Run-Time Checking Multi-threaded Java Programs

Frank S. de Boer[1,3](✉) and Stijn de Gouw[1,2]

[1] CWI, Amsterdam, The Netherlands
[2] SDL, Amsterdam, The Netherlands
[3] Leiden University, Leiden, The Netherlands
f.s.de.boer@cwi.nl

Abstract. Assertion checking traditionally focused on state-based properties. In a multi-threaded environment, approaches based on shared-state require complex locking mechanisms to ensure that specifications are checked atomically (in the same state). In addition to this increased complexity, locks also negatively affect performance.

In this paper, we extend both the underlying theory and the practical implementation of SAGA, a run-time checker for single-threaded Java programs, to multi-threading, while avoiding locks.

1 Introduction

Runtime assertion checking is an important practical method for finding bugs. However, the scope of run-time assertion checking is restricted mainly to sequential programs. The main problem of run-time assertion checking of parallel shared-variable programs in general is because of interference. Take for example the very simple statement `assert x==x;`, where x is a field of an object. If after retrieving the value of the first occurrence of x *another* thread modifies x then the assertion may evaluate to false! To prevent this whenever an assertion is checked, the entire parallel execution has in principle to be "frozen". This thus requires control of the underlying execution platform and will in most cases give rise to severe loss in performance.

In [5] we enhance run-time assertion checking with attribute grammars [9] for describing properties of *histories*, e.g., sequences of method calls and returns. This supports strict programming to interfaces because it allows for interface specifications abstracting from the state as represented by the program variables.

The main contribution of this paper is an extension to multi-threaded Java programs which avoids in a natural manner interference problems. As an example of the expressivity and generality of our approach, we show how to detect at runtime deadlocks in multithreaded Java programs.

Related Work. To the best of our knowledge, our extension of run-time assertion checking to multi-threaded programs is the first solution which supports interface specs and does not require fine-grained control of the JVM execution platform to

This research is partly funded by the EU project FP7-610582 Envisage.

© Springer-Verlag Berlin Heidelberg 2016
R.M. Freivalds et al. (Eds.): SOFSEM 2016, LNCS 9587, pp. 217–228, 2016.
DOI: 10.1007/978-3-662-49192-8_18

prevent interference in assertion checking as in the purely state-based approaches of [1,7]. Other state-based approaches like [2,15] require complex atomicity specifications and locking mechanisms. These latter two approaches further require substantial extensions both of the specification language and corresponding tool support for run-time verification.

Besides run-time assertion checking, there are many other tools for run-time verification of multi-threaded programs. In [16] also an automated code instrumentation technique is introduced for the generation of counter-examples of a given property. These counter-examples are given in terms of partial orders of events which record particular modes of access to the state. This state-based approach consequently does not support a design by contract methodology. The run-time verification of multi-threaded programs as described in [10] affects the running system by blocking those threads which would violate the property. Other approaches focus on specific classes of properties, like data-races [3], deadlock [13] and restricted protocol-properties [6].

In [12] a process algebraic approach based on an extension of CSP is introduced which supports a strictly separate run-time verification of properties of data and protocols. In contrast, our approach builts on well-established parser technology and integrates the run-time verification of both data- and protocol-oriented properties of multi-threaded programs.

2 Extending the Framework

Our approach to multi-threading is based on the following perspectives:

Thread view: here we specify the behavior of each thread in isolation.
Object view: here we specify the behavior of objects individually.
Global view: here we specify global properties of a program.

All of the above views can be supported by a single formalism: attribute grammars extended with assertions, but the interpretation of the grammars differs between the various perspectives. We first discuss the required extensions to communication views and grammars. We then illustrate each of the above views with a running 'dining philosophers' example. The behavior of the philosophers is specified by a corresponding thread class. The interfaces of the classes defining the resources, i.e., the forks and the pasta are defined in Fig. 1.

```
interface Fork {
  void get();
  void release();
}

interface Pasta {
    void eat();
}
```

Fig. 1. Interfaces forks and pasta

2.1 Multi-threaded Events

A communication view (as defined in [5]) is a partial mapping which associates a name to each event, e.g., method calls and returns. In multi-threaded programs, due to scheduling and locking, there can be a delay between when a method is called, and when its body starts executing. For synchronized methods, a method call indicates that a lock was requested, whereas the start of the execution of a method body indicates that the lock was acquired successfully. To distinguish these two events, we introduce an 'exec' event, that indicates the start of execution of a method body (and thus, implies acquisition of the lock). See Fig. 4 for a communication view that uses an 'exec' event. Returns of synchronized methods indicate the release of the lock.

