A Formal Model for the SCJ Level 2 Paradigm

Matt Luckcuck

Department of Computer Science, University of York,
York, YO10 5GH, UK

ml881@york.ac.uk

1 Introduction

Safety-Critical Java (SCJ) [12] is the product of an international effort to provide a Java-based language for applications that must be certified using the avionics standard ED-12C/DO-178C [4] at Level A, which defines software that would prevent continuous safe flight and landing in the event of failure. To aid certification, SCJ is organised into three compliance levels that ascend in complexity from Level 0 to Level 2.

The SCJ standard does not cover verification techniques. Verification has been addressed and results obtained for Level 1, but not Level 2. We focus on providing verification for SCJ Level 2 programs. SCJ Level 2 has received little attention from practitioners and researchers, even its intended uses are unclear from the standard, and in [14] we present the first examination of the uses of its features and present example applications for Level 2.

The SCJ API ensures a hierarchical program structure and supports several real-time execution abstractions. SCJ programs are centred around missions, which each contain several real-time tasks that perform a particular function. Uniquely for SCJ, a Level 2 program may have many concurrent missions, which allows Level 2 programs to adopt more complex structures than those at the other two compliance levels. Tasks from any active mission may preempt each other, based on their priorities; there is no assumption that tasks from a particular mission have precedence. Level 2 tasks may use all four SCJ execution patterns: periodic, aperiodic, run-once after a time offset, and run-to-completion. Finally, Level 2 programs may use the familiar Java suspension features.

Our work makes three contributions to the state of the art on verification of SCJ Level 2 programs. Firstly, we model the SCJ Level 2 paradigm using the state-rich process algebra *Circus* [15]. Our model can be used to identify potential errors in the programs that it represents. *Circus* combines Z [10] for modelling state, CSP [7] for modelling behaviour, and Morgan's refinement calculus [9]. A *Circus* program is organised around processes, which may have a state component to hold variables and actions to perform behaviours. Communication between processes is achieved via CSP channels. Our model uses features from other members of the *Circus* family. *OhCircus* [2] introduces a

notion of object orientation and inheritance, and we use features from *Circus* Time [13] to specify time budgets and deadlines.

We provide a mechanised translation strategy that enables the automatic transformation of SCJ Level 2 source code into faithful *Circus* models. As a secondary objective, we also provide a strategy for translating our models back into SCJ programs.

Our second contribution rests on our model capturing the API separately from the program-specific behaviour. Because of this separation we can show that the SCJ API does not introduce undesirable behaviour, such as deadlock or livelock, under the circumstances that we capture.

There is a body of previous work involving *Circus* and SCJ, including a model of SCJ Level 1 [16] – upon which our work is based. A refinement strategy [3] has been devised to transform abstract specifications into concrete specifications that capture the SCJ paradigm. This refinement strategy facilitates the development of SCJ programs that are correct by construction.

Our final contribution is that our model provides the refinement strategy [3] with a target for models of SCJ Level 2. While this refinement strategy is out of scope for our work, our model enables it to consider Level 2 programs.

Previous approaches to ensuring the safety of SCJ programs include using annotations to provide run-time checks [11] or to specify checkable program constraints [6]. RSJ [8] is a tool that explores all possible schedulings of the threads within an SCJ program to check for scheduling-dependent errors. However, none of these techniques are specifically aimed at Level 2.

ABS [1] is an executable specification language that has similar capabilities to *Circus*. Both ABS and *Circus* have an object-oriented model that is similar to Java's and capture concurrency. However, *Circus* contains a notion of refinement that ABS does not. Refinement is important for our third contribution.

In the next section we describe our model of SCJ and what analysis it facilities. Finally, in Section 3 we summarise our research and contributions, and describe the further work needed to complete this research.

2 Model and Translation

We capture the paradigm of SCJ Level 2, agnostically of its implementation in Java, using two components. The framework model captures the behaviour of the API classes of SCJ and is reused for each program. Conversely, each program is represented by an application model that captures its particular behaviour.

The framework and application models both contain a process for each of the SCJ API classes. The framework processes control the program flow and hand off to their application counterparts wherever the program runs application code, including where API methods are overridden.

We capture Java exceptions but only when they indicate a misuse of the SCJ paradigm, never when they indicate a purely Java problem (such as a `null` parameter). If the program uses locking or suspension, then we capture this in extra elements added to the framework model.

The translation strategy that we are developing contains formal rules that build the specification of a given program. Our work provides the first formal semantics of SCJ Level 2. As there is nothing else formal to compare our

semantics to, we can not consider its soundness, but it will be validated using tools and case studies.

A *Circus* model checker is in development; in the mean time we translate our *Circus* model into CSP to validate our specifications using FDR3 [5]. We animate its behaviour and compare it to that described in the SCJ standard. We model check it to identify properties (such as deadlock, livelock, and non-termination) that represent program errors and SCJ-specific problems, such as Java exceptions that indicate a misuse of the SCJ paradigm. This also gives us confidence that our model of the SCJ infrastructure is correct and helps to verify the SCJ API itself, because we model it separately.

