thread AT1
public static final String PT1_id = "AT1";
public static final int AT1_Cost = 30;
public static final int AT1_Deadline = 50;
public static final int AT1_Priority = 25;
```

Listing 1. An example of the real-time application requirements

During the transformation, threads' priorities are constructed as `javax.realime.PriorityParameter` objects and other parameters (e.g., cost, deadline) are modelled into `javax.realtime.ReleaseParameters` objects, as shown in Figure 3. Depending on whether a given thread is associated with a period, the release parameter objects are further modelled to either `PeriodicParameters` or AperiodicParameters[10] objects provided in `java.realtime.package`. In addition, it is to be noted that the `DeadlineMissHandler` transferred from the pre-defined `faultRecovery()` method is also integrated into the `ReleaseParameters` object, as defined by the RTSJ specification.

With above real-time thread parameters and logic constructed, a standard Java thread can be transferred into a RTSJ thread via passing these parameter objects and the Runnable object into the construction method, where the Runnable object is called inside the run() method. For periodic threads, method `waitForNextPeriod()` is added into its run() method to achieve a periodic release behaviour. Note that as we assume the presence of a real-time GC, real-time threads are allowed to execute in Heap memory so that those memory area parameters associated to the real-time threads

[9]This Runnable object does not replace the Runnable of the real-time thread. It is passed into the real-time thread and executed by invoking its `run()` just for the execution the logic implementation in the method. The `run()` method of a real-time thread may contain extra implementation for realising its activation behaviours.

[10]The Class `PeriodicParameters` and AperiodicParameters are realised by RTSJ as the sub-classes of `ReleaseParameters`in package `javax.realtime`

are set as empty, which by default are executed in Heap by JamaicaVM.

Finally, it is to be noted that we used a hybrid (i.e. imperative and declarative) transformation approach. The transformation rule in Figure 3 is one of the rules we define for the entire transformation. There are also rule dependencies, for example, we also define a *faultRecovery2DeadlineMissHandler* transformation rule, this rule should be called within our *Thread2RealtimeThread* transformation rule. This execution behaviour is typical for hybrid transformations and we recommend using the Epsilon Transformation Language [27] to write and execute the transformation.

4.3.2 Java Synchronisation to Real-Time Resource Control

Besides thread scheduling, another major difference between stand Java application and RTSJ is thread synchronisation approach, where in RTSJ each shared resource must be protected by proper resource sharing protocols (i.e., `javax.realtime.MonitorControl`) to ensure bounded resource accessing time for each resource access. As described in Section 3.3, the PCP is applied in the proposed toolchain for managing shared resources in transformed RTSJ appellations. This section describes the transformation rule *Lock2RealtimeLock* that transforms standard Java synchronisation to RTSJ PCP facility[11]. In the current version of the proposed toolchain, we assume that each thread can only can only access one resource at a time.

To perform proper transformation, we define a set of rules towards thread synchronisation in the input source code and real-time application requirements, as described below.

- The user should be aware of all shared objects (i.e., ones that are read and written by more than one threads) and the threads that access those shared objects in the input Java source code.
- Each shared resource must be implemented as a class with the required operations implementing its methods properly protected by the standard Java synchronisation approach, i.e., via synchronised coding blocks and methods.
- The use of `wait()`, `notify()` and `notifyAll()` facilities in Standard Java are not allowed i.e., threads are not self suspended.
- For a given shared resource, say r^k, the user should provide its ceiling priority (i.e., the maximum priority of threads that access r^k) in the real-time application requirements.

Below we provide an example of a valid input source code of a shared object class `SharedResource` and the associated application requirements conforming to the rules defined above.

[11]The PCP in RTSJ is implemented by class `PriorityCeilingEmulation` in package `javax.realtime` by extending class `MonitorControl`.

```
// real-time application requirements
public static final String SR1_id = "sr1";
public static final int SR1_Ceiling = 25;
...
