Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking

Willem Visser and SeungJoon Park
RIACS
NASA Ames Research Center
Moffet Field, CA 94035

John Penix
Computational Sciences Division
NASA Ames Research Center
Moffet Field, CA 94035

ABSTRACT

While it is becoming more common to see model checking applied to software requirements specifications, it is seldom applied to software implementations. The Automated Software Engineering group at NASA Ames is currently investigating the use of model checking for actual source code, with the eventual goal of allowing software developers to augment traditional testing with model checking. Because model checking suffers from the state-explosion problem, one of the main hurdles for program model checking is reducing the size of the program. In this paper we investigate the use of abstraction techniques to reduce the state-space of a real-time operating system kernel written in C++. We show how informal abstraction arguments could be formalized and improved upon within the framework of predicate abstraction, a technique based on abstract interpretation. We introduce some extensions to predicate abstraction that all allow it to be used within the class-instance framework of object-oriented languages. We then demonstrate how these extensions were integrated into an abstraction tool that performs automated predicate abstraction of Java programs.

1. INTRODUCTION

Model checking is becoming an increasingly successful technique for analyzing software requirement specifications and software design models [1, 4, 9]. The main reason for this trend is that, at high levels of abstraction, the limitations of model checking are often avoided with minimal cost. This is convenient because it is well known that discovering errors early in the software life cycle is very cost-effective. However, it is also the case that some errors cannot be discovered in the requirements and design stages. This might be because the details of the system are not elaborated to sufficient detail to reveal problems until implementation, or simply because errors are made during implementation.

The state of the art techniques for finding errors at the implementation level are static analysis and testing. However, testing is not well suited to giving high degrees of behavioral coverage of a system, especially when considering a concurrent system where testing has little, or no, control over the scheduler. Static analysis has better success dealing with concurrency, but it is sometimes challenging to obtain accurate results [16]. Model checking, however, can provide more extensive behavioral coverage in two ways. First, the model checker can evaluate every possible interleaving of concurrent processes/threads in the system. Second, model checkers support nondeterministic operations that can be used to construct an environment model to close a system for verification. This can allow the model checker to generate all combinations of environmental behaviors as the closed system is checked.

There have been many cases of applying model checking to programs. However, the standard approach is to extract relevant portions of the code, create a model of its behavior and then check the model. This has the drawback that the modeling activity requires expertise in the use of the model checking tools and will not, in general, allow software developers to check their own code during development. Ideally, one would like a model checker that could operate directly on the source program to find the errors that testing might miss.

There are two obvious problems with applying model checking to programs: getting the program source code into the notation of your favorite model checker, and avoiding the state-space explosion problem. The former is not a fundamental problem and some work has been done to create automatic translations from popular programming languages to the input notations of model checkers [12, 15, 7]. We used one such approach to translating object-oriented software to Promela, the input notation of the Spin model checker [14], for the work described in this paper. Avoiding the state space explosion problem is more difficult than the translation problem. The implementation detail in the program that was not present in the design now is a doubly edged sword: on one side it is required to find some errors, but on the other it may cause the model checking to exhaust memory resources before finding any errors.

Abstraction has long been a favored method for reducing the state-space size of a system to allow efficient model checking [5]. Abstraction techniques are often based on abstract interpretation [8] and require a user to give an abstraction function relating concrete datatypes to abstract datatypes. One approach is *predicate abstraction*, where an abstraction function is specified as a number of predicates over the concrete data [11]. For example, one might be interested in whether an integer x is positive, negative or zero and hence the following predicates will be used to represent the state space of the abstract system: $x > 0$, $x < 0$ and $x = 0$.

In this paper, we report on an investigation into the use of abstraction to reduce the state space of a software system written in C++ (+-3000 lines). To the best of our knowledge, this is the first time predicate abstraction has been applied to a real software system at the source code level. In section 2 we give an overview of the abstraction techniques used in the paper. Section 3 contains a brief description of the DEOS real-time operating system as well as the approach we took to model check the system. The remaining sections contain the major contribution of this paper, namely the application of predicate abstraction to the DEOS model as well as the introduction of dynamic predicate abstraction and its influence on the DEOS verification. We then show that dynamic data manipulation cannot be handled by automated (static) predicate abstraction techniques and we then introduce *dynamic predicate abstraction* to solve this problem. Section 6 describes the integration of dynamic predicate abstraction into an automatic abstraction tool for Java. We then demonstrate the use of this tool on another real program example. Section 7 contains concluding remarks and points out future areas of research.

2. ABSTRACTION FOR VERIFICATION

Program abstraction is a very general concept with many potential applications in program analysis, compilation and verification. A common application of program abstraction in verification is to reduce the complexity of a program in order to make a verification algorithm more tractable. In the specific case of model checking, abstractions are used to reduce the state-space size of a program in an attempt to overcome the memory limitations of model checking algorithms.

There are two main approaches to model checking using abstraction:

- an abstract state graph can be generated for model checking by executing the concrete transitions over the abstract data, or
- the concrete transitions can be abstracted and the resulting abstract system can be model checked.

