Modeling and Validation of Java Multithreading Applications using SPIN

C.Demartini, R.Iosif, and R.Sisto

demartini@polito.it, iosif@athena.polito.it, sisto@polito.it

Dipartimento di Automatica e Informatica, Politecnico di Torino
corso Duca degli Abruzzi 24, 10129 Torino (Italy)

Abstract

This paper presents some issues about the design and implementation of a concurrency analysis tool for deadlock detection on Java programs based on Promela and SPIN. An abstract formal model expressed in Promela is generated from the Java source using the Java2Spin translator. Then the model is analyzed by SPIN and possible error traces are converted back to traces of Java statements and reported to the user. We carried out a set of experiments, to evaluate the extent to which this approach is feasible, and found that non-trivial Java programs can be successfully analyzed.

KEY WORDS: concurrent programming; deadlock; formal analysis; PROMELA; SPIN

Introduction

Java [6] is one of the few object oriented programming languages that offer a simple and integrated support for threads, allowing widespread use of concurrent programming. This feature makes Java one of the natural contexts where programmers may get benefit from the application of formal verification techniques aimed at excluding the presence of concurrency faults such as deadlocks.

Our focus here is not on formal verification of high-level models of concurrent applications, but on static analysis of implementations, to be performed when the source code is available. This kind of analysis is not in contrast with the former, rather it is complementary, because it can weed out errors that may be introduced while refining and implementing high-level models.

The validation method for Java concurrent applications that we propose here is based on reachability analysis and in particular on the use of the SPIN tool [5]. The analysis procedure consists of three steps:

1. Creation of an abstract formal model describing the behavior of the program. The Java source code is converted into a Promela [4] formal description file by the Java2Spin translator tool.
2. Analysis of the model to find error states. The validation of the model is performed by the SPIN analysis tool.
3. In case errors are found, the sequences of program states that lead to the errors are built. The error traces generated by SPIN are converted back into sequences of original program statements and reported to the user.

JCAT (A Java Concurrency Analysis Tool) is the tool that implements this kind of analysis. It was designed as an integrated environment provided with a simple and easy-to-use graphical interface. Its aim is to automatically perform all the above operations in a transparent way, by embedding the facilities of the Java2Spin translator and the ability of running the SPIN validation tool.

The choice for the Promela language and the SPIN validation tool was driven by several reasons, among which the flexibility of the former, that gives among other features the possibility of modeling dynamic process creation and destruction, and the efficiency of the latter [2].

Deriving an abstract formal model from the source code is the most crucial point, because the model must be accurate enough to yield meaningful results, but at the same time it must be simple enough to enable analysis of non-trivial applications in reasonable time. Also, generality conflicts with efficiency because the model can be kept so much simple as the number of different language features to be modeled is small. One first approach to simplify the model in our tool is to enable neglecting unessential details, such as the contents of selected variables, and to use as simple as possible abstractions for the basic synchronization mechanisms and native operations. Another technique that we used to simplify the model is enforcing more partial order reductions than those automatically applied by SPIN.

Another critical aspect to be addressed is how to translate object-oriented features in Promela. For example, how to represent objects and classes and how to deal with polymorphism and inheritance.

The rest of the paper goes through these issues and illustrates the solutions that we devised. First the main related approaches are reviewed, and the Java concurrency support is briefly described. Then, the design issues of the JCAT tool are presented, and finally some experimental results about the tool performances are given.

Related Work

Most concurrency analysis tools operate on high-level models of concurrent applications. Only few of them are designed for source code analysis. While in principle similar techniques are applicable at both levels, the pragmatic considerations are different. In particular, details irrelevant to the concurrent behavior of the program are unavoidably present in the source code, and the relevant parts of the program must be identified and extracted.

The most common approach to concurrent programs analysis is similar to the one adopted in our tool, i.e. to translate the program source code into an analysis formalism by using some abstraction technique.

Translation from Ada to Petri nets was described in several papers among which [9]. The Task Interaction Graph (TIG) model of Ada tasking programs was introduced by Long and Clarke in [7]. A practical application of this model was the Concurrency Analysis Tool Suite for Ada Programs (CATS), described in [10], that uses a compiler front end to generate the representation of Ada tasks according to the TIG model and performs temporal logic model checking on such representation.

To our knowledge, no concurrency analysis tool based on these approaches has ever been developed for the Java language. A work that has some aspects in common with the one presented here regards the formal analysis of sC++, a concurrent extension of C++ [1]. The method outlined in [1] relies on the SPIN analyzer as we do, and the object-oriented nature of sC++ puts problems similar to the ones found when modeling Java applications. The main differences between our work and that of [1] are the input languages, that differ under various respects among which inter-process communication mechanisms, and the general approach to model extraction. In [1], the aim is to deal with a very restricted subset of the language, that is meant to be used as a high-level design language, to be refined successively by means of the other language features. In our tool instead we intend to limit restrictions on the input language as much as possible, and insist on the use of abstraction techniques to get a simplified model.

