dSPIN: A Dynamic Extension of SPIN

Claudio Demartini[1], Radu Iosif[1], and Riccardo Sisto[1]

Dipartimento di Automatica e Informatica, Politecnico di Torino
corso Duca degli Abruzzi 24, 10129 Torino, Italy
demartini@polito.it, iosif@athena.polito.it, sisto@polito.it

Abstract. The SPIN extension presented in this article is meant as a way to facilitate the modeling and verification of object-oriented programs. It provides means for the formal representation of some run-time mechanisms intensively used in OO software, such as dynamic object creation and deletion, virtual function calls, etc. This article presents a number of language extensions along with their implementation in SPIN. We carried out a number of experiments and found out that an important expressibility gain can be achieved with at most a small loss of performance.

1 Introduction

It is nowadays a common approach to use concurrent programming along with object-oriented techniques in order to increase robustness and re-usability of concurrent software. A number of new problems in software design and verification arise due to the increase in program complexity, namely run-time complexity. As an obvious consequence, formal verification of this kind of software requires special features. A previous attempt to elaborate formal models of OO programs is presented in [3]. It regards the possibility of automatically generating PROMELA models of programs written in sC++, a concurrent extension of C++. However, the modeling approach used has an important number of limitations, in particular the lack of models for: object creation and deletion, pointer and reference variables, polymorphic function calls.

Our previous experience regarding the translation of Java programs into PROMELA is presented in [4]. One of the critical aspects in our model regards object creation. We discovered that dynamic object creation can be represented in PROMELA in several ways but always in exchange of an important increase in memory requirements for verification. The approach used for Java multithreading applications regarded both passive and active objects (i.e., *thread* objects) and used predefined vectors of a maximal size in order to keep the object data. An index into the vector was used as a pointer (reference) to the object. In addition to the fact that the number of created objects is strongly bound by the maximal vector size, the model does not cover type casting operations and polymorphism, because of the lack of underlying support for representing this kind of information within SPIN.

An analysis of the Java run-time system led us to the conclusion that implementing the following features in PROMELA would suffice in order to efficiently model any kind of Java construct[1]:

- object references.
- dynamic object creation.
- function definition and call.
- function code reference.

In the following we refer to the above presented mechanisms as *dynamic features*, intended as representations of dynamic information regarding the program. Here we distinguish between static and dynamic information, the former referring to the program information that can be known at compile-time using static analysis techniques (e.g., data flow analysis) while the later referring to information that occurs while the program is running.

The SPIN extension presented in this article is called dSPIN, which stands for *dynamic* SPIN. Its intention is to provide SPIN with a number of dynamic features which allow for object-oriented programs to be modeled in a natural manner and efficiently verified. The new features introduced by dSPIN can be divided into:

- **memory management** features concerning dynamic memory allocation and reference mechanisms.
- **functional management** features concerning function declaration, call and reference but also local scoping.

Even if the above mentioned mechanisms are currently implemented in both the SPIN simulator and model checker our attention focuses on the implementation of the model checker and, specifically on the changes made to the representation of the program state and transition system. We tried to exploit the standard philosophy as much as possible in order to achieve a high degree of compatibility with the SPIN distribution. Nevertheless, new aspects had to be introduced in order to ensure the correctness of dSPIN verifications and their compliance with the complexity reduction mechanisms used by SPIN.

The paper is organized as follows: Sect. 2 presents the language features, while Sect. 3 is concerned with their implementation in dSPIN. Sect. 4 discusses the backwards compatibility with the original software and finally Sect. 5 presents some experimental results.

2 The Language Features

In order to make this paper self-contained, the present section discusses the syntax and semantics of the main extensions added to PROMELA in dSPIN. A full description of the dSPIN language extension can be found in [5]. As already mentioned, we classify dynamic features into memory management and functional

[1] We did not mentioned exception handling because SPIN 3.2.4 already provides support for it by means of a different interpretation of the `unless` construct.

management features. Memory management regards the possibility of referring memory objects, namely statically declared and dynamically allocated variables, as well as memory allocation and release mechanisms. Functional management is concerned with function declaration and call, function code reference mechanism and local scoping (visibility) issues. The following covers the main issues of both classes of extensions.

