Java2CSP: A System for Verifying Concurrent Java Programs

Hui Shi

Bremen Institute for Safe Systems, University of Bremen
Postfach 33 04 40, 28334 Bremen, Germany
shi@informatik.uni-bremen.de

1 Introduction

More and more software systems are used in safety critical areas. In practice, the quality of such software, or its correctness, is still assured by inspecting its source code, or testing. Experience in the areas of railway control systems [5] and space station control systems has shown the formal verification of concurrent software systems is necessary and feasible, especially if one concentrates on those properties relevant for synchronisation. Verified Systems GmbH and the Bremen Institute for Safe System (BISS) have successfully analysed a fault-tolerant data management system [3] for the International Space Station (ISS) (under the contract with DaimlerChrysler Aerospace, DASA). In that project, some properties like deadlock [1] and livelock freedom [2] have been investigated based on CSP specifications and the model-checking tool FDR.

Until now there exist only a few practical tools to support the automatic analysis of existing software systems, let alone tools for translating them into a special formal method like CSP so that the associated model-checking tools can be applied. An important reason for choosing them, except for some previous experiences with CSP and FDR, is to take advantage of the HOL-CSP system [10], a semantic encoding of CSP in Higher-Order Logic within the theorem prover Isabelle [7], for performing those verification tasks that exceed the power of a model-checker. In the industrial project mentioned above, existing programs were manually transferred to CSP specifications. In this procedure, errors can be introduced, regardless how carefully it is done. In addition, this process is very inefficient and requires deep knowledge of the formal method used.

This paper presents the system Java2CSP which translates concurrent Java programs into CSP processes. Our goal is to verify automatically the synchronization behaviour, such as deadlock and livelock, of the original Java programs with the model-checking tool FDR.

2 Implementation Approach

In contrast to programming languages CSP was designed as a notation and theory for describing and analysing systems whose primary interest arises from the ways in which different components interact at the level of communication. CSP differs from concurrent programming languages in three important aspects:

– CSP was designed to be a notation for purely communicating systems: the computations internal to the component state (variables, assignments, etc.) are ignored.
– CSP supports no structured types, like strings, arrays, etc.
– The CSP notation for describing concurrency is quite simple: a set for communication events and a set of parallel operators like || for synchronised or ||| for asynchronised composition.

To interpreter concurrent Java programs in CSP the system Java2CSP makes an *abstract representation* of a Java program by removing all parts that do not influence its synchronisation behaviour, in order to reduce it to a manageable size. Then, CSP *process patterns* are introduced for simulating the concurrency concepts of Java, like shared variables, threads, and monitors.

For the verification of synchronisation behaviours of concurrent programs it is not necessary to inspect every detail of the original code, since only a subset of the programmed statements have impact on properties like deadlock and livelock freedom. Theoretically, every program with finite states can be verified by a model-checker like FDR, but it is still necessary to consider the problem of state space explosion. In fact, the main obstacle of using model checking for industrial projects is to solve this problem. It is therefore necessary to generate a CSP specification which represents an abstract version of the original Java program showing only the amount of detail which is relevant for the verification of its synchronisation behaviour. The important abstractions we applied are: *abstract object-model*, *abstract interpretation of types* and *abstract representation of algorithms*.

CSP processes which simulate those mechanisms that are supported by Java but not by CSP such as shared variables, threads and monitors, are called *process patterns*. The following CSP-process, for example, simulates a shared integer variable, where channels *readNum* and *writeNum* are for the reading and writing operation, respectively:

```
VarNum(var,val) = writeNum.var?x -> VarNum(var,x)
                  []
                  readNum.var!val -> VarNum(var,val)
```

The parameters are for the variable's name and its current value. The reading operation does not change its value, while after a writing operation the variable gets a new value.

3 The System

Java2CSP is implemented in Java. Figure 1 gives an overview of all system components. The system is divided into three phases: the bytecode analysis, the syntax-tree analysis and the code generation.

