Adding Active Objects to SPIN

First Steps Towards Program Verification in SPIN

Willem Visser[1], Klaus Havelund[2] and John Penix

[1] RIACS
[2] Recom Technologies

Automated Software Engineering Group
NASA Ames Research Center

Abstract. We adapt the SPIN system to allow more efficient program verification. Specifically, we add a feature to the language that makes it possible to model check PROMELA systems containing active objects, i.e. objects that are capable of executing autonomously, and the contents of which can be accessed via object references.

1 Introduction

Model checking is a popular method for analyzing the correctness of designs. The advantages of model checking at the design level are two-fold: firstly, finding errors during the design phase is well known to be extremely cost-effective, and secondly, designs tend to be more abstract than implementations and as a consequence the state explosion problem that model checking often suffers from is less acute. However, it is our experience within the Automated Software Engineering group at NASA Ames[1], that we often apply model checking posterior, i.e. after the system is built we extract a model of the system from the source of the implementation, which we then use for model checking [HLP98,HS99]. In both the design verification and the "after-the-fact" verification the problem exists that the model of the system is often in a different notation than the implementation of the system. We believe this gap should be closed.

The approach we suggest for doing this is to add to a model checker the capability to do *program verification*, i.e. enhance the notation for describing the input to the model checker with common programming structures and change the checker so that it can handle these new constructs during model checking. This would allow us to have one notation for both design and program verification. We picked the SPIN model checker [Hol97] to try out the idea of program verification, primarily since its input notation, PROMELA, already contains many concepts available in modern programming languages, e.g. records, arrays, loops, conditional branching, shared memory, processes etc.

In the rest of this paper, we will only focus on using program verification principles within the framework of after-the-fact verification, since this is what our group is currently concerned with. However, we believe the benefits of enhancing PROMELA with programming constructs would allow other avenues for research within the SPIN community. For example, one can imagine developers of software systems, doing a high-level design in PROMELA then model check it, after which they refine it more and more, after each stage checking the model, until finally producing executable code in a commercial programming language. At some stage during the refinement

[1] `http://ase.arc.nasa.gov`

process the size of the model might become intractable when doing exhaustive verification, at which point the model checker can be considered a debugger, i.e. it looks for errors but cannot guarantee to find them (by for example running in the super-trace mode). Of course, given a formal semantics of the *extended* PROMELA one can also do refinement proofs between the different levels of implementation.

Currently, we have two major program verification exercises within our group: model checking a real-time operating system of about 3000 lines of C++ code (called *DEOS*) and doing general JAVA verification with a home-grown tool called *JAVA PathFinder* (JPF) [HP99]. In both these projects we translate the original code to PROMELA and use SPIN for model checking: in the DEOS verification we do it by hand and the JPF tool does an automatic translation. In both cases the translation is non-trivial since PROMELA does not support objects directly. It was therefore obvious that we require support for manipulating objects within PROMELA. Furthermore, JAVA in fact supports *active objects*, i.e. objects that can have their own thread of control. Active objects are interesting constructs, since they unify the two seemingly different areas of concurrent and object-oriented programming. That is, an active object has a thread of control in addition to public methods/data. In case there is no thread of control, we have a passive object. In case there is a thread of control but no public methods/data, we have a process.

This paper describes the experiment of adding the capability to the SPIN system to handle active objects. We will show how we made minimal changes to SPIN in order to both model check and simulate programs with active objects. Throughout we will try and quantify how difficult the changes were, and what insight was required to make them. It is our hope that such a description might be useful to others thinking of making changes to PROMELA/SPIN. We evaluate the active objects addition to SPIN by comparing the translation effort and verification efficiency between the old translations used for DEOS and JPF and the new translations to the system with active objects.

The structure of the rest of the paper is as follows. In section 2 we show how the translation of objects to standard PROMELA is achieved at the moment, and in section 3 we show the changes we made to the system to handle active objects. In section 4 we evaluate the work by comparing it to the standard translation used for the DEOS and JPF projects and in section 5 we discuss future work.

