A Survey on Model Checking Java Programs

Sean Thompson

Abstract

Due to the recent advancements of the Internet, Java has become a dominant programming language, especially for coding concurrent applications. Finite state verification is a powerful technique for detecting subtle errors in a program. This paper reviews recent attempts to transform Java source code into a model that can be checked using verification tools. Two classes of techniques are reviewed. The first is translating Java source code into a known modeling language such as Promela. The second is building a finite state model based on the source and performing state reductions. Once state reductions are performed and the model size reduced, the resulting model can be checked for desired properties.

1. Introduction

Finite state verification tools have immense potential; they are highly automated and can detect hard to reproduce errors. Unfortunately, the acceptance of these tools by the programming community has been limited. This is partly due to the differences between conventional programming languages and modeling languages. Most modeling languages (SMV, Promela) only superficially resemble mainstream languages. To overcome this, the extraction of models from program source code has been investigated.

Verifiable models contain a higher level of abstraction than program source code. For a model from a program source to be verified, the model must be sufficiently small. Unfortunately, most systems with enough complexity to benefit from model checking are limited by the state explosion problem. However, many concurrent systems contain many errors that are subtle and hard to detect. The size of such systems may still be small enough to keep model checking tractable in size. Because of this, extraction of models from program source has been mainly focused on detecting deadlocks in concurrent systems. Until recently, ADA has been the language of choice for such research [8].

The Java language offers a simple package for integrating threads into applications. Compiled Java code can run under any operating system with an interpreter (i.e. the Java Virtual Machine). As a result, the World Wide Web has become an increasingly popular environment for concurrent Java programs. Deadlock detection by simply analyzing Java source code would be highly useful for web application developers, among others.

Unlike ADA, Java is an object-oriented (OO) language. OO languages naturally group related data and routines into distinct modules. These modules (i.e. classes) provide an initial level of abstraction that may be difficult to automate using an imperative language. Simple concurrency support and OO structure make Java an attractive language for model extraction. Some OO concepts are easily represented in common model-checking tools. For example, message passing between objects is similar to message passing between processes. Other concepts, such as polymorphism, can be more difficult to model because standard modeling languages were not designed with these in mind.

Extracting finite-state models from Java source is a non-trivial task. The most common approach so far is to syntactically translate Java source code into Promela. Promela is the modeling language associated with the SPIN model checker. SPIN is particularly effective at checking concurrent systems for deadlock and other concurrency problems. This is due partly to the effectiveness of the message passing constructs and concurrency support within the Promela/SPIN system. Promela also contains processing and syntactical elements similar to that of conventional programming languages, in particular, the language C. It is convenient that the Java language syntax is also based on C.

The second approach is building finite models directly from the source code. The finite models can then be processed to reduce their size. Once these models are optimized, they can be analyzed directly, given the proper tools, or can be used to construct code in a modeling language, such as Promela or SMV. This approach has the potential to overcome some of the problems inherent in techniques using strictly syntactic translation.

This paper proceeds as follows. Section 2 provides an overview of the translation techniques. Section 3 looks at creating and reducing finite models from the Java source code. Section 4 compares the two methods and discusses other advancements that will help develop this technology. Finally conclusions are drawn in Section 5. The reader is assumed to have some familiarity with Promela/SPIN and the Java language.

2 Translating Java to Promela

There are two tools that can translate Java into Promela code. These are the Java2Spin translator [4,5], and the Java Pathfinder [10,11,12]. Cattel [1] also has done work translating sC++ (an extension to C++ to provide concurrency) into Promela, and while the source language differs from Java, the underlying principles for translating an object-oriented language apply. Other work has been done in translating ADA programs [8] although Java offers several new challenges over ADA [2]:

1. Java has a more dynamic nature than ADA. Most ADA variables are stored on the local stack, while Java variables are commonly stored on the heap.
2. Java threads are created dynamically, making it difficult to determine the number of threads a program will create.
3. Java has a unique locking mechanism to synchronize data access.

These challenges undoubtedly impose certain limitations on the ability to translate Java into Promela. Promela has primarily been used to check hardware and protocols; and unlike Java programs, both of these use a predetermined amount of resources. Some basic assumptions about resource requirements are thus needed to be made before Java can be checked by re-modeling the program using Promela.

2.1 Concurrency in Java

Java threads are object instances of the class, or a subclass of, *Thread*. In order to create a user-defined thread, the *Thread* class must be subclassed and a *run()* method defined. The *run()* method contains the code for the thread to execute. Once an instance of the thread object is created, the *start()* method (defined in *Thread*), and not the *run()* method, is instantiated to enable the code to run concurrently. There are several methods defined in the *Thread* class that allow for the control of threads. The method *wait()* is used to halt the thread until another thread sends a message to the blocked thread with either a *notify()* or *notifyAll()* command. The *notifyAll()* command will unblock all threads suspended with *wait()*, whereas *notify()* will awaken only a single arbitrarily chosen thread.