Recently there has been much work in automating both these approaches by using decision procedures to determine the abstract states and/or abstract transitions given the abstraction function, initial state(s) and concrete transitions [2, 6, 10, 11, 19, 20]. In the approach where abstract transitions are generated, the number of calls to the decision procedures is bound by the size of the concrete system. The abstract state graph approach is a dynamic abstraction, and hence, will in most cases require many more calls to the decision procedures. The abstract state graph approach can however be more precise since it can use dynamic information about the abstract state-space to generate abstractions.

2.1 Property Preservation

When using abstraction to assist verification, the main concern is that the abstractions must be *property-preserving.* There are two forms of property preservation:

Weak Preservation – An abstraction is a weakly preserving abstraction of a concrete system if a set of properties true in the abstract system has corresponding properties in the concrete system that are also true.

Strong Preservation – An abstraction is a strong preserving abstraction if a set of properties with truth-values either true or false in the abstract system has corresponding properties in the concrete system with the same truth-values.

Note that strong preservation does not seem to allow for much scope in simplifying the system during abstraction. However, property preservation is with respect to a specific set of properties and properties outside of the set can be disregarded. In fact, an abstraction is often applied in order to preserve a single property. Therefore, strong preservation is often useful in practice. But, as might be expected, abstractions that are only weakly preserving can be much more aggressive in reducing the state-space and therefore are more popular for verification purposes.

2.2 Over and Under Approximation

Over-approximation of the behaviors of the system occurs when more behaviors are added than were present in the concrete system. This approach provides a very popular class of weakly preserving abstractions for universally quantified path properties (for example LTL properties). Weak preservation in this case follows trivially: if more behaviors (i.e. more execution paths) are added and a property is true for all paths then it is true for any subset of those paths, including the subset that describes the behavior of the concrete system. Unfortunately, over-approximations often only work well for invariant properties, since liveness properties may be erroneously invalidated by one of the extra paths that was added. This highlights a drawback of over-approximation, it can add behaviors that invalidate a property in the abstract system that is true of the concrete. These spurious errors must then be removed by constraining the over-approximation, i.e. reduce the degree of over-approximation. This is often difficult and is the reason why invariant properties are the properties of choice when doing abstraction with an over-approximation of the system behavior. The problem encountered in dealing with over-abstraction are analogous to the problems encountered when attempting to make static analysis more precise [21].

Under-approximation, i.e. where behaviors are removed when going from the concrete to the abstract system, is clearly not weakly preserving for invariant properties, but is often used during model checking of such properties. This is because it can often be shown that the behaviors removed by under-approximation do not influence the verification result. Typical ways of making under-approximations are to reduce a buffer size, restrict the number of processes in a system, etc. Under-approximations are also often found in the construction of an environment for a system to be checked: for example, checking the system with just 5 input values instead of infinitely many. In fact, testing is a under-approximation of this type. In practice it is difficult to determine whether an under-approximation in the environment might influence the truth-value of a property.

2.3 Predicate Abstraction

Predicate abstraction, introduced by Graf and Saidi [11], is a popular form of over-approximation and forms the basis of a number of automated abstraction tools [10, 19, 20]. The basic idea of predicate abstraction is to replace a concrete variable by a boolean variable that evaluates to a given boolean formula (a predicate) over the original variable. This concept is easily extended to handle multiple predicates and, more interestingly, predicates over multiple variables. For example, assume we have a program with two integer variables, x and y, which can grow infinitely. Since this program will have an infinite state-space, model checking cannot be complete in general, although a property might be (in)validated in a finite portion of the state-space. However, closer inspection may reveal that the only relationship of interest between the two variables is whether or not they are equal. We can then define a predicate to represent this relationship, $B_0 : x == y$, and use it to construct an over-approximation of the system's behavior as follows: wherever the condition $x == y$ appears in the program we replace it with the predicate B_0, and whenever their is an operation involving x or y we replace it with an operation changing the value of B_0 appropriately.

Replacing concrete transitions with abstract transitions, can be performed automatically with the aid of decision procedures [2]. Furthermore, it can take place dynamically during state generation [10, 19] or statically before state generation begins [6, 20]. In both approaches, over-approximation occurs when not enough information is available for the decision procedure to calculate a deterministic next action or state. For example, the operation $y := y + 1$ could lead to two nondeterministic abstract transitions: $B_0 := false$ or $B_0 := true$, because if B_0 is false, it is unknown whether it should become $true$ (if $y = x - 1$). In the abstract state graph approach, since it is dynamic, this information may be available, hence showing one area in which this approach is better than the transition generation approach.

Invariants of the system can often allow more precise abstractions (i.e. less nondeterminism, and hence less over-approximation). For example, the invariant $x \leq y$ would allow the concrete transition $y := y + 1$ always to be abstracted to $B := false$. Note that this is an interesting case in predicate abstraction: when the predicate abstraction introduces no nondeterminism, over-approximation does not occur and strong preservation is achieved [19]. One might believe that this would be a rare occurrence, but we show (in section 4) that a specific class of infinite state programs that occur frequently in practice can be transformed to finite state programs by a predicate abstraction that introduces no nondeterminism.