In general, our framework incorporates built-in attributes of an event which store the objects involved, e.g., the actual parameters. Here we introduce a new built-in attribute `Long threadId` which stores the identity of the thread in which the event occurred. Thread id's are used in both the object and the global view.

The final addition to our framework are `reset`-actions. Suppose a user desires to check the history only when a specific event occurs, and subsequently reset the history to start a new session. We support this by labeling an event with a `reset(b)` action, where `b` is a `boolean` expression in which both grammar attributes *of the previous session* (the attributes of the grammar start symbol) and the objects involved in the event can be used. Grammar attributes are prefixed by the keyword 'session'. If the condition `b` is true, the history is parsed and subsequently reset, with the caveat that attributes values of the previous session are retained. Figures 2 and 4 both depict views with `reset`-actions.

2.2 Grammars and Interference Freedom

The context-free grammar underlying an attribute grammar generates in our approach the valid histories, i.e., traces of events. Event names form the terminal symbols of the grammar, whereas the non-terminal symbols specify the structure of valid sequences of events. In our approach, a communication history is valid if and only if it and all its prefixes are generated by the grammar. The attributes are used in assertions to specify the *data-flow* of the valid histories.

State-based specifications for multi-threaded programs introduce race conditions. During evaluation of an assertion in one thread, another thread can change the state. This implies that different parts of the same assertion are evaluated in different states. For example, `assert o.x==o.x;` is not necessarily true anymore, if `o.x` is changed by another thread. To solve this problem, we must ensure exclusive access to all objects occurring in the assertion. This requires complex locking mechanisms and a fine-grained control over the underlying execution platform. Furthermore, such a complete 'lock-down' of the system can have a severe negative impact on performance. Our history-based approach avoids these problems, provided that a grammar is *well-formed* in the following sense.

Definition 1. *A grammar is well-formed if and only if no built-in attributes (of terminals) are dereferenced.*

This condition is easily checked, and is natural from a conceptual point of view: whenever an event occurs, *only* the actual parameters and return value are communicated between the caller and the callee. Dereferencing an attribute accesses the underlying state of the program, and would mean that the grammar does not only depend on the current history, but potentially on the entire heap! The above condition ensures that the grammar depends only on the history. Well-formed grammars allow an elegant solution to the interference problems mentioned above. See Sect. 4.2 for more details.

3 Multi-threaded Perspectives

Thread View. In the thread perspective, we specify the behavior of each thread in isolation by a corresponding communication view and grammar. We associate a history to each thread, and the grammar generates the set of valid histories of the thread. Semantically, such thread-local histories can be obtained from the global history by projection on the value of the `threadId` attribute[1].

The thread perspective is tailored for the specification of properties that each thread must obey, independently from that of any other threads. Consider for example a client-server scenario. To avoid blocking access to the Server while handling the requests of a Client, the server creates a new thread for each client after accepting an incoming connection, and each client must follow a protocol.

We illustrate the thread view using the running dining philosophers example. Figure 2 presents the communication view for the Phil thread class. It introduces the grammar terminals "start", "get", "release", and "eat" for the corresponding events. Only events from implementations of the `Fork` interface with `synchronized` versions of `get` and `release` are selected.

```
thread view Phil grammar Phil.g {
  return Phil.Phil(Fork left, Fork right) start reset(true),
    call synchronized void Fork.get() get,
    call synchronized void Fork.release() release
               reset(callee==session.right),
    call void Pasta.eat() eat
}
```

Fig. 2. Communication view philosophers

Note that we have included the constructor method of the class Phil in the communication view, which allows us to use its parameters in the attribute

