A Desktop 3D Printer in Safety-Critical Java

Tórur Biskopstø Strøm
Department of Informatics and Mathematical Modeling
Technical University of Denmark
torur.strom@gmail.com

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

It is desirable to bring Java technology to safety-critical systems. To this end The Open Group has created the safety-critical Java specification, which will allow Java applications, written according to the specification, to be certifiable in accordance with safety-critical standards. Although there exist several safety-critical Java framework implementations, there is a lack of safety-critical use cases implemented according to the specification.

In this paper we present a 3D printer and its safety-critical Java level 1 implementation as a use case. With basis in the implementation we evaluate the specification and its usability for developers of safety-critical systems.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

Keywords

Safety-critical Java, scoped memory, RepRap

1. INTRODUCTION

The popularity of Java has spawned projects that have brought Java into real-time systems. A continuity of this process is done by The Open Group with the safety-critical Java (SCJ) specification, such that Java can be used in certifiable safety-critical applications. This specification is a work in progress. To evaluate the expressiveness of the specification, the simplicity of the API, and the ease in which safety-critical applications can be written in Java, we need implementations of SCJ and very important use cases. These use cases shall show the strength and weakness of SCJ.

Implementations of SCJ are already on the way [19, 16]. However, use cases are still very rare. In this paper we evaluate the specification by implementing a RepRap 3D desktop printer as a use case. A RepRap is a desktop printer capable of creating 3-dimensional (3D) objects in plastic [8]. Some of the components of a RepRap are printable by the RepRap itself. Therefore, a RepRap is partially self-replicable. The 3D drawings are interpreted by a host computer (a normal PC) and printing instructions are sent to the RepRap controller. The RepRap controller interprets the instructions, moves the printing head, heats the plastic and extrudes it. This controlling has real-time constraints. In our project we have substituted the microcontroller with an FPGA board and rewrote the original C based firmware as a SCJ application.

As SCJ platform we use the Java processor JOP on a Altera DE2-70 FPGA platform. The FPGA platform allows us to build application specific I/O devices to access the sensors and actuators of a RepRap system. Besides implementing the controller software in SCJ, we have built the RepRap printer hardware, electrical circuits to interface the stepper motors, the the FPGA platform, and use simple analog-digital converter for the melding temperature measurement.

The main contribution of the paper is the first real SCJ-based application controlling a robot and providing it in open-source. We also provide feedback on the SCJ specification and API from the point of view of a Java programmer

The paper is organized as follows: The following section presents background on safety-critical Java and the Java processor JOP and related work on SCJ use cases and the RepRap project. Section 3 describes the implementation of the RepRap controller in SCJ on top of JOP. The Evaluation in Section 4 gives feedback on our programming experiences with SCJ and compares the SCJ implementation against an implementation in C. Section 5 provides links to the source of the application and Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

The intention of this work is the evaluation of safety-critical Java; how expressive this specification is and where the restrictions are. Therefore, we present a brief background on safety-critical Java here. Details can be found in the public draft of SCJ [5].

As JVM we use a Java processor, which is implemented in an FPGA. Therefore, we also give some background information on JOP.

2.1 Safety-Critical Java

Safety-critical Java (SCJ) [5] is intended for future safety-critical systems that need certification. To allow certification of Java programs only a very restricted subset of Java is defined. SCJ itself is based on the real-time specification for Java (RTSJ) [1]. It is a subset of RTSJ with some additional class files. It is so defined that it can in principle be implemented on top of RTSJ.

The SCJ specification is developed within the Java community process (JCP) under specification request number JSR 302. To cover different criticality levels, SCJ defines three different levels with increasing complexity of implementations and increasing ex-

JTRES 2012 October 24-26, 2012, Copenhagen, Denmark

pressive power for the application programmer. Level 0 provides a single-threaded cyclic executive. All memory areas (immortal, mission, and private) are available in level 0. As individual executions of handler releases are not preempted, the backing store for the private memory of a handler can be reused by the next handler. Level 1 introduces preemptive scheduling with ceiling based locks. Furthermore, interrupt handlers, written in Java, are allowed in level 1. Level 2 provides the notion of nested missions for more dynamic systems. With the work presented in this paper we are interested in the evaluation of SCJ level 1.