3. DEOS VERIFICATION

The Honeywell DEOS operating system is a portable micro-kernel based real-time operating system which provides both space and time partitioning between applications. The enforcement of time partitioning involves innumerable interleavings of program executions due to the rich set of scheduling primitives supported by DEOS. The developers understood from the beginning of the DEOS development that testing was going to be inadequate for ensuring the correctness of the scheduler. This led to a collaboration between NASA Ames and Honeywell to investigate the application of model checking to assist the verification of DEOS [18]. A slice of the scheduler code including one of the most subtle errors detected during the DEOS development was selected by Honeywell and delivered to NASA for analysis.

Because our ultimate goal was to integrate our verification technique as part of the software development process, we decided to apply model checking directly on the source code, without first extracting a model of the system. We adopted a translation scheme similar to the one used within the Java PathFinder tool [12]. The important point of our translation is that we translated the code line for line, such that there is nearly a one-to-one mapping between the C++ and the Promela code[1].

DEOS threads can run in different *scheduling periods* and require a certain amount of CPU time to be allocated to them, called their *budget*, within their period. The scheduling period and budget of each thread is fixed at thread creation. During execution if a thread uses more time than its remaining budget then it will be interrupted by the kernel and a new thread will be scheduled. The property that we wanted to verify within DEOS was that of *time partitioning*: that all threads in the system are guaranteed their requested budget within their scheduling period.

3.1 Environment Modeling

In order to model check DEOS, we had to define an environment for the kernel to execute in. Although environment modeling is not the main focus of this paper, we include this short discussion to provide context for the remainder of the paper and to highlight the fact that environment modeling is a critical task involving abstraction which proves to be quite difficult in practice.

The environment for DEOS had to contain a model of the threads that were to be scheduled and a model of the hardware that was needed to provide interrupts. In our version of the kernel, a thread could either delete itself, yield the CPU or continue running until interrupted. In the last case, the kernel will stop a thread if it uses all of its allocated budget. Therefore, we simply allowed a thread to make a nondeterministic choice between these three options. This model of a thread is an over-approximation of a thread's behavior and will preserve any time partitioning errors that might exist in the kernel.

The model of the hardware component of the system consists of a system tick generator and a timer. The system tick must be generated periodically and the timer is used to determine when a thread has used up its budget. To provide an accurate model of the environment, these two events must be synchronized. Otherwise, it becomes possi-

[1] Due to certain Promela limitations, some C++ statements would translate to more than one statement in Promela.

ble for a system tick to occur before a timer interrupt, even though the thread requests less time than is remaining in the scheduling period. To deal with the large number of potential execution times after which a thread could yield or delete itself, we used the following under-approximation: whenever the kernel requests the remaining time, the timer chooses nondeterministically one of the following values: 0, the time the timer was started with, or a value halfway between these. This abstraction of time worked well for this verification effort, but it is difficult to assess whether it preserves all time partitioning errors. We are currently investigating techniques to support environment modeling and abstraction for DEOS.

Although we realize that generating the environment for DEOS was made more difficult due to the lack of real-time support in Spin, we still argue that our experience indicates that environment modeling is non-trivial when considering complex systems. On the one hand, over-approximations allow a clean way of introducing property preserving abstractions, but on the other hand they introduce many spurious errors that would hamper its uptake in the world of everyday software developers. Our solution was to reduce the spurious errors by introducing possibly unsafe under-approximations.

3.2 Verification Results

Although we were told there was a time partitioning error in the system, we often found ourselves considering the option that the Honeywell team might have accidentally removed it from the code that they gave us. As it turned out, the real problem was our attitude towards the state-space explosion: we were constantly under-approximating the system behavior (by disabling features such as dynamic thread creation and deletion) to avoid running out of memory. These under-approximations however did not preserve the time partitioning error.

After realizing that the under-approximations were probably unsafe, we changed our approach from exhaustive verification of some part of the system behavior to looking for errors only up to a certain depth of execution. On the very first model checking run with 4 threads to schedule and each with all of its possible behaviors enabled, the error was found. The error only occurred if a thread deleted itself after using a large amount of its allocated budget. At this point the deleted thread's budget is returned to the *main* thread in the process, which had the option of using this budget again. Hence the budget was used twice, causing another thread to be starved and time partitioning to fail. Spin can find this error at a minimum depth of 2556 steps.

4. PREDICATE ABSTRACTION FOR DEOS

Although we rediscovered the time partitioning error without introducing any abstractions within the DEOS code itself (all our abstractions were in the environment) we were still unsatisfied for several reasons. First, we were not guaranteed to discover the error with the approach we used because the model could not be exhaustively checked, even to a depth of 2500 transitions[2]. Second, we were interested in checking other properties of the system that would required

[2]A transition corresponds roughly to a program statement due to our nearly 1-to-1 translation.

```
void
StartOfPeriodEvent::pulseEvent( DWORD systemTickCount )
{
  countDown = countDown - 1;
  if ( countDown == 0 )
  {
    itsPeriodId = itsPeriodId + 1;
  ...
}

void
Thread::startChargingCPUTime()
{
  // Cache current period for multiple uses here.
  periodIdentification cp = itsPeriodicEvent->currentPeriod();
  ...
  // Has this thread run in this period?
  if ( cp == itsLastExecution )
  {
    // Not a new period.  Use whatever budget is remaining.
    ...
  }
  else
  {
    // New period, get fresh budgets.
    ...
    // Record that we have run in this period.
    itsLastExecution = cp;
    ...
  }
  ...
}
```

Figure 1: Slice for `itsPeriodId`

an exhaustive check of the state-space of the system. For example, Honeywell sent us the "fix" for the error to verify that it indeed resolved the problem. It was clear that we needed to find abstractions for some parts of the DEOS code in order to do exhaustive verification. In this section we related the steps we took to first introduce and ad-hoc abstraction into the system, and then to refine the abstraction using predicate abstraction.