A method or section of code can also be declared *synchronized*. A *synchronized* section of code means only one thread may be executing such code at any given point. Synchronizing code is a simplistic way to perform locking operations. Each object has an associated lock with it. The Java Virtual Machine will implicitly request and wait for an object's lock before entering a *synchronized* block of code, and will release it immediately following.

2.2 Classes

Classes are the fundamental structure in Java. A class defines both the methods and variables associated with an object of that type. Two related approaches can be taken to translate a class to Promela. The first uses a Promela *typedef* statement to create a record of the variables in the class. Inheritance is achieved in this case by inclusion of superclass variables into the *typedef* statement. To represent multiple instances of a class an array of the newly defined type is created. Methods are handled separately and described in the next section.

```
package test.one;

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

```
#define MAX_INSTANCES 10

typedef REFERENCE {
      int i_no;
};

int test_one_SomeClass_x;
int test_one_SomeClass_y[MAX_INSTANCES];
int test_one_SomeClass_i_cnt;
REFERENCE
  test_one_SomeClass_super[MAX_INSTANCES];
```

Figure 1: Java Definition of a Subclass (left) and its Promela Translation (right)

The second approach defines each variable of the class as a unique Promela global variable. Instance variables are defined as an array, while static variables (in which one value holds for all objects of a class) are treated as a single variable. An example from [5] is given in Figure 1. A special counter variable (*test_one_SomeClass_i_cnt*) is kept for each class to hold the number of instances that have been created. Inheritance is accomplished by creating a *REFERENCE* type for the superclass with a variable to represent the instance number. Special naming conventions are used to keep variable names unique. The second approach is illustrated in figure 1 with an example from [4].

Both approaches suffer from the same problem. The model needs to be limited by defining the maximum number of instances of a class that can occur. This clearly does not represent Javas dynamic object nature. For many problems the number of instances of a particular class is small so this solution can suffice.

A second problem is in translating variables. Promela does not have definitions for floating point numbers or strings, both of which are fundamental types in Java. Two approaches have been taken to permit verification of source code containing one or both of these variable types. Applications of Java to Promela translators have been primarily focused towards detection of deadlock situations. Floating-point and string variables may not have any effect on whether or not a deadlock situation can possibly occur. A good example of this is strings used to output information to the user. They offer no change to the state of the program. If a string or a floating-point variable has no effect on the concurrency of an application, it can be ignored (omitted) in the Promela code. Omitting these variables is generally not an automated procedure. In most cases it must be performed manually. The other solution translates floating-point variables into integer variables. This does not represent the exact state space of the Java model and can allow for subtle errors to be introduced (e.g. division by 0.3 becomes division by 0). These solutions have been adequate for the Java source code that model checking has been applied to, but this is still an issue that will need to be addressed further to make this procedure applicable in more general cases.

2.3 Methods

There are two approaches to translating a method. The first approach is to use inline methods and macros. The body of a Java method is translated into Promela code. This code is made into an inline function using one of either the Promela *#define* or *inline* constructs. Extra parameters are added to the new inline function to provide the class name and the class instance number. Any local variables in the method must be made global in Promela.

All instance and class variables used must be prefixed by the appropriate class name. Any reference to an instance variable of a class in the Java code must be appended with an array index using the instance number parameter when translated into Promela code. Inline method calls are replaced by the appropriate Promela code during