Concurrence is represented as *handlers* in SCJ, similar to RTSJ style event handlers. In fact the SCJ handlers are a subclass of RTSJs BoundAsyncEventHandler. These handlers are either periodic or event triggered.

SCJ has the notion of missions. An application can consist of several missions, where each might represent a different operation mode. The mission itself consists of the handlers and a mission memory. The handlers within a mission are created at initialization phase and the number of handlers is fixed for a mission. Handlers come in two flavors: a periodic event handler to be released time-triggered and an aperiodic handler released by an event. The event to release an aperiodic handler can be a software event or an interrupt.

A mission consists of three phases: initialization, execution, and cleanup. At the initialization phase the mission memory is created by the SCJ implementation and all handlers and data created during initialization is by default allocated in the mission memory. Data shared between handlers needs to be allocated in mission memory. Data shared between missions needs to be allocated in immortal memory.

The SCJ application is started on the transition to the execution phase. Temporal objects are allocated in the handlers private memory. After the cleanup phase, the mission memory is cleared and a new mission can be started. Our example is a single mission example. There is no need for mission sequences. We assume that most safety-critical applications are single mission applications.

A SCJ application is represented by a class that implements Safelet and at least one class that extends Mission. Simple program, consisting of a single mission, can use one class that extends Mission and implements Safelet.

Three different memory areas are available for an SCJ application: immortal memory, mission memory, and anonymous private scope memories. Immortal memory is the same as immortal memory in the RTSJ. It contains static fields, objects that are created during class initialization, and application data that needs to be preserved over mission boundaries. Mission memory, as the name implies, exists as long as a mission is active (in any of the three phases).

Each handler has an initial private memory, which is cleared after a release has finished. Therefore, no data can survive individual releases. To allow more flexibility with in the release of a handler, the handler can enter nested private memories. Different enters in nested private memories might be sized different. Therefore, the implementation of the private memory area needs to be able to resize a memory area [13].

2.2 The Java Processor JOP

The Java processor JOP [12] is an implementation of the JVM in hardware. The bytecodes of the JVM are the native instructions of the processor. JOP has been optimized to be time-predictable, while still performing well. The execution pipeline is a 4 stage in-order pipeline. The execution time of individual bytecodes are independent of each others. These properties enabled building worst-case execution time (WCET) analysis tools for Java that target JOP [15, 4].

The execution time of individual bytecodes, the instruction set of the JVM, is known cycle accurate. Most bytecodes have a constant execution, which means that the WCET and the best-case execution time (BCET) are equal. Variability in the execution time mainly results from the instruction cache. To simplify WCET analysis of the instruction cache, JOP caches whole methods [10]. Therefore, cache misses can only happen on method invocations and on a return from a method. This method cache also simplifies the timing model for WCET analysis.

The implementation in an FPGA also allows to add hardware devices in the same FPGA. Within the RepRap project we added the motor driver interface and an ADC interface for the temperature sensor. Those I/O devices are represented as hardware objects in Java [14]. In a standard SCJ or RTSJ implementation access to the I/O devices would be via *raw memory* interfaces.

In this project we use the SCJ implementation on top of JOP [16]. Only the I/O devices are platform specific (as usual in an embedded system), but the application code shall be standard SCJ. We intend to verify this by building a self-contained benchmark, where I/O devices are simulated. With this benchmark we can evaluate different SCJ implementations.

2.3 SCJ Use Cases

While there is a lack of use cases for SCJ, some work has been done. In [6] the CD*x* benchmark is ported to the SCJ level 0 compliant oSCJ framework. The benchmark is used to evaluate performance of the framework compared to the equivalent C code. In this paper we move to a level 1 compliant framework and focus more on the problems a developer may encounter.

In [17] a framework called PERC Pico is described, which slightly diverges from SCJ. The paper describes the porting of it to two ARINC 653 compliant operating systems. A simplified flight warning system developed by THALES is used to validate the porting. It is desirable to implement real safety-critical use cases in SCJ, such as a flight warning system, however they are not readily accessible. The RepRap is freely available and, depending on the implementation, provides a real-time use case capable of testing a large part of the SCJ specification.

A very recent use case, a cardiac pacemaker, compares the implementation in Ravenscar Ada [2] against one in SCJ [18]. The conclusion of this experiment is that SCJ is missing a watchdog timer. They propose (and expect) that a future version of SCJ will support one-shot timers. In our use case all tasks are time triggered, or the input of a serial port is periodically polled. Therefore, we have not missed a one-shot timer.