4.1 Ad-hoc Abstraction

The first step toward introducing abstractions into the system was to determine whether their actually were any chances to reduce the state space of the system without grossly under-approximating the system behaviors. We were guided by several experiments showing traces through the system that were 2,000,000 steps long. Based upon our limited intuition of how DEOS works, this seemed too large because the system's behavior is cyclic in nature: at the end of the longest scheduling period, the system should return to a state where all threads are available to be scheduled with all of their budget available. These extremely long traces indicated that some data was being carried over these longest period boundaries. We were able to identify this data by running a simulation and observing the Spin data values panel; The `itsPeriodId` data member for the `StartOfPeriodEvent` class was operating as a counter, incrementing every time the end of the corresponding period was reached. In addition, the `itsLastExecution` variable in the Thread class was also climbing, because it is periodically assigned the value of the `itsPeriodId` counter for the `StartOfPeriodEvent` corresponding to the thread's scheduling period.

The section of the DEOS kernel involving `itsPeriodId` and `itsLastExection` is shown in Figure 1. These variables are used to determine whether or not a thread has executed in the current period; If it has not, then its budget can be safely reset. When a thread starts running, `itsLastExecution` is assigned the value of `itsPeriodId` (the return value of `currentPeriod()`) whenever the two are not equal. Therefore, `itsLastExecution` will always increase by *exactly* one if a thread is scheduled every period. If this is true, then both variable types can be replaced with much smaller ranges (namely bits) and still maintain the exact behavior of the system.

Whether or not a thread is scheduled every period is very difficult (if not impossible) to determine by inspection, because it is control flow dependent upon the core of the scheduling code. To test this hypothesis, we placed an assertion in the code to check that `itsLastExecution` was only incrementing by one. Spin failed to find an assertion violation to a search depth of 76,000, but could not exhaustively verify the system due to the same memory limitation that we were attempting to circumvent. We then changed the increment code for the `PeriodId` to roll over at 8 (rather than 256) and ran the verification again. This resulted in the assertion being violated at roll over (because 0=8+1 is false). While this result is not a complete confirmation, it was consistent with our belief that that `itsLastExecution` only increments and increased our level of confidence in our code inspection.

Assuming that `itsLastExecution` can only increment by one, we can safely use modulo 2 arithmetic (i.e. a bit) to determine whether the thread has executed in the current period. Therefore, we changed the `PeriodId` code to roll over at 2. Table 1 shows formal characterization of this abstraction, where the effect of using modulo 2 arithmetic is modeled by toggling a boolean variable. With this abstraction in place, we found that the number of states dropped to under one million (980197) and we could now exhaustively verify a system with one user thread using 260MB of memory.

4.2 Predicate Abstraction

It turns out that, in the slice of the DEOS system being verified a thread runs every period. However, in the full DEOS system there are synchronization mechanisms, such as events and semaphores, that may cause threads to wait for arbitrary amounts of time. In this case, our assumption that a thread will execute every period, and consequently the preservation property of the abstraction, breaks down. Therefore, a more general solution was required if the abstraction was to be used in a broader context.

Discussions with Honeywell revealed that the information that actually needs to be maintained is simply a boolean variable that indicates whether a thread has executed in the current period. These flags would then be reset at every period boundary. However, this approach can not be implemented in the system for efficiency reasons: all kernel algorithms must be O(1), where as resetting the flags is O(n), where n is the number of threads.

This realization led us to try predicate abstraction. We replaced the variables `itsPeriodId` and `itsLastExecution` by a single boolean variable, `executedThisPeriod`, defined by the predicate `itsPeriodId` == `itsLastExecution`.

To generate an abstract program, the statements that manipulate the variables must map to statements that properly update the predicate variable. In this case, it is obvious that the statement `itsLastExecution = itsPeriodId` should be mapped to `executedThisPeriod = TRUE`. However, the mapping for the program statement `itsPeriodId = itsPeriodId + 1` is nontrivial because, depending on the previous values of `itsPeriodId` and `itsLastExecution`, the value of the predicate after the increment could be either `TRUE` or `FALSE`. However, in the real system, `itsPeriodId` is always incremented, and `itsLastExecution` is only ever assigned the value of `itsPeriodId`. Therefore, it is easy to “prove” (by inspection of the code in Figure 1) that `itsPeriodId` will always be greater than or equal to the value of `itsLastExecution` and therefore the result of incrementing `itsPeriodId` will be that the predicate becomes `FALSE`. This abstraction mapping is shown in Table 2.

In practice, the case where `itsPeriodId` rolls over (at MAXINT) is an exception to the above assumption. However, the correct behavior of the real system implementation also depends on this assumption (specifically, that `itsPeriodId` does not roll over and catch up with `itsLastExecution`, meaning that a thread will not wait MAXINT periods). This is precisely the case where the above predicate abstraction will become invalid. Therefore, this abstraction does not introduced any stronger assumptions on the system than those imposed by the implementation and is therefore a strongly preserving abstraction of the code.