From the methodological point of view, our work was mainly inspired by the rich experience with Ada programs, even though the Ada solutions could not be applied directly because of the heavy differences between the Java and Ada languages.

A preliminary and simplified version of the basic approach followed in our tool to derive an abstract formal model from a Java source program was first presented in [3].

The Java Concurrency Support

A Java Virtual Machine (JVM) can support several execution threads at once [6]. Threads independently execute Java code that operates on objects residing in a shared main memory. Threads are created and managed by the built-in classes `Thread` and `ThreadGroup`. The only way to create a thread in Java is to create a `Thread` object. Each Thread object has an associated `run` method, that contains the code to be executed by the thread. A newly created thread object is not yet active; it begins running (i.e. executing its `run` method) when its `start` method is called, and stops when it reaches the end of its `run` method or when its `stop` method is called. The execution of a thread can be temporarily suspended by means of the `suspend` method. A suspended thread can proceed only if its `resume` method is called by some other thread. A call to the `join` method of a thread causes the caller thread to wait until the target thread is no longer alive.

Threads can basically communicate via variables from a shared main memory, which can lead to safety and liveness problems. An usual way to avoid them consists in using the standard primitives for synchronizing the concurrent activity of threads. To synchronize threads, Java uses *monitors* which are high-level mechanisms that allow only one thread at a time to execute a region of code protected by the monitor. The behavior of monitors is explained in terms of locks. A lock is associated with every object. A *lock* action done by a thread on a specified object has the effect of acquiring a claim on the object lock, whereas an *unlock* action releases a previously acquired lock associated with the object.

In Java there is no way to perform separate *lock* and *unlock* actions. Instead, they are implicitly performed by *synchronized* statements and *synchronized* methods. These are high-level language constructs that arrange always to pair such actions correctly. A *synchronized* statement first computes a reference to an object; it then attempts to perform a *lock* action on that object and does not proceed further until the *lock* action is successfully completed. After the completion of the *lock* action the body of the synchronized statement is executed. If the execution of the body is ever completed then an *unlock* action on the same lock is automatically performed. A *synchronized* method acts as if the body of the method were contained in a *synchronized* statement. If the method is an instance method, it locks the lock associated with the instance object in behalf of which it was called. If the method is declared `static` it locks the lock associated with the `Class` object that represents the class in which the method was declared. Java monitors are re-entrant, that is, a thread is allowed to acquire more than once the same lock without blocking itself.

A method that is not marked as synchronized may execute immediately when called, even if some other synchronized method is currently executing. Therefore, *synchronized* is not equivalent to *atomic* because it is not enough to ensure exclusive access to some region of code; every unsynchronized method can execute it concurrently. Java guarantees that only the most primitive operations are atomic and will work safely without synchronization in a multithreading environment. These operations include individual accesses and assignments to all built-in scalar types, except `long` and `double`.

In addition to *synchronized* statements and methods, the Java standard library provides the methods `wait`, `notify` and `notifyAll` defined in class `Object`. Rather than spinning in a busy-wait loop, that is, continuously testing if the state of an object has changed, a thread can suspend itself by calling `wait`, until another thread awakens it by calling `notify` or `notifyAll`. This mechanism is used mainly in circumstances where threads have a producer-consumer relationship, rather than a mutual exclusion one.

The `wait`, `notify` and `notifyAll` methods must be called exclusively within synchronized statements or methods and make use of the so-called *wait queue* associated with each locked object. As the result of calling `wait`, a thread performs the following actions:

1. the object lock is released.
2. the thread is disabled for further scheduling purposes.
3. the thread is appended to the object wait queue.

The thread remains dormant until some other thread calls the `notify` or `notifyAll` method of the same object. In this case the thread may be extracted from the object wait queue and re-enabled for scheduling, but it cannot proceed until it acquires again the object lock. The state of the object lock on return from the `wait` method is exactly the same as it was when `wait` was invoked. A variant of the `wait` method is the *timed wait*, that blocks the caller only for a specified amount of time. After the expiration of the time quanta, the thread is automatically awakened and competes with the other threads in order to acquire the object lock.

If the `notify` method is called, one thread is picked at random from the locked object wait queue, and enabled for thread scheduling, whereas if the `notifyAll` method is called all the threads in the locked object wait queue are awakened. Obviously, the awakened threads will not proceed until the current thread completes the notification method and relinquishes the object lock. The `notify` and `notifyAll` methods have no effect if the locked object wait queue is empty.

The Modeling Approach

The main objective in the design of our tool was to obtain an efficient model of the source program without sacrificing too much the model accuracy.