2.1 Memory Management

The main extension concerning dynamic memory management is the memory reference mechanism, called briefly *pointer*. A pointer may hold at run-time a reference to a previously defined or dynamically generated object. By object we mean a variable of any type, including basic types, array types and user defined types. In order to make use of pointers one should be able to assign and read reference values, called *left values*, to and respectively, from them. Left values are produced by the left-value operator and the new object creation statement, presented later in this section.

Pointer Syntax and Semantics. A pointer variable is declared by prefixing its name with the `&` (ampersand) symbol in the declaration. Unless it is initialized in declaration, a pointer variable contains the `null` value, where `null` represents a dSPIN literal that can be used in programs in order to denote the value of an undefined pointer.

The use of pointers is quite simple because they do not need to be dereferenced. It is done automatically, according to the context. Let us consider first the case when a pointer variable occurs on the left-hand side of an assignment statement. In this case the assignment changes the pointer's left value only if the right-hand side of the assignment is a pointer variable, a left-value expression or a new object creation expression. Otherwise, an assignment to a pointer changes its *right value* that is, the value of the object to which it points. Any attempt to change the right value of an undefined (`null`) pointer generates a run-time error.

The second case to be considered is when a pointer occurs on the right-hand side of an assignment. If the left-hand side variable is a pointer then the pointer will evaluate to its left value, otherwise the pointer will evaluate to its right value, namely the value of the object to which it actually points. In the last case the pointer needs not be `null`, otherwise a run-time error will be raised.

Pointers can also be used along with comparison operators `==` and `!=`. An equality comparison between two pointers evaluates to true if and only if their left values are equal that is, if they point to the same object or they are both `null`. In all other types of expressions and statements, a pointer will evaluate to its right value.

The Left-Value Operator. The left-value operator is an unary operator (`&`) which takes as argument any kind of variable of a basic, array or structured

type. It cannot be applied to a pointer variable. The left-value operator returns a reference to its argument. This reference can then be assigned to a pointer variable.

The New and Delete Statements. The new object creation statement, called briefly the *new* statement, allocates an amount of memory in order to hold a number of objects of any basic or structured type. The reference to the newly allocated area is assigned to a pointer variable. The formal syntax for the new statement is presented below:

```
<pointer> '=' new <type> [ '[' <bound> ']' ]
```

Here `pointer` stands for a pointer variable, `type` stands for any type name and `bound` is an integer specifying the size in number of objects to be allocated. The bound specifier is optional, the default size being of one object of the specified type.

The object deletion statement, briefly called the *delete* statement performs the reverse action i.e., it deallocates the memory space previously allocated by a new statement. The formal syntax of a delete statement is presented below:

```
delete '(' <pointer> ')'
```

where `pointer` stands for a pointer variable. Only heap variables can be used along with delete statements. Any attempt to delete a static variable raises a run-time error.

Another point to be stressed here concerns the executability of the new and delete statements. Both are always executable. The order in which objects can be deleted does not depend on the order they were created that is, a process will never block attempting to delete an object.

2.2 Functional Management

The main concept regarding functional management features is the *function*, defined in dSPIN as it is in most programming languages: a function is a parameterized sequence of code whose execution may be invoked at a certain point in the program and which, upon termination, makes control return immediately after the invocation point. dSPIN functions are executed synchronously that is, the caller process execution thread does not span, rather it is continued by the function call. The statements contained within the function are executed with interleaving along with other processes. The way functions are defined in dSPIN allows for recursiveness in a natural manner.

Functions represent also "objects" supporting a reference mechanism. Indeed, a function can be referred by a special kind of pointer, which provides a mean to model polymorphic function calls used in most object-oriented languages.

Function Definition and Call. The syntax of function definition is similar to the standard PROMELA `proctype` definition. Formally it looks like:

```
function <name> '(' <list> ')' [ ':' <type> ]
'{' ( <declaration> | <statement> )* <statement>+ '}'
```

where `name` stands for the function name, `list` stands for the formal parameter list and `type` is any type name (including pointer types) , specifying the function return type. The return type is optional, which means that a function is not required to return a value i.e., it can be a procedure. A function definition also specifies the function body which consists of a number of declarations and statements. It is required for a function to contain at least one statement.