[1] There is one subtle technicality: thread id's can be reused in Java. Hence, two events that occur in different threads can share the same thread id. This can be detected by monitoring the `void run()` method of a Thread class in the communication view. A call to `run` signals the creation of the thread, and a return indicates termination.

```
S ::= start
     (S.left=start.left; S.right=start.right;)
  |  gf1=get gf2=get eat rf1=release rf2=release
     (S.left=start.left; S.right=start.right;)
     { assert gf1.callee == S.left && gf2.callee == S.right; }
     { assert rf1.callee == S.left && rf2.callee == S.right; }
```

Fig. 3. Attribute grammar for philosopher session

grammar to specify the behavior of the thread instances of class Phil. This communication view marks each `start` event, and each `release` event for which its `callee` equals the attribute of the previous session, as a reset.

The grammar in Fig. 3 defines the behavior and the attributes of a session[2]. The grammar introduces the aliases `gf1`, `gf2`, `rf1` and `rf2` to distinguish different occurences of the same terminal.

Object View. In the object view of a Java program, we specify the interaction of a single object by means of a corresponding communication view and grammar. The grammar generates the set of all valid traces of events that the object may engage in. In a multi-threaded environment, several threads can be active (executing) in a single object. Intuitively, the local object histories can be obtained from the global history by projection on the values of the built-in attributes `caller` and `callee`.

The object view is convenient for specifying a resource that is shared between different threads. We illustrate this by a specification of the forks in the dining philosophers example. Figure 4 presents the communication view.

```
local view ForkView grammar Fork.g specifies Fork {
      exec synchronized void get() get,
    return synchronized void release() release reset(true)
}
```

Fig. 4. Communication view of a fork

As explained above, the keyword "exec" indicates events which are triggered by the start of the execution of the specified method. The grammar in Fig. 5 then defines the behavior of a session of a Fork object. Note that mutual exclusion here is simply expressed by the identity between the thread id's corresponding to the events which indicate the execution of the get and release methods.

[2] Terminals have built-in attributes, which refer to the objects involved. Non-terminals have user-defined attributes defining properties of sequences of terminals.

S ::= get release { `assert get.threadid==release.threadid;` }

Fig. 5. Attribute grammar for fork session

Global View. In the global view, we treat the Java program as a single entity that we wish to specify. Traditional approaches based on global invariants [4,11] are supported in this perspective. The grammar generates the set of all valid *global* traces of the entire program. In particular, the user can specify the desired interleavings between events from different threads.

We illustrate the global view by means of a general method for dynamic detection of deadlocks. We focus on deadlocks that arise from synchronized methods. Given a multi-threaded program we first introduce the communication view in Fig. 6, which includes all events of the synchronized methods.

```
global view DeadlockMyProgram grammar deadlock.g {
    call synchronized T C.m() C_m,
  return synchronized T C.m() ret_C_m,
    exec synchronized T C.m() exec_C_m,
  ...
}
```

Fig. 6. Global communication view

Since unsynchronized methods cannot cause a deadlock, the communication view filters those out. A thread blocks if it calls a synchronized method on an object that is already locked by another thread. The general idea is to build a directed "wait-for" graph to capture such dependencies between threads. A deadlock corresponds to a cycle in the wait-for graph.

In more detail, the nodes of the graph are thread id's, and there is an edge from t_1 to t_2 if t_1 calls a method on some object that is locked by t_2. To build the graph we define three inherited attributes:

– An attribute `reqLock` of type `Map<Long, Object>` that maps a thread id (a `Long`) to the object for which it requested, but has not yet acquired the lock.
– An attribute `hasLock` of type `Map<Long, Map<Object, Integer> >`. Given a thread id and an object, this map returns the number of times the lock on that object has been acquired but not released by the thread[3]. If for a given thread id, the count becomes 0 for an object, the entry is removed.
– An attribute `g` storing a directed graph. Any graph library can be used as long as it has a method `void addEdge(Object node1, Object node2)` for adding an edge between `node1` and `node2` to the graph, and a (pure) method

[3] Due to reentrance, locks in Java can be acquired more than once by the same thread.