The concurrency primitives used in Java programs (i.e., threads, synchronization mechanisms) are translated in PROMELA according to a template model. Every thread is modeled by a separate process. Synchronization among threads is modeled using message queues and global variables. Since PROMELA is a specification language, rather than a conventional programming language, it lacks several run-time mechanisms, such as dynamic memory management and function call-return. In order to deal with the gap between the source and the target languages, the translator uses a set of approximation techniques that emulates the Java run-time system making use only of the static mechanisms allowed in PROMELA. In respect to this, the modeling power is bounded by some restrictions (e.g., the maximum number of processes, message queues and variables) imposed by the state explosion problem and by some specific limitations of SPIN. The model complexity is kept low by avoiding explicit modeling of library classes, by letting the user specify which variables need to be modeled and by applying other model reduction techniques.

The remainder of this section will first present the approach in modeling Java objects and threads. Then we present a set of model templates for describing the thread synchronization primitives and some model optimization techniques.

The Object Model

As stated before, the PROMELA language lacks the possibility of memory allocation and object reference. Under some limitations, these run-time mechanisms can be emulated using predefined vectors of objects of a convenient size. The term "convenient size" means large enough to allow all object instantiations during the execution of a Java program to be performed, but still small, not to exceed the bounds imposed by the state space explosion problem. As the number of class instantiations during the execution of a program cannot be always statically determined, the task of approximating the right size of the object vector cannot be automatically performed by the translator, being left entirely to the user.

The object reference mechanism uses an integer value as a vector subscript in order to distinguish among several instances of the same class. More precisely, the reference types from Java are translated into a PROMELA predefined *reference* type.

```
typedef REFERENCE {
        int i_no;
};
```

The i_no field of the structure is used to keep track of the class instance number. The formal description of a Java program uses a naming convention in order to distinguish between several class types. A field variable that can be accessed in Java using a qualified expression as PackageName.ClassName.FieldName becomes a global variable named PackageNameWithoutDots_ClassName_FieldName where PackageNameWithoutDots stands for the package name string in which all dots are replaced by underscores. For example, the Java class declaration:

```
package Test.One;

class SomeClass extends SomeSuperClass {
    static int x;
    int y;
}
```

is modeled by the following series of declarations in PROMELA:

```
#define MAX_INSTANCES     10

int Test_One_SomeClass_x;                          /* Class variable */
int Test_One_SomeClass_y[MAX_INSTANCES];           /* Instance variable */
int Test_One_SomeClass_i_cnt;                      /* Instance counter */
REFERENCE Test_One_SomeClass_super[MAX_INSTANCES]; /* Super class reference */
```

A *class variable* is a field declared using the keyword static within a Java class declaration or without the keyword static within an interface declaration. In the Java run-time a class variable is created when its class or interface is loaded and is initialized to its default value. Every class variable is translated into a global variable, using the above presented naming convention.

An *instance variable* is a field declared within a class declaration without using the keyword static. If a class contains an instance variable, then a new variable is created and initialized to its default value as part of every newly created object of the class or of any of its subclasses. The instance variable effectively ceases to exist when the object to which it belongs is no longer referenced. Every instance variable declaration is represented by a global vector of a predefined size. In the above presented case, up to MAX_INSTANCES objects of class SomeClass can be created during the execution of the Java program that is, a vector of MAX_INSTANCES instance variables of class SomeClass is declared. Unlike the Java run-time that performs garbage collection of objects that are no longer referred, the translation model does not provide a way to reuse instance variable vector slots. Implementing a garbage collection mechanism in PROMELA would greatly increase the size of the model state space.

Every class model in PROMELA has an associated *instance counter* variable used in new instance creation expressions. A *super class reference* is declared in order to keep track of the super class object of the current class instance. The following example explains the use of the instance counter and the super class reference. Let us consider the class instance creation expression:

```
SomeClass object = new SomeClass();
```

It results in the following sequence of PROMELA statements:

```
REFERENCE object;

Test_One_SomeClass_i_cnt ++;         /* class instance creation */
object.i_no = Test_One_SomeClass_i_cnt;

Test_One_Some_SuperClass_i_cnt ++;   /* super class instance creation */
```

```
Test_One_SomeClass_super[object.i_no].i_no = Test_One_SomeSuperClass_i_cnt ++;
```

The class instance counter is first incremented, then its reference value is assigned to the `i_no` field of the reference variable. Then a new instance of the super class is created, its value being assigned to the object super class reference. The object reference index is used to distinguish between class instances when performing *lock* and *unlock* operations and also to refer class instance fields. For example, the expression `object.y` that denotes the instance field `y` of class `SomeClass` is translated in PROMELA as `Test_One_SomeClass_y[object.i_no]`. The inherited fields of the object are accessed via the super class reference.

It is to be noted that instance indexes are one-based numbers, a zero reference index being used only to denote a static reference or a reference to the `Class` object associated to every class in the system. In particular, this is used only by static *synchronized* methods that perform *lock* and *unlock* operations on the class lock.