4.3 Implementation Details

The reasoning described above to determine the abstract program statements for DEOS is quite formal, and can be automated using existing program abstraction tools. However, due to the object-oriented nature of the program, the actual abstraction that was required was slightly more complex than the mapping in Table 2 and existing tools could not be used to generate the abstract program. The problem arises because the C++ code describes classes while the state space that we are attempting to abstract is composed of instances of these classes. Existing abstraction techniques do not address this distinction between class and instance variables and this becomes problematic in practice. The fact that there are multiple instances of the classes that we want to abstract means there must be multiple predicates introduced to perform the abstraction and that abstracted program statements must be generated that properly manipulate all of these predicates at once.

The specific problem in DEOS was the one-to-many relationship between `StartOfPeriodEvents` and `Threads` because more than one thread can execute within a period. Therefore, a predicate was required for each `Thread` instance to store the relationship between `itsLastExecution` and the `itsPeriodId` of its `StartOfPeriodEvent`. When constructing the abstract program, we need to know which of these predicates should be updated when either `itsPeriodId` or `itsLastExecution` is updated. From the perspective of the thread this is simple because there is only one predicate per thread. However, from the perspective of the period, this meant that whenever the `itsPeriodId` was incremented we needed to go through all the threads that can run in that period and update the appropriate predicates.

Concrete Program	Abstract Program
int itsPeriodId; int itsLastExecution;	bool itsPeriodId; bool itsLastExecution;
itsPeriodId = itsPeriodId + 1;	itsPeriodId = !itsPeriodId;
itsLastExecution = itsPeriodId;	itsLastExecution = itsPeriodId;

Table 1: Abstraction of `itsPeriodId` and `itsLastExectution` to booleans

Concrete Program	Abstract Program
int itsPeriodId; int itsLastExecution;	bool executedThisPeriod;
itsPeriodId = itsPeriodId + 1;	executedThisPeriod = FALSE;
itsLastExecution = itsPeriodId;	executedThisPeriod = TRUE;

Table 2: Abstraction of `itsPeriodId` and `itsLastExectution` to a single boolean

This is a precise approximation of the original system behavior because the predicates are always assigned to false if `itsPeriodId` is incremented and true when the assignment `itsLastExecution = itsPeriodId` is executed. Note that this algorithm corresponds precisely to the O(n) updating algorithm that could not be used in the implementation. However, the O(1) real-time constraint does not apply to the verification model, so this is not a problem.

This predicate abstraction allowed us to exhaustively check the original (faulty) system as well as the fixed system for the configuration with 4 threads with their full behavior. We found another error on the very first run with the fixed software, which turned out to be an error that the DEOS engineers already found themselves and had corrected. Unlike the original time partitioning error this second error could have been found easily with traditional testing.

4.4 The Event-Counter Pattern

The target of our abstraction within DEOS was an instance of a more general pattern where a counter is used to indicate that an event has occurred. This is a form of *time-stamping* that is common in distributed and database programming. We believe that finding a precise approximation to remove the infinite behavior associated with this common pattern could be an important result in the quest to model check programs. In fact, this event-counter pattern was also used within NASA's Deep Space 1 Remote Agent control system, which we have also used as a case study for program verification. In Section 6 we show how this same abstraction can enable model checking to discover a deadlock in the program that actually occurred during flight [13].

In there general case, this pattern consists of an `Event` class containing a counter and any number of other `Listener` classes which monitor the occurrence of events by keeping a local copy of the event counter and periodically comparing the two values. This general case removes several simplifying assumptions that occurred in the DEOS system, most importantly that listeners can monitor more than one event. In the following sections, we describe a generalization our approach used for DEOS that can be used to support automatic generation of abstract programs for these types of object-oriented abstractions.

5. DYNAMIC PREDICATE ABSTRACTION

One drawback of the abstraction approach described in the previous section is that it required a fairly deep understanding of the system to be able to introduce the code to achieve the predicate abstraction. Ideally, one would want to simply identify the abstraction predicate and the verification system would automatically create the abstract transitions. This is how the current automated predicate abstraction tools work, but in these systems the predicates relate *static* variables, whereas in our case the predicate relates variables from different objects that get created *dynamically*.

We propose the use of dynamic predicates in order to perform predicate abstractions in programs that uses dynamic data. Dynamic predicates are simply predicates augmented with dynamic information. For example, if we want to create a predicate abstraction for the program statement $q.x == r.y$, where q is an object of class Q and r is an object of class R, we use a dynamic predicate $B : (q, r, q.x == r.y)$ which not only holds the static predicate but also the dynamic information relating the specific objects q and r. Note that for each program statement, there will typically be many dynamic predicates in each abstract program. In the above example involving two dynamic objects we would require in the worst-case $|Q| \times |R|$ dynamic predicates, where $|Q|$ and $|R|$ refers to the number of objects instantiated of class Q and class R, respectively.