Upon declaration the name of a function can occur within a function call statement. In dSPIN statements of a function are executed with interleaving, reflecting the behavior of real concurrent programs.

There are three types of function invocation statement. The first one concerns only non-void functions i.e., functions for which the return type is specified in definition. Its formal syntax is shown below:

```
<variable> '=' <name> '(' <list> ')'
```

Here `variable` stands for a declared variable identifier, `name` stands for a declared function name, and `list` for the function actual parameter list. This statement causes function `name` to be called, its return value being assigned to `variable`.

A function may return a value by means of the dSPIN `return` statement, formally specified as:

```
return [ <expression> ]
```

where `expression` stands for any expression that can be atomically evaluated. The `return` statement causes control to leave the function body and return to the statement immediately below the invocation point. The return expression is only required when the `return` statement occurs within a non-void function definition.

For procedures i.e., functions for which the return type is not specified or the return value is not of interest at the invocation point, there is a more simple type of invocation:

```
<name> '(' <list> ')'
```

Here `name` stands for a procedure name and `list` for the actual parameter list. The above statement causes function `name` to be called with actual parameters from `list`, any possible return value being discarded.

The third and last type of function invocation statement was introduced in order to facilitate the writing of tail-recursive functions. It has the following formal syntax:

```
return <name> '(' <list> ')'
```

where `name` stands for a non-void function name and `list` specifies the actual parameter list for the function invocation. This statement causes the function `name` to be called, its return value being passed back as the return value of the caller function.

A process attempting to execute a function call statement will always pass to the first statement of the function without blocking. Analogously, a return statement is always executable. However, the executability of a function call statement depends on the executability of the statements residing inside the function body, therefore it cannot be evaluated a priori. For this reason, function call statements cannot be used within deterministic regions of code (i.e., `d_step` sequences).

Function Pointers. As previously mentioned, functions represent objects which can be referred using a special kind of variables called *function pointers*. A pointer to a function actually holds a reference to the beginning of the function code.

A function pointer is declared as a variable of a predefined type named `ftype`. Unless initialized in declaration, a function pointer is undefined i.e., its value is `null`. A function pointer can be assigned a reference to any declared function. Upon assignment, the function pointer can be used in any form of function invocation statement, instead of the function name.

Local Scopes. The notion of function introduces also the concept of *local scope* for variables. Variables can be declared inside a function, being visible only within the function scope. Moreover, a local scope can be declared inside a process, function or local scope by enclosing the program region into curly braces. Local scopes can be nested and the same variable name can be used in different scopes without conflicts. Variables defined within a local scope are automatically initialized when the control enters the scope.

3 Implementation Issues

This section presents some issues regarding the implementation of the dynamic features discussed in Sect. 2 in dSPIN. As a first remark, let us note that almost all language extensions previously mentioned require dynamic memory space that is, the ability to dynamically increase or decrease the amount of memory used by the current program state. This appears obvious in the case of `new` and `delete` statements and moreover, function calls require the caller process to expand in order to hold the function actual parameters and local variables. A linear representation of the current state (i.e., state-vector representation) would not be convenient for our purposes, because every dynamic increase or decrease of the memory space could require the relocation of many components of the state (e.g., processes, queues, variables), affecting the overall performance of the model checker. Our solution was to adopt a different representation for the current

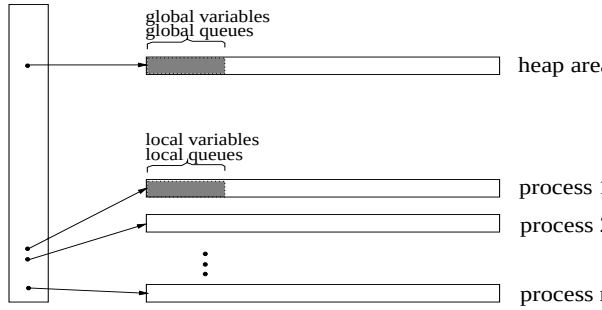

Fig. 1. The State Structure

state. This representation is non-linear, rather composed of a number of different vectors. In the following we will refer to the current state representation as the *state structure*. Fig. 1 depicts the general organization of the state structure.

All memory areas that may shrink or expand are organized into separate vectors of variable sizes. There is one vector for each currently active process, which we refer to as *process vector* and another global one called *heap area* used to hold global data and dynamically created objects. Global variables and queues reside at the beginning of the heap area, while local variables and queues are kept at the beginning of the process vector.