Due to the lack of a garbage collection mechanism in the formal model, Java programs that create new objects in an infinite loop will always generate untraceable models. The design choice was made here between the possibility of analyzing a large class of programs using a complex model and analyzing a smaller class of programs using simple models. In practice, complex models usually fail at validation stage because the amount of memory required by the analysis tool is far too large than available.

Object methods are not modeled as separate entities, but they are incorporated in the thread models as explained in the next section.

The Thread Model

Java threads are modeled in PROMELA using `proctype` constructs. Each thread class declared in a Java program has a corresponding PROMELA process type that contains a formal representation of the code within its `run` method. Sequential Java programs can be seen as having a single thread that executes code from the `main` method. The formal representation of this code is contained by the `init` process of the PROMELA specification.

The method and constructor invocations performed by the thread are inlined within the thread's body and translated into corresponding PROMELA code. The use of function inlining is intended to keep the number of active processes from the model as small as possible. This technique can only be applied in cases of non-recursive methods and constructor invocations. A recursive method is modeled by means of a separate process type.

The thread modeling approach distinguishes four states in which a thread can be found after being created:

1. *ready.* A newly created thread remains in the ready state until its `start` method is called.
2. *running.* After being started a thread enters the running state.
3. *suspended.* A thread enters the suspended state when its `suspend` method is called. A subsequent call to `resume` changes the thread state back to running.
4. *dead.* A thread enters the dead state when it reaches the end of its `run` method or its `stop` method is called by some other thread.

In addition to its field variables, the model of a Java thread class contains a thread status variable used to keep track of the current thread state. For example, the following Java declaration:

```
class SomeThread extends Thread {
    ...
}
```

translates into the following PROMELA declaration:

```
mtype = {ready, running, suspended, dead};

mtype Test_One_SomeThread_status[MAX_INSTANCES];
```

A process type that models a thread is declared to receive a *reference* type parameter that refers the thread instance object. Let us assume that the process which models the behavior of the above declared thread is called `thread0`. The instantiation and starting of the thread are modeled by the following PROMELA statements:

```
REFERENCE thread;
                            /* create a new thread instance (new expression) */
SomeThread_i_cnt ++;
thread.i_no = SomeThread_i_cnt;
SomeThread_status[thread.i_no] = ready;
         ...               /* start the newly created thread (start method) */
if
:: atomic {
        (SomeThread_status[thread.i_no] == ready);
        SomeThread_status[thread.i_no] = running;
        run thread0(thread);
      }
:: else ->
        skip;
fi;
```

First, a new instance of the thread class is created and the status variable of the thread instance is set to the `ready` value. The newly created thread is then started (i.e., in Java its `start` method is called). The model of the thread starting sequence consists in a case selection statement with two guards. If the thread status variable is `ready` then it is set to the `running` value and a new process is spawned, otherwise no action is taken, because subsequent calls of the `start` method on behalf of the same thread must be ignored.

Every instruction within the `run` method of a thread occurs inside an `atomic` sequence whose first statement is a test of the thread status variable. A thread does not proceed until its status variable has the `running` value. A call to the `suspend` method for a thread changes the value of its status variable from `runnable` to `suspended`, preventing the thread from taking any further action. The `resume` method restores the value of the thread status variable to `runnable`, therefore allowing it to proceed.

The PROMELA code that models the `run` method of a thread is generated inside the main sequence of an `unless` construct whose escape sequence consists from an equality test between the thread status variable and the `dead` value. The thread can proceed until its status variable becomes `dead`, then the escape sequence becomes executable, interrupting the execution of the thread. A call to the `stop` method of a thread sets its status variable to `dead`, therefore interrupting its execution. A call to the `join` method is simply modeled by an equality test between the thread status variable and the `dead` value. The caller of the `join` method does not proceed until the thread enters the `dead` state. When a thread reaches normally the end of its execution body (i.e., nobody called its `stop` method) it automatically sets its status variable to `dead` in order to unblock the eventual callers of its `join` method.

The Synchronization Model

Every Java object has an associated lock and a wait queue used by the thread synchronization primitives. The translator converts them into the following PROMELA structure:

```
typedef sync1_t {
        chan lock = [1] of { bit };
        chan notify = [0] of { bit };
        byte nwait;
};
```

The `lock` channel is declared to keep up to one message of the size of one bit. The messages passed via this channel are:

```
#define LOCK    0
#define UNLOCK  0
```

The same message was redefined in order to achieve clarity in the presentation of the synchronization model.

The `notify` channel is a rendez-vous port that can pass one bit values. The message passed via the `notify` channel is:

```
#define NOTIFY  1
```

The `nwait` variable is a counter that keeps track of the number of processes blocked due to a call of the `wait` method. The message passed via the `lock` channel can be seen as the lock claim acquired by a thread when entering a *synchronized* block or method. The `notify` port together with the `nwait` counter are used to represent the object's associated wait queue.