To calculate the abstract transitions for the dynamic predicates, we use the techniques for generating abstract transitions for static predicates and then augment the resulting transitions with information available during run-time. For the $B : q.x == r.y$ example from above, decision procedures are used to calculate the following abstract transitions from the corresponding concrete transitions (assuming that there is an invariant that states $q.x \leq r.y$ hence there is no non-deterministic choice for the third statement):

$$q.x == r.y \quad \rightarrow \quad B \qquad (1)$$

$$q.x := r.y \quad \rightarrow \quad B := true \qquad (2)$$

$$r.y := r.y + 1 \quad \rightarrow \quad B := false \qquad (3)$$

For each of the above abstract program statements, we require a "wrapper" function in order for it to be used dynamically. Lets assume we have a list of dynamic predicates with each predicate having the following structure: (q, r, B) where B is the predicate from above. We replace the statements $q.x == r.y$ with the function $isEqual(q, r)$ defined by the following pseudo-code:

```
boolean isEqual(qq,rr) {
  find the dynamic predicate with (q==qq) and (r==rr)
  return the value of B for this predicate
}
```

The statements $q.x := r.y$ are replaced by the function $setEqual(q, r)$:

```
void setEqual(qq,rr) {
  find the dynamic predicate with (q==qq) and (r==rr)
  set the B field for this entry to true
}
```

Lastly the statements for $r.y := r.y + 1$ is replaced by the function $inc(r)$:

```
void inc(rr) {
  find all the dynamic predicates with (r==rr)
    for each one set the B field to false
}
```

The third line of each of these functions is taken directly from the translations calculated (by the use of decision procedures) from the static predicates. This provides a straightforward way of allow the use of predicate abstraction in a dynamic setting.

One potential inefficiency of this approach is that it may introduce too many predicates. For example, if we have 10 objects of class Q and 10 from class R then there will be 100 dynamic predicates for $q.x == r.y$. However, it might be the case that during program execution only 1 of the objects of class Q interacts with all the objects of class R and therefore only 10 dynamic predicates were really required. This is the case in DEOS where threads monitor a single start of period event during program execution. We believe this problem can be overcome by either allowing more user interaction or using static analysis techniques to determine which objects interact and hence can be used to minimize the number of dynamic predicates. One potential source of information that can be used to refine the abstractions may be from UML class diagrams. The fact that DEOS threads only monitor one event can be expressed using multiplicity constraints in UML.

Description	States	Time (s)	Memory (Mb)
Manual	1255010	100	126
Dynamic (con)	1670880	143	135
Dynamic (opt)	1291510	113	126

Table 3: Predicate Abstractions for DEOS

The results of applying the different predicate abstraction techniques to the DEOS kernel is shown in Table 3. The table shows the number of states visited, space requirements and time taken during a deadlock detection run, during which Spin generates the full state space of the system. The DEOS system configuration used to obtain these results was four schedulable threads with their full capabilities enabled. The results were obtained on a SUN ULTRA60 with 512Mb of memory. The Manual abstraction refers to the predicate abstraction described in Section 4 where a considerable amount of user intervention was required. Next we considered a dynamic predicate abstraction that conservatively creates dynamic predicates for all possible thread and period objects, regardless of whether the thread can execute within the period. Finally, we look at an optimization of the conservative approach that only creates dynamic predicates for the thread and period combinations that actually occur within the program. The results indicate that the optimized dynamic abstraction performs almost as well as the method that requires considerable user intervention.

6. AUTOMATED TOOL SUPPORT

In the preceding sections we showed how predicate abstractions can make model checking tractable when analyzing object-oriented programs. Although we alluded to the fact that we believe the dynamic predicate abstractions can be automated, all the work was done by hand. In this section we describe an automated abstraction tool, which converts a Java program to an abstract program with respect to user-specified abstraction criteria. We will illustrate how to use the tool on an example derived from flight software used within the NASA Deep Space 1 Remote Agent experiment [17][3]. The Java program, given in Figure 2, is a fragment of code translated from the original Lisp code that illustrates a deadlock that happened during flight [13]. In fact, the program in Figure 2 makes use of the same event-counter pattern encountered within DEOS and therefore has an unbounded state-space.

To interact with the abstraction tool, a user specifies abstractions by removing variables in the concrete program and/or adding new variables (currently the tool only supports adding boolean types) to the abstract program. This is illustrated by the calls to the methods `Abstract.remove` and `Abstract.addBoolean` in Figure 2. Given a Java program and such abstraction criteria, the tool generates an abstract Java program in terms of the new abstract variables and remaining concrete variables. Part of the abstract Java program for Figure 2 is shown in Figure 3. To compute the conversion automatically, we use a decision procedure, SVC (Stanford Validity Checker), which checks the validity of logical expressions [2]. The tool extracts information from the concrete Java program during parsing, translates the Java statements to SVC notation to determine what the abstract statements should look like and translates the results back to Java. In the translation back to Java the tool embeds the SVC results inside wrapper code, see Figure 3, to handle the dynamic nature of the predicates as described in Section 5.