Static variables (i.e., variables explicitly declared in the program) are assigned an offset into the containing vector, global variables being represented by heap offsets, while local variables being defined by offsets into the process vector that contains them. This representation of variables somehow contrasts with the standard SPIN representation because SPIN directly converts PROMELA variables into C variables of the corresponding type. In dSPIN we use type information to compute the storage size of a variable and determine its offset.

An important point which needs to be stressed here regards the way in which the state structure representation affects state comparisons and storage performed by the model checker during verification runs. The standard version of SPIN takes great advantage from the linear state representation (i.e., state-vector) in order to optimize the run-time performance of the model checker. Indeed, the comparisons between states are implemented as byte-vector comparisons. Moreover, state-space storing procedures (e.g., hash indexing store, minimized automaton store[2]) are byte-vector operations. For example, the computation of a hash index takes only one pass through the vector in order to accomplish its task. As previously discussed, the state structure representation keeps the state information organized into several vectors of various sizes. Making the state comparison and store routines iterate through all vectors that make up the state structure greatly decreases the run-time performance of the model

[2] dSPIN considers only the storing techniques used up to and including SPIN version 3.2.0

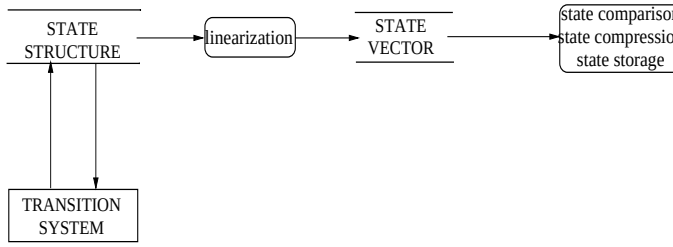

Fig. 2. The State Structure Linearization

checker. Also state compression can be difficult to implement when using a non-linear structure.

The implementation solution used in dSPIN interfaces the state structure representation with linear comparison, compression and storing techniques by first performing a linearization of the state. The linear representation of the state is subsequently referred to as the *state vector*. It is obtained from the state structure by copying the relevant state information from the heap area and process vectors into the state vector. The state vector uniquely identifies the state that is, every state change into the state structure is reflected into the corresponding state vector. This is then used along with state comparison and storing routines instead of the original state structure representation. Fig. 2 shows the use of the state structure along with standard SPIN comparison, compression and storage routines.

3.1 Implementation of New and Delete Operations

As already mentioned, dynamically allocated objects are modeled into the heap area. A dynamic object is represented by an integer index into two tables that keep the information needed in order to locate the object. The first table holds the offset of the object into the heap area, while the second one holds the object size. In the following, we will refer to the first table as the *offset table* and to the second one as the *size table*. The representation of dynamic objects somehow resembles the representation of message channels within standard SPIN.

A new operation takes as parameters a type specifier and a number of elements of the specified type in order to compute the storage size of the object. The heap area is then increased by the size increment in order to contain the newly created object, the first free slot into the offset and size tables is found and the retrieval information is written into the tables. In order to avoid false partial matches of the state structure the type of a newly created object is encoded at the end of the object area. Finally, the new operator returns the table index of the object that is subsequently used to compute the actual reference to the object data.

A delete operation receives as parameter a reference to the object that is intended to be deleted. Such a reference specifies the index number of the object

that is, its index into the offset and size tables. The heap area is decreased by a decrement equal to the size of the object. Unless the deleted object was placed at the end of the heap area, a compaction of the heap area is needed in order to avoid memory losses caused by fragmentation. In practice, the compaction is a simple operation which involves only the relocation of the objects and message channels situated after the deleted object into the heap area by decreasing their offset values by the size of the deleted object. The run-time overhead of heap compaction is therefore limited.

3.2 Implementation of Object References

The object reference mechanism called pointer was presented in Sect. 2. Let us recall that a pointer can hold a reference to a statically declared local or global variable or to a heap variable. We distinguish among three kinds of object reference: the *local variable reference*, the *global variable reference* and the *heap variable reference*. The implementation of object references follows a symbolic approach that is, rather than storing the physical memory address into the pointer variable we have chosen to encode the information needed to locate the object into the pointer by using a symbolic format. This approach allows for an easy run-time check of dangling references. In the following, we model pointers as unsigned 32-bit integers[3]. The first two bits of the integer are used to encode the reference type, which may be one of the above mentioned.