2.4 The RepRap Project

A RepRap is a desktop printer capable of creating 3-dimensional (3D) objects in plastic [8]. Some of the components are printable, meaning that the RepRap is partially self-replicable. In a standard setup the host software, running on a computer, reads a 3D drawing, such as an STL file from a CAD application, and sends printing instructions, called G-codes, to the RepRap. The RepRap firmware interprets the instructions and ensures that the printing head is moved to the instructed coordinates while heating and extruding plastic.

The RepRap does not fall under the normal definition of a safety critical system where human lives are in danger [20], however it is still a useful SCJ use case. The firmware has to control the stepper-motors that move the printing head (extruder) and extrude plastic, read the end-stop sensors, and also control and measure the tem-

perature of the head. The temperature has to be high before the plastic is optimally extruded, e.g. as high as 230°C for PLA [7], but it should not overshoot, as too high temperatures might destroy the plastic and components. It is therefore reasonable to implement the firmware as a SCJ application, where all measurements and controls are done within well defined timing boundaries.

3. IMPLEMENTATION

The implementation is comprised of the host software, an FPGA, an interface board and the RepRap hardware, connected as shown in Figure 1.

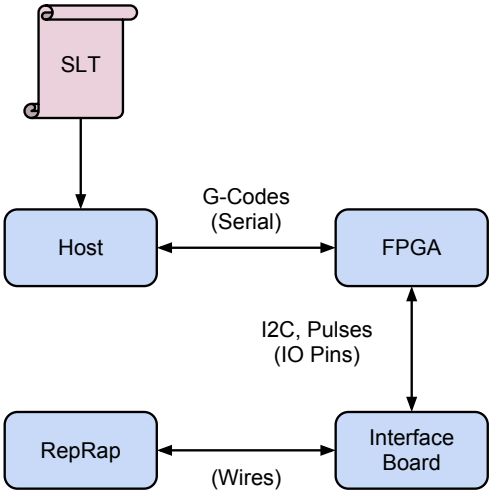

Figure 1: Hardware Layout

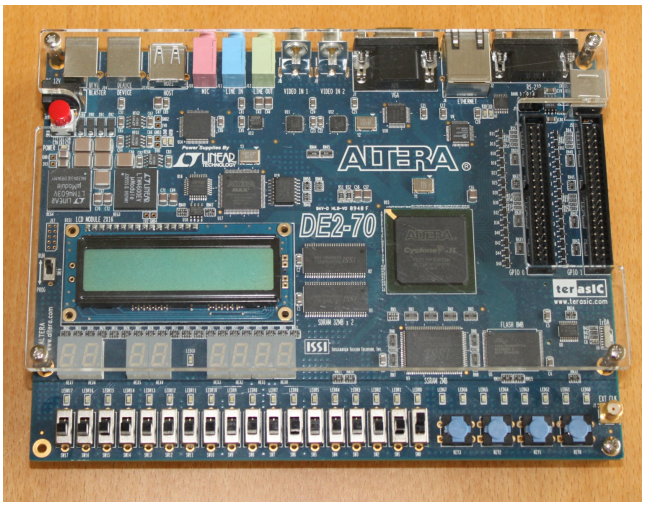

Figure 2: FPGA

The host software is the official Java RepRap host software. It is capable of slicing 3D drawings into G-codes and send the codes to the firmware over a serial line. The communication between the firmware and the host software uses a simple protocol: At startup the firmware is waiting for G-codes. Each code received from the host must be confirmed by sending an acknowledgement back to the host. Since the host does not send any further codes until it receives an acknowledgement, the firmware can control the communication flow and avoid overflowing the serial buffers. If checksums are used in the communication a corrupted code triggers a resend request by the firmware to the host, which resends the requested code line.