`boolean noCycle()` for cycle detection. For example, `DirectedGraph` from the `JGraphT` library suffices for our purposes.

The two maps and the graph are initialized to empty. Figure 7 shows how the attributes are updated whenever an event occurs. The graph is built in the last ϵ-production using the `reqLock` and `hasLock` maps. The assertion formalizes the no deadlock requirement.

```
S ::= C_m
      (
       S.reqLock.put(C_m.threadId, C_m.callee);
      )
      S
  |   ret_C_m
      (
       Map<Object, Integer> m = S.hasLock.get(ret_C_m.threadId);
       Integer cnt = m.get(ret_C_m.caller);
       if( cnt == 1)m.remove(ret_C_m.caller);
       else m.put(ret_C_m.caller, cnt-1);
      )
      S
  |   exec_C_m
      (
       S.reqLock.remove(exec_C_m.threadId);
        Map<Object, Integer> m = S.hasLock.get(exec_C_m.threadId);
        int newCnt = 1;
        if(m == null){
         m = new HashMap<Object, Integer>;
         S.hasLock.put(exec_C_m.threadId, m);
        } else if(m.get(exec_C_m.callee)!= null)
         newCnt = m.get(exec_C_m.callee)+1;
        }
        m.put(exec_C_m.caller, newCnt);
      )
      S
  |   ϵ
      (
       for(Long rl : S.reqLock.keySet())
        for(Long hl : S.hasLock.keySet())
          if(hl.get(rl).containsKey(reqLock.get(rl)))
           g.addEdge(rl, hl);
      )
      { assert g.noCycle(); }
```

Fig. 7. Attribute grammar specifying deadlock

3.1 Combining the Views

Above, we described three different perspectives on a multi-threaded Java program: the thread view, the object view and a global view. Thus, to properly support a separation of concerns, multiple attribute grammars can be present, each grammar focusing on a specific behavioral property of the program. We check all grammars independently. If each grammar is considered to be a formal language (generating the valid histories), this has the same effect as taking the intersection, or conjunction of the formal languages involved. Note that this also provides a simple way to specify the intersection of two context-free languages. Although the intersection of two context-free languages is in general not context-free, we can specify them separately by two different context-free grammars.

Combining the above views on the dining philosophers ensures that

– All philosophers use their (shared) resources correctly (thread view).
– The resources themselves behave properly (object view).
– No deadlock arises during execution (global view).

4 Tool Architecture

In this section we describe the tool architecture of SAGA (Fig. 8). SAGA integrates four different components: a state-based assertion checker, a parser generator, a monitor to intercept events and a general tool for meta-programming. How and when these components are used is explained below by means of a workflow. The tool architecture in a single-threaded setting was previously described in [5], and successfully applied to a large industrial case[4] with 150,000 lines of code and 44 classes, specified by 5 different attribute grammars of 28, 65, 115, 25 en 37 lines, respectively. It is important to note that this application thus involves storage, updates and parsing of different histories. The main difference with respect to multi-threaded programs resides in the additional event "exec" and the different views (i.e., thread, object and global view). Below we describe how the extensions at the implementation level needed for multi-threading can be easily incorporated without affecting the overall tool architecture, which has proven its use in practice, as shown above. Further, we discuss how these extensions avoid the interference problem as explained in the introduction.

4.1 Meta-Program

Suppose that during execution of a multi-threaded Java program, an event listed in a communication view occurs. The corresponding history should be updated to reflect the addition of the new event. Hence, the first question is: *how to represent the history*? A meta-program written in Rascal [8] generates for each event in the communication view a 'token class' with fields for storing: the *caller*

[4] This case study has successfully been carried out in the EU HATS project (http://www.hats-project.eu).