A reference to a local variable uses the next 6 bits in order to encode the declaring process identifier. This allows for local variables from at most 2^6 processes to be referred. The last 24 bits will hold the local variable offset into the process vector, allowing for a maximum of 2^{22} different 32-bit integer objects to be referred inside a process vector.

The global variable reference uses all remaining 30 bits for the representation of the variable offset within the heap area. This allows for a maximum of 2^{28} different 32-bit integer objects to be referred inside the heap area.

The heap variable reference uses the remaining 30 bits in order to keep the object index number that uniquely identifies the heap object. This allows for a maximum of 2^{30} different heap objects to be referred.

3.3 Implementation of Functional Management Mechanisms

It was mentioned in the presentation of the state structure that every active process in the system is represented by a separate vector of a variable size. This implementation choice was inspired by the real world run-time systems that use a separate stack for every process in order to hold the actual values of function parameters and local variables during a function call. Fig. 3 shows in more detail the organization of the process vector as implemented in dSPIN.

The process vector is composed of a static area situated at the beginning, which we will refer to as the *prefix*. It contains information regarding control

[3] The implementation of pointers on different architectures requires minimal changes.

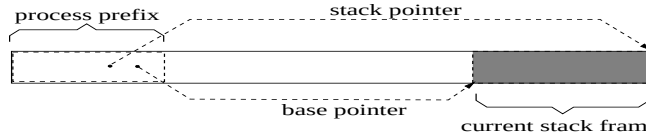

Fig. 3. The Process Vector

flow, such as process type and current state number. As in the standard version of SPIN, this information is needed in order to identify the current process transition from the transition table. Stack management is performed by means of two variables, called *stack pointer* and *base pointer*, residing also in the prefix area. The remainder of the process vector is referred to as the *process stack*. The top offset of the process stack is pointed to by the stack pointer. The area between the offset pointed to by the base pointer and the top offset is called the *current stack frame*. It contains information regarding the currently executing function, namely local variables and actual parameters, as well as function return information.

Sect. 2 briefly mentioned a couple of aspects regarding the semantics of function calls. The first one we recall here states that the instructions contained in a function are executed with interleaving along with other processes. As a consequence of this, a function is represented by means of a separate finite state machine having its own entry into the global transition table. A process that makes a function call temporarily suspends the execution of its finite state machine and starts executing the function FSM. After a return from the function, the execution of the caller process FSM is resumed. The second aspect regarding the execution of function calls is that the caller process execution thread is actually continued by the call. In order to give a more detailed explanation of this, let us consider the linear sequence of states explored by the depth-first search during verification. Let us consider also that the system is in state S_a just before performing a function call. After firing the function call transition a new state S_{a1} is generated. This state reflects only the changes committed on the caller process stack by the function call transition but it has no relevance from the verification point of view that is, it does not carry information that may be useful in order to check system properties. Therefore it makes no sense to store this state in the state space. The first relevant state after the function call is generated by firing the first transition from the function body. Let us call this state S_b. Fig. 4 shows the behavior of the model checker during a function call.

The transitions depicted using dotted lines represent the function call and return from function transitions. In the following, we denote this kind of transition a *virtual transition*. Even if it does not create a new state into the state space, a virtual transition must be recorded on the trail stack in order to allow the model checker to unwind it by performing a backward move.

A function call transition increases the process stack size by first pushing the return information onto the stack that is, the current process type identifier and

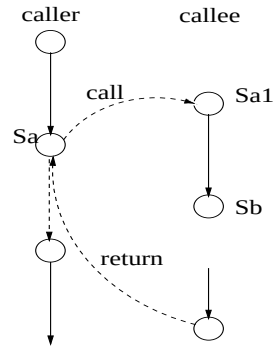

Fig. 4. The Function Call

state number. Following, the left value of the return variable is pushed onto the stack. This is a reference to the object meant to keep the function return value, if there is one. Finally, the current value of the base pointer is saved on the stack, the base pointer is set to the current value of the stack pointer, and the process prefix is actualized in order to reflect the control state change. This is done by storing into it the function type identifier and the identifier of the function FSM initial state in the process prefix area. At this point, the actual parameters are evaluated and their values pushed onto the stack. It is to be mentioned that each process local variable is uniquely identified by its offset relative to the beginning of the stack frame. When the base pointer is set to the stack pointer value a new stack frame is actually created, in order to hold the function parameters and local variables that are not visible from the caller process scope[4].