```
void doSomething()                #define doSomething(class, inst) \
{                                         if
      if (x > y)                          :: (##class_x[inst]> ##class_y) ->
            x = x - y;                       ##class_x[inst] = ##class_x[inst] -
      else                                                     ##class_y;
            x++;                          :: else -> ##class_x[inst]++;
}                                         fi
```

Figure 2: Translation of a Java Method to Promela using Macros (Inline code)

preprocessing of the model. Constructors are commonly modeled using inline methods. Using this approach can increase the size of the Promela model when Java methods are called frequently. An example of inline method translation is given in Figure 2.

The second approach is to translate a method into a Promela process. In order for this to run sequentially, a special lock variable is used to block the caller until the method has returned. While this keeps the Promela code shorter, it can not be used frequently because of the needed resources of a new process. Each new process contains a local stack. Nested methods and a high number of processes containing method calls can cause the SPIN model checker to run out of memory. To illustrate this point, assume there is a process that needs to call a method containing a nested method. For each process that runs concurrently, at some point of execution (assuming both the methods are called) there are three stacks allocated. This results in a reduction of a third (in the worst case) in the number of processes that can be run concurrently by SPIN, and subsequently reduces the maximum size of a model that the model checker can verify.

Recursion is not handled correctly using inline methods. Simple text replacement techniques would, in theory, create a method with infinite length. Of course the system would run out of memory and/or disk space first. Using processes as methods can handle recursion. However, the problem of running out of memory becomes worse. Each recursive call would require a local stack, and very little recursive depth could be achieved. Generally, the best solution to translating a recursive method is to first make the method iterative.

In order to translate a Java method, the basic language constructs need to be translated. The similarities between Java and Promela make these translations trivial. Decision statements in Java (*if, select*), can be converted to Promelas *if-fi* statement (see Figure 2). Java looping constructs (*do, while, for*) are converted to Promelas *do-od* statement. Variable assignments work similarly for both languages.

2.4 Threads and Synchronization

Java threads are modeled using the Promela *proctype* construct. Each subclass of Thread that is defined is translated into a *proctype* structure. The code within the *proctype* structure is taken from the run method of the associated Java *Thread* subclass. Multiple instances of a (*Thread*) class are simulated by running the associated process multiple times. As was the case with modeling classes, the number of runnable threads must be limited to keep the model size tractable.

With each thread object there is an associated lock. This is modeled in Promela by associating a locking mechanism with each process. The locking mechanism consists of a set of variables and channels bound together to form a Promela type (by using a *typedef* statement). The channels allow the thread to receive lock requests and notify requests. The standard *Thread* functions are not translated. They are prewritten to provide a communication mechanism that provides the same interface as in the Java language, but with an implementation that utilizes Promela's message passing constructs. Macros (i.e. inline code) are used to code the *wait, notify, notifyAll* and *synchronization* methods and statements.

```
typedef synct1_t {
      chan lock = [1] of { bit };
      chan notify = [0] of { bit };
      byte nwait;
};

#define LOCK            0
#define UNLOCK    0
#define NOTIFY          1

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

Figure 3: Object Locking Mechanism is Promela

Figure 3 gives shows an example of a locking mechanism using channels as locks. The *nwait* variable keeps track of the number of processes that are currently blocked. The implementation of the *wait()* function is first divided into three atomic statements. The first unlocks the object by writing an *UNLOCK* message to the objects lock channel. The next statement waits for a *NOTIFY* and the third attempts to reacquire the lock for the object by writing a *LOCK* message to the channel. The *notify()* code checks for a blocked process, then sends a message on the notify channel. The command *notifyAll()* repeats this process until there are no more blocked processes for that object. Blocks of code that are declared *synchronized* in the Java can simulate the process in Promela by placing *_lock* and *_unlock* macro calls at the beginning and end of the block.

Demartini et al [6] present a further extension that allows for a single thread to reacquire the lock several times (to deal with the recursive-lock situation). This is done by adding a lock count and a process id variable (of the thread with the lock) to the lock structure.

2.4 Exceptions

Exception handling in Java is very similar to the *unless* clause of Promela. The *try/catch* Java construct can be modeled by the *unless* Promela construct for simple exception handling. When nested exception handlers are used, the Promela unless clause will 'catch' the exception with the outer-most *unless* clause. In contrast, the *try/catch* construct will 'catch' the exception with the inner-most handler. To illustrate this point more clearly figure 4 shows an example of a section of code with nested exception handlers. When an exception occurs in the *S1* routine, the *S2* method will be called to handle the exception. In the Promela code the *S3* routine will be called in the same scenario. Newer versions of SPIN have been modified with a –J (for Java) option that will make the *unless* clause run 'inside-out' to emulate the Java exception handling.

Another slight difference between the *try/catch* and *unless* clauses is the necessary conditions for activation. The *try/catch* clause requires an exception to be 'thrown'. The *unless* clause requires that a specific variable state be met (e.g. x >3). Promela does not have an equivalent statement to the Java *throw* command. Each Promela thread is thus associated with a variable (*exn_E* in Figure 4) for dealing with exceptions. This variable normally has a value of zero. To simulate a *throw* statement a throw macro is defined that sets this exception variable to a value greater than zero. A catch macro is also defined to trigger when the exception variable becomes non-zero. The catch clause also runs the exception procedure and returns the exception variable to zero. Otherwise the outer *unless* clause would also trigger.

```
try {
      try {S1} catch (E x) {S2}
}
catch (E y) { S3 }
```

```
{
      S1 unless { catch(exn_E, x, S2)}
} unless { catch(exn_E, y, S3) }
```

Figure 4: Exception Handling in Java (left) and in Promela (right)

2.5 Performance Enhancements

The main goal of translating Java source code has been for deadlock detection in concurrent software. Demartini et al [4] use special code annotations to specify variables that should appear in the formal Promela model. The remaining variables and any assignment statements that use these variables are ignored. This can be used to remove code not associated with concurrency when performing deadlock detection. When an ignored variable appears in a conditional statement, the condition is removed and the resulting Promela code contains a non-deterministic choice. The resulting model is simplified but at the same time less accurate.