// input source code
class SharedResource{
    String id;

    public synchronised void access(){
        critical_section;
    }
}

SharedResource sr1 = new SharedResource("sr1");
```

Listing 2. An example of RTSJ synchronisation with PCP applied

With above rules defined, the standard Java synchronisation approach can be effectively transferred to real-time resource sharing techniques. First, the source Java model generated in Section 4.1 (i.e., the model that extracts all objects in the input source code) is able to identify all shared objects (i.e., classes) and required operations by detecting the *synchronised* keyword. Then, for each object created based on these classes, its associated priority ceiling priority can be found in the application requirements. With above information, RTSJ implementation can be generated by adding a `PriorityCeilingEmulation` instance to that object, as shown below.

```
SharedResource sr1 = new SharedResource("sr1");
// With PCP enforced.
PriorityCeilingEmulation PCP =
     PriorityCeilingEmulation.instance(25);
MonitorControl.setMonitorControl(sr1, PCP);
```

Listing 3. An example of RTSJ synchronisation with PCP applied

The transformation first generates a `PriorityCeiling Emulation` instance for the shared object with its ceiling priority assigned based on the requirements. Then, the control policy for this shared resource is set to this PCP instance so that each thread accesses this object will raise its priority, and later on restore its original priority once the lock is released. This is performed automatically by JamaicaVM, assuming the transformation is conducted successfully.

Finally, note that the priority ceiling priorities of shared objects must be correctly assigned in the real-time application requirements. Otherwise (e.g., the ceiling is lower than that of the accessing thread), a `CeilingViolationException` will be thrown by JamaicaVM.

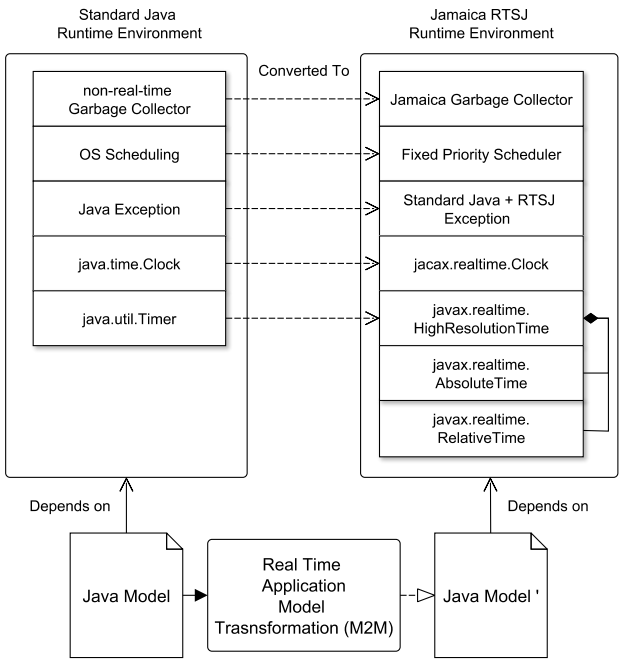

Figure 4. Transformation rule to resolve run-time environment dependencies

4.3.3 Transformation Rules for Run-Time Environment Dependencies

Besides the transformation towards those major RTSJ facilities, RTSJ applications require dedicated run-time Environment in order to be executed with real-time properties. Therefore, there is a need to execute a transformation to convert standard Java run-time Environment dependencies into RTSJ run-time Environment dependencies.

Figure 4 illustrates the transformation rule that converts these run-time environment dependencies. As shown in the figure, a typical standard Java run-time environment is equipped with a non-real-time garbage collector, a system clock with granularity in milliseconds, utility timers, standard Java exceptions. In addition, standard Java threads are mapped to native level threads and are scheduled by the underlying operating system.

