The dSPIN User Manual

RaduIosif

iosif@athena.polito.it

What is dSPIN[1] ?

The name dSPIN stands for *Dynamic* SPIN. This paper explains the use of a language extension for the model checker SPIN [1, 2], assuming that the reader has some familiarity with SPIN and its input language PROMELA. It does not cover the implementation aspects of our extension and does not discuss the backward compatibility with the original software.

We shall first attempt to explain the intended meaning of the term *dynamic* in this particular context and then proceed with a general overview of the tool extension. The following sections will describe in detail the language constructs implemented in dSPIN.

What does "dynamic" mean?

In the compiler developers community *dynamic* information defines the kind of information that can only be known at run-time, which can only be approximated by static analysis techniques (e.g., data flow-analysis) during the compilation of a program. This information regards among others the effective value of a variable at a certain point in the program for a given execution path or which function is actually called by a polymorphic function call. The "dynamic" extension of SPIN provides some language features for the explicit representation of this information. Namely, it provides two kinds of features:

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

The development of the dSPIN extension was motivated by the need to apply model checking techniques to complex object-oriented software as for example C++ or Java programs which make heavy use of dynamic information when dealing with situations of object creation and deletion, polymorphism, pointer access, etc. dSPIN uses the standard SPIN verification power in combination with the above mentioned aspects regarding mainly OO programs, in order to obtain more accurate models for this kind of software and be able to verify them efficiently.

Memory Management

The main extension concerning dynamic memory management is the memory reference

[1] This user manual is updated for dSPIN Version 3.2.0 + 0.3. As a convention, the number just before + represents the standard compatibility version of dSPIN that is, the version of standard SPIN dSPIN is compliant with. The number right after + is the dSPIN version number.

mechanism, called briefly *pointer*. A pointer may hold at run-time a reference to an object previously defined or dynamically generated. By object we mean a variable of any type or a function. This section will describe only the use of pointers to variables, the notion of pointers to functions being covered by the next one.

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. For example:

```
int x, &y;
```

declares two variables, `x` of type `int` and `y` of type `int pointer`. The last one can now be used as a container for left values. 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 an undefined pointer value.

The use of pointers is quite simple because they don't need to be dereferenced. It is done automatically, according to the context. Let us consider first the case 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 will generate a run-time error. In the following example:

```
int &x, &y;
 ...
x = y;
x = 1;
```

the first assignment makes `x` point to the same object as `y`, while the second one changes the value of the integer pointed by both `x` and `y` to `1`. The pointer-to-pointer assignment is not restricted in any way by static type checking. Any two pointers can be assigned one to another. The assignment itself does not cause an error. For example, one can declare:

```
int &x;
byte &y;
```

and assign `x` to `y` without getting any compile 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, as in the case of the first assignment statement from the example above. Otherwise the pointer will evaluate to its right value, namely the value of the object to which it actually points. In this 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 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 automatically evaluate to its right value. If we consider for example the statement

```
printf(``%d'', x + y);
```

in the scope of the above declaration, it will print the sum between the right value of `x` and the right value of `y`.

No arithmetic can be done on pointers, with respect to this they act like the C++ references [3]. To summarize, a defined pointer which points to a variable can be seen as another name for that variable.

Pointers and Arrays

Pointers can be used in combination with standard PROMELA arrays. Let us consider first the declaration of an array of pointers:

```
int &x[10];
```

which declares an array of ten pointers to integer. An expression like `x[3]` evaluates to the fourth pointer within the array `x`, which represents a left value and acts like a common pointer. But if `x` were declared like:

```
int &x;
```

or, using the standard PROMELA equivalent form:

```
int &x[1];
```

then `x` is no more considered to be an array of pointers, rather a pointer itself. Consequently, the expression `x[3]` stands for the value of the fourth integer from the area pointed to by `x`. It is to be noted that pointers can still be dereferenced (even if this is not usually needed) by an expression of the form `x[0]`, which in this case yields always the value of the first integer pointed to by `x`.

