First Steps for Performance Monitoring of Petri Net Simulator Renew with Kieker

Marcel Hansson and Daniel Moldt
{firstname}.{lastname}@uni-hamburg.de
University of Hamburg, Hamburg, Germany

Abstract

To investigate how Kieker can be used for Java applications with dynamically loaded modules of the Java Platform Module System (JPMS), we conducted experiments with Renew. Renew is a Java-based editor and simulator for reference nets, a nets-in-nets Petri net formalism that allows the execution of Java code during transition firing. Monitoring Renew's simulator is challenging because Renew loads core components as plugins in dynamic JPMS module layers and can reach high simulation speeds. We demonstrate how the Kieker observability framework can be applied using Kieker source instrumentation. The high execution throughput made the Kieker text writer unsuitable without limiting fired transitions. Nevertheless, the collected traces revealed specific areas for future optimization.

1 Introduction

Kieker does not allow direct monitoring of modularized Java that uses the JPMS layer concept in the usual manner. We have chosen the *Reference Net Workshop* Renew as a testbed. The goal of this work is to monitor and analyze the simulator component of Renew to identify performance bottlenecks and at the same time get insights for Kieker. Renew has a plugin-based architecture. Each plugin is loaded in its own JPMS layer. Therefore, all the challenges of monitoring JPMS-based applications with Kieker are present.

We faced two main obstacles. First, simple, automatic Java agent instrumentation (via AspectJ) only works for the static boot layer, with special Java virtual machine command line options that are not applicable to dynamic modules. Second, Renew's simulation is quite fast and consists of many methods, producing monitoring data at rates that exceed the throughput of Kieker's text file writer in our setup.

We addressed the first challenge by applying Kieker source instrumentation [8] to insert the Kieker monitoring code directly into the source code. The second challenge was addressed by restricting the simulation to a lower number of transition firings and adapting the Kieker file writer's queue size. After overcoming these challenges, we recorded traces that revealed method-level hotspots in the simulator for further investigation.

The remainder of the paper is structured as follows: First some foundations are introduced. Next, we explore the two faced obstacles and the solutions. Afterward we present as an example some findings in Renew. In the end, we give a small conclusion with further research topics as an outlook.

2 Foundations

In this section we quickly present Renew, the JPMS and Kieker.

2.1 Renew

Renew [10] is a Java-based tool for modeling and simulating various Petri net formalisms, in particular reference nets. Petri nets are a mathematical modeling formalism for discrete event systems, defined as bipartite graphs of places and transitions with tokens representing system states. Their execution semantics, based on transition firing, enable analysis of properties such as concurrency, synchronization, and reachability (see [3] for a broader overview and definitions). Reference nets are a high-level form of Petri nets defined in [1]. They allow developing whole applications based on Petri nets, with Renew acting as the execution and development environment [4]. Renew features a plugin architecture [2] allowing easy expansion and modularity. Even core components of Renew, like the drawing framework, are implemented as plugins.

As research software, Renew is used in many contexts: in lectures and exercises to teach the concepts of Petri nets, in practical courses for hands-on software development, for long-running simulations for analysis aspects and as a runtime environment for Petri net-based applications. Especially the latter would benefit from monitoring parts of the simulation component.

2.2 Java platform module system and Renew

The *Java Platform Module System (JPMS)*, introduced with Java 9, provides a framework for structuring Java applications into *modules* with explicit declarations of dependencies, exported packages, and service interactions. This enhances encapsulation, main-

tainability, and runtime integrity [5]. Renew uses the JPMS since release 4.0 [7]. Every Renew plugin is integrated in one module. At runtime for each plugin (=module) a module layer is created. Module layers[1] are immutable hierarchical groupings of resolved modules, rooted in the boot layer, that govern class loading and visibility. The module layers enable the dynamic loading and unloading of plugins and their constituent classes, which was not easily possible with the previous architecture.

2.3 Kieker

Kieker is an observability framework for monitoring and analyzing the runtime behavior of concurrent Java applications [6, 11].

3 Obstacles

Two main hurdles arose when attempting to monitor the simulator component of Renew: The quite unique architecture of Renew and the simulation speed.

3.1 Architecture

Renew's plugin architecture together with the Java Platform Module System (JPMS) results in runtime creation of hierarchically ordered module layers, as mentioned. Each plugin is a JPMS module and at runtime is loaded into a module layer.

The problem now with automatic instrumentation with, for example, the AspectJ-based Kieker Java agent is that a module has to explicitly declare which other modules it reads. By default, it can only read modules and not the classpath (which would contain required classes from AspectJ). With a command-line option for the Java command, it is possible to declare that a module reads the unnamed module (which includes everything on the classpath), but this only works for modules known at start time. It does not work for the dynamically loaded modules/plugins of Renew. It would therefore be possible to monitor the module and plugin system of Renew, which reside in a Loader component as part of the boot layer, but not its simulator component or other plugins.

As the automatic approach with the Java agents did not work, we tried direct source adaptation. There is a tool for Kieker that directly adds instrumentation code to the source code [8]. The tool wraps the body of each method with the code to create Kieker monitoring records. This adds roughly 40 lines of code to each monitored method, making the source code unmaintainable. In contrast, if the approach with AspectJ had worked, it would have required no code changes or just simple annotations to define the monitored methods.

As mentioned, Renew is used in several different contexts. The source code instrumentation could only be applied during the build process for versions used in the contexts where it would make sense and only directly before deployment as part of continuous deployment. This way the developers could work on the unaltered code without the bloated source code. At the moment this is not applied, as we still want to try a more automatic approach.