In order for the output application with the RTSJ facilities generated in the above sections to execute successfully and to satisfy the application's timing requirements, a RTSJ run-time environment is required. The right side of Figure 4 shows an example of JamaicaVM-based RTSJ run-time environment. This JaimaicaJM RTSJ run-time is equipped with a dedicated Real-Time garbage collector running in the Heap memory, a real-time wall clock, finer-grained HighResolutionTime objects with granularity in nanoseconds and additional RTSJ related exceptions and a Fixed Priority preemptive scheduler mechanisms. The mappings from Standard Java run-time environment to RTSJ run-time environment is drawn in Figure 4 using dashed lines.

Among the facilities considered in the targeted RTSJ runtime environment, the real-time garbage collector is enabled and the fixed priority scheduler is applied as default by JaimaicaVM. The standard Java clock is transformed to `javax.realtime.Clock` in `javax.realtime` package so that the invocations to obtain the current system time[12] is replaced by the method `Clock.getRealtimeClock().getTime()`. Further, as required by the RTSJ specification, time units in RTSJ should be modelled by `HighResolutionTime` as either a `AbsoluteTime` or a `RelativeTime` object, where the later two time units are sub-classes of the former. For instance, the temporal properties (e.g., period, cost and deadline) for a real-time thread will be generated as the `RelativeTime` objects before they are assigned to the construction method of `RealtimeThread`. Finally, additional exceptions introduced by Class RTSJ is generated into the output implementation where applicable. For instance, for each synchronised method, a `CeilingViolationException` exception should be thrown for illegal ceiling priority assignment. After this transformation is executed, the target model should have dependencies to RTSJ run-time resolved.

5 Open Challenges and Further Research Directions

Plenty of open questions and research opportunities are introduced from this work. In this section, we discuss some of the challenges and point towards selected future research directions.

First, the current version of the proposed toolchain assumes the presence of a real-time garbage collector (e.g., the one supported by JamaicaVM), which allows the execution of real-time threads in *Heap* memory. However, in situations where a real-time GC is not available, an explicit memory management model must be constructed by `ScpoedMemory` to guarantee temporal requirements of real-time Java applications, as executing in Heap memory will suffer from unpredictable interference of standard Java garbage collector. One major challenge of this Java to RTSJ automation approach is to provide a generic memory management model that suits all types of RTSJ applications. The memory management model in RTSJ is highly specific to the application characteristics (especially, the correlation of those real-time threads) and is difficult to generate based merely on the knowledge from the input source code.

A possible workaround is to enforce an SCJ (Safety-Critical Java)-like programming model [39], which imposes restrictions towards the application structure but is sufficient to provide the required functionalities. In the SCJ, threads are grouped into *missions*, which are executed by one or more *mission sequencers* (i.e., missions can be executed concurrently). This programming model conforms to a specific

[12] The method `System.currentTimeMillis()`.

memory management framework [38]. In the SCJ, each mission has its own memory block and each thread in that mission is also assigned with a private memory area, building upon the memory block of the associated mission. Once a mission is finished (i.e., all its threads are signalled to be terminated), its associated memory block (and subsequently, memory areas of its threads) will be reclaimed during a mission *cleanUp* phase. However, applying this memory model in the proposed methodology requires extra information describing the correlation between those real-time threads in order to allocate them correctly into each individual group for memory allocation.

Second, as an initial attempt on the topic, we have targeted at a simple and widely-applied uniprocessor environment and focused mainly on the functionality of the proposed toolchain. There is a trend that most of the existing real-time programming specifications are extended to support multiprocessor and distributed systems [50]. The proposed approach can also be extended to support multiprocessor features with multiprocessor scheduling policies and resource sharing techniques taken into account. In addition, as the application scenarios of real-time systems become more sophisticated, supporting complex system semantics (e.g., in the presence of release jitters or shared resources) is also desirable and should be investigated.