The abstraction tool was designed to be used as a front-end tool for our Java model checking tools. Java PathFinder1

[3]We cannot demonstrate the tool on DEOS because DEOS in written in C++ and the code of interest cannot be fully published.

```
class Event{
  int count = 0;
  public synchronized void wait_for_event(){
    try{wait();}catch(InterruptedException e){};
  }
  public synchronized void signal_event(){
    Abstract.remove(count);
    count = count + 1; notifyAll();
} }

class FirstTask extends Thread{
  Event event1,event2;
  int count = 0;
  public FirstTask(Event e1, Event e2){
    this.event1 = e1; this.event2 = e2;
    Abstract.remove(count);
    Abstract.addBoolean("FcntEQEcnt", count == event1.count);
    Abstract.addBoolean("FcntLEEcnt", count <= event1.count);
  }
  public void run(){
    count = event1.count;
    while(true){
      if (count == event1.count)
        event1.wait_for_event();
      count = event1.count;
      event2.signal_event();
} } }

class SecondTask extends Thread{
  Event event1,event2;
  int count = 0;
  public SecondTask(Event e1, Event e2){
    this.event1 = e1; this.event2 = e2;
    Abstract.remove(count);
    Abstract.addBoolean("ScntEQEcnt", count == event2.count);
    Abstract.addBoolean("ScntLEEcnt", count <= event2.count);
  }
  public void run(){
    count = event2.count;
    while(true){
        event1.signal_event();
        if (count == event2.count)
          event2.wait_for_event();
        count = event2.count;
} } }

class START{
  public static void main(String[] args){
    Event event1 = new Event();
    Event event2 = new Event();
    FirstTask  task1 = new FirstTask(event1,event2);
    SecondTask task2 = new SecondTask(event1,event2);
    task1.start(); task2.start();
} }
```

Figure 2: Example Java Program with a Deadlock

```
class Event{
  public synchronized void wait_for_event(){
    // Abstract.remove(count);
    try { wait();} catch(InterruptedException e){}
  }
  public synchronized void signal_event(){
    Verify.beginAtomic(); // model checker trap
    // count = count + 1;
    FcntEQEcnt.incEcnt(this); FcntLEEcnt.incEcnt(this);
    ScntEQEcnt.incEcnt(this); ScntLEEcnt.incEcnt(this);
    Verify.endAtomic();   // model checker trap
    notifyAll();
  }
  Event() {
    FcntEQEcnt.setEvent(this); FcntLEEcnt.setEvent(this);
    ScntEQEcnt.setEvent(this); ScntLEEcnt.setEvent(this);
} }

// Class defined for a new multi-class
// abstraction variable
class FcntEQEcnt {
  static final int MAX = 3;
  static public int numFirstTask = 0;
  static public FirstTask[] objFirstTask
                            = new FirstTask[MAX];
  static public void setFirstTask(FirstTask obj){
    objFirstTask[numFirstTask++] = obj;
  }
  static public int getFirstTask(FirstTask obj){
    for(int i = 0; i < numFirstTask; ++i)
      if(obj == objFirstTask[i]) return i;
    return MAX + 1;
  }
  static public int numEvent = 0;
  static public Event[] objEvent = new Event[MAX];
  static public void setEvent(Event obj){
    objEvent[numEvent++] = obj;
  }
  static public int getEvent(Event obj){
    for(int i = 0; i < numEvent; ++i)
      if(obj == objEvent[i]) return i;
    return MAX + 1;
  }
  static public boolean[][] pred
                            = new boolean[MAX][MAX];

  static public void incEcnt(Event event) {
    for(int i = 0; i < numFirstTask; ++i){
      // pre-image calculation
      if (pred[i][getEvent(event)] ||
          FcntLEEcnt.pred[i][getEvent(event)])
        // ``false'' generated by SVC
        pred[i][getEvent(event)] = false;
      else
        // nondeterminism generated by SVC
        pred[i][getEvent(event)] = Verify.randomBool();
  } }
  static public void setEqual(FirstTask task,
                              Event event) {
    for(int i = 0; i < numEvent; ++i){
      if (i == getEvent(event))
        // ``true'' generated by SVC
        pred[getFirstTask(task)][i] = true;
      else
        // nondeterminism generated by SVC
        pred[getFirstTask(task)][i] =
                                 Verify.randomBool();
  } }
  static public boolean isEqual(FirstTask task,
                                Event event) {
    return pred[getFirstTask(task)][getEvent(event)];
} }
```

Figure 3: Section of Abstracted Java Program

(JPF1) is based on a translation from Java to Spin [12]. Currently we are working on Java PathFinder2 (JPF2) which is a model checker built on top of our own Java virtual machine [3]. Since the abstraction tool does a source to source translation, both JPF1 and JPF2 can use it as a front-end. Java does not support nondeterminism, so our model checkers trap special method calls, `random(n)` and `randomBool()` from the `Verify` class to introduce respectively nondeterministic values between 0 and n or *true* and *false*. Since the abstraction tool currently only support predicate abstraction it relies heavily on the `randomBool` method call to introduce over-approximations). Furthermore, the abstract Java code often contains more statements than the concrete program, hence in order to assure correctness, all the abstract code related to one statement in the concrete program must be placed between `beginAtomic()` and `endAtomic()` methods that are also trapped by the model checkers to ensure atomicity (see Figure 3).