2 Background

Translation tools from programming languages to PROMELA, to allow model checking with SPIN, have recently gained the interest of the research community. The JAVA PathFinder [HP99] developed at NASA Ames translates from JAVA to PROMELA. Another JAVA to PROMELA translator is described in [IDS98], whereas in [Cat98] a concurrent extension of C++ is considered (called sC++, and in fact is extending C++ with active objects). In addition we are also translating C++ code to PROMELA within the DEOS project, but unlike the above mentioned tools, we do the translation by hand (we are however trying to stay as close as possible to the original code in our translation). In this section we only focus on how objects and threads are translated to PROMELA, since this is the capabilities we want to add to PROMELA in the form of active objects. Furthermore, since the synchronization between active objects in sC++ is very limited (all method calls to an object are synchronized), we will only discuss the more general translation used within the two JAVA tools and the DEOS project (these three translations are very similar). We will therefore focus on objects (of a specific class) as defined in C++ and JAVA, and threads as defined in JAVA. We will not discuss these concepts in any great detail, but will rather focus on the general concepts of objects and threads.

For the translation of JAVA and C++ classes the general idea is the following: the data part of a class is modeled by a record and the methods of the class by macro definitions. Since for each object instantiation of the class there will be allocated a new data part, we keep track of the different objects' data members in an array of records. For each class an index variable is used to keep track of the next free slot in the array (initially this index is 0) and its value is used as an *object pointer* for the next object instantiation for this class. An obvious drawback of using arrays to keep track of object instantiations is that in PROMELA the size of an array is fixed and hence, dynamic object creation in its full generality is not possible.

```
class Spin {
  public int concurrency;

  public Spin(int con) {
    concurrency = con;
  }
}

class ActiveSpin extends Spin {
  public int objects;

  public ActiveSpin(int con,
                    int obj) {
    super(con);
    objects = obj;
  }
}
```

```
#define MAX_OBJECTS 3

typedef Spin_Class {
  int concurrency;
};
Spin_Class Spin_POOL[MAX_OBJECTS];
byte Spin_Index = 0;

typedef ActiveSpin_Class {
  int concurrency;
  int objects;
};
ActiveSpin_Class ActiveSpin_POOL[MAX_OBJECTS];
byte ActiveSpin_Index = 0;
```

Fig. 1. PROMELA Object Arrays for two JAVA classes: JAVA (left) and PROMELA (right).

In Figure 1 the object arrays for two simple JAVA classes that have no methods, but show inheritance, are given in PROMELA. For each class we create an array of records for the data of the class (of maximum size 3). Note how we include the `concurrency` field from class `Spin` in the definition of the data for class `ActiveSpin`, due to the inheritance between the classes.

A particular object is referenced by an object reference that indicates not only what the class of the object is (i.e. in which array to look up the data for the object), but also which instance it is (i.e. at what index in the array is the object's data stored). There are a number of different approaches to encode this information in an object reference. In DEOS we used a simple scheme where each object reference is a record with two fields: one for the class of the object and one for the index to the specific instance. Both the class and index field would be set when a new object is instantiated. For example the constructors for the two JAVA classes of Figure 1 would be translated as follows (`inline` is a macro expansion):

```
#define Spin        0
#define ActiveSpin  1

typedef ObjRef {
  byte CLASS;
  byte INDEX;
}
```

```
inline Spin_NEW(this,con) {
  this.CLASS = Spin;
  this.INDEX = Spin_Index;
  Spin_POOL[Spin_Index].concurrency = con;
  Spin_Index++;
}

inline ActiveSpin_NEW(this,con,obj) {
  this.CLASS = ActiveSpin;
  this.INDEX = ActiveSpin_Index;
  ActiveSpin_POOL[ActiveSpin_Index].concurrency = con;
  ActiveSpin_POOL[ActiveSpin_Index].objects = obj;
  ActiveSpin_Index++;
}
```