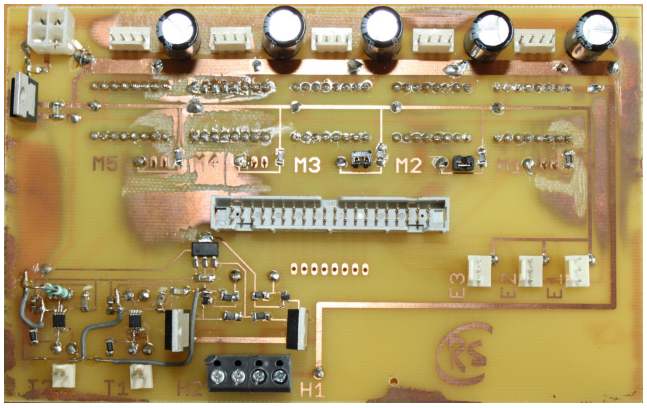

Figure 3: Interface Board

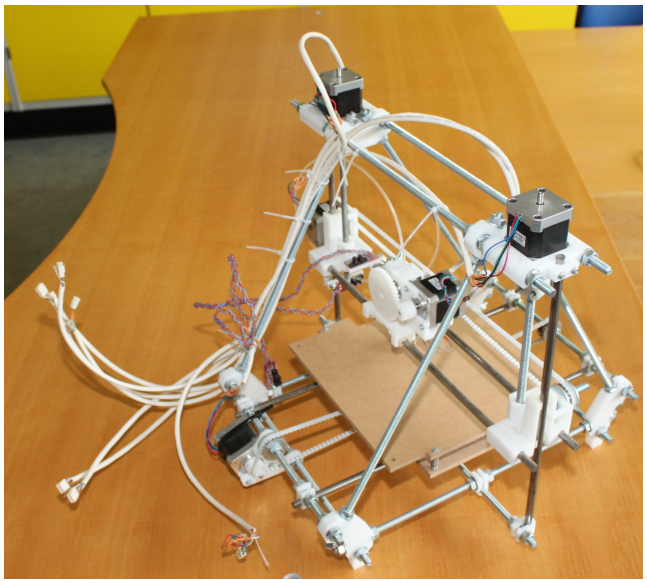

Figure 4: RepRap

3.1 Hardware

The FPGA is used for the RepRap firmware and is therefore connected to the host. We use the Altera DE2-70 board shown in Figure 2. To drive the motors, temperature and sensors of the RepRap, we have built the interface board shown in Figure 3. The RepRap itself is shown in Figure 4. Figure 5 shows the complete RepRap system consisting of the robot, a power supply, and FPGA board and the motor interface. The host PC is not shown.

The FPGA is configured with the implementation of JOP. JOP is loaded with a SCJ level 1 compliant framework, on top of which the firmware is running (see Figure 6).

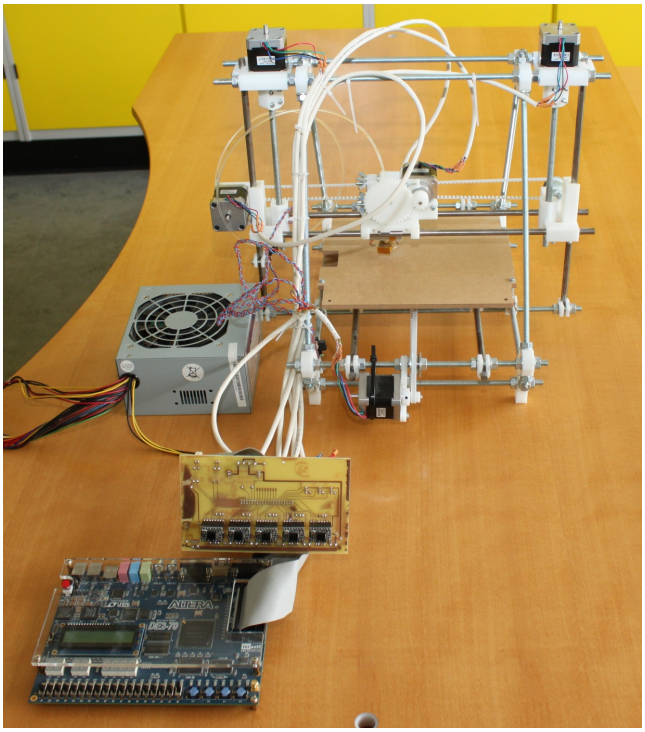

Figure 5: RepRap setup without the host

3.2 Software

The RepRap firmware is implemented from scratch. It is designed as a SCJ level 1 single mission application and consists of 4 PeriodicEventHandlers (PEHs), as shown in Figure 7: HostController, CommandParser, CommandController and RepRapController. The idea is to have a pipeline where a G-code is received, parsed and executed. For each type of G-code there is a respective command class who's instances represent received valid G-codes. Each command class has an object pool with at least one instance of itself that resides in mission memory. Instead of creating a new command object when the G-code has been parsed, the respective object is retrieved from mission memory. This allows the object reference to be passed between PEHs. Command classes that have an object pool represent G-codes that are buffered, i.e., the firmware should send an acknowledge to the host as soon as the code is verified, and not wait until it has executed like the other codes.