In addition, as illustrated in Figure 1, there is an open question to be answered when the given applications are found unschedulable after model validation. One possible solution would be the reconfiguration of system scheduling parameters to achieve better schedulability (i.e., transferring systems that are deemed unschedulable into feasible real-time systems). Such reconfiguration is worthwhile especially for complex systems (e.g., multiprocessor systems with shared resources), where optimal scheduling solutions may not be available. In such cases, a search-based algorithm could be applied for searching threads' parameters and feasible resource sharing protocols that can achieve a schedulable system [49]. In addition, further improvement can be made towards other perspectives of real-time systems, such as sustainability and robustness in the presence of additional interference.

From the programming language perspective, the proposed automated toolchain can be generalised to support different programming languages (e.g., C/C++ and Ada) and their real-time, safety-critical and high-integrity extension profiles (e.g., MISRA C/C++ [19, 44] and Spark Ada [6]). Such efforts are worthwhile as they remove the restriction on the usage of a specific programming language (and its extensions) in the proposed automated toolchain and provide solutions towards those major programming languages in embedded systems.

From the model-driven perspective, for those programming languages where reverse engineering facilities may not be available (e.g., C and Ada), modelling real-time systems from system specification directly and then generating implementation via code generation facilities would be desirable. There are several modelling languages which are capable of modelling real-time systems, e.g., the Architectural Analysis and Design Language (AADL), the Unified Modelling Language (UML), the Systems Modelling Language (SysML), the Modelling and Analysis of Real-Time Embedded Systems (MARTE) UML profile, and the AADL for UML profile are all feasible languages for modelling real-time systems.

However, there are shortcomings in these languages discussed above. AADL is not an open modelling language, and there is a lack of modelling capabilities for the system behaviour. UML is a general modelling language. However, it lacks the formalism needed in modelling of the real-time systems. SysML shares the same problem as UML. MARTE provides extensive modelling capabilities, which leads to the complexity of the language itself. Consequently, a MARTE model could get complex quickly, leading to complex models and diagrams which are hard to manage. A new modelling language is therefore needed for the real-time systems community to address the above shortcomings. With this modelling language, we could generate the real-time applications in programming languages such as Java, Ada and C.

6 Conclusion

This paper proposes a model-driven methodology that automatically transforms time-sharing Java applications to real-time applications in RTSJ. This methodology eases the development of real-time systems by allowing software engineers to construct real-time Java applications without necessary knowledge of the RTSJ programming specification. In addition, the proposed methodology is favourable to those organisations with a need to re-develop their products to possess real-time features. The proposed methodology provides a real-time system development solution that reduces software development cost, increases productivity and eliminates human-related errors. In this paper, a complete standard Java to RTSJ conversion automation architecture is presented with required actions during each transformation phase described in detail. In addition, transformation rules are presented for generating major RTSJ facilities and the RTSJ run-time environment based on the JamaicaVM with the given inputs.

The proposed methodology opens up plenty of research questions and possible research directions, which can be investigated together by the embedded systems, programming languages as well as MDE communities. They have been discussed with motivation and preliminary approaches. In future, we aim to provide a complete and fully functional toolchain for the proposed Java to RTSJ automated methodology to prove the concept presented in this paper and to evaluate the efficacy of the proposed Java to RTSJ automated toolchain.

References

[1] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. 2007. A real-time Java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 1 (2007), 5.

[2] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal* 8, 5 (1993), 284–292.

[3] Neil C Audsley, Yu Chan, Ian Gray, and Andy J Wellings. 2014. Real-Time Big Data: the JUNIPER Approach. (2014).

[4] Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbruster, Edward Pla, and David Holmes. 2006. A real-time java virtual machine for avionics-an experience report. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE, 384–396.

[5] Konstantinos Barmpis, Dimitrios Kolovos, and Justin Hingorani. 2018. Towards a framework for writing executable natural language rules. In *European Conference on Modelling Foundations and Applications*. Springer, 251–263.

[6] John Barnes. 1997. *High integrity Ada: the SPARK approach*. Vol. 189. Addison-Wesley Reading.

