Alias Analysis by Means of a Model Checker[+]

Vincenzo Martena and Pierluigi San Pietro

Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci, 32. Milano 20133, Italia
{martena, sanpietr}@elet.polimi.it

Abstract. We study the application of a standard model checker tool, Spin, to the well-known problem of computing a may-alias relation for a C program. A precise may-alias relation can significantly improve code optimization, but in general it may be computationally too expensive. We show that, at least in the case of intraprocedural alias analysis, a model checking tool has a great potential for precision and efficiency. For instance, we can easily deal, with good precision, with features such as pointer arithmetic, arrays, structures and dynamic memory allocation. At the very least, the great flexibility allowed in defining the may-alias relation, should make it easier to experiment and to examine the connections among the accuracy of an alias analysis and the optimizations available in the various compilation phases.

1 Introduction

Two symbol or pointer expressions in a program are an *alias* when they reference the same memory location.

Alias analysis is the activity of detecting which expressions, at a given point in a program, are not aliases of each other. Static (i.e., compile-time) alias analysis is very important for generating efficient code [1], since many compiling optimizations rely on knowing which data could be referenced by a load or store expression [2,3]. Currently, in compilers for Instruction-Level Parallelism (ILP) processors alias analysis is even more important since it can improve the performance of the instruction scheduler [4]. However, exact alias analysis is impractical, and in general undecidable [5], because of the difficulties in determining which objects are referenced by pointers at a given point in a program. Hence, every approach to alias analysis makes some conservative approximations to the alias relation, determining what is called a *may-alias* relation. A pair of expressions $(e1,e2) \in$ *may-alias* in a given point of the program if a static analysis determines that "there is a chance" that *e1* and *e2* address the same memory cell at that point during some execution of the program. The relation must be *conservative*, i.e., if $(e1,e2) \notin$ *may-alias* then it is impossible that *e1* and *e2* may reference the same cell at that point: if this were not the case, a code optimization allocating different addresses to *e1* and *e2* would deliver an incorrect program. Clearly, the may-alias relation is an approximation that must be a compromise between preci-

[+] Work partially supported by STMicroelectronics and by CNR-CESTIA.

R. Wilhelm (Ed.): CC 2001, LNCS 2027, pp. 3–19, 2001.

sion (leading to efficient code) and the temporal and spatial efficiency of the computation.

Approximated static alias analysis has attracted a great body of literature and it is usually distinguished in flow-sensitive or flow-insensitive, context-sensitive or context-insensitive and interprocedural or intraprocedural [2]. In practice, even approximated solutions may be computationally expensive for large programs (e.g., see [6,7]): a precise analysis should be executed only after some other kind of imprecise-but-efficient analysis has shown the necessity of improving the precision for some parts of a program. It is also debatable how much precise an analysis should be to be considered really cost-effective in terms of optimization. The goal of our work is to build a tool that, rather than being adopted directly in a compiler, may be used to explore and assess the importance and usefulness of various approximations to the may-alias relation and also to check the results computed by other alias analysis tools.

The novelty of our approach is the adoption of a standard *model checking* tool, Spin [8]. A model checker is essentially a highly efficient analyzer of properties of finite state machines, using highly optimized concepts and algorithms, developed in years of study and experiments. In our approach, the relevant features of a C program are abstracted into a suitable finite-state automaton and the alias analysis problem is transformed into a reachability problem for the automaton. The advantage of this method is that no analysis algorithm has to be defined and implemented: the model checker takes care of the actual analysis, without having to program or develop any algorithm ourselves. Hence, our approach readily allows to study, extend, and experiment with various advanced features that are usually ignored in alias analysis [9,7,10]. For instance, the prototype we built so far is able to deal, with better precision than usual, with features such as dynamic memory allocation, aggregates (i.e., arrays and structs), multi-level pointers and pointer arithmetic in C.

Our approach to alias analysis may be especially useful in deciding whether a precise analysis effectively improves the performance of the code generated by a compiler for an ILP processor, by assessing whether the treatment of one or more precise features is really useful and worth incorporating in a compiler.

A long-term goal of our research is also the integration of our toolkit with other less precise analysis tools, such as the one being developed in our group [11] and others [12,13]. The model checking tool could then be used to check only those parts of a program that are found in need of greater optimization and thus require more precision in the analysis. Alternatively, if experiments with the model checker may show that a certain kind of precise analysis is feasible and useful for optimization, specific algorithms and tools may also be developed.

The paper is structured as follows. Section 2 briefly describes a model checker and introduces its usage in alias analysis on some short, but "difficult" examples. Section 3 summarizes the experimental results obtained so far, discussing issues of efficiency and extendibility. Section 4 draws a few conclusions and directions of future research.