Pointers and Structures

A pointer variable can be declared inside a PROMELA `typedef` construct. It can be declared as pointer to any basic type or structure type, including the structure itself. For example, the following may be the declaration of a simply linked list node:

```
typedef node {
  int value;
  node &next;
};
```

There is no difference between structure access using standard PROMELA variables and structure access using pointers, for example, in the following:

```
node &list;
 ...
list.value = 1;
list = list.next;
```

the first statement assigns value `1` to the `value` field of the structure object pointed by `list`. In this case `list` acts as a standard variable declared of type `node`. The second statement will assign the value of the `next` field of the structure pointed by `list` to `list`. It is a pointer-to-pointer assignment, so as previously discussed, it only concerns left values. A run-time error will occur if an undefined (i.e., `null`) pointer is used in a field access expression.

It is worthy to mention here another extension concerning `typedefs`. A structure can be first declared as, for example:

```
typedef something;
 ...
something someone;
```

and then actually defined:

```
typedef something {
  /* definition of something */
};
```

Upon declaration the name `something` can be used in variable declarations before the structure is defined as it appears in the declaration of `someone`. This feature may be useful for the definition of mutually recursive structures. Note that it is however an error to define mutually recursive structures other than by means of pointers. For example:

```
typedef second;

typedef first {
  second x;
};

typedef second {
  first x;
};
```

is not correct because no one can determine the actual representation for the `first` and `second` structures. Here pointers should be used for the field declaration in at least one of the above structure definitions.

Note that there is no possibility to declare pointers to pointers in dSPIN. Still, one can simulate this using `typedef` constructs. This restriction is similar to the one regarding multi-dimensional arrays from the standard PROMELA specification, and the solution to overcome it is roughly the same.

The Left-Value Operator

The left-value operator is an unary operator which takes as argument any kind of standard PROMELA variable, of a basic type or structured type. It cannot be applied to a pointer variable. Field access expressions can also be used as an argument to the left-value operator. The left-value operator returns the reference to its argument. This reference can then be assigned to a pointer variable. For example, in the following code:

```
int x, &y;

y = &x; /* operator applied to x */
```

the assignment statement actually makes `y` point to the `x` variable. For now on and until the next change on the left value of `y`, both `x` and `y` will represent the same thing i.e., an integer number. If one writes then:

```
x = 1;
printf(``%d'', y);
```

the result is that the value `1` will be printed, because the right value of `y` has also changed due to the assignment to `x`. Of course,

```
y = 2;
```

will cause `x` to evaluate `2` in the future because its right value has changed due to the assignment to the pointer `y`.

The left-value operator does not return the left value of an array element, rather it treats its argument as a whole. In the following code:

```
int x[10], &p;
p = &x[3];
```

the assignment makes `p` point to the entire `x` array, not just to its fourth element.

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 type or structured type. The reference to the newly allocated area is assigned to a pointer variable. The formal syntax for the new statement is given below:

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

Here `ptr` stands for 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.

For example, the new statement:

```
int &x;
 ...
x = new int[10];
```

makes x point to an area capable of containing up to ten integers. In the future x will act as if it was statically defined like:

```
int x[10];
```

although it is defined at run-time. The memory region where the space is taken from is called *heap*, analogously to the C++ run-time system. We distinguish between variables that reside on the heap and statically defined variables. The former ones are called *heap variables*.

Note that a new statement is a stand-alone statement (it must occur between separators) therefore it cannot be used to write pointer variable initializers as:

```
int &p = new int; /* wrong */
```

The newly allocated heap space is automatically initialized. In the example above this means all ten integers are set to zero, but there are cases in which this has more relevance. Let us take the example of a structure containing a channel declaration:

```
typedef one_chan {
  chan c = [1] of { int };
};
```

The following new statement:

```
p = new one_chan[10];
```

causes ten channels to be defined, one for every field of the one_chan structure. In consequence a send or receive operation involving one of those fields is perfectly legal.

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 given below:

```
delete '(' <ptr> ')'
```

where ptr stands for a pointer variable. Only heap variables can be used within a delete statement. An attempt to delete a statically defined variable will yield 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.