Another technique for state space reduction involves analysis of the inheritance structure. When subclassing is used variables can be inherited but never actually utilized in the child class. These can effectively be ignored when translating from Java to Promela without consequence in the resulting model. This is a common scenario in object-oriented programming. Parent classes can be used as a superclass to a wide variety of child classes, requiring an extensive range of attributes.

The Promela *atomic* statement can also be used to reduce the size of the state space. This is a generalized technique for Promela code and by no means restricted to the translation of Java into Promela. The number of states in a Promela process is proportional to the number of statements in the associated *proctype* structure. Multiple statements will be treated as a single state when enclosed in an *atomic* construct. A *synchronized* block of code can be considered to be a single state, so the translated code can be placed inside an *atomic* construct. Consecutive statements accessing only local variables to a process can also be made atomic. This will cause a reduction in the state space during verification by SPIN. Section 3.1 describes this set of techniques in more detail with reference to general finite state models.

The reduction of the size of the models is important. A small reduction in the number of states can mean a large reduction in the amount of time required to perform model verification. Currently, the most effective technique for reducing the number of states is having the developer manually edit the source code to remove code irrelevant to the property to be verified. This can include removing variables used exclusively for output. While it is not a desirable trait to have to edit the model (i.e. the Java source code), in many cases it can reduce the state space of the model to a tractable size.

2.6 Extending SPIN

In order to handle many of the problems encountered with translation of Java code into Promela, Demartini et al [6,9] extended the SPIN model checker and subsequently the Promela language. The new checker, called dSPIN for dynamic SPIN, handles the problems of dynamic memory allocation and functional management. The memory management additions include pointer variables and the addition of new and delete statements to Promela. A pointer is a reference mechanism that allows the memory address of an object to be stored. An operator (*&*) was also added that returns the memory address of an existing variable. The *new* and *delete* operators allow objects to be dynamically created and destroyed. The *new* operator returns a pointer, and can be used to create any type (basic or user defined) including arrays.

dSPIN also permits function calls within the extended Promela language. This is similar to a function in most conventional programming languages. Introduced to eliminate the problems with inline and process methods, the function returns control to the calling process at the point where it was called, allowing more structured sequential programming. Variables declared inside a function have scope for that function only. An addition type exists in dSPIN called *ftype*. The *ftype* type is special type that can store the memory location of a function. The addition of the *ftype* pointer will allow for polymorphism in the Promela models by being able to associate functions with objects.

The performance of dSPIN with respect to SPIN is remarkably good. When the dSPIN checker is used to verify standard SPIN examples, there is a small tradeoff between memory overhead and completion time. dSPIN tends to use more memory but takes less time to complete the verification. Unfortunately no translator from Java to Promela for dSPIN is yet available.

Visser et al. [16] recognized a similarity between Promela proctypes and Java objects. By representing a Java object by a Promela process, the upper bound limitation on the number objects of a class created can be removed. However, object attributes are difficult to associate with a *proctype* in Promela. Visser et al [16] show a potential solution to this problem by creating an extension of SPIN that allows access to local variables of a process by another process. This is analogous to objects referring other objects. Local variables of a process represent instance variables of a class. The proposed method is only intended to show the feasibility of such an approach. Several issues, such as the visibility of variables, have not been addressed. No distinction is made between private, protected and public variables; all variables are treated as public.

While processes can contain variables to simulate objects, there are drawbacks to implementing method calls. Inline method calls are required. Because objects are represented as processes, using processes to represent methods as well can become costly, even for small programs. Constructors need to be called explicitly, unlike the *new* command in Java. Using this approach, recursive methods are again limited. Objects are categorized by being active or passive. Active objects may contain extra code in the proctype (for example, code in the *run()* method if the object is a thread). Passive objects only contain one executable statement, that being '*0*,' to block the process from terminating.

The only addition to the Promela language to use the SPIN extension was an operator (~) to allow reference to a variable within a process (see example in Figure 5 [16]). This operator works similarly to the dot operator of Java, or the dereference operator (->) found in C and C++.