When an object is instantiated, in C++ or JAVA, with a call `x = new Spin(1)` we translate it to the following PROMELA code: `Spin_NEW(x,1)` where `x` is of type `ObjRef`. Inheritance however allows polymorphism between the objects of a class and its subclasses and this is where the translations become more cumbersome. If, for example, the field `concurrency` of an object of class `Spin` is accessed, we don't always know whether the field is in the array for `Spin` or `ActiveSpin`. In DEOS we therefore translate the following code `y = a.concurrency` (where `y` is an integer and `a` is declared as an object of class `Spin`) as follows:

```
if
::a.CLASS == Spin       -> y = Spin_POOL[a.INDEX].concurrency
::a.CLASS == ActiveSpin -> y = ActiveSpin_POOL[a.INDEX].concurrency
fi;
```

Similar conditional code is used when assigning a value to the field of an object where polymorphism might be present. We found this way of handling polymorphism in the DEOS translation satisfactory, since we did the translation by hand and furthermore inheritance is only used in one part of the system (in the definition of doubly linked lists, which forms the basis of all the queue manipulations within the operating system). The translation scheme do however cause a serious code blow-up in some places: for example, the innocuous looking C++ code `this->previous->next = otherList->next` translates to 32 lines of PROMELA, since no type information is available and each of the references could be of one of two types.

In the JPF tool a slightly different approach was taken: the class and index fields from the object reference were combined into one field of type int. Creating this combined reference involves using a formula with a specific offset for where the class and index information can be distinguished: specifically a reference to a new object of class c and index i is created by $c \times 100 + i$. Retrieving the index and class of an object is therefore simply *div* and *mod* operations: respectively `getclass(x) = x div 100` and `getindex(x) = x mod 100`. This approach of encoding the object reference into one integer field was found to be more convenient when doing automatic translations than the `CLASS` field approach used within DEOS. However, a code blow-up is still present and in fact, in an early version of the JPF tool this code blow-up caused the generated C code (from the PROMELA generated from a 1000 line JAVA program) to be so large that it caused the gcc compiler to fail and hence made verification impossible.

In the next section we discuss a small change to PROMELA and SPIN to allow for a more efficient translation from object oriented languages to PROMELA.

3 First Steps to Adding Active Objects to PROMELA/SPIN

It should be said right up-front that the change to PROMELA/SPIN was very minor and took only about one man-week to do. It is not a general solution for handling objects in SPIN, but was rather intended to show *proof-of-concept*.

Our basic motivation was the following: how can we reduce the effort involved for translating objects to PROMELA, and, do it with minimum changes to the current system. In early attempts to translate JAVA/C++ we modeled objects as PROMELA processes, but this didn't work. The problem was that although the data part of a class translated to variables declared within a process, and, an object instantiation translated to a process being started, what did not work was the translation of method calls. One could model method calls as synchronous messages to the process representing an object, but in JAVA, for instance, at any point in time more than one *thread* can call the same object's method if the call is not *synchronized*, and this would not be possible when translating method calls as messages and method bodies as code fragments inside a process. Even if there is only one thread of control in a system, as is the case in DEOS, then this translation approach fails when one object calls a method in another object that then calls a method in the first object before returning: this would cause a deadlock. What we learned from this is that processes and objects is a perfect match when considering data, but not for calling methods. Hence, if we could find a way to keep method calls as inline macro expansions within the body of the calling process, but somehow relate the data within the methods to variables within another process (one that contains the object's data) then we would have a satisfactory solution for the translation of objects. This last observation motivated the change we required in the SPIN system: *a mechanism to reference variables declared within another process.* To illustrate how this would work we show in Figure 2 how the two JAVA classes defined in Figure 1 would be translated to a SPIN system that allows access of variables inside another process. The following notation is used to access variable `var` in process `proc`: `proc~var`.

```
#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);
}
```

Fig. 2. PROMELA with Access to the Variables of a Process.