Garbage Collection (GC)

Heap-allocated objects that are not reachable by any chain of pointers from the program variables are called *garbage* [4]. The memory occupied by garbage should be reclaimed as soon as possible. This task is accomplished by the dSPIN run-time system. There are two garbage collection algorithms implemented in dSPIN. They are known in the literature as *reference counting* and *mark and sweep* collection.

Reference counting collection keeps track of the number of pointers to each object. When an object reference count becomes zero, the object is automatically deleted. The advantage of the reference counting collection consists in its rather small time overhead. Usually, the collector does not need to scan all reachable objects in order to decide which ones must be deleted. There is a major problem though, cycles of garbage cannot be reclaimed because objects belonging to a loop will have the reference count of at least 1 even if they are not reachable from program pointers. Reference counting should not be used along with cyclic data structures.

Mark and sweep collection marks all objects that are reachable from program pointers. All unmarked objects are then deleted. Mark and sweep collection always reclaims all garbage objects even if they are cyclically linked. The time amount required by mark and sweep collection is larger than the one needed by reference counting because the collector must scan all reachable data.

Reference counting collection is the default GC mode in dSPIN due to its low time overhead. It is activated by the automatic definition of the GC0 macro in the pan.h file. Mark and sweep collection can be activated by defining the GC1 macro, which disables the reference counting GC mode. Garbage collection can be totally disabled by the definition of the NOGC macro.

Functional Management

The basic concept introduced by dSPIN concerning the functional management aspect is the *function*, defined in most programming languages as C, Pascal, C++, etc. Briefly, a function is a "piece" of code whose execution may be invoked at a certain point by the program and who, upon termination, makes control return immediately below the invocation point. Even if dSPIN functions can receive arguments and return values, we are here concerned more about the way control flows during a function invocation. The way functions are defined in dSPIN allows for recursiveness in a natural manner.

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

This section will first present the way functions are defined and how they may be called. The remainder will present the function reference mechanism and finally discusses some scoping (visibility) issues.

Function Definitions

A function definition syntax is similar with 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. Note that the return type is optional, which means that a function is not required to return a value, i.e., it can only be a procedure. Furtherly, a function definition must specify a list of declarations and/or statements. It is required for a function to have at least one statement. Therefore the most simple function definition may be:

```
function foo()
{
  skip;
}
```

Before its definition a function can be only declared, as for example:

```
function foo;
```

This allows for the definition of mutually recursive functions i.e., functions that may invoke each other.

It is required for a function declaration/definition to occur within the global scope that is, a function cannot be defined inside a proctype or inside other function. Moreover, functions need to have unique names. It is an error to define or declare two functions by the same name.

Function Invocations

Upon declaration, a function name can occur within an invocation statement. It may seem awkward that we consider a function invocation to be a statement, rather than an expression. This means that something like:

```
x = x * foo(x - 1) + 3;
```

is not a legal dSPIN expression. The idea behind this follows the standard SPIN philosophy which considers expressions to be atomic pieces of code i.e., it automatically executes them without interleaving with other processes, because it considers only top-level statements to represent FSM transitions, regardless of the amount of computation involved in a statement. This assumption holds pretty well in case expressions are "small enough" that is, they contain operations that are normally executed without interleaving in a real system, but no one can assume that the execution of a function call is atomic.

There are three ways in which one can call a function. The first type of function invocation concerns non-void functions i.e., functions for which the return type is specified in definition. For example, the following function:

```
function inc(int x) : int
{
  return x + 1;
}
```

can be invoked like:

```
int a;
 ...
a = inc(1);
```

The above statement causes `inc` to be called with parameter `1`. The value returned is automatically assigned to `a`. Note that this is considered to be a function invocation rather than an assignment statement, although it apparently has the same syntax.

The above example also introduced a new dSPIN statement, the `return` statement, which can be 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. A `return` statement can occur only within a function definition. Non-void functions require that `return` specifies an expression to be evaluated and passed back as return value. If used within procedures (void functions) `return` must not specify an expression.

In case of non-void function declarations it is a good programming practice to ensure that every execution path within the function ends up with a `return` statement. However this is not a requirement, a non-void function which reaches the end of its body automatically returns zero or `null` if the return type is a pointer type.