2.7 V-Promela

It is briefly worth mentioning a different perspective to the problem of extracting models from source code. Many object-oriented software engineering approaches contain graphical models developed early on in the software development cycle. These models are used to develop source code. VIP [14] is an extension to Promela that provides a graphical interface for developing OO models. The formalism, V-Promela [13], is based on current OO software engineering tools. VIP can take the models developed in V-Promela and translate them into Promela code to be checked by SPIN. VIP models tend to be more abstract than Java source and have the potential to handle larger software models than these extracted from source code. Translators from Java to Promela require at least some human interaction to reduce the size and complexity of the source code. Currently, VIP allows a higher degree of automation for developing Promela code. For larger software development, V-Promela may be more beneficial than model extraction techniques. The same model used in design can be used to write code and verify

```
#define passive 0

proctype Spin_Class() {
     int concurrency;
     passive
};

proctype ActiveSpin_Class() {
     int concurrency;
     int objects;
     passive;
}
```

```
inline Spin_NEW(this, con) {
     this~concurrency = con;
}

inline ActiveSpin_NEW(this, con, obj) {
     this~concurrency = con;
     this~objects = obj;
}

init {
     Spin_Class spin_obj;
     ActiveSpin_Class activespin_obj;

     spin_obj = run Spin_Class();
     Spin_NEW(spin_obj, 1);
     activespin_obj = run
                ActiveSpin_Class();
     ActiveSpin_NEW(activespin_obj, 1,1);
}
```

Figure 5: An Example of Active Objects in Promela

properties (such as absence of deadlock). When the problem size is small or pre-existing code is being used, the overhead of developing the complete V-Promela model for SPIN gives it the same disadvantage as standard model checkers. The Java extraction techniques would be better suited to such cases to reduce the overhead of developing the separate models.

3. Building Models from Java Source Code

An alternative approach to model extraction is to build models from Java source. The difference is subtle; building models with static analysis is used to construct a graph of possible states, as opposed to trying to reproduce the model inherent in the source code in different syntactical form. The focus of research in this area is on reducing the number of possible states in the graph before verification. The paths of execution can also be modeled and checked for concurrency problems.

3.1 State Space Reductions

Each execution point of a Java program can be modeled by a state. A state is an assignment of values to a finite set of state variables. The state variables are all instance, class and local variables. Transitions are defined as maps from state to state. Assignment of a value to a variable in Java would correspond to a single transition. It is important to keep the state variables restricted to as small of a domain as possible. Corbett [3] describes a situation where a single Java *long* type without restrictions has 2^{64} possible states, and to check all of the possible states would take millions of years of computation and all existing RAM.

The transitions of a state transition system can be classified as visible or invisible. If a transition is classified as invisible it can be composed with the successor transition to produce a single transition. For example, if one invisible transition updates a value of x and the next releases a lock, then the composed transition would perform both actions in a single transition. By performing these compositions on the entire state transition system defined by the source program, a reduced transition system is defined. This reduced system is only useful if [3]

1. It is equivalent to the original transformation system for the purposes of verification.
2. It is constructable directly from the program without constructing the original transformation system.

Creation of the entire state space would allow properties to be checked without applying reductions. The amount of time and resources required to construct such a space may outweigh any benefits from verification. The resulting model may not even fit into memory. Eliminating the need to construct the complete state system is thus crucial.

A transformation is considered invisible if for some specific property to be verified the truth-value of the system with respect to that property remains unchanged. This means that the reduced transition system has to be recalculated with each property to be verified. Several techniques, collectively known as *virtual-coarsening* [2,3], can be used to reduce the original transition system.

The first transition in the Java source that can be determined as invisible is local variable reduction. Transformations where the only variables accessed are locally scoped to a particular thread may be made invisible, provided they do not change the truth-value of the property to be verified. Determining which variables are local to a particular thread is not a trivial task. There are three situations where this can occur [2]:

1. The variable is allocated on the stack, as in a formal parameter or within a method body.
2. The variable is statically allocated and referenced by at most one thread.
3. The variable is heap allocated and its associated object is accessible from one thread.

The second transformation is a lock reduction. If a variable is changed in a synchronized (i.e. locked) section of code then no other object can change that variable until the lock is released. The variable change can be considered invisible and the transition appended to the unlock transition. This situation requires knowledge of which locks are associated with which variables. The most trivial, yet still common, case is an instance variable of a class being changed inside a synchronized section of code within the same particular class. Occasionally a thread may attempt to acquire and release a lock it already has acquired. This can occur when one synchronized method calls another. The subsequent lock and unlock requests can also be made invisible. Only the outer most lock and unlock requests have any visible effect to the state of the system. As well, *notify()* or *notifyAll()* commands can be made invisible if there are no processes blocked by the *wait()* command.