Bytecode analysis
The Java bytecode, instead of the Java source text, of a Java program has be used as the source code for the translation, since

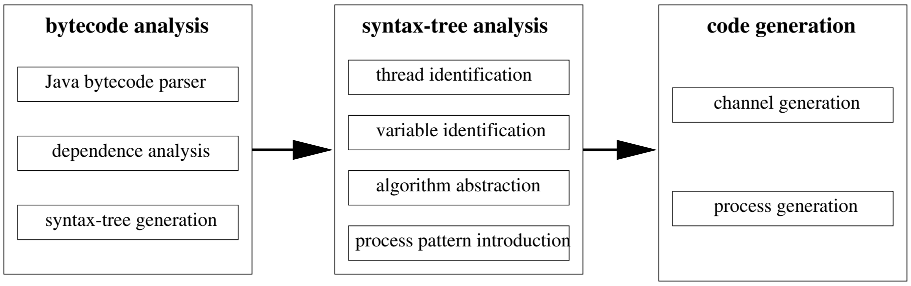

Fig. 1. The system architecture

- The Java Class File Format is a standard determined by Sun Microsystem.
- The Java Class File Format can not only be used for user applications written in Java, but for applications which can be compiled into this format.
- A Java program can include some auxiliary sources from libraries or from other users, which are sometimes only available in their bytecode representations.

The *bytecode parser* consists of two elementary parts. One reads a Java class file and writes the class information from a code segment into a file and one initiates the parsing process and resolves the dependency relation between classes (*dependence analysis*). The generated constructs maintain a shallow branch hierarchy and use only references as a structure. With such information it is difficult to make lower level analysis later. For this reason the parsed information will be transformed into a syntax-tree at an abstract level by the *syntax-tree generation*.

Syntax-tree analysis

For the evaluation of concurrent Java programs it is important to *identify threads* which can run concurrently, and to *identify variables* which influence the behaviour of threads in different ways. A *structure variables*, for example, influences the control flow of some threads; or a *synchronization variable* called by *wait*- and *notify*-method can change the state of some threads.

A program contains not only code to control its concurrent and synchronous behaviour, it may contain parts which take over the functional tasks and have no influence on its control flow. It is not necessary to translate them into CSP specifications for the verification of synchronization properties of the original program. *Abstracting algorithms* of a Java program from such codes is another important step for generating compact CSP specifications and for efficiently model-checking.

Code generation

The code-generation phase converts the above generated syntax-trees with all necessary information into a CSP specification. Such a CSP specification consists usually of a set of channel definitions, a set of auxiliary processes and a main process.

Within a generated specification the main process specifies always the main-method of the original Java program, and all in the Java program created threads will be translated into processes, as well. The whole system is composed of all these single generated processes which synchronise with process patterns specifying shared variables, threads and monitors, etc.

Figure 2 shows a screenshot of Java2CSP's simple graphical user interface. On the left are three windows showing the original java file (*Source File*), the corresponding CSP specification (*Output File*) and the system output (*Java2CSP Output*), respectively. The generated CSP specification can be saved in a FDR file using the *Save* button. Together with the specification two assertions will always be generated for the model-checker FDR to prove the deadlock and livelock behaviours of the original Java program.