A return from function transition performs the reverse actions. First the current stack frame is popped from the stack by simply setting the stack pointer to the current value of the base pointer. Then the control flow information is retrieved from the stack in order to resume the caller process type identifier and state number. Finally the original stack frame is restored by assigning the base pointer with the previously saved value. If the function returns a value this is assigned to the object referred by the left value that was also saved on the stack. Subsequently, the model checker attempts to fire the transition immediately following the function call, resuming in this way the execution of the caller process FSM.

The implementation of local scopes also makes use of the process stack mechanism. A local scope can be seen as a parameterless function call for which no return information is needed because the transitions residing inside the scope are part of the same finite state machine as the ones situated outside it.

Sect. 2 presents the function reference mechanism, called function pointer. The implementation of function pointers uses a 32-bit integer value in order to represent the information needed to uniquely identify a function object. The

[4] The dSPIN language extension does not provide support for the dynamic binding of variables.

first 16 bits are used to encode the function identifier that is, its entry into the global transition table. This allow for a maximum of 2^{16} different functions to be referred. The last 16 bits are used to encode the state number of the function FSM initial state. The function pointer implementation uses a symbolic approach that allows for run-time consistency checks to be easily performed.

4 Backwards Compatibility

The previous section presented the implementation of the dSPIN language features and discussed some issues related to the correctness of formal verifications taking into account these new aspects. In the following, we will discuss the way in which our implementation choices affect the standard SPIN complexity reduction techniques, considering state compressions and partial order reductions. Some explanations regarding the run-time overhead introduced by dSPIN are also given.

4.1 Compliance with State Compressions

Sect. 3 presented the state representation used in dSPIN and its use along with the standard SPIN state comparison, compression and storage techniques. Let us recall that all these techniques are based on a linear representation of states in order to optimize the run-time performance of the model checker. In dSPIN a non-linear state representation is used along with the standard approach, which is made possible by linearization of the state structure. In the following, we consider the two main state compression techniques used in SPIN, namely byte-mask compression and recursive indexing compression (i.e., COLLAPSE mode) [6]. These compression modes are the main ones in the sense that all other compression routines (e.g., minimized automaton) take as input the state vector already compressed by one of byte-masking or recursive indexing routines. An important optimization of dSPIN was obtained by combining linearization with one of the two above mentioned techniques. In this way, the state vector obtained by applying linearization to the state structure representation is already in the compressed form and can be used for further compressions, comparisons and hash storing. The time taken up by linearization in this case is almost equal to the time needed by the standard SPIN byte-mask or recursive indexing, yielding a very small overhead in comparison with the original version. It is to be noted that linearization combined with byte-mask compression can be implemented using block copy operations yielding a further speed increase. In exchange, some redundant information is copied into the state vector as it is the case of rendez-vous queues which, unlike in the original byte-mask compression are copied along with their containing vectors.

4.2 Compliance with Partial Order Reductions

The current version of SPIN uses a static partial order reduction of the interleavings between processes known as a *persistent set* technique [10]. The imple-

mentation of partial order reduction in SPIN is based on a static analysis of the processes in order to identify transitions that are *safe*. One case in which a transition is considered to be unconditionally safe regards any access to exclusively local variables, as discussed in [7]. While in the standard PROMELA language it is quite straightforward to find out if some identifier represents a local variable, in dSPIN this is not so. Indeed, an access to a local variable referred by a global pointer variable cannot be considered an unconditionally safe transition. Moreover, by-reference parameter passing in function calls introduces similar problems. As previously mentioned, a static analysis of the source code cannot determine exactly the dynamic information, still a conservative approximation of it can be obtained using data flow analysis techniques. Our solution, implemented in dSPIN uses an iterative algorithm in order to solve what is known as an *aliasing problem* [1]. The algorithm computes, for every non-pointer variable accessed at a certain point in the program, an *alias list* that is the list of all pointers that may refer it. The alias relation is symmetric, therefore every pointer variable at a certain point in the program also has an associated list of objects that it may point to. The safety condition is restricted. We consider a transition to be unconditionally safe if none of the accessed variables is global or has a global alias. More precisely, an access to a local variable that is aliased by a global pointer, as well as an access to a pointer that refers a global object are considered unsafe. By making this assumption we can only err on the conservative side, choosing not to apply partial order reduction in some cases where it still can be safely applied, but never applying it for unsafe transitions.