3.2 Flow Analysis

Another approach to the verification problem is data flow analysis. FLAVERS (FLow Analysis for VERification of Systems) is a known tool for the analysis of concurrent ADA programs. Naumovich et al. [14] are currently extending FLAVERS to the analysis of Java programs. As was the case with translation into Promela, using FLAVERS with Java mainly deals with the detection of concurrency errors.

The flow analysis technique first creates a control flow graph (CFG) for each method in the Java source code. Each node in the CFG represents an executable statement from the Java source code. The CFGs are reduced by only allowing actions that are observable and are to be reasoned about. A trace flow graph (TFG) is then computed for the program by connecting the inlined CFGs for the methods. The start and end points of the TFG are labeled with *initial* and *final* labels respectively. The alphabet of the TFG is the set of all events associated with the TFG. The language is the set of event sequences that occur on paths from the initial node to the final node. An example of a simple TFG is given in Figure 6 [15].

Verification is done by representing properties as deterministic finite state automata (FSA) over the TFG language. This provides an over-approximation of the possible execution paths. This is not a problem when a FSA is successful in every state. If the TFG ever enters a violation state then the property fails to hold, even if the entire sequence has not yet been performed. When a transition through an FSA fails (i.e. ends in a violation state) the execution sequence must be checked for validity because it may not be a possible sequence of the TFG. This is a minor drawback to the method. Infeasible paths maybe removed by the analyst using feasibility constraints. Feasibility constraints are FSAs that provide additional restrictions to execution sequence.

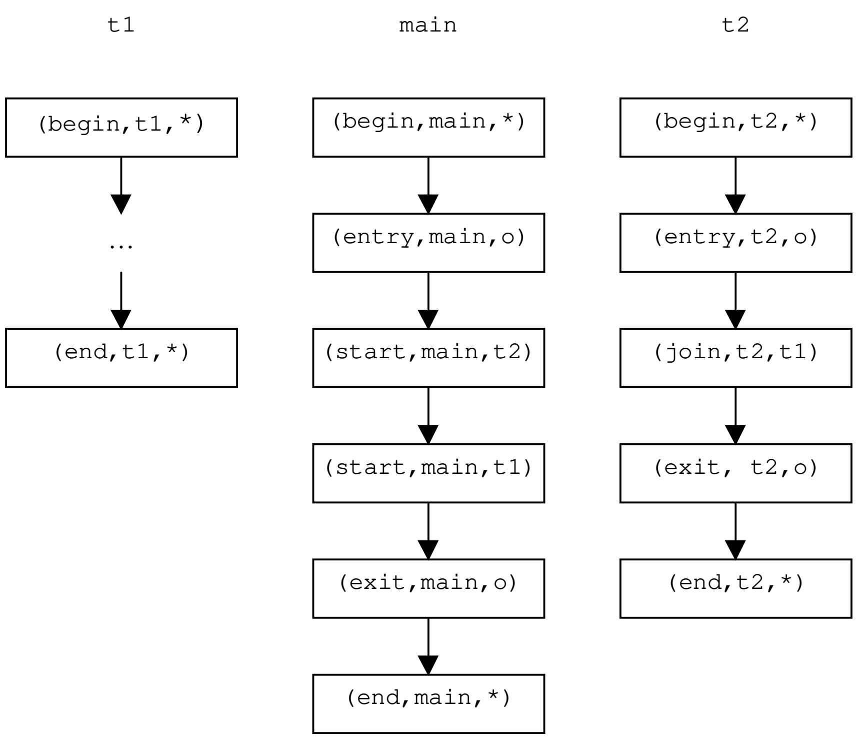

Figure 6: Example of a Simplified Trace Flow Graph for 3 Threads from [15]

Currently a complete version of Java/FLAVERS does not exist. Program models have been created by hand. Automating this method is currently being investigated. An advantage to the approach is the complexity of the algorithm, which is $O(SN^2)$. S is the size of the product of all the finite state automata used for verification and feasibility constraints. N represents the number of events of interest in the program. The low-complexity comes from the state space not being dealt with directly since only the paths of execution are modeled.

4 Discussion

Although there has been considerable advancement recently, there still is a lot of research to be done to have a full model extraction algorithm that can perform generalized verification procedures. Most research has been focused on concurrency models to detect deadlock situations. SPIN is also capable of verifying properties described in Linear Temporal Logic (LTL). Demartini et al. have specified the syntax and semantics for embedding LTL into Java source code [7]. Special annotations are used to distinguish the LTL properties inside standard Java comments. In this manner, no changes need to be made to compile the same source file. Havelund[12] suggests a special Java class (Verify) with methods to supply LTL properties. Currently, the only actual implementation is a static *assert* method in Verify, whose sole purpose is to be translated into a Promela *assert* statement. Both Java Pathfinder and Java2Spin lack the implementation of translating LTL properties to be verified by SPIN. Being able to perform verification, including deadlock detection and general verification of properties, on source code will be necessary to allow model checking to become more commonplace in software development.