A *lock* action performed by a thread on an object lock is modeled as a receive operation of one message from the object's `lock` channel. The object is identified by the class descriptor string `class` and the class instance number `inst_no`. The name of the synchronization structure associated to a class is composed of the symbolic name of the class prefixed by the `sync_` string:

```
#define _lock(class, inst_no)               \
        sync_##class[inst_no].lock ? LOCK;
```

An *unlock* action performed by a thread on an object lock is modeled as a send operation of one message to the object's `lock` channel:

```
#define _unlock(class, inst_no)             \
        sync_##class[inst_no].lock ! UNLOCK;
```

The `wait` method translation model requires three atomic actions. Although they are not explicitly declared atomic, the translator will always arrange them to be enclosed in `atomic` sequences at the code generation stage. First, the `nwait` counter is incremented and an `UNLOCK` message is sent via the `lock` channel in order to release the previously acquired lock:

```
#define _wait1(class, inst_no)              \
        sync_##class[inst_no].nwait ++;     \
        sync_##class[inst_no].lock ! UNLOCK;
```

The translator performs a static check in order to ensure that `wait` methods are not called outside *synchronized* blocks or methods. At run-time level this guarantees that a send operation to the `lock` channel will never block the current process. Then the caller process blocks attempting to receive a `NOTIFY` message from the `notify` channel:

```
#define _wait2(class, inst_no)                \
        sync_##class[inst_no].notify ? NOTIFY;
```

After receiving the `NOTIFY` message from another process, the current process attempts to reacquire the object lock by receiving a `LOCK` message from the object's associated `lock` channel.

```
#define _wait3(class, inst_no)            \
        sync_##class[inst_no].lock ? LOCK;
```

The `notify` method invocation involves one atomic step in the PROMELA model of the Java program. The template model used for the `notify` method is presented below:

```
#define notify(class, inst_no)                          \
        if                                              \
        :: (sync_##class[inst_no].nwait > 0)  ->        \
                sync_##class[inst_no].notify ! NOTIFY;  \
                sync_##class[inst_no].nwait -- ;        \
        :: (sync_##class[inst_no].nwait == 0) -> skip;  \
        fi
```

If there is at least one process blocked due to a call to the `wait` method then the `nwait` counter has a value greater than zero. In this case a `NOTIFY` message is sent to the `notify` channel associated with the object and the `nwait` counter is decremented. Let us note that `notify` is a rendez-vous port, therefore the waiting process execution will be resumed at the same moment the `NOTIFY` message is sent via the `notify` channel. In case there are no previously blocked processes in the monitor, no action is performed.

The translation model for the `notifyAll` method invocation is similar to the one used for the `notify` method, but notifications are executed repeatedly, as long as there are still processes blocked in the wait queue:

```
#define notifyAll(object, inst_no)                       \
        do                                               \
        :: (sync_##object[inst_no].nwait > 0)  ->        \
                sync_##object[inst_no].notify ! NOTIFY;  \
                sync_##object[inst_no].nwait -- ;        \
        :: (sync_##object[inst_no].nwait == 0) -> break; \
        od
```

The model presented above does not work in the so-called *recursive lock* situation, when a thread acquires several times the same lock without blocking itself. According to the model, performing a *lock* action means receiving a `LOCK` message from the object `lock` channel. Once the message has been extracted from the channel, the attempt to receive another message will deadlock the caller process. In order to deal with this situation, an alternative synchronization model is provided. It uses a modified synchronization structure, as shown below:

```
typedef sync2_t {
        chan lock = [1] of { bit };
        chan notify = [0] of { bit };
        byte nwait;
        byte lockcnt [MAX_THREADS];
};
```

In addition to the non-recursive lock structure, it has the field `lockcnt`, declared as a vector of counters. There is a counter corresponding to each thread in the system. It keeps track of the number of *lock* actions performed by the thread on the object, which are not yet matched by *unlock* actions. In this way, a thread performing *lock* and *unlock* actions will access the `lock` channel only at the outermost level. The modified versions of *lock* and *unlock* actions are not presented here for conciseness.

It is to be noted that the size of the counter vector `lockcnt` in the recursive variant of synchronization structure is bounded by the `MAX_THREADS` value. This is a direct consequence of the fact that in PROMELA there are no possibilities to dynamically increase the size of an array, therefore it must be previously defined at its maximum size. Obviously, this limits the number of threads in the concurrent system to a maximum value. On the other hand, the use of the lock counter vector greatly increases the size of the state vector at validation stage, affecting the analysis time. The translator performs static analysis of the input program in order to detect multiple lock situations. Nested *synchronized* blocks on behalf of the same object are modeled performing only one *lock* and *unlock* operation for the out-most block, using the non-recursive model. There are however situations where a static decision is not enough to ensure that a thread will not deadlock due to a recursive *lock* action. In those cases the recursive lock model must be used instead. In practice, the use of the lock counter vector is limited only to recursive synchronized method invocations. However, Java concurrent programs make little use of recursive methods marked as *synchronized* because invocations of *synchronized* methods are more expensive in run-time than the non-synchronized ones.