The HostController represents the receiver and handles the serial communication with the host computer. As PEHs have guaranteed response times, no characters are left unprocessed as long as the host does not send characters faster then the agreed upon baud rate. The controller strips the received G-code string of any comments before marking it as ready.

The CommandParser represents the parser. It polls the HostController for a ready code string and, if ready, copies it. The string is then parsed. If the string is valid the respective command object is pulled out of its pool, and enqueued in the CommandController. The HostController does not save further characters until the CommandParser is finished with the last command, however the functionality is split into two PEHs to decrease the WCET, allowing greater throughput.

The CommandController is a part of the command object execution. It has a command object queue which is traversed in FIFO

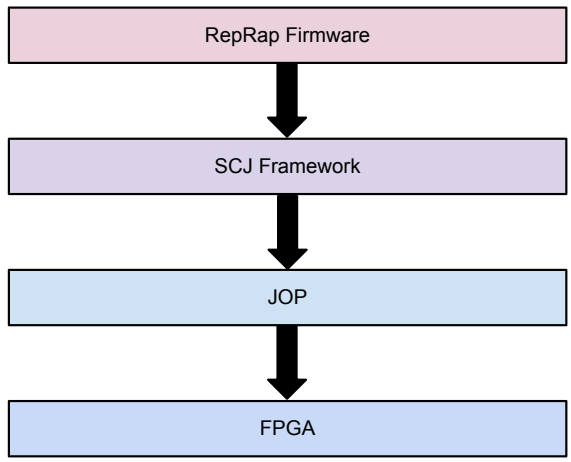

Figure 6: Firmware layers

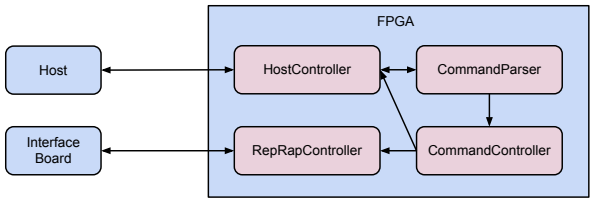

Figure 7: PeriodicEventHandler communication

order. When an object is pulled out of the queue the CommandController calls the command's execute method, which performs the command. If the command needs to interact with the RepRap or host as part of the execution, it calls the necessary methods in the RepRapController or HostController. After a command has executed it is returned to its respective command pool.

The RepRapController handles all communication with the RepRap hardware. It ensures that all axes move toward their target points. It also reads all necessary measurements, such as temperature and end-stop signals. The RepRapController uses SimpCon [11] to map a"hardware object" to the input/output pins on the FPGA. The pins are connected to the interface board, which in turn is connected to all the RepRap hardware.

4. EVALUATION

From a programmer's perspective there can be several reasons to want to use Java technology for safety-critical applications, e.g. avoid/reduce memory and timing management, overall ease of use, used to Java programming, etc. SCJ aims to bring Java to safety-critical systems. However, SCJ restricts Java in several areas, so the question is whether the difference between SCJ and Java alienates Java, and other, developers.

4.1 SCJ Programming Experience

Being used to Java threads, programming with PEHs is similar, since the application can be functionally distributed across PEHs in the same manner as threads. One difference is that PEHs are cre-

ated at mission start-up and are periodic, which might be a problem where one would create threads on the fly in Java, such as when processing large data sets. However this is not necessarily a problem for safety-critical applications. In Java, to create a periodic task, one would typically create a thread and override the run method, as shown in Figure 8. In SCJ this done by overriding the handleAsyncEvent method, as shown in Figure 9.

```
@Override
public void run()
{
        boolean loop = true;
        while(loop)
        {
                /*
                 * Do work
                 */
                try
                {
                        wait(10);
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }
        }
}
```

