A Comparison of Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems

Crystal Chang Din[1], Richard Bubel[2] and Olaf Owe[1]

[1] University of Oslo, Norway
[2] Technische Universitat Darmstädt, Germany
{crystald,olaf}@ifi.uio.no, bubel@cs.tu-darmstadt.de

Distributed systems play an essential role in society today. For example, distributed systems form the basis for critical infrastructure in different domains such as finance, medicine, aeronautics, telephony, and Internet services. It is of great importance that such systems work properly. However, quality assurance of distributed systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. It is highly challenging to test such distributed systems after deployment under different relevant conditions. These challenges motivate frameworks combining precise modeling and analysis with suitable tool support. In particular, *compositional verification systems* allow the different components to be analyzed independently from their surrounding components. Thereby, it is possible to deal with systems consisting of many components.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [15]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [2]. Concurrent objects communicating by *asynchronous method calls* have been proposed as a promising framework to combine object-orientation and distribution in a natural manner. Each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process is executing on an object at a time. Asynchronous method calls allow the caller to continue with its own activity without blocking while waiting for the reply, and a method call leads to a new process on the called object. The notion of *futures* [6, 19, 12, 20] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to wait for method results. However, futures complicate program analysis since programs become more involved compared to semantics with traditional method calls, and in particular local reasoning is a challenge. ABS [17] is a high-level imperative object-oriented modeling language, based on the concurrency and synchronization model of Creol [18]. It supports futures and concurrent objects with an asynchronous communication model suitable for loosely coupled objects in a distributed setting. In this work, we present our testing and verification tools for ABS programs.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [8, 14]. At any point in time the communication history abstractly captures the system state [10, 9]. In fact, traces are used in the semantics for full abstraction results (e.g., [16, 1]). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of possible histories, expressing safety properties [5].

In our reasoning system, we formalize object communication by an operational semantics

based on five kinds of communication events, capturing asynchronous communication, shared futures, and object creation, where each event is visible to only the object generating it. Consequently, the local histories of two different objects share no common events. For each object, a local history invariant can be derived from the class invariant by hiding the local state of the object. Modularity is achieved since history invariants can be established independently for each object, without interference, and composed at need. This results in behavioral specifications of dynamic system in an open environment. Such specifications allow objects to be specified independently of their internal implementation details, such as the internal state variables. In order to derive a global specification of a system composed of several components, one may compose the specification of different components. Global specifications can then be provided by describing the observable communication history between each component and its environment.

In this work we implement a runtime assertion checker and extend the KeY theorem prover for testing and verifying ABS programs, respectively. For runtime assertion checking the ABS interpreter is augmented by an explicit representation of the global history, reflecting all events that have occurred in the execution. And the ABS modeling language is extended with method annotations such that users can define software behavioral specification [13], i.e., invariants, preconditions, assertions and postconditions, inline with the code. We provide the ability to specify both state-based and history-based properties, which are checked during simulation. History wellformedness, i.e. the order of the events, the non-nullness of the calling objects and the characteristic of futures, is proved and need not be checked during execution. Although by using runtime assertion checking, we gain confidence in the quality of programs, correctness of the software is still not fully guaranteed for all runs. Formal verification may instead show that a program is correct by proving that the code satisfies a given specification. As formal verification tool we use and extend a variant of the KeY verification system [7], which supports ABS as target language. In particular, KeY features a semi-automatic theorem prover based on dynamic logic. The design of its Gentzen-style sequent calculus follows the symbolic execution paradigm. For the ABS formalisation in dynamic logic, we follow the approach developed in [4, 3], but use the improved history formalisation as presented in [11]. The characteristic feature of the calculus is that it achieves to stay in a sequential setting while reasoning about properties of concurrent and distributed systems.

ABS runtime assertion checking and theorem proving of ABS programs in KeY are illustrated via two examples: a fair version of the reader/writer example and a publisher/subscriber example. The first example shows how we verify the class implementation by relating the objects state with the communication history. The second example shows how we achieve compositional reasoning by proving the order of the local history events for each object. Along these two examples, we evaluate and compare both approaches with respect to their scope and ease of application. In particular, we investigate their strengths and weaknesses concerning the different properties. We give recommendations on which approach is suitable for which purpose as well as the implied costs and benefits of each approach. Finally, we identify areas where improvements are needed and provide directions of future research.

References

[1] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. Journal of Logic and Algebraic Programming, 78(7):491–518, 2009.

[2] A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. Theoretical Computer Science, 389(3):341 – 410, 2007.

[3] W. Ahrendt and M. Dylla. A verification system for distributed objects with asynchronous method calls. In K. Breitman and A. Cavalcanti, editors, *Proc. International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of Lecture Notes in Computer Science, pages 387–406. Springer, 2009.

[4] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. Science of Computer Programming, 77(12):1289–1309, Oct. 2012.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[6] H. G. Baker Jr. and C. Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of Lecture Notes in Computer Science. Springer, 2007.

[8] M. Broy and K. Stølen. *Specification and Development of Interactive Systems.* Monographs in Computer Science. Springer, 2001.

[9] O.-J. Dahl. Object-oriented specifications. In *Research directions in object-oriented programming*, pages 561–576. MIT Press, Cambridge, MA, USA, 1987.

[10] O.-J. Dahl. *Verifiable Programming.* International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.

[11] C. C. Din, J. Dovland, and O. Owe. Compositional reasoning about shared futures. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc. International Conference on Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of Lecture Notes in Computer Science, pages 94–108. Springer, 2012.

[12] R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[13] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

[14] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.

[15] International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[16] A. S. A. Jeffrey and J. Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *Proc. European Symposium on Programming*, volume 3444 of Lecture Notes in Computer Science, pages 423–438. Springer, 2005.

[17] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of Lecture Notes in Computer Science, pages 142–164. Springer, 2011.

[18] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling, 6(1):35–58, Mar. 2007.

[19] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.

[20] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, Nov. 1986.