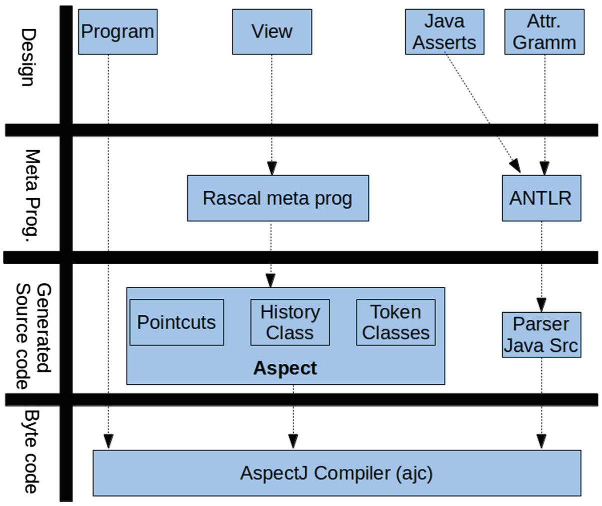

Fig. 8. SAGA tool architecture

and *callee*, the *thread identity*, the actual parameters and (for return events only) the *return value*. We can then represent a history as a Java `List` of instances of token classes.

4.2 Monitoring and Interference

The monitoring component updates the history whenever a relevant event occurs, and (possibly) triggers the parser.

AspectJ is tailored for monitoring. It can intercept method calls, executions and returns conveniently with pointcuts, and weave in user-defined code (advice) that is executed before or after the intercepted event. Each pointcut corresponds to an event listed in the communication view. The advice is the code that updates the history.

The code for the aspect is generated from the communication view automatically by the Rascal meta-program. Advice is woven into Java source code, byte code or at class load-time fully automatically by AspectJ. We store thread-local histories in a `Map<Long, History>`, which assigns to each thread id (of type `Long`) its history, and use the inter-type declarations of AspectJ to store the local history of an object in the object itself in a new field named `h`. This ensures that whenever the object goes out of scope, so does its history and consequently reduces memory usage. Furthermore, compared to storing a single global history, this method avoids the calculation (by projection upon the global history) of the local object and thread histories. Since in thread-local histories, the `threadId` is the same for every event, we reduce memory usage by avoiding storing it altogether. We store global histories inside a separate Aspect class.

Figure 9 shows a generated aspect. The first five lines together form a pointcut, the sixth and seventh line is the advice. The third line identifies the method name. The fourth line binds the variables ‘clr’ and ‘cle’ to the appropriate objects, and together with the first line, determines the full method signature (thereby distinguishing overloaded methods). The fifth line ensures that the advice is executed only when assertions are enabled. Assertions can be enabled for each communication view (and associated grammar) individually. The sixth line retrieves the thread id, and the seventh line calls a `void synchronous update(Token t)` method that appends the new event to the history. Here, `exec_get` is the token class (generated by the Rascal meta program) corresponding to the event. Since the event was defined in a local object view, the history is saved in a field `cle.h` of the callee and will not persist in the program indefinitely. It will be garbage collected as soon as the callee object itself is destroyed.

```
    /* exec synchronized void get(); */
before(Object clr, Fork cle):
 (execution( synchronized void *.get())
  && this(clr) && target(cle)
  && if(ForkViewAspect.class.desiredAssertionStatus() )) {
    long threadId = Thread.currentThread().getId();
    cle.h.update(new exec_get(clr, cle, threadId));
}
```

Fig. 9. Aspect for the event ‘exec synchronized void get()’ from Fig. 4

Our approach avoids interference, as a consequence of which concurrently running threads of the program need not be locked, because of the following three main characteristics:

- Whenever the history should be updated with a new event, we create a new instance of the appropriate token class to store the objects involved in the event. This new token class object is not changed (or even visible) in the program under test.
- Assertions in well-formed grammars do not refer to the actual state of the program: they are completely determined by the values of the built-in attributes stored in these newly created objects.
- Event updates that arise from different threads can cause an update to the same history variable: global histories and local object histories are shared between threads. We provide exclusive access to the history with update methods.

4.3 Parser Generator and Assertion Checker

After a history update, SAGA must decide whether it still satisfies the specification given by the grammar. A history can be seen as a sequence of tokens (in our setting: events). Since the grammar together with the assertions generate the set of all valid histories, checking whether a history satisfies the specification reduces to deciding whether the history can be parsed by a parser for the grammar, where moreover during parsing the assertions must evaluate to true. For events labelled with a reset action, if the associated condition is true, the parser is triggered and the history subsequently reset (see below for more information on parsing).