Figure 8: Periodic Java thread

```
new PeriodicEventHandler(new PriorityParameters(1),
        new PeriodicParameters(null, new RelativeTime(10,0)),
        new StorageParameters(50, null, 0, 0), 40)
{

        @Override
        public void handleAsyncEvent()
        {
                /*
                 * Do work
                 */
        }
};
```

Figure 9: PeriodicEventHandler

Whilst it is an improvement knowing that the handleAsyncEvent method is guaranteed to be called in the specified period, the storage parameters are troublesome. It is possible to count the number of objects and primitives used in a memory scope, however the size of objects is platform dependent, so unless the programmer has a thorough knowledge of the platform, the correct value for the parameters is not clear. Development would be made easier if a tool was available to statically analyse the maximum memory usage of the handleAsyncEvent method and thereby get a PEH's necessary storage parameters.

Java programmers are used to the JVM handling memory management, with a garbage collector taking care of object deallocation. In SCJ there is no garbage collector. Instead objects are created in memory scopes and deallocated when the scope is exited. Since each PEH has a private scope this means that all objects created in one execution of the PEH are deallocated when the PEH is released. It is therefore not a problem to create temporary objects during execution, similarly to using a garbage collector. However, the objects cannot be referenced from anywhere outside the execution. This is a major difference to Java, where the programmer can freely pass references between threads. If one PEH generates a result object that is needed in another PEH, the primitive values of the object have to be copied to a shared object in either mission or immortal memory. This is why the SCJ RepRap firmware uses object pools in mission memory. Initially the pools were created in immortal memory, however the PEHs have to be created in mission memory and some Command objects need references to some of the PEHs, which results in illegal referencing. Using immortal memory can therefor be problematic. It is also not evident when an object is initialized in immortal memory, e.g. is it only after the Safelet is created or also when using static initializers? Having an initialization method in the safelet similar to Mission.initialize would be easier.

The referencing problem also affects the use of library code which creates new objects. For example StringBuilder automatically creates a new array in the append method if its buffer is full. If the StringBuilder was created in mission memory and append is called from a PEH's execution context, this results in an illegal reference. If the StringBuilder is created in the execution context, it does not live after the PEH's release. It is therefore not usable in HostController, where a command string can be built over several executions. The absence of a garbage collector therefore requires more effort from programmers. To capture faulty references during development we used the on-line scope checker shipped with JOP. Optimally any wrong references can be caught with static analysis instead [3].

4.2 Schedulability Tests

Similar to the problem with the storage parameters, the programmer cannot be sure if the PEH's priority and periodic parameters present a feasible schedule until the WCET of each PEH is found. Since the execution time of each line of code is platform dependent, the analysis must cover the application, the framework, and the platform. For JOP the WCET analysis tool has this capability. As it is the handleAsyncEvent method that is called at each PEH's execution, the WCET tool is used on this method for each PEH. The results are shown in Table 1. The priorities are in decreasing order, i.e. the PEH with the highest priority is the RepRapController. The tool presents the results in cycles, which are converted to execution time (in ms) when the clock frequency (60 MHz in our case) is known. The tool is not able to include the maximum time a PEH can be blocked due to a lower priority thread taking the same resource. This is found by manually tracing a PEH's execution and finding each synchronization lock. Each other block of code that uses the same lock and is called by another PEH is analysed with the WCET tool. The largest one is added to the table.

To check if the PEHs are schedulable the utilization test from [21, p. 137] is used in Figure 10, which takes into account potential blocking times. All inequalities are satisfied so the PEHs are schedulable. Note that in this analysis the switching time is not included, as context switching code is not reachable from the handleAsyncEvent methods. Analysing the system's entry method should result in a full system analysis, however this requires a lot of changes. To achieve the current analysis it was necessary to modify some framework libraries, since these were not analysable or resulted in WCETs that were far too high.