Model Optimizations

The performances of a formal model validation depend on the size of the model state space, that is, the amount of memory needed in order to perform an exhaustive verification. The size of the state space can be estimated as the product between the state-vector size and the number of states in the reachability graph. State size reductions can be achieved by reducing the number of variables, vectors and message queues. In order to minimize the number of states in the reachability graph, the average number of states per process must be reduced. The state size optimizations usually result in a linear reduction of the analysis time. Instead, the state number reductions result in an exponential optimization, as the number of states in the reachability graph depends on the product between the number of states per each process.

The Java2Spin translator accepts a special form of comments that are interpreted as code annotations. This provides the user with the possibility of eliminating useless information from the source code, that is, information irrelevant for the concurrent behavior of the program. In addition, the translator automatically performs a set of code optimizations that does not alter the model's accuracy (i.e., the generated model after performing optimizations will have the same properties as the original one). The present section will describe the use of code annotations and the model optimizations automatically performed.

Code Annotations

The use of code annotations refers only to Java variables. Annotating a variable means taking it into consideration for the formal model generation. Ignoring a variable implies ignoring all the expressions in which it appears. Annotations can be applied to every kind of Java variable, that is, to class or interface fields, local variables and function formal parameters. For example, in the following declaration:

```
/*^ x y */        // annotation of x and y
int x, y, z;      // declaration of x, y, and z
```

only the `x` and `y` variables will be modeled in the formal description of the program. Every expression that contains a reference to the `z` variable will be ignored in further code generation. Variable annotations should be used with care because they tend to collapse all the expressions that make use of them, resulting in a simplified but less accurate model, which may produce several spurious error reports at validation stage. However, the reduction in complexity can be significant, due to the minimization of both state size and number of states in the model.

State Size Reductions

The size of a concurrent program state depends on the sum of all global and local variable resources, as variables, vectors and message queues. The translator performs a static analysis on the input program in order to eliminate redundant resource declarations from the model.

In practice, the code of an object-oriented program declares more variables than really needed. This is a consequence of the fact that previously written classes are reused by class inheritance. However, a derived class usually does not make use of the whole functionality of its base class. The formal model of a concurrent Java program is optimized without sacrificing accuracy by representing only the class members referred in the reachable code of the program and only the classes that are actually instantiated. Also, object synchronization structures will not be declared if no thread ever uses them for synchronization purposes. The translator individuates the above cases during static analysis.

State Number Reductions

A straightforward way to reduce the number of states in the model is to enclose all expressions that refer process local variables into `atomic` blocks. A local variable can result from a thread's `run` method local variable, an inlined function local variable or formal parameter. In addition, final fields within a class or interface declaration are treated as constants, therefore can be referred inside an atomic block. The above presented optimization does not affect the system properties, nor it introduces new spurious ones.

Some Java concurrent programs make use only of fully synchronized objects (i.e., objects belonging to classes in which every method was declared synchronized [8]). In this case, the code within a synchronized method can be treated as atomic because there is no other method that can execute concurrently with it. All statements within a *synchronized* method or block are generated in atomic sequences sliced by the eventual occurrences of wait methods. As discussed above, a wait method performs three different actions that must reside in separate atomic blocks. All other statements between a *lock* and an *unlock* action will belong to an atomic block.

The Analysis Tool

The basic structure of the JCAT tool is illustrated in Figure 1. A graphical front-end is provided in order to transparently perform all operations implied by the analysis and debugging processes.