With the use of the constant `passive` in Figure 2 we give a hint at another interesting side-effect of this form of translation from objects to processes, namely that both *active* and *passive* objects can be translated trivially. The two classes in Figure 2 are both *passive*, i.e. they have no body to execute, which is modeled by them having the unexecutable statement 0 as their body. The 0 body is required to ensure the process does not terminate. Obviously, when translating an active object one would replace the 0 with the body of the active object. For example, we show below how the `ActiveSpin_Class` would look if it had an endless loop as a body. Note how a new method is now required to *start* the execution of the body, to ensure that the body is not executed before it is correctly initialized.

```
proctype ActiveSpin_Class() {
  int concurrency;
  int objects;
  bool STARTED = 0;

  STARTED;
  do
  ::1 -> ...
  od;
}

inline ActiveSpin_Start(this) {
  this~STARTED = 1;
}
```

In our extended system one can therefore now declare object variables (references) that are of the type of an already defined process. The variables of an object (process) can be of any PROMELA type, including other objects. None of the functionality of standard PROMELA is affected, hence non-object variables within a process can be used as before. Object variables can be passed over channels and as parameters to processes, since an object variable is nothing more than a `byte`, with extra type information (i.e. which type of process it is pointing to).

3.1 Implementation Details

In order to understand how we implemented the access to the variables of a process, one needs to understand how certain parts of the SPIN model checker works. Specifically, how do SPIN generate the *next* state from the current system state during model checking. The PROMELA compiler generates C code that gets compiled by a C compiler (gcc in our case), where after the generated executable file is executed to do the model checking. This means that every PROMELA statement is essentially translated to a fragment of C code, that when executed, will change the current state of the system, i.e. generate the next state for model checking. In order for each fragment of C code to execute correctly it must be viewed within a dedicated run-time environment, which we will refer to as the PROMELA run-time environment. In fact, most of the PROMELA run-time environment is just the C environment, for example addition in PROMELA translates to addition in C. Of course, in areas, where no equivalent C code exists, for example communication statements in PROMELA, the PROMELA run-time environment is much more elaborate, but for the purpose of this paper we will not focus on these areas. Our focus is rather on the memory model of the PROMELA run-time environment, and only in so far as the parts that concern our goal of accessing variables of a process.

In the memory model of the PROMELA run-time environment a record data-structure (called `struct` in C) is used to keep track of the variables of a process. In the current system state the this reference always refer to the structure of the current process executing. Global variables are stored in a special structure called State and can be accessed through a reference called now. As an example, the record structure for process Spin_Class above would be:

```
typedef struct P0 { /* Spin_Class */
        unsigned _pid : 8;  /* 0..255 */
        unsigned _t   : 3; /* proctype */
        unsigned _p   : 5; /* state    */
        int concurrency;
} P0;
```

Referencing the value of variable concurrency would require execution of the following C statement: ((P0 *)this)->concurrency. This would assume however that the current reference of the variable occurs within the process Spin_Class, since the this pointer is assigned the reference to the data of the currently executing process within SPIN. In other words when the process was instantiated with the run statement a new block of memory containing its data was created and every time this process is scheduled to execute the this pointer points to the block of memory. See how the block of memory the this pointer points to is cast to the correct structure type so that all the fields (i.e. variables) of the process can be accessed.

In order to generate code for the statements that refer to the variables within another process we need to know where the block of memory for the process resides and then access the correct fields (variables) within this block. Since it was clear that within the SPIN source it was possible to find out where the block of memory for a specific process starts (how else would the value of the this pointer be known!), it was just a question of finding out how it was done. Looking up all assignments to the this pointer within pan.c (the main code that gets executed for the SPIN model checker) turned up a macro called pptr(x) that would return a pointer to the memory block for a process with instantiation number (x). The instantiation number, or process id (_pid), is returned by the run call and is therefore available during run-time. The only other information we require is the record name used for a process' memory block, since it is required to cast the block of memory into the correct structure to allow a field (variable) to be accessed. This information is however available statically from the type declaration of the object. Furthermore, these structures have a fixed naming scheme: they are referred to Pn where n indicates the number of processes that have been parsed during compilation (i.e. the first process in the file will have structure name P0, the next process P1 etc.). In fact, there is even a function in the SPIN compiler that would return the value of n when given the name of a process (proctype) as a character string (called fproc(char *s)).