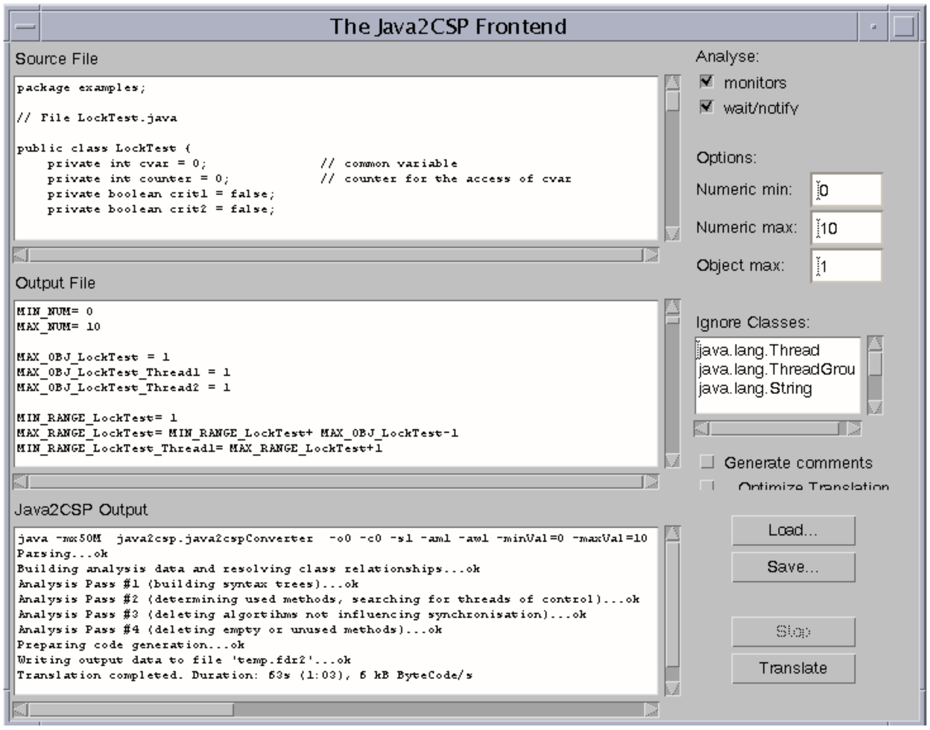

Fig. 2. The Graphical User Interface of Java2CSP

4 Conclusion

When we began with this work we thought the most difficult part of the system should be in the area of compiler construction, because the translation of Java bytecode to efficient structures for the analysis is in no means trivial. In reality, it turned out, however, that the representation of Java programs as CSP specifications is more difficult. For this purpose some abstraction methods have been implemented, such as the abstract representation of Java object-model, of Java types and algorithms. A set of CSP process patterns have been developed as well, which specify the general behaviour of a set of Java mechanisms, like shared variables, threads and monitors. Both of them are the most important parts of the system.

At present the system Java2CSP is in an experimental stage. The representation of floats and strings has not been implemented, nor the timed synchronisation. Although it can translate a lot of concurrent Java programs, like any other compiler output format, the output of Java2CSP is again the input for FDR, the improvement of the quality of the output format with respect to the readability and the recoverability is also an indispensable effort to make the system practical useful for large Java programs.

References

1. B. Buth, M. Kouvaras, J. Peleska and H. Shi: Deadlock analysis for a fault-tolerant system. In M. Johnson (ed.), *Algebraic Methodology and Software Technology. Proceedings of the AMAST'97.* LNCS 1394, (1997).
2. B. Buth, J. Peleska and H. Shi: Livelock analysis for a fault-tolerant system. In A. M. Haeberer (Ed.): *Algebraic Methodology and Software Technology. Proceedings of the AMAST'98.* LNCS 1548, (1998).
3. DaimlerChrysler Aerospace: DMS-R FTC Detailed Design Document Volume 3 (FML Software).
4. Formal Systems: *Failures Divergence Refinement FDR2.* FDR2 User Manual. Formal Systems (Europe) Lts (1998).
5. A. Haxthausen, J. Peleska: Formal Development and Verification of a Distributed Railway Control System. In *Proceedings of the 1st FMERail Workshop*, (1998).
6. C. A. R. Hoare: *Communicating Sequential Processes.* Prentice-Hall International (1985).
7. L. C. Paulson. *Isabelle - A Generic Theorem Prover.* LNCS 828, Springer Verlag, (1994).
8. A. W. Roscoe: *The Theory and Practice of Concurrency.* The Prentice-Hall International (1998).
9. H. Shi, J. Peleska and M. Kouvaras: Combining Methods for the Analysis of a Fault-Tolerant System. In B. Werner (ed.) *Proceedings of 1999 Pacific Rim International Symposium on Dependable Computing.* IEEE Computer Society, (1999).
10. H. Tej and B. Wolff: A Corrected Failures-Divergence Model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas (eds.) *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods.* LNCS 1313, Springer Verlag, (1997).