A concurrent program represented in dSPIN consists of a set of processes and functions. Every such entity is described by a reduced flow graph that is, a graph whose nodes represent sequences of statements and whose edges represent possible control flow paths. A node of such a flow graph is also known as a *basic block*. It is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without possibility of branching except at the end. Every basic block has an associated equation relating the alias relationships among all variables at the beginning of the basic block with the alias relationship at its end. The set of all such equations is known as the *data-flow equation system*. Solving the data flow equation system means finding the smallest conservative estimation of every basic block input and output relations. It is done by an iterative algorithm that is a slightly modified version of the ones described in [1]. The iteration is repeated for every process and function flow graph, one at the time, until a stable solution is found that is, no more changes occur to any input or output relation. In order to estimate the overall complexity of the algorithm let us consider for every process an extended flow graph obtained by adding to the original process flow graph the flow graphs of all functions that may be called during its execution. As pointed out in [1], the number of iterations through a depth-first ordered flow graph is bounded by the depth of the flow graph which is at most $logN$, where N represents the number of basic blocks in the extended flow graph. As a consequence, the complexity

of the entire data flow computation is $O(NlogN)$ where N represents here the maximum number of basic blocks over all process extended flow graphs.

5 Experimental Work

We have applied dSPIN to perform verification of a number of standard SPIN specifications in order to make a performance comparison between the tools. The experience we had analyzing standard examples is reported in the first part of this section. The remainder reports a number of tests carried out with a dSPIN specification of a concurrent B-tree structure in order to give a glimpse of the tool's capability of verifying programs that make use of dynamic information. [5]

Table 1 shows a comparison of the results obtained by verifying a number of specifications with dSPIN and standard SPIN. All examples are taken from the standard SPIN distribution and are denoted by the corresponding file name.

Table 1. Standard Examples

File	States	Transitions	Memory (Mb)	Time (sec)
i. Using dSPIN				
leader	108	108	1.373	0.1
leader2	28898	42274	7.065	1.1
sort	6965	6965	10.193	0.8
pftp	678785	917632	187.184	35.9
erathostenes	444602	628697	180.121	24.3
ii. Using standard SPIN				
leader	108	108	1.493	0.1
leader2	28898	42274	6.613	1.9
sort	6965	6965	9.276	1.9
pftp	678785	917632	191.957	59.3
erathostenes	444602	628697	109.628	38.8

While the number of states and transitions are exactly the same in both cases, we notice that dSPIN sometimes tends to use a slightly larger amount of memory in exchange of a small verification speedup. The cause of this overhead resides in our current implementation of byte-masking compression that uses block copy operations in order to increase speed but at the same time copies from the state structure some redundant information, as discussed in Sect. 4. However, the memory overhead is limited and in most cases it does not increase with the global number of states.

[5] All analysis time reports are obtained from the Unix `time` command on a 256 Mb RAM UltraSparc 30 at 300MHz workstation. Small times (under 0.5 seconds) tend to be inaccurate because of the system overhead.

The experiments regarding the modeling of dynamic aspects have been carried out using a B-tree structure [8] accessed concurrently by updater processes that insert values into it. The system model was specified using pointer variables as well as dynamic object creation and deletion operations. The mutual exclusion protocol follows the approach described in [2]. Each node has an associated lock channel used to ensure exclusive access to its data. When performing insertion into a node, an updater process holds only the locks to its ancestors considered to be *unsafe* (i.e., the safety property is verified for all nodes that do not split due to an insertion). When a certain depth of the tree is reached, the updater processes stop their execution, in order to avoid an unbounded growth of the data structure. The B-tree example considered here is highly scalable, the scaling parameters being the B-tree order, referred to as K, the B-tree maximum depth, referred to as D and the number of updater processes, referred to as P.