We added one field to describe a *Symbol* within the PROMELA compiler to indicate the name of the process an object variable refers to (called *procname* and declared as a character string). When an object variable is declared we can assign this field the name of the proctype it refers to. For example in the declaration Spin_Class spin_obj; from Figure 2 the *procname* field for spin_obj will be assigned Spin_Class. If we assume Spin_Class was the first process to be declared within a PROMELA file then the code for the assignment spin_obj˜concurrency = 1; (where spin_obj = run Spin_Class()) within init (which we assume is the third process in the file), would be:

```
((P0 *)pptr(((P2 *)this)->spin_obj))->concurrency = 1;
```

Since the assignment is within the currently executing process we first need to get the value of the variable spin_obj within the memory block pointed to by this (note the cast to $P2$ to ensure the `spin_obj` field is accessible); next we need to find the block of memory for the process that `spin_obj` points to by using the `pptr` call and lastly we need to ensure again that the memory is cast into the correct structure namely that for `Spin_Class()` which is $P0$.

Polymorphism is also easily handled by the translation. Let us consider the record structure containing the data variables for `ActiveSpin_Class` that extends `Spin_class`:

```
typedef struct P1 { /* ActiveSpin_Class */
        unsigned _pid : 8;  /* 0..255 */
        unsigned _t   : 3; /* proctype */
        unsigned _p   : 5; /* state    */
        int concurrency;
        int objects;
} P1;
```

Now if we have an object reference of type `Spin_Class` then the actual object might be of the type of the subclass `ActiveSpin_Class`, but one can notice that the data member that is shared between the two classes is stored in the same location in the record structure for both processes. This means when the following reference is encountered `obj~concurrency` and the type of `obj` is `Spin_Class` then the memory block, `obj` points to, can be casted to $P0$ regardless of whether the actual memory block has the structure of $P1$ (i.e. was allocated with a `run ActiveSpin_Class()` call), and the correct value of the `concurrency` variable will be found. Of course, casting the memory block to type $P0$ and then to try accessing the `objects` variable will not be allowed by the C compiler, but this is not allowed in object oriented languages anyway (i.e. one cannot reference variables of the subclass when the object reference has the superclass' type[2]).

3.2 Observations

From having the initial idea to having a system going that could handle a very simple verification example, took no more than one day of work. An immediate need however was to be able to not only do verification, but also to simulate the extended PROMELA programs. Unfortunately SPIN compiles and executes C code for verification, but for simulation it interprets *different* code. This meant one also needed to extend the simulation code. As it turned out, this was in fact rather easy and involved very few changes in the functions that *get* and *set* the values of a variable. However, arriving at the precise places to make the changes required understanding the whole simulation engine. Making changes to the simulation engine took in all about 2 days.

On the verification side the most time was spent extending the implementation beyond the initial phase so that it could be compatible with the different features of SPIN. For example allowing arrays of objects and indexing arrays with the value of a field of an object etc. This took approximately 3 days, but was interleaved with the translation of the PROMELA version of the DEOS model to extended PROMELA. We believed if a truly large system like DEOS can be handled by the extended SPIN system, then it would be a sign that the system was indeed useful.

One interesting experience was when we attempted to do a verification of a temporal property for the first time with the extended system; it didn't work! The problem was that when doing verification with a *never* claim all instantiation numbers move up one, since the process for the never claim is assigned the first instantiation number (0). In fact this is indicated by a constant

[2] In both JAVA and C++ it can however be achieved by type casting, but this would not be a problem since the cast would be explicit at compile time.

value `BASE` that becomes 1 when doing temporal verification and is 0 otherwise. Hence when calling `pptr(x)` it should really be `pptr(BASE+x)`. We learned this lesson the hard way.

We took an extremely conservative view of the effect object references would have on the use of partial order rules [HP94] within SPIN, by viewing all object manipulations as *global* references. This means all statements that contain object references will be considered *visible* and hence will be interleaved with all other visible transitions during verification. Also note that manipulating "local" variables within an active object must be considered global references now, since these variables might be accessed from outside the object (by calling methods of the object).