We use ANTLR [14] to create a parser for the given attribute grammar.

During parsing, ANTLR calls a state-based assertion checker to evaluate the assertions in the grammar. We used standard Java assertions in our grammars. This ensures compatibility with all Java compilers. The result of the parsing process is either a parse or assertion error, which indicates that the history violates the specification given by the attribute grammar, or a parse tree decorated with new attribute values.

5 Conclusion

The new version of SAGA can be obtained from https://github.com/cwi-swat/saga. Although we illustrated our framework using synchronized methods, general locks as provided in the package `java.util.concurrent.locks` can be handled just as easily by tracking the methods `lock`, `tryLock` and `unlock` in the communication view. In [5] we have already reported on a successful application of our tool to an industrial case study. This led to the integration of SAGA into the software lifecycle at SDL. Currently, in the context of the European Envisage project, we are extending this case to the specification, run-time verification and *monitoring* of a distributed cloud application.

References

1. Aftandilian, E., Guyer, S.Z., Vechev, M.T., Yahav, E.: Asynchronous assertions. In: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, Part of SPLASH 2011, Portland, OR, USA, pp. 275–288 (2011)
2. Araujo, W., Briand, L.C., Labiche, Y.: Enabling the runtime assertion checking of concurrent contracts for the java modeling language. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, pp. 786–795 (2011)
3. Bodden, E., Havelund, K.: Racer: effective race detection using AspectJ. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, pp. 155–166 (2008)
4. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 480–494. Springer, Heidelberg (2010)
5. de Boer, F.S., de Gouw, S., Johnsen, E.B., Kohn, A., Wong, P.Y.H.: Run-time assertion checking of data- and protocol-oriented properties of java programs: an industrial case study. In: Chiba, S., Tanter, É., Bodden, E., Maoz, S., Kienzle, J. (eds.) Transactions on AOSD XI. LNCS, vol. 8400, pp. 1–26. Springer, Heidelberg (2014)

6. Hurlin, C.: Specifying and checking protocols of multithreaded classes. In: ACM Symposium on Applied Computing (SAC 2009), pp. 587–592. ACM Press (2009)
7. Kandziora, J., Huisman, M., Bockisch, Ch., Zaharieva-Stojanovski, M.: Run-time assertion checking of JML annotations in multithreaded applications with e-OpenJML. In: Proceedings of the 17th Workshop on Formal Techniques for Java-Like Programs, FTfJP 2015, Prague, Czech Republic, pp. 8:1–8:6. ACM, New York (2015)
8. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain specific language for source code analysis and manipulation. In: Walenstein, A., Schupp, S. (eds.) Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), pp. 168–177 (2009)
9. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory **2**(2), 127–145 (1968)
10. Luo, Q., Rosu, G.: EnforceMOP: a runtime property enforcement system for multi-threaded programs. In: International Symposium on Software Testing and Analysis, ISSTA 2013, Lugano, Switzerland, pp. 156–166 (2013)
11. Mizuno, M.: A structured approach for developing concurrent programs in Java. Inf. Process. Lett. **69**(5), 233–238 (1999)
12. Möller, M., Olderog, E.-R., Rasch, H., Wehrheim, H.: Integrating a formal method into a software engineering process with UML and Java. Formal Asp. Comput. **20**(2), 161–204 (2008)
13. Nonaka, Y., Ushijima, K., Serizawa, H., Murata, S., Cheng, J.: A run-time deadlock detector for concurrent Java programs. In: 8th Asia-Pacific Software Engineering Conference (APSEC 2001), Macau, China, pp. 45–52 (2001)
14. Parr, T.: The Definitive ANTLR Reference. Pragmatic Bookshelf, Lewisville (2007)
15. Rodríguez, E., Dwyer, M.B., Flanagan, C., Hatcliff, J., Leavens, G.T., Robby: Extending JML for modular specification and verification of multi-threaded programs. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 551–576. Springer, Heidelberg (2005)
16. Rosu, G., Sen, K.: An instrumentation technique for online analysis of multithreaded programs. Concurrency Comput. Pract. Experience **19**(3), 311–325 (2007)