When the Java program in Figure 2 is checked for deadlock with JPF1/JPF2, no result is obtained since the model checker runs out of memory. This is due to the fact that the program has a very large state-space caused by the incrementing of the `count` variables when an event is signaled. However, when it is observed that the `count` variables within the program are really only used to test equality, predicate abstractions can be used to reduce the state-space. Unfortunately, if only the equality predicates are introduced, over-approximation can introduce a spurious deadlock: the `count = count + 1` statements can set the predicates to `true` or `false` nondeterministically and hence make both `FirstTask` and `SecondTask` wait for a signal. To avoid this problem, the user must notice that the Tasks' count variables will always be less or equal to the Events' count variables and add these predicates to the system, as shown in figure 2. This allows the abstraction tool to refine the abstraction and remove the over-approximations. The reason for the refinement lies in the tool's use of pre-image calculations to guard the execution of the abstract Java program. For example, the abstract statement corresponding to `count = count + 1` first tests whether the less-or-equal predicate holds and if so sets the equality predicate to false. Otherwise it picks a nondeterministic value (see comments in `incEcnt` methods in the code of Figure 3). JPF2 finds the deadlock in the abstract program in in 184 steps (transitions) from the initial state.

7. CONCLUSIONS AND FUTURE WORK

The results of this investigation have shown that by extending predicate abstraction techniques to support object-oriented languages, they can be very effective in practice for reducing programs for model checking. Because predicate abstraction is a very general (and somewhat simple) technique, we believe that it will be applicable to many other programming patterns beyond the event-counter pattern that we have shown here. In general, the framework of abstract interpretation on which predicate abstraction is based allows abstraction to be applied in a controlled and minimal fashion, which helps to provide an understanding of exactly what can be done to avoid state space explosion.

We are continuing to work on extending the applicability of predicate abstraction and to integrate it with related abstraction techniques [7]. We are also planning to investigate the use of predicate abstraction to support environment generation, because this is usually the most time consuming aspect of performing model checking in practice.

Acknowledgments

We would like to thank Eric Engstrom, Aaron Larson, Nickolas Weininger and Robert Goldman at Honeywell Technology Center for their collaboration and support in the translation and verification of DEOS. We would also like to thank Phil Oh, Klaus Havelund, Charles Pecheur, Michael Lowry, Thomas Uribe, Hassen Saidi, Matt Dwyer, John Hatcliff, David Dill, Satyaki Das and Jens Skakkabaek for numerous technical discussions that contributed to this work.

8. REFERENCES

[1] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21 of *SIGSOFT Software Engineering Notes*, pages 156–166. ACM, October 1996.

[2] C. Barrett, D. Dill, and J. Levitt. Validity Checking for Combinations of Theories with Equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, November 1996.

[3] Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. Model checking programs. In *Proceedings of the 14th IEEE International Automated Software Engineering Conference*. IEEE Computer Society Press, September 2000.

[4] W. Chan, R. Andersen, P. Beame, D. Jones, D. Notkin, and W. Warner. Decoupling Synchronization from Local control for Efficient Symbolic Model Checking of Statecharts. In *Proceedings of the 21st International Conference on Software Engineering*, pages 142–151, Los Angeles, May 1999.

[5] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Translactions on Program Languages and Systems*, 16(4), sep 1994.

[6] M. Colón and T. Uribe. Generating Finite-state Abstractions of Reactive Systems using Decision Procedures. In *Proceedings of the 10th Conference on Computer-Aided Verification*, volume 1427 of *LNCS*, July 1998.

[7] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from java source code. In *In Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[8] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 4(2):511–547, August 1992.

[9] Z. Dang and R. Kemmerer. Using the ASTRAL Model Checker to Analyze Mobile IP. In *Proceedings of the 21st International Conference on Software Engineering*, pages 132–141, Los Angeles, May 1999.

[10] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Proceedings of the 11th International Conference on Computer Aided Vericifaction '99*, volume Lecture Notes in Computer Science 1633, pages 160–171, 1999.

[11] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Vericifaction*, Lecture Notes in Computer Science 1254, pages 72–83, 1997.

[12] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 1999.

[13] Klaus Havelund, Michael Lowry, SeungJoon Park, Charles Pecheur, John Penix, Willem Visser, and Jon L. White. Formal analysis of the remot agent before and after flight. In *Lfm 2000: Fifth NASA Langley Formal Methods Workshop*, 2000.

[14] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[15] G. Holzmann and M. Smith. A practical Method for Verifying Event-Driven Software. In *Proceedings of the 21st International Conference on Software Engineering*, pages 597–607, Los Angeles, May 1999.

[16] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data flow analysis for cehcking properties of concurrent java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410. ACM Press, May 1999.

[17] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan Execution for Autonomous Spacecrafts. In *Proceedings of the International Joint Conference on Artificial Intelligence*, August 1997. Nagoya, Japan.

[18] John Penix, Willem Visser, Eric Engstrom, Aaron Larson, and Nicholas Weininger. Verification of time partitioning in the deos scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, June 2000.

[19] H. Saidi. Modular and Incremental Analysis of Concurrent Software Systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 92–101, October 1999.

[20] H. Saïdi and N. Shankar. Abstract and Model Check while you Prove. In *Proceedings of the 11th Conference on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 443–454, July 1999.

[21] D. A. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In G. Levi, editor, *Proceedings of the 5th Static Analysis Symposium*, volume 1503 of *LNCS*. Springer, sep 1998.