4 Impact on DEOS and JPF

Adapting the JPF tool to work with the extended SPIN system is still in progress, hence we cannot yet determine the impact of the new system. We have however done some experiments where we translated from JAVA to PROMELA and then hand-translated the PROMELA to the extended PROMELA. We relate here the results achieved by running the JPF tool versus the extended SPIN system on the producer-consumer example from [HP99]. Since JPF must estimate the size of the arrays required to store object data, and these estimates are always conservative, the extended SPIN system used less memory and time (due to hashing being slower on longer state vectors). The number of states generated by both systems were about the same. When we optimized the PROMELA code produced by JPF to have the optimal array sizes the two systems were within 5% of each other with respect to time and memory usage, with JPF showing better performance. However, when we changed the system configuration by adding an extra consumer (active) object, JPF outperformed the extended SPIN system significantly, most notably the number of states was different by more than $100K$. This is due to our conservative treatment of partial order rules (section 3.2) and the fact that the consumer object is active and therefore introduces many more possible interleavings.

The impact of the extended SPIN system on the DEOS verification was much more apparent. For example, one method for merging two lists, translated from 7 lines of C++ code to 193 lines of standard PROMELA, but had an almost one-to-one mapping with the C++ code in translation to extended PROMELA (see Figure 3). The reason why the code blow-up was so much for the standard PROMELA translation was that there is polymorphism present here: each of the nodes in the list could be of the superclass ThreadList (with data members, previous and next) or the subclass ThreadListNode that extends the superclass with one more data member (called itsParent). Each variable reference therefore required an `if` construct to determine the class of the variable before looking up the value in the correct array (as described in section 2).

The translation effort to get from the original C++ for DEOS to extended PROMELA was negligible. Furthermore, since so many control states were removed in a part of the code that was heavily used, the amount of states generated between the old and new model of DEOS in PROMELA was very significant. In fact, the old model of DEOS was intractable (i.e. SPIN ran out of memory on a 512M machine) for small configurations of the system, i.e. where the DEOS operating system only had to schedule 2 threads. The new extended PROMELA model however, easily completed (i.e. did not run out of memory) with 3 threads to schedule.

We did one more experiment to determine the amount of overhead that is required to check extended PROMELA models. We created a version of DEOS where we removed the inheritance in the link list code, by just merging the sub- and superclass into one class with all the combined

```
void ThreadList::mergeList(ThreadList &otherList) {
  if ( ! otherList.isEmpty() ) {
    previous->next = otherList.next;
    otherList.next->previous = previous;
    previous = otherList.previous;
    otherList.previous->next = this;
    otherList.next = &otherList;
    otherList.previous = &otherList;
  }
}
```

```
inline ThreadList_mergeList(this,otherList) {
  ThreadList_isEmpty(isEmpty,otherList)
  if ::!isEmpty ->
    this~previous~next = otherList~next;
    otherList~next~previous = this~previous;
    this~previous = otherList~previous;
    otherList~previous~next = this;
    otherList~next       = otherList;
    otherList~previous   = otherList;
  ::else
  fi;
}
```

Fig. 3. Code for merging two lists: C++ (left) extended PROMELA (right)

fields and methods. Hence, we now had a PROMELA model from which we can translate to extended PROMELA without removing any control states. Of course, the number of states generated now were (almost[3]) the same for both systems. The extended system was however 4 times slower. One reason for the slower speed was that SPIN spent time trying to execute statements in passive processes, which had no executable statements. We changed the SPIN scheduler to ignore passive processes during scheduling and that made the extended system only 2 times slower. This is acceptable, since the DEOS model is highly object oriented and the extra time is obviously going into doing the accesses of other processes' variables (i.e. the `pptr` calls).

In summary, the effect of the extended SPIN system is that it reduces the size of the state space when polymorphism is present, but it is slower than a system that just encodes objects in standard SPIN. Arguably, extended SPIN's main contribution is however the amount of effort that is saved in translating object oriented software in order for it to be model checked by SPIN.