For the intended purpose of deadlock detection, the models generated from Java to Promela performed reasonably well. Havelund successfully detected a deadlock situation in a game server application, although only a small subset of the code was verified. Several reductions had to be made to the game server code before a successful Promela model could be made. Havelund et al [12] also applied the Java Pathfinder to Java source code for the producer/consumer problem. SPIN found no errors in the resulting Promela code. Deadlocks and *assert* violations were then introduced into the Java source and the Java Pathfinder reapplied. The Java code was also compiled with each errors and executed to provide a comparision for the model checker. SPIN detected each error that was introduced, while the executed code detected some errors on only about half of the runs performed.

Several examples are given in [5] of Java source being translated using the Java2Spin translator and then using SPIN to perform model checking on the resulting Promela code. The most notable example given was the verification of the Jigsaw HTTP server. The original code contained 12,000 lines of Java code. The source file had to be modified to remove instances of polymorphism and verify only a given subset of Java was used. The result was the detection of an obscure deadlock. Once the error was fixed the application was verified to be correct.

To compare the Java Pathfinder and the Java2Spin translators based on their test cases would be misleading. Each performed as expected for their respective sets. The test cases given were tailored to the subset of Java that each translator was capable of handling. The results, however, should not be disregarded. Each translator showed the promise of its corresponding approach. Both showed the ability to detect deadlock in Java programs consisting of a small number of classes. The approaches are complimentary; they could be combined to encorporate a larger subset of the Java language. The Java2Spin translator could easily be extended by adding the exception handling of the Java Pathfinder. Java2Spin was able to handle larger amounts of source code more effectively. The state space reduction techniques used to achieve this could also be applied to the Java Pathfinder.

The translators also show the limitations of syntactically changing Java into Promela. Polymorphism is difficult to represent. Redundant variables increase the complexity of model, which can be expensive due to the state explosion problem. Representing a simple programming language function is difficult and costly. A lack of representation for some basic data types limits the effectiveness of the model. Demartini et al [6,9], and Visser et al [16] both provide useful extensions to SPIN. dSPIN provides several extensions to overcome problems inherent with the Java2Spin translator. The addition of functions and pointers to the Promela modeling language can be viewed as a step towards making Promela a programming language. It does, however, recognize the need to have a specialized model checker to overcome problems due to object-oriented models.

Visser et al [15] provide an extension to SPIN that recognizes this same need. In this extension bjects and processes become indistinguishable from each other; both are modeled using the Promela *proctype* construct. This is a reasonable approach for concurrent applications, but objects not based on the *Thread* class are given incorrect behavior. The only advantage of providing variable access is the upper bound limitation on the number of object instances is effectively removed. However, dSPIN contains dynamic object creation that provides the same advantage while maintaining the distinction between objects and processes. Adding variable access to processes provided no benefit to the performance of the SPIN checker. dSPIN provides a slight decrease in verification time, but there is a small tradeoff as the models tend to have more memory overhead.

Corbett [3] and Naumovich et al [15] attempted the a more general approach to program verification. Instead of working Java source into a form understandable by an existing model checker, the code is represented by a finite state model. Once Java source code can be represented in this form, verification of general properties can be performed. The intention of modeling Java programs was first and foremost detection of concurrency errors. Corbetts[3] representation could easily lend itself to verifying LTL properties. Using a prototype tool Corbett[3] was able to perform virtual coarsening on two systems (Producer/Consumer and Observer) to check for deadlock. The tool performed only simple state enumeration (model checkers such as SPIN use more efficient techniques) and several examples of unreduced transition systems that could not be verified due to memory limitations verified once a reduced model was generated. To draw a direct comparison of results between this prototype tool and Java2Spin or Java Pathfinder would be evaluating the efficiency of the model checkers rather than the approach.

Modeling Java source code by trace flow graphs has relatively low complexity, but several disadvantages. Error detection requires extra work to determine the validity of the errors produced. Developing the finite state automaton is also error prone and costly. This approach works with concurrency errors because concurrency errors can be represented by distinct set of ordered method calls from the *Thread* class. Extending this to general property verification (e.g. using Linear Temporal Logic) would be difficult. The creation of finite automaton to represent the desired properties of a program would be tedious. Current model checkers, such as SPIN, have convenient property specification syntax and semantics (e.g. CTL, LTL) but have modeling languages that are too unlike conventional programming languages. Analyzing Java code using flow analysis does not solve the problems associated with model checkers. The modeling language (Java) is convenient and familiar. However the specification language for properties becomes tedious and difficult to use. The scenario associated with standard model checkers is simply inverted.