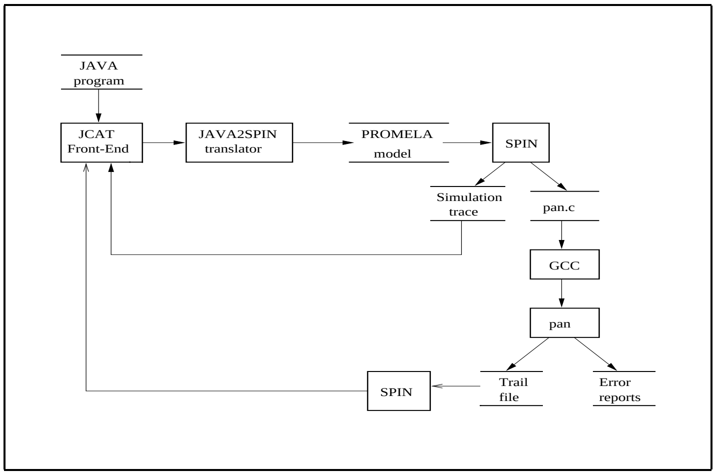

Figure 1: **The structure of JCAT verification and simulation.**

The Java source code is translated into a PROMELA model file by the Java2Spin translator. Then SPIN is used to produce a validator of the model (file pan.c) or just a simulation trace that is directly interpreted and displayed by the front-end. When used to perform a verification JCAT compiles and executes the validator program. The current version of JCAT uses SPIN only to perform deadlock detection of the model. The analysis results together with some statistic information (e.g., total number of states and transitions) are made available to the user. If errors were detected a trail file is automatically generated. In this case SPIN is invoked to match the trail file, the execution sequence that lead to an error being converted back into a sequence of lines from the original Java program, allowing the error to be traced by means of a graphical interface.

The Java2Spin translator was designed in order to generate the PROMELA description of a Java multithreading application. The translator front-end acts roughly the same as the standard Java compiler, performing the syntactic and semantic checks described in [6]. In addition some static checks are performed to distinguish recursive lock situations and carry out the above presented model optimizations.

Code optimization techniques are currently an open subject for research. Although there is no universal approach to reduce the complexity of the target program, several techniques related to flow analysis can be applied by the translator back-end in order to obtain an efficient formal description model. A large analysis overhead is added by recursive methods which cannot be inlined within the caller thread body and must be modeled as separate processes. One of the further development issues consists in the recognition of tail-recursive methods, which can be modeled using inlined iteration loops. Moreover, particular cases or non-tail recursive methods can often be converted to the tail-recursive form by means of an auxiliary parameter used to form the result.

Some Experimental Results

We have applied JCAT to analyze a variety of canonical problems that involve concurrency. Experience with two of these (the dining philosophers example and the readers and writers problem) is presented below in order to give a glimpse of the tool's capability, expressed in terms of state space size that is, the number of states and transitions in the model[1].

Dining Philosophers

The dining philosophers problem is a well-known example of a deadlock situation, often used as a case study for concurrent programming. We assume the reader has some familiarity with the problem. The Java class that models the behavior of a philosopher is presented below:

```
class Philosopher extends Thread
{
  Object left, right;

  Philosopher(Object l, Object r) {
    left = l;
    right = r;
  }

  public void run() {
    while (true) {
      synchronized (left) {
        synchronized (right) { /* eating */ }
      }
    }
  }
}
```

Each fork is represented by a generic object. Taking a fork means simply acquiring a claim on the fork object lock. In order to eat, a philosopher executes two nested `synchronized` statements, the outer one on his left fork and the inner one on his right fork. Table 1 shows the results obtained by having 3, 4, 5, 6 philosophers acting concurrently in a non-deadlocking manner (i.e., the last philosopher always taking the forks in the reversed order).

The test case case involving 9 philosophers shows the upper limit in complexity for this problem. It was analyzed using the bit state hashing technique.

Readers and Writers

The readers and writers problem corresponds to a family of concurrency control designs that involves the control of inspective (reader) threads versus mutative (writer) threads. Any number of reader threads can be executed simultaneously as long as there is no writer thread, while writers require exclusive access. The Java class that implements the readers and writers example

[1] All analysis time reports are obtained from the Unix `time` command on a 64 Mb Pentium Pro 150 MHz machine. Small times (under 0.5 seconds) tend to be inaccurate because of the system overhead. The analysis results (i.e., number of states and transitions) are related to SPIN version 2.9.4.

Philosophers	States	Transitions	Bit state hashing	Time (min:sec)
3	576	1475	no	0:0.66
4	3966	13450	no	0:0.90
5	27265	113938	no	0:6.00
6	184876	914019	no	0:49.00
9	2.99697e+06	2.26987e+07	yes	18:10

Table 1: Dining philosophers

is presented in [8][2]. The thread control policy is implemented by means of a set of shared variables accessed and modified within `synchronized` blocks. Tests were performed on a fully detailed model and on a reduced model that ignores all thread control variables. The results are presented in Table 2.

Threads (readers, writers)	States	Transitions	Bit state hashing	Time (min:sec)
i. Full model				
1, 1	217	396	no	0:0.50
1, 2	3470	9673	no	0:0.85
2, 2	61751	226683	no	0:1.00
6, 6	3.15117e+06	2.69938+07	yes	34:24
ii. Reduced model				
1, 1	66	96	no	0:0.50
1, 2	318	708	no	0:0.50
2, 2	1800	5430	no	0:0.50
6, 6	2.53435e+06	1.60765e+07	yes	12:20

Table 2: Readers and Writers

The table presents the test cases that involve 1, 1, 2 readers and 1, 2, 2 writers respectively. The case in which 6 readers and 6 writers act concurrently was considered as an upper bound of the problem's complexity. It was analyzed using bit state hashing. The analysis of the reduced model reported the occurrence of a spurious deadlock state in the reachability graph, due to the lack of control flow conditions in the program. Both the readers and writers are allowed to non-deterministically block while executing respectively the `read` and `write` methods. This case shows both the risk implied by the elimination from the model of certain shared variables used in control flow statements and, on the other hand, the significant reduction in complexity that can be achieved.