PEH	Priority	Period (ms)	WCET (ms)	Maximum time potentially blocked (ms)
RepRapController	4	1	0,0718667	0,0016
HostController	3	1	0,42593	0,1529833
CommandController	2	20	0,9138333	0,1529833
CommandParser	1	20	3,5771167	0,1529833

Table 1: WCET for the PeriodicEventHandlers

RepRapController

$$\frac{0{,}0718667}{1} + \frac{0{,}0016}{1} \leq 1 * (2^{\frac{1}{1}} - 1) \Leftrightarrow 0{,}0734667 \leq 1$$

HostController

$$\frac{0{,}0718667}{1} + \frac{0{,}42593}{1} + \frac{0{,}1529833}{1} \leq 2 * (2^{\frac{1}{2}} - 1) \Leftrightarrow 0{,}65078 \leq 0{,}8284271$$

CommandController

$$\frac{0{,}0718667}{1} + \frac{0{,}42593}{1} + \frac{0{,}9138333}{20} + \frac{0{,}1529833}{20} \leq 3 * (2^{\frac{1}{3}} - 1) \Leftrightarrow 0{,}55113753 \leq 0{,}7797631$$

CommandParser

$$\frac{0{,}0718667}{1} + \frac{0{,}42593}{1} + \frac{0{,}9138333}{20} + \frac{3{,}5771167}{20} + \frac{0{,}1529833}{20} \leq 4 * (2^{\frac{1}{4}} - 1) \Leftrightarrow 0{,}729993365 \leq 0{,}7568285$$

Figure 10: Utilization test

To produce an analysable application it is necessary to program while keeping in mind schedulability, e.g. the time PEH1 blocks PEH2 is relevant to PEH2's utilization test, which is why blocking times must be diminished. Figure 11 shows a design with this intent. Instead of locking the entire handleAsyncEvent method, only a small part is synchronized with the getInputStatus and setInputStatus methods. Although the locking could be done with synchronization blocks, thereby avoid the overhead of method calls, they are not available according to the SCJ specification. It is not evident why this is so.

The //@WCA loop=16 line acts as an annotation for the WCET tool and indicates that the loop will run a maximum of 16 times. This annotation is necessary for most non-trivial loops, as the tool is otherwise unable to determine the execution time. This is relevant to system classes such as String that are designed for strings of almost arbitrary size. As the tool is not able to see if an application only uses strings with a fixed length, the maximum length must be manually added to the loops. Another problem is while loops. From the tool's perspective they are unbounded. It is therefore necessary to annotate or avoid them. This becomes especially troublesome when the framework itself uses them for blocking reads/writes, e.g. System.in.read(). Reading and writing to streams needs to be organized such that a PEH can check availability before reading/writing. If characters on the stream are not available, the PEH can be released allowing other PEHs to execute.

4.3 Safety-Critical Java vs. C

There are several RepRap firmwares written in either C or C++ for micro-controllers. One of them is the Teacup firmware, a rewrite of the original FiveD firmware, written entirely in C [9]. It does not use any real-time framework, so there are no guarantees for execution times. The application handles the timing itself, such as setting up and deactivating timer interrupts. The SCJ firmware lets the framework handle timing, which seems simpler for the programmer. The major difference between the two firmwares is the use of two very different programming languages, but basic advantages/disadvantages in using an object oriented languages will not be discussed here. Instead, we highlight the performance cost of using SCJ based firmware, as opposed to using a minimalistic firmware such as Teacup, in Table 2. The firmware size includes all framework and application code. The size difference is not as bad as expected, especially considering that the SCJ firmware size can be further reduced by using an optimization tool that removes unused methods from classes. The steps indicate how many steps the firmware can move the motors per second. The frequency shows the frequency of the CPU. Teacup has a clear advantage even at a lower frequency. The SCJ firmware steps are based on the period of the RepRapController, which is limited by the framework. The RepRapController WCET analysis and utilization test shows that it can almost run with a period of 0.1 ms. This is certainly achievable by increasing the number of cores in JOP from 1 to 4 so that the utilization factor of the RepRapController doesn't affect the other PEHs. However this is not as interesting in the schedulability analysis, since the PEHs would simply run in parallel. In any case, the Teacup stepping performance is roughly 3 times better than the best SCJ firmware performance. If the RepRapController's responsibilities are delegated to hardware components, such as writing the stepping controls in VHDL, this gap can be reduced and even reversed. The question is therefore whether SCJ is better suited for high-level computing, whereas low-level and performance intensive computing is delegated to other systems/components.