5. Conclusion

For model checkers to be commonly used tools by programmers, familiar syntax is necessary to bridge the gap between programming languages and modeling languages. The recent attention devoted to the use of Java not only as a programming language but also as a modeling language is a step in this direction. Verification of models built or extracted from Java source code is a reality for deadlock detection, and should be for other types of verification in the foreseeable future. Translation of Java into Promela code has shown that reliable models that detect obscure errors can be constructed. Problems that are associated with the input languages can be solved by tailoring the model checking tools to the source language constructs, as is shown with dSPIN.

The major drawback is the need for manual analysis to combat the state explosion problem, but this is unavoidable in most areas of model checking. One way to combat the state explosion problem is the use of finite state models. The number of states that are represented in the finite state model can be significantly reduced using techniques such as virtual coarsening. The reduced models can either be analyzed directly or translated into a modeling language to use pre-existing model checkers.

For a successful model extractor and verifier to be created several issues need to be addressed

1. The techniques described need to be integrated.
2. Some benchmark examples needs to be defined that include deadlock scenarios of various size and situations for both trivial and non-trivial properties to have verification applied to.
3. The full Java language needs to be addressed, including polymorphism and the inclusion of all basic Java types (e.g. floating point numbers and strings).

The techniques presented in this paper show the ability to at least detect concurrency errors by model-checkers using Java source code. If the above issues are fully addressed, then a finite state verification tool that uses Java as a modeling language and performs verification on a reasonable set of properties (perhaps even a subset of LTL) will be a likely result.

References

1. T. Cattel: Modelling and Verification of sC++ Applications*, Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, *LNCS* 1384. Springer-Verlaag (April 1998), pp 232-248.

2. J. Corbett: Constructing Compact Models of Concurrent Java Programs, *Proceedings of the International Symposium on Software Testing and Analysis*, (March 1998)

3. J. Corbett: Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs, Submitted for publication, (January 1999).

4. C. Demartini, R. Iosif, and R. Sisto: Modeling and Validation of Java Multithreading Applications using SPIN, *Proceedings of the 4th Workshop on Automata Theoretic Verification with the SPIN model checker*, Paris, France (November 1998), pp 5-19.

5. C. Demartini, R. Iosif and R. Sisto: A Deadlock Detection Tool for Concurrent Java Programs, *Software: Practice & Experience*, Volume 29, Issue 7 (July 1999), pp 577-603.

6. C. Demartini, R. Iosif and R. Sisto: dSPIN: A Dynamic Extension of SPIN*, Proceedings of the 6th International SPIN Workshop*, in Theoretical and Practical Aspects of SPIN Model Checking, Toulouse, (September 1999).

7. C. Demartini, R. Iosif and R. Sisto: Specifying Linear Temporal Logic Properties In Java Source Code, Submitted for publication, (Novermber 1999).

8. M. B. Dwyer and L.A. Clarke: Data Flow Anaylsis for Verifying Properties of Concurrent Programs, Proceedings of the Second Symposium on Foundations of Software Engineering, (December 1994), pp 62-75

9. R. Iosif: The dSPIN User Manual. http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml

10. K. Havelund, J. Skakkebaek: Applying Model Checking in JAVA Verification, *LNCS* 1680. Springer-Verlag (September 1999), pp 216-231.

11. K. Havelund and T. Pressburger: Model Checking Java Programs Using Java PathFinder, to appear in *International Journal on Software Tools for Technology Transfer.*

12. K. Havelund: JavaPathfinder, A Translater from Java to Promela, *LNCS* 1680. Springer-Verlag (September 1999), pp 152.

13. S. Leue and G. Holzmann: V-Promela, A Visual, Object-Oriented Language for SPIN, *Proceedings of the Second IEEE International Symposium on Object-oriented Real-time Distributed Computing ISORC '99*, Saint Malo, France, (May 1999).

14. M. Kamel and S. Leue: VIP: A Visual Interface for Promela, *LNCS* 1680, Springer-Verlag (September 1999), pp 153.

15. G. Naumovich, G.S. Avrunin, and L.A. Clarke: Data Flow Analysis for Checking Properties of Concurrent Java Program, Proceedings of the International Conference on Software Engineering, Los Angeles, (May 1999), pp 399-410.

16. W. Visser, K. Havelund, J. Penix: Adding Active Objects to SPIN, *Proceedings of the 5th SPIN Workshop*, Trento, Italy, (July 1999).