Future Work

The object translation model previously presented allows the dynamic creation and reference of a limited number of objects in a PROMELA formal specification. An object instance field is referenced by an integer array index. As previously discussed, the choice for a certain array is based on the type information obtained at the static checking level. Analogously, object references for synchronized statements and method invocations are computed using type information. Even if this method results in a good approximation of the real cases, there are still situations in which the type of the object cannot be determined during translation to formal model. The declared type of an object reference can be modified during the execution of a Java program by explicit or implicit cast operations. A translator cannot decide during static analysis which

[2] Pages 131-133.

object lock will be used or which virtual method will be called at run-time. Such situations can be handled by some dynamic extensions of the formal specification language used to model a Java concurrent program. An evaluation of the Java run-time requirements showed that implementing the following mechanisms in PROMELA would suffice to model any Java construct:

- object references.
- dynamic object creation and deletion.
- stack-based function call.
- function code references.
- exceptions.

All the above extensions can be implemented in SPIN with only a linear increase in analysis time. The memory space can be kept as small as possible by means of a different organization of the state vector. Optionally, a fast garbage collection algorithm can be used to improve the analysis performances in terms of required memory space.

The current version of the translator can safely model a class of programs that does not use cast operations. A future version is intended to make use of the above presented extensions in order to allow a wider class of Java applications to be modeled.

Another goal to be achieved in a further version of our project is to provide the user with the possibility of making use of the whole functionality of SPIN in analyzing concurrent software. At present, only deadlock detection is performed. We intend to exploit the ability of checking general system properties expressed as LTL (Linear Temporal Logic) formulas by developing a language interface between LTL and Java.

Conclusions

This paper has presented the main issues related to the design of JCAT [3], a tool for modeling Java concurrent applications in Promela and using SPIN to perform deadlock analysis.

The possibility of analyzing Java source code directly and of applying model simplifications automatically has the advantage of aiding the programmer in design problems not adequately covered by tools based on high-level models that must be subsequently mapped onto the target implementation language. Also, if we consider that the abstraction and object-orientation mechanisms of the Java language support the step-wise refinement of high-level Java programs into more detailed implementations, JCAT could be used at various development stages and not only at the implementation stage.

Some preliminary experience with the tool makes us optimistic about the role static analysis can play, in combination with other techniques, for analyzing concurrent Java software. While the well-known state explosion problem makes the analysis of complex programs intractable, task interactions in subsystems well beyond the comprehension of unaided programmers can be exhaustively analyzed in a few minutes, and, even when this is not feasible, at least a partial verification or an extensive simulation of the program behavior can improve the program confidence.

The positive preliminary results obtained stimulate us to proceed in the refinement and improvement of JCAT, so as to get a tool capable of dealing with most of the Java language features and of supporting full model checking functions. Also, we found that JCAT could take advantage of possible improvements in the Promela/SPIN engine.

[3] JCAT source code can be downloaded from URL: http://www.polito.it/~iosif/jcat.

References

[1] Thierry Cattel, "Modeling and verification of sC++ applications", *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, LNCS 1384, Springer, (April 1998), pp. 232-248.

[2] J.C. Corbett, "Evaluating deadlock detection methods for concurrent software", *IEEE Trans. on Software Engineering*, Vol. 22, No. 3 (March 1996), pp. 161-180.

[3] Claudio Demartini and Riccardo Sisto, "Static analysis of java multithreaded and distributed applications", *Proceedings of the 1998 International Workshop on Software Engineering for Parallel and Distributed Systems*, Kyoto, Japan, (April 1998), pp. 215-222.

[4] Gerard J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[5] Gerard J. Holzmann, "The model checker SPIN", *IEEE Trans. on Software Engineering*, Vol. SE-23, No. 5 (May 1997), pp. 279-295.

[6] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison Wesley, 1996.

[7] Douglas Long and Lori A. Clarke, "Data flow analysis of concurrent systems that use the rendezvous model for synchronization", *Proceedings of the Symposium on Software Testing, Analysis and Verification*, (October 1991).

[8] Doug Lea, *Concurrent Programming in Java*, Addison Wesley, 1996.

[9] Sol M. Shatz, Khanh Mai and Cristopher Black, "Design and implementation of a petri net based toolkit for ada tasking analysis", *IEEE Transactions on Parallel and Distributed Systems*, (October 1990).

[10] Michal Young, "A concurrency analysis tool suite for ada", *SERC Technical Report TR-1288-P*, (November 1994).