2 Model Checking for Alias Analysis

Model Checking is the automated verification that a (often, finite state) machine, described in a suitable format, verifies a given property. The property must be described with a formal notation, which can be either the description of another machine or a temporal logic formula.

If the verification of the property fails, the Model Checker (MC) tool builds a counterexample, i.e., an execution trace of the specified system that leads to the violation of the desired property. By examining such traces, it is usually not difficult to identify the cause of the failure.

Model checking techniques have received great attention in the last years, due to the successes of the automatic verification of finite-state systems describing protocols, hardware devices, reactive systems. A symbolic MC [14] may routinely check systems with 10^{10} reachable states, and the case of 10^{8} states is routine also for on-the-fly model checkers as Spin. In certain cases, much larger numbers have been obtained, but they usually correspond to less than a few hundreds bits of state space. This means that any non-trivial software system cannot be checked as it is. The approach for applying a MC to software programs and specifications is to use some form of *abstraction* to reduce the number of states. An abstraction omits many details, such as data structures, of the system to be checked. Finding the right abstraction is a very difficult problem, and there is in general no guarantee that the verification of the abstracted system brings any information about the real system. In case of alias analysis, however, we are mainly interested in pointer expressions and we can thus abstract away significant parts of the code to be checked, without loosing the correctness of the results. Also, various conservative approximations may be applied to the original code (e.g., replacing branch conditionals with nondeterministic choices), leading to small, and efficiently analyzable, finite state machines, but still providing a good precision of the analysis. The abstraction we propose is one of the main contributions of this paper.

2.1 The Spin Model Checker and the Promela Language

Spin is a widely distributed software package that supports the formal verification of distributed systems. For the sake of brevity, we do not include here a description of the tool itself and of its use, which is widely available also via web.

The input language of Spin is called Promela. We cannot describe here a language rich and complex such as Promela, whose description is widely available. The syntax of Promela is C-like, and we ignore here Promela's communication aspects. Conditional expressions may be defined as: `(expr1 -> expr2 : expr3)`, which has the value of `expr3` if `expr1` evaluates to zero, and the value of `expr2` otherwise. A special `skip` statement denotes the null operation, i.e., it does nothing (its use derives from the syntactic constraints of Promela that do not allow "empty" statements). Processes may be declared with a `proctype` declaration. We are only interested in declaring one process, `active proctype main()`, which basically corresponds to the `main()` function of a C program. Functions and procedures may be declared with an

inlining mechanism. The control structures are a selection (`if`) statement and a repetition (`do`) statement, but it is also possible to use labels and `goto` statements. Selection has the form: if `:: statements ... :: statements fi`. It selects one among its options (each of them starts with `::`) and executes it. An option can be selected if its first statement (the guard) is enabled (i.e., it evaluates to a number greater than 0). A selection blocks until there is at least one selectable branch. If more than one option is selectable, one will be selected at random. The special guard `else->` can be used (once) in selection and repetition statements and is enabled precisely if all other guards are blocked. Repetition has the form: `do :: statements ... :: statements od`. It is similar to a selection, except that the statement is executed repeatedly, until the control is explicitly transferred to outside the statement by a `goto` or a `break`. A `break` will terminate the innermost repetition statement in which it is executed.

2.2 Applying Model Checking to Alias Analysis on Demand

The application of the Spin model checker to alias analysis requires to deal with two distinct problems:

1) how to encode the program to be analyzed into the Promela language of Spin;
2) how to encode and retrieve the alias information.

Encoding C into Promela.
The problem of encoding C into Promela is essentially the issue of finding the right abstraction to solve the may-alias problem, since the model checker cannot deal exactly with all the data and variables used in a real C program. In alias analysis, the important task is to find aliasing information for pointer expressions: a good abstraction could remove everything not directly related to pointers (such as the actual data values). Actual address values (integers) may be dealt with by Spin, provided their value range is finite (32-bit integers are supported). Hence, each nonpointer variable in a program may be replaced by an integer constant, denoting its address. Also a pointer variable must have a static address, but it must also *store* an address: it corresponds to an integer variable (actually, an array of variables is introduced, one cell of the array for each pointer, to make it easier to reference/dereference dynamic structures and multi-level pointers).

Also, as it is usually the case in alias analysis, we can ignore the actual conditions in conditional branching instructions, by replacing them with nondeterministic choices, at least when they do not involve pointer expressions. Notice that Spin, when confronted with a nondeterministic choice among two alternatives, always keeps track that only one is actually taken. This is not often implemented in alias analysis. For instance, the C statement:

```
if (cond) {p1=&a; p2=&b;}
else {p1 = &b; p2 = &a;}
```

is translated into Promela with a nondeterministic choice between the two branches, but no mixing of the information is done: Spin does not consider `p1` and `p2` to be aliases.