Our approach is limited to capturing the behaviour of SCJ programs. We do not capture use of resources, in particular memory usage. Further, while we capture time for the purposes of deadline detection, our models cannot be used to calculate the worst-case execution time of a program.

We have used FDR3 to show that our model of the SCJ infrastructure is free from program errors, meaning that if our specification of a program exhibits these errors, then they must arise from the application model. Further, we have translated several small example applications into our model, by hand, to show that our model can capture the SCJ Level 2 paradigm. Using FDR3 we have proved that these examples do no throw exceptions, are free from deadlock and livelock, and that they terminate.

3 Summary and Further Work

In summary, we model the paradigm of SCJ Level 2 as a combination of a framework model, that captures the SCJ API, and an application model, that captures the behaviour of the program being modelled. Our model of SCJ Level 2 contributes to both top-down development of correct SCJ programs, as a target for the refinement strategy presented in [3], and to bottom-up development of correct SCJ programs, as a tool for the identification of program errors. Further, it can be used to verify the SCJ API because we capture it separately in our model.

Our framework model and the skeleton processes for the application model are both complete. We have modelled several small example programs, to show that we can capture the common features of the SCJ Level 2 paradigm, and shown that these examples do not exhibit any undesirable properties.

The remaining work is to formalise the translation of a program into our model. Translation will then be automated using a tool that will take SCJ programs as an input and output *Circus* models. We envisage minor restrictions on the form of the SCJ programs, similar to those presented in [16]; for example, each SCJ class should be in its own file. Automatic translation not only validates our model, but also enables the verification of SCJ Level 2 programs, by allowing a simple translation to our model to enable model checking. More work on analysing our model is needed and the basic properties we can already prove will be augmented by properties that capture SCJ exceptions that indicate a misuse of the paradigm and application-specific properties.

Acknowledgements

This work is funded by the hiJaC project, backed by the EPSRC grant EP/H017461/1. We would like to thank Ana Cavalcanti, Andy Wellings, Frank Zeyda, Alan Burns, and Thomas Gibson-Robinson.

References

[1] Bubel, R., Montoya, A.F., Hähnle, R.: Analysis of Executable Software Models. In: Formal Methods for Executable Software Models, pp. 1–25. Springer (2014)

[2] Cavalcanti, A., Sampaio, A., Woodcock, J.: Unifying Classes and Processes. Software & Systems Modeling 4(3), 277–296 (2005), http://dx.doi.org/10.1007/s10270-005-0085-2

[3] Cavalcanti, A., Wellings, A., Woodcock, J., Wei, K., Zeyda, F.: Safety-Critical Java in Circus. In: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 20–29. JTRES '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2043910.2043915

[4] EUROCAE and RTCA: Software Considerations in Airborne Systems and Equipment Certification. Norm ED-12C, EUROCAE (2012)

[5] Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: Failures Divergences Refinement (FDR) Version 3 (2013), https://www.cs.ox.ac.uk/projects/fdr/

[6] Haddad, G., Hussain, F., Leavens, G.T.: The Design of SafeJML, a Specification Language for SCJ with Support for WCET Specification. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 155–163. JTRES '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1850771.1850793

[7] Hoare, C.A.R.: Communicating Sequential Processes. http://www.usingcsp.com/cspbook.pdf (2004)

[8] Kalibera, T., Parizek, P., Malohlava, M., Schoeberl, M.: Exhaustive Testing of Safety-Critical Java. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 164–174. JTRES '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1850771.1850794

[9] Morgan, C.: Programming from Specifications. Prentice-Hall, Inc. (1990)

[10] Spivey, J.M.: The Z Notation: A Reference Manual. International Series in Computer Science (1992)

[11] Tang, D., Plsek, A., Vitek, J.: Static Checking of Safety Critical Java Annotations. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 148–154. ACM, Prague, Czech Republic (2010)

[12] The Open Group: Safety-Critical Java Technology Specification. Tech. rep., The Open Group (27 December 2014)

[13] Wei, K., Woodcock, J., Cavalcanti, A.: New Circus Time. University of York, Tech. Rep., February (2012)

[14] Wellings, A., Luckcuck, M., Cavalcanti, A.: Safety-Critical Java Level 2: Motivations, Example Applications and Issues. In: Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems. pp. 48–57. JTRES '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2512989.2512991

[15] Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002:Formal Specification and Development in Z and B, Lecture Notes in Computer Science, vol. 2272, pp. 184–203. Springer Berlin Heidelberg (2002)

[16] Zeyda, F., Lalkhumsanga, L., Cavalcanti, A., Wellings, A.: Circus Models for Safety-Critical Java Programs. The Computer Journal (2013), http://comjnl.oxfordjournals.org/content/early/2013/07/02/comjnl.bxt060.abstract