5. SOURCE ACCESS

The SCJ RepRap implementation is open source and hosted at `https://github.com/torurstrom/jop`. The basic application is according to the current SCJ specification and not dependent on the JOP platform. However, access to I/O devices is always platform dependent. And in the current form the project currently uses SimpCon based hardware objects and is therefore dependent on JOP. It is hosted as a JOP fork, with the RepRap Java code located in `https://github.com/torurstrom/jop/tree/master/java/target/src/rtapi/org/reprap`.

Work is currently being done to create a version that removes the I/O dependency of JOP (and the FPGA board). The input sensors (and the UART) and the actuators will be simulated. This will allow the SCJ RepRap project to act as general test-case for other SCJ frameworks.

6. CONCLUSION

The RepRap implementation reveals several points of interest in safety-critical Java that affect both vendors and developers. Tools should be available to analyse WCET and maximum memory usage of the applications. Platforms, frameworks, and libraries must be modified so the tools are able to perform the analysis. This means that code should not use unbounded loops or otherwise block Pe-

	SCJ firmware	Teacup
Firmware size (KB)	79	∼32
Maximum steps per second	500 @ 60 MHz	15570 @ 20 MHz

Table 2: Firmware performance costs

riodicEventHandlers indefinitely. Java programmers will have familiarity with safety-critical Java but must learn to code with more responsibility. The lack of garbage-collection means that the programmer has to be careful where created objects are referenced. The programmer must have a deeper knowledge of the library code to ensure that objects aren't created and wrongly referenced during execution. This can be made easier with static analysis tools that check possible references. A safety-critical Java implemented firmware has a much larger execution overhead than an optimized C based firmware, which could indicate that safety-critical Java applications are better suited for high-level computing.

7. ACKNOWLEDGMENTS

This work is part of the project "Certifiable Java for Embedded Systems" (CJ4ES) and received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159.

8. REFERENCES

[1] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[2] A. Burns, B. Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, pages 263–275. Springer-Verlag, 1998.

[3] A. E. Dalsgaard, R. R. Hansen, and M. Schoeberl. Private memory allocation analysis for safety-critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[4] T. Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.

[5] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety-critical Java technology specification, public draft, 2011.

[6] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 95–101, New York, NY, USA, 2010. ACM.

[7] RepRap Project. Polylactic acid. `http://reprap.org/wiki/PLA`, June 2012.

[8] RepRap Project. The reprap project website. `http://reprap.org/wiki/Main_Page`, June 2012.

[9] RepRap Project. Teacup firmware. `http://reprap.org/wiki/Teacup_Firmware`, June 2012.

[10] M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[11] M. Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[12] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[13] M. Schoeberl. Memory management for safety-critical Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, pages 47–53, York, UK, September 2011. ACM.

[14] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, 10(4):42:1–42:40, November 2011.

[15] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[16] M. Schoeberl and J. R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[17] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy. Use of perc pico in the aida avionics platform. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 169–178, New York, NY, USA, 2009. ACM.

[18] N. K. Singh, A. Wellings, and A. Cavalcanti. The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[19] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[20] The Free On-line Dictionary of Computing. safety-critical system. `http://encyclopedia2.thefreedictionary.com/safety-critical+system`, July 2012.

[21] A. Van Tilborg, G. Koob, and U. S. O. of Naval Research. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Kluwer international series in engineering and computer science: Real-time systems. Kluwer Academic Publishers, 1991.

```
synchronized private void setInputStatus(boolean status)
{
	inputStatus = status;
}

synchronized private boolean getInputStatus()
{
	return inputStatus;
}

@Override
public void handleAsyncEvent()
{
	char[] output = outputBuffer.getChars(16);
	for (int i = 0; i < output.length; i++) //@WCA loop = 16
	{
		SP.write(output[i]);
	}
	//Input buffer is still full so do nothing
	if (getInputStatus())
	{
		return;
	}
	for (int i = 0; i < 16; i++) //@WCA loop = 16
	{
		char character;
		if (!SP.rxFull())
		{
			//No input
			return;
		}
		character = (char)SP.read();
		if (character == ';')
		{
			comment = true;
		}
		else if (character == '\n')
		{
			comment = false;
			if (inputCount > 0)
			{
				setInputStatus(true);
				return;
			}
		}
		else if (!comment) //Ignore comments
		{
			if (inputBuffer.add(character))
			{
				inputCount++;
			}
		}
	}
}
```

Figure 11: Excerpt of HostController.java