We may also make Spin to deal with `malloc()` instructions, which generate a new address value and allocate new memory, by simulating this generation. However, since in general the number of executions of a `malloc` may be unbounded, we make the conservative assumption that each occurrence of a `malloc` in a C program may generate explicitly only a certain fixed number of actual addresses (e.g., just one), and, after that, the `malloc` generates a fictitious address that is a potential alias of every other address generated by the same occurrence of the `malloc` statement. Records (`struct`) types may be dealt as well, by allocating memory and computing offsets for the pointer fields of a new `struct` and by generating new addresses for the nonpointer fields.

Encoding and Retrieving the Alias Information.
The typical application of a model checker is to build counterexamples when a given property is violated, or to report that no violation occurs.However, for recovering the may-alias relation after verification with Spin, we need to have aliasing results rather than counterexamples. Our idea is to use a "byproduct" of model checking analysis, namely unreachability analysis. A model checker like Spin is able to report the unreachable states of a Promela program. Hence, we can add a simple test stating that a pair of pointer expressions are aliases of each other, followed by a null (`skip`) operation to be executed in case the test is verified: if the analysis shows that the null operation is unreachable in that point, then the two expressions cannot be aliases.

In this way, it is possible to compute the complete may-alias relation in each point of the program, even though this would mean adding a great number of pairs (test,null operation) to the Promela program, since the total number of states only increases linearly with the number of may-alias tests. It is however more natural and convenient to compute only the part of the may-alias relation that is deemed useful by a compiler.

2.3 The Translation Scheme

A simple translator is being designed to compile C programs into Promela. In this section, we illustrate the translation scheme by translating a few examples of simple, but not trivial, C programs. All the programs will be monolithic, i.e., not decomposed in functions. This is not a limitation, since nonrecursive function calls may be replaced by code inlining; however, the inlining could cause a significant slowdown for the interprocedural analysis of large programs.

Pointer Arithmetics and Static Arrays.
The first program, called `insSort.c`, shown in Fig. 1, is a simple insertion sort of an array of integers, using pointer arithmetics. The program reads 50 integers, store them in an array `numbers` and then prints them in increasing order. The line numbers are displayed for ease of reference.

For instance, we may be interested in computing alias information about `position` and `index` at the beginning and at the end of the inner loop (lines 13 and 18). The `int main()` declaration corresponds in Promela to the declaration `active proctype main()`. We now show how to deal with variable declarations. Each non-pointer variable declaration is replaced by the assignment of a progressive integer

constant (the address) to the variable name. The addresses conventionally start at one. For instance, `int key;` is replaced by: `#define key 1`.

Each pointer declaration corresponds to the Promela declaration of an integer variable. To allow a homogenous treatment of multi-level pointers and of dynamic memory allocation, an array of integers is declared, called `Pointers`: each pointer is an element of the array. Each pointer name is defined in Promela by an integer constant (the static address of the pointer), which is used to index the array. Since no dynamic memory allocation is used in this program, the dimension of this array is simply the number of pointers in the program, i.e., two. We use three instead, since for various reasons we prefer to ignore the first element of the array. Each pointer variable is then replaced by the constant index of the array.

Hence, the declarations: `int *index;  int *position;` are replaced by the Promela declarations:

```
1.int main(){
2.  int key;
3.  int numbers[50];
4.  int *index;
5.  int *position;
6.    for  (index=numbers;  index<numbers+50;  index++)  /*read
  array */
7.    scanf("%d", index);
8.  index=numbers+1;
9.  while(index<numbers+50){ /*sort the array */
10.   key=*index;
11.   position=index;
12.   while (position>numbers){
13.    /*position and index are aliases here? */
14.    if ( *(position-1) > key) {
15.                  *position= *(position-1);
16.    }
17.          position--;
18.          /*position and index are aliases here? */
19. }
20. *position=key;
21. index++
22.  }
23. printf("Sorted array\n");
24.  for  (index=numbers;  index<numbers+50;  index++)  /*print
  array */
25. printf("%d\n",*index);
26.}
```

Fig. 1. The `insSort.c` program: a simple insertion sort in a static array

```
int Pointers[3];
#define index 1;
#define position 2;
```

The actual address referenced by the pointer `index` is denoted with `Pointers[index]`, while the address `&index` is denoted by the name `index` itself. To make pointer expressions easier to write and generalize, we prefer to introduce a C-preprocessor macro, called `contAddr` ("content at the given address"), which returns the address stored in a pointer:

```
#define contAddr(expression) Pointers[expression]
```

Hence, `contrAddr(position)` corresponds in Promela to the address that in C is denoted by `position`, while `position` actually denotes in Promela what in C is denoted by `&position`.