[7] Jean Bézivin. 2005. On the unification power of models. *Software & Systems Modeling* 4, 2 (2005), 171–188.

[8] Gregory Bollella and James Gosling. 2000. The real-time specification for Java. *Computer* 33, 6 (2000), 47–54.

[9] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. Modisco: A model driven reverse engineering framework. *Information and Software Technology* 56, 8 (2014), 1012–1032.

[10] Alan Burns and Andy Wellings. 2016. *Analysable Real-Time Systems: Programmed in Ada.* CreateSpace Independent Publishing Platform.

[11] Alan Burns and Andrew J Wellings. 2001. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education.

[12] Wanli Chang and Samarjit Chakraborty. 2016. Resource-aware automotive control systems design: A cyber-physical systems approach. *Foundations and Trends in Electronic Design Automation* 10, 4 (2016), 249–369.

[13] Wanli Chang, Dip Goswami, Samarjit Chakraborty, and Arne Hamann. 2018. OS-aware automotive controller design using non-uniform sampling. *ACM Transactions on Cyber-Physical Systems* 2, 4 (2018), 26.

[14] Wanli Chang, Dip Goswami, Samarjit Chakraborty, Lei Ju, Chun Xue, and Sidharta Andalam. 2017. Memory-aware embedded control systems design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 4 (2017), 586–599.

[15] Wanli Chang, Alma Pröbstl, Dip Goswami, Majid Zamani, and Samarjit Chakraborty. 2015. Reliable CPS design for mitigating semiconductor and battery aging in electric vehicles. In *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*. 37–42.

[16] Robert I. Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *Acm Computing Surveys* 43, 4 (2011), 1–44.

[17] Ian Gray, Neil Cameron Audsley, Jamie Garside, Yu Chan, and Andrew John Wellings. 2015. FPGA-based acceleration for Real-Time Big Data Systems. In *9th HiPEAC workshop on Reconfigurable Computing*.

[18] Ian Gray, Yu Chan, Jamie Garside, Neil C. Audsley, and Andy J. Wellings. 2015. FPGA-based hardware acceleration for Real-Time Big Data systems.

[19] Les Hatton. 2004. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology* 46, 7 (2004), 465–472.

[20] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. 2009. Closing the gap between modelling and java. In *International Conference on Software Language Engineering*. Springer, 374–383.

[21] Thomas Henties, James J Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. 2009. Java for safety-critical applications. In *2nd international workshop on the certification of safety-critical software controlled systems (SafeCert 2009)*.

[22] Erik Yu-Shing Hu, Eric Jenn, Nicolas Valot, and Alejandro Alonso. 2006. Safety critical applications and hard real-time profile for Java: a case study in avionics. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*. ACM, 125–134.

[23] Ari Jaaksi. 2002. Developing mobile browsers in a product line. *IEEE software* 19, 4 (2002), 73–80.

[24] A Juan, Jorge Garrido, Juan Zamorano, and Alejandro Alonso. 2014. Model-driven design of real-time software for an experimental satellite. *IFAC Proceedings Volumes* 47, 3 (2014), 1592–1598.

[25] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. 2009. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*.

[26] Timothy Patrick Kelly. 1999. *Arguing safety: a systematic approach to managing safety cases.* Ph.D. Dissertation. University of York York, UK.

[27] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*. Springer, 46–60.

[28] Dimitrios S Kolovos, Ran Wei, and Konstantinos Barmpis. 2013. An approach for efficient querying of large relational datasets with ocl-based languages. In *XM 2013–Extreme Modeling Workshop*. 48.

[29] Björsn Konieczek, Michael Rethfeldt, Frank Golatowski, and Dirk Timmermann. 2015. Real-time communication for the internet of things using jcoap. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, 134–141.