5 Conclusion and Future Work

We illustrated how a small change to PROMELA can allow easier modeling of systems written in an object oriented language. Specifically, we showed that by allowing access to the local variables of a process from outside of the process one can easily model objects in PROMELA. We gave some intuition of how we came up with the idea and also showed that one can do verification of the extended PROMELA models by reusing some functions already available inside the SPIN code. Furthermore, we showed that the change we made to PROMELA, also allows one to model active objects in PROMELA. Active objects unify the concepts of concurrency and objects and therefore we believe it is important to be able to handle active objects in PROMELA.

Since objects are essentially modeled by an extension of processes in our approach, and processes can be created dynamically in SPIN, we get dynamic object creation for free (so to speak). However, one major drawback of our approach is that we cannot handle object deletion efficiently. Although, SPIN allows processes to terminate, the space in the state vector occupied by the processes' data variables, is not reclaimed. Therefore, if many objects are created and deleted in a program then the state vector will grow quickly and essentially make verification impossible (due to memory problems). A very interesting avenue for further research is therefore how to implement a form of garbage collection within SPIN.

[3] The difference was less than 1%, and is due to the slightly different notation used between the two PROMELA designs.

Partial order rules that reduce the number of interleavings of statements in concurrently executing processes, is often essential to allow exhaustive verification within SPIN [HP94]. These rules use a dependency relation between the different statements to determine when reductions can be achieved. This dependency relation is determined statically and is very weak, in the sense that it considers certain statements to be dependent when they really are not. For example, all statements that reference global variables are considered dependent on each other (also called *visible*). Visible statements will always be interleaved. Since we consider all object references to be *global* references, i.e. equivalent to updating or reading from a global variable, all object references are visible. Our extended SPIN system is therefore in need of a more advanced dependency analysis technique (so to is the current version of SPIN we believe). Sadly, of course, we have essentially introduced the notion of a pointer, and therefore aliasing, with our extension of SPIN and it is well known that alias analysis complicates dependency calculations a lot [WL95]. This would however still be an interesting avenue for research, since traditional dependency analysis is often focussed on compiler optimizations and performance issues (for example how to parallelize a sequential program), whereas here we are interested in reducing the state space during model checking.

Although the extended SPIN system allows us to model check object oriented programs more efficiently, by essentially reducing the translation effort to PROMELA, we would like to enhance the system even further so that we can also define and access methods in a more straight-forward way. We are currently looking into this.

It has recently come to our attention that at least one other group is working on extending SPIN to allow for program verification [Ios99]. Their extension to SPIN is called *dSPIN* and is a much bigger change to the original system than the one described here. Essentially they added the more general notion of *pointers* to PROMELA, where a pointer can point to any type of variable, not just to processes as we have here. They also added functions and pointers to functions.

References

[Cat98] T. Cattel. Modeling and Verification of sC++ Applications. In *Proceedings of TACAS98: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, LISBON, April 1998.

[HLP98] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the Fourth Workshop on the SPIN Verification System*, Paris, November 1998.

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[HP94] G.J. Holzmann and Doron Peled. An Improvement in Formal Verification. In *Proc. FORTE94*, Berne, Switzerland, October 1994.

[HP99] K. Havelund and T. Pressburger. Model Checking JAVA Programs Using JAVA PathFinder, 1999. To appear in Special Issue of Software Tools for Technology Transfer. Copy available from havelund@ptolemy.arc.nasa.gov.

[HS99] K. Havelund and J. Skakkebæk. Practical Application of Model Checking in Software Verification, 1999. Submitted for publication. Copy available from havelund@ptolemy.arc.nasa.gov.

[IDS98] R. Iosif, C. Demartini, and R. Sisto. Modeling and Validation of JAVA Multithreaded Applications using SPIN. In *Proceedings of the Fourth Workshop on the SPIN Verification System*, Paris, November 1998.

[Ios99] R. Iosif. Personal Communication. iosif@athena.polito.it, March 1999. http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml.

[WL95] R. Wilson and M. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation.*, June 1995.