For procedures (i.e., functions for which we don't need to know the return value) there is a more simple way for invocation. They can appear as a stand-alone program statement:

```
function hello()
{
  printf(``Hello world!\n'');
}
 ...

hello(); /* procedure invocation */
```

The third and last way in which functions can be invoked was introduced in order to facilitate the writing of tail-recursive functions. It has the following 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 as the return value of the caller function. Below there is an example of a tail-recursive function which computes the factorial:

```
function fact(int x; int r) : int
{
  if
  :: x == 0 ->
        return r;
  :: else ->
        return fact(x - 1, x * r);
  fi
}
```

The top-level invocation for this function may be like:

```
a = fact(5, 1); /* factorial of 5 */
```

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 stated before, 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`, as for example:

```
ftype fp_one, fp_two;
```

declares two function pointer variables. Function pointers can be declared anywhere a variable declaration may occur, i.e., in the global scope, inside processes and functions, inside structure definitions, etc. Arrays of function pointers can also be declared. Unless initialized in declaration, a function pointer is undefined i.e., it holds the `null` value. A function pointer can hold a reference to any declared function. The assignment statement in the following example:

```
function bar()
{
 ...
}

fp_one = bar;
```

makes `fp_one` point to the `bar` function. Assignments between two function pointers are allowed, like:

```
fp_two = fp_one;
```

but no assignments involving a function pointer and any other kind of expression are legal. Upon assignment, a function pointer can be used in function invocations, instead of the function name:

```
fp_one();
```

The above statement, assuming that it occurs after the assignment of `fp_one` to `bar` determines the invocation of the `bar` function. A function pointer can be used in any form of function invocation statement. Any attempt to make a function call using an undefined (`null`) pointer will raise a run-time error.

Function pointers can be used in equality comparison expressions involving the `==` and `!=` operators. An equality between two function pointers yields true if both operands point to the same function or they are both `null`.

Another point that must be addressed in order to complete the specification of the function invocation mechanism regards the parameter passing. Normally, any expression can be passed as a function actual parameter in invocation, including pointer variables and function pointers. There is however one restriction concerning pointers (common pointers, not function pointers) which are passed as actual parameters to a function: at run-time they will always evaluate to their left value. This restriction is raised by the fact that function invocations by means of function pointers do not carry out information concerning the type of formal parameters. In order to minimize confusion we extended this restriction to all function invocations, for example in the following code:

```
int &p;
 ...
some_function(p);
```

the invocation `some_function(p)` will receive as actual parameter the left value of `p`.

Local Scopes

The notion of function introduces also the concept of *local scope* for a variable. Variables can be declared inside a function, being visible only within the function scope. Moreover, a local scope can appear inside any process, function or local scope. The local scopes can therefore be nested, as shown by the following code:

```
{
  int x = 1;
   ...
  {
    int x = 2;
     ...
    printf(``%d\n'', x);
  }
  printf(``%d\n'', x);
}
```

Here two nested local scopes are defined, each one containing a local variable declaration `x`. Note that the second declaration of `x` overrides the first one, therefore the above code will print first the value `2` and then `1`. As a general rule, we consider that every sequence of statements enclosed in curly braces, except the `atomic` and `d_step` sequences, and also the escape sequence of an `unless` construct defines a local scope. The restriction about `atomic`, `d_step` and escape sequences arises due to a somehow subtle reason. When entering a local scope an invisible action is performed. This is always executable, which may affect the behavior of the `atomic`, `d_step` and escape sequence. Variables defined within a local scope are automatically initialized when the control enters the scope.

Conclusions

This paper presents a language extension for the model checker SPIN, called the "dynamic" SPIN. Here "dynamic" has roughly the same meaning as it has in the context of compiler development, identifying the run-time information which cannot be obtained just by performing a static analysis of the source program. dSPIN provides support for modeling this kind of information in order to aid people in writing models of complex object-oriented software and verify them, while the standard SPIN is mostly intended for software protocols verification.

References

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

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

[3] Bjarne Stroustrup, *The C++ Programming Language*, Wesley, 1991.

[4] Andrew W. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press, 1998