[30] Shaoshan Liu, Jie Tang, Chao Wang, Quan Wang, and Jean-Luc Gaudiot. 2017. Implementing a Cloud Platform for Autonomous Driving. *arXiv preprint arXiv:1704.02696* (2017).

[31] Shaoshan Liu, Jie Tang, Chao Wang, Quan Wang, and Jean-Luc Gaudiot. 2017. A unified cloud platform for autonomous driving. *Computer* 50, 12 (2017), 42–49.

[32] HaiTao Mei, Ian Gray, and Andy Wellings. 2016. Real-Time stream processing in java. In *Ada-Europe International Conference on Reliable Software Technologies*. Springer, 44–57.

[33] Parastoo Mohagheghi and Vegard Dehlen. 2008. Where is the proof?-A review of experiences from applying MDE in industry. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 432–443.

[34] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[35] Ben Potter, David Till, and Jane Sinclair. 1996. *An introduction to formal specification and Z.* Prentice Hall PTR.

[36] Ragunathan Rajkumar. 2012. *Synchronization in real-time systems: a priority inheritance approach*. Vol. 151. Springer Science & Business Media.

[37] Douglas C Schmidt. 2006. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-* 39, 2 (2006), 25.

[38] Martin Schoeberl, Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Stephan E Korsholm, Anders P Ravn, Juan Ricardo Rios Rivas, Tórur Biskopstø Strøm, Hans Søndergaard, Andy Wellings, and Shuai Zhao. 2017. Safety-critical Java for embedded systems. *Concurrency and Computation: Practice and Experience* 29, 22 (2017), e3963.

[39] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P Ravn. 2007. A profile for safety critical java. In *10th IEEE International*

Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07). IEEE, 94–101.

[40] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. 39, 9 (1990).

[41] Fridtjof Siebert. 2007. Realtime garbage collection in the JamaicaVM 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. Citeseer, 94–103.

[42] Fridtjof Siebert. 2010. Concurrent, parallel, real-time garbage-collection. In *ACM Sigplan Notices*, Vol. 45. ACM, 11–20.

[43] Rashmi P Sonar and Rani S Lande. 2018. Javolution-Solution for Real Time Embedded System. In *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*. IEEE, 1–10.

[44] Chris Tapp. 2008. An introduction to MISRA C++. *SAE international journal of passenger cars-electronic and electrical systems* 1, 2008-01-0664 (2008), 265–268.

[45] Kleanthis Thramboulidis. 2007. IEC 61499 in factory automation. In *Advances in Computer, Information, and Systems Sciences, and Engineering*. Springer, 115–124.

[46] Kleanthis Thramboulidis and Alkiviadis Zoupas. 2005. Real-time Java in control and automation: a model driven development approach. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, Vol. 1. IEEE, 8–pp.

[47] Christian Wawersich, Michael Stilkerich, and Wolfgang Schröder-Preikschat. 2007. An OSEK/VDX-based multi-JVM for automotive appliances. In *Embedded System Design: Topics, Techniques and Trends*. Springer, 85–96.

[48] Andrew J Wellings. 2004. *Concurrent and real-time programming in Java*. John Wiley New York.

[49] Shuai Zhao. 2018. *A FIFO Spin-based Resource Control Framework for Symmetric Multiprocessing*. Ph.D. Dissertation. University of York.

[50] Shuai Zhao, Andy Wellings, and Stephan Erbs Korsholm. 2015. Supporting multiprocessors in the ICECAP safety-critical java run-time environment. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, 1.

[51] Athanasios Zolotas, Ran Wei, Simos Gerasimou, Horacio Hoyos Rodriguez, Dimitrios S. Kolovos, and Richard F. Paige. 2018. Towards Automatic Generation of UML Profile Graphical Editors for Papyrus. In *Modelling Foundations and Applications*, Alfonso Pierantonio and Salvador Trujillo (Eds.). Springer International Publishing, Cham, 12–27.