It should be noted, that we still had to manually adapt source files (module-info to add additional requirements) to make the source instrumentation work. This includes dependencies of Kieker, which could not be resolved automatically, as Kieker itself is not yet modularized. As well as a requirement to Kieker itself. This was something we wanted to avoid and similar adaptation may would have the Java agent approach with AspectJ allowed to work as well. The tool for Kieker source instrumentation though, could be easily adapted to add Kieker (and its requirements if not modularized) as a requirement automatically.

So to quickly summarize this aspect: The Kieker source instrumentation can be used to monitor JPMS layer-based architectures, and with small modification in the future, even without manual editing. Of course an automatic approach without any adaptation (or minimal with annotations) of the source code would still be preferable.

3.2 Simulation Speed

Renew's simulator can reach high throughput in terms of transition firings. In practice this created a second obstacle: the volume of monitoring records generated during realistic runs overwhelmed the Kieker text file writer in our environment. This resulted in a termination of the monitoring during the simulation. Also, the binary writer of Kieker, which compresses the data to binary files before writing, did not solve the problem. It only allowed more transition firings before termination. To obtain usable traces, we restricted monitored runs to a lower number of transition firings and increased queue sizes of Kieker for writing the records from the default 10,000 to 1,000,000. This made text-file logging viable to at least get some data.

Of course, in an actual monitoring context, we would not monitor every method. But as a first step, we wanted to profile the Renew simulator to gain insights into the simulator component and identify points of interest for improvements and monitoring candidates. This was a testing scenario to later monitor the Renew simulations in larger contexts.

4 Results

With the obstacles solved, instrumenting the simulator component produced monitoring traces that could be analyzed with Kieker's tools. As a test scenario, a simple Petri net was created with two places and two transitions arranged in a circle. The simulation was run for 200 rounds, which means there were 400

[1] https://docs.oracle.com/javase/9/docs/api/java/lang/ModuleLayer.html

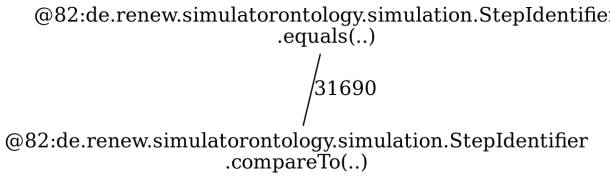

Figure 1: Zoomed in view of a call tree.

Table 1: The ten most common calls.

# Calls	Method
88850	StepIdentifier.getComponents()
88850	StepIdentifier.equals(..)
88850	StepIdentifier.compareTo(..)
19555	Aggregate.getReferences()
15171	Tuple.hashCode()
12473	Searcher.getStateRecorder()
9797	Unify.isBound(..)
8896	Variable.getValue()
8896	Reference.getValue()
8238	Unify.isComplete(..)

transition firings.

Figure 1 shows a zoomed-in view of an aggregated call tree diagram extracted from the produced traces with the Kieker trace analysis tool. The figure showcases the highest number found in the graph. The exact same method is also called at other points in the graph. Adding these together results in 88,850 calls. In table 1 the ten most common calls are shown.

For 400 transition firings these all seem to be unusual high numbers, especially the *StepIdentifier* ones and are therefor promising starting points for performance optimization.

5 Conclusion

We demonstrated that the Kieker framework can be applied to a JPMS and dynamic module-layer-based application like Renew with source instrumentation. The monitoring traces provide actionable insights into simulator behavior and identify hotspots for future optimization. Planned next steps include instrumenting a wider set of Renew plugins and improving monitoring by, e.g., remote TCP writing to a separate machine, limiting the monitored methods, or using compression to hopefully be able to monitor larger simulation runs.

Also, further investigations to enable automatic instrumentation are planned to avoid or reduce the source code adaptations. Analyzing the performance within nets themselves is also ongoing work [9].

Acknowledgement This research is funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. 528713834.

References

[1] O. Kummer. *Referenznetze*. Berlin: Logos Verlag, 2002.

[2] M. Duvigneau. "Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen". `https://ediss.sub.uni-hamburg.de/handle/ediss/3023`. Dissertation. Vogt-Kölln Str. 30, D-22527 Hamburg: University of Hamburg, Department of Informatics, Oct. 2009.

[3] W. Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.

[4] L. Cabac, M. Haustermann, and D. Mosteller. "Renew 2.5 - Towards a Comprehensive Integrated Development Environment for Petri Net-Based Applications". In: *Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings*. Ed. by F. Kordon and D. Moldt. Vol. 9698. Lecture Notes in Computer Science. Springer-Verlag, 2016, pp. 101–112.

[5] N. Parlog. *The Java Module System*. Manning, 2019.

[6] W. Hasselbring and A. van Hoorn. "Kieker: A monitoring framework for software engineering research". In: *Software Impacts* 5 (June 2020).

[7] L. Clasen et al. "Enhancement of Renew to Version 4.0 using JPMS". In: *Proceedings of the International Workshop on Petri Nets and Software Engineering 2022 co-located with the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Bergen, Norway, June 20th, 2022*. Ed. by M. Köhler-Bußmeier, D. Moldt, and H. Rölke. Vol. 3170. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 165–176.

[8] D. G. Reichelt, S. Kühne, and W. Hasselbring. "Towards solving the challenge of minimal overhead monitoring". In: *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. 2023, pp. 381–388.

[9] M. Hansson. "First Investigations of the Possibilities of Performance Monitoring and Analysis of Reference Nets with Kieker". In: *AWPN 2025 workshop proceedings*. Ed. by R. Lorenz. Algorithmen und Werkzeuge für Petrinetze (AWPN). [in press]. 2025.

[10] O. Kummer et al. *Renew – The Reference Net Workshop*. Release 4.2. Aug. 2025.

[11] S. Yang et al. "The Kieker Observability Framework Version 2". In: *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*. 2025, pp. 11–15.