Table 2 shows the results obtained by performing analysis of the model for some different configurations of the scaling parameters. The last column of the table specifies the compression modes activated by the corresponding compilation options. Complexity increases exponentially with the maximum depth of

Table 2. The B-Tree Example

K, D, N	States	Transitions	Memory (Mb)	Time (min:sec)	Options
2, 2, 2	9215	18354	3.022	0:02.0	COLLAPSE
2, 2, 3	300577	895141	9.886	1:44.2	COLLAPSE
2, 2, 4	2.87937e+06	1.12389e+07	66.417	23:27.6	COLLAPSE
2, 3, 2	719925	1.50269e+06	40.330	3:03.5	COLLAPSE
2, 3, 3	1.152e+06	3.40875e+06	3.150	5:54.9	BITSTATE
4, 2, 2	13019	25490	3.283	0:2.8	COLLAPSE
4, 2, 3	691256	2.0539e+06	19.060	4:03.2	COLLAPSE
4, 2, 4	1.54303e+06	5.50518e+06	2.824	8:31.9	BITSTATE
4, 3, 2	651896	1.30644e+06	4.009	2:27.3	BITSTATE
4, 3, 3	1.26746e+06	3.72083e+06	4.018	6:45.1	BITSTATE

the B-tree and the number of processes too. The most trivial cases were verified using the recursive indexing (i.e., COLLAPSE) compression mode, while the others were analyzed approximatively using the supertrace technique (i.e., BITSTATE).

6 Conclusions

An extension of the model checker SPIN, providing support for modeling and verification of programs that use dynamic mechanisms was presented. Here the term "dynamic" has roughly the same meaning it has in the context of compiler development, identifying the information that occurs exclusively at run-time. In

order to support the representation of dynamic information, some new concepts were introduced at the front-end level, among them: pointer variables, new object creation and deletion statements, function declarations and calls, function pointers and local scopes. The implementation of these language extensions uses an alternative representation for states that is interfaced with the existing state compression and storage techniques of the standard SPIN version. Nevertheless, new approaches had to be used for the representation of variable references and function calls. The correctness of verifications performed in the presence of the new language features has also been discussed. As pointed out, systems that make use of the dynamic extensions can be verified at the expense of a limited memory overhead by taking advantage of all standard state compression techniques. The number of cases where partial order reductions can be applied is smaller with dSPIN than with SPIN, due to the use of pointers and function calls. The applicability of partial order reductions is driven by a conservative approximation of the variable alias relationship, which is computed by a polynomial complexity data-flow analysis algorithm.

Experiments carried out in order to make a comparison between dSPIN and the standard version of SPIN showed that the overhead in memory space and execution speed can be neglected. Moreover, experience with a non-trivial dynamic data structure used in a concurrent environment gives a glimpse of the tool's capability of modeling and verifying programs that use dynamic mechanisms.

References

1. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers, Principles, Techniques and Tools. Addison-Wesley (1986)
2. R. Bayer and M. Schkolnick: Concurrency of Operations on B-Trees. Acta Informatica, Vol. 9. Springer-Verlag (1977) 1–21
3. Thierry Cattel: Modeling and Verification of sC++ Applications. Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal, LNCS 1384. Springer-Verlag (April 1998) 232–248
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) 5–19
5. R. Iosif: The dSPIN User Manual. http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml
6. Gerard J. Holzmann: State Compression in SPIN: Recursive Indexing and Compression Training Runs. Proceedings of the 3rd workshop on automata theoretic verification with the SPIN model checker, Twente, Holland (April 1997)
7. Gerard J. Holzmann: An Improvement in Formal Verification. Proceedings FORTE 1994 Conference, Bern, Switzerland (October 1994)
8. Knuth, D.E.: The Art of Computer Programming. Vol. 3. Sorting and Searching. Addison-Wesley (1972)
9. Bjarne Stroustrup: The C++ Programming Language. Addison-Wesley (1991)
10. Pierre Wolper and Patrice Godefroid: Partial-Order Methods for Temporal Verification. CONCUR '93 Proceedings, Lecture Notes in Computer Science, Vol. 715. Springer-Verlag (August 1993) 233–246