An array declaration, such as `int numbers[maxEl]`, is considered as a declaration of a group of non-pointer variables, in this case `maxEl` variables. Hence, we allocate `maxEl` consecutive integer constants for the addresses of the array elements, without allocating any memory for the array itself. The array declaration is replaced by `#define numbers 2;`

Lines 6 and 7 only read values in the array and are thus ignored. The assignment of line 8 means that the address of the second cell of the array `numbers` (i.e., the address `numbers + 1`) is assigned as the new value of the pointer `index`. To make the Promela program more readable and extendable, we introduce a macro of the C preprocessor for this kind of assignments, called `setAddr`:

```
#define setAddr(P,expression) Pointers[P] = expression
```

Hence, `index=numbers +1` becomes `setAddr(index,numbers+1)`. The external `while` loop of the insertion sort (line 9) must be replaced by a Promela loop. Hence, a `while(C) B;` statement is replaced by the Promela statement: `do ::P(C)-> P(B); ::else -> break; od` where `P(C)` and `P(B)` are the Promela translation of the conditional `C` and of the (possibly compound) statement `B`, respectively. In this case, the condition is translated to `(contrAddr(position)>numbers)`

The assignment statement of line 10 is ignored, since no pointer value is involved, while the one of line 11 (between pointers) becomes `set(position,index);` where `set` is defined by the following C-preprocessor macro:

```
#define set(P,ex) setAddr(P,contAddr(ex))
```

Both `set` and `setAddr` assign a value to the cell indexed by the first argument `P`, but `setAddr` considers this value to be directly the second argument, while `set` considers this value to be the content of the cell whose address is the value of the second argument. Hence, the C fragment: `int a; int *p, *q; p = &a; q=p;`

is translated into: `setAddr(p,a); set(q,p);`

The conditional `if` statement of lines 14 to 16 can be eliminated. In fact, the condition cannot be computed on pointer values only and should be replaced by a nondeterministic Promela if statement. However, since line 15 must be eliminated as well, the translation of the conditional would be:

```
if ::true ->  skip; ::true -> skip; fi
```

which obviously has no effect. Line 17 becomes `Pointers[index]++;` or, alternatively, `setAddr(index,contAddr(index)+1)`, and the other lines may be ignored.

The alias information is introduced, in the points corresponding to the lines 13 and 18, by using a simple test, defined via a pair of macro as follows:

```
#define mayAlias(X,Y) if ::(Pointers[X] == Pointers[Y]) ->
skip;
#define notAlias   ::else-> skip; fi
```

The complete translation of the C program above is reported in Fig. 2.

When Spin is run to analyze the program, it reports that the `mayAlias` condition of line 16 is the only unreachable part of the code. Hence, `position` and `index` do not belong to the may-alias relation at line 18 of the original program. Instead, since the other `mayAlias` condition is reachable, the two pointers do belong to the may-alias relation at line 13 of the original program. Notice that in the latter case the relation is not a must-alias relation (since also the `notAlias` condition is reachable as well). The relationship between the speed of Spin in providing the answer, the total memory usage and the size of the static array is reported in Section 3.

Multi-level Pointers.
Multi-level pointers can be easily dealt with. For instance, a pointer declared as `int** p;` is declared in Spin again as a static address for `p`, used as an index of the pointer array. The operator `contAddr` can be used to dereference a pointer expression. Consider the following fragment of code:

```
char **p2; char * p1; char a;
...
*p2 =&a; p2 = &p1; *p2 = p1; p1 = *p2;
...
```

This can be easily translated into Promela:

```
#define a 1
#define p1 2
#define p2 3
...
setAddr(contAddr(p2), a);
setAddr(p2,p1);
set(contAddr(p2),p1);
set(p1, contAddr(p2));
```

Structures and Dynamic Memory Allocation.
Traditionally, in alias analysis, programs with dynamic memory allocation are either ignored or dealt with in a very limited way: when a `p =(T*) malloc(sizeof(T));` instruction occurs in a program, the expression *p is considered to be an alias of every other pointer expression of type `T`. At most, some separate treatment is introduced by distinguishing among the pointers initialized with `malloc` instructions at different lines of the program [7,9]. This is a conservative assumption, assuring that the may-aliases relation always includes every pair of poi experessions that are aliases, but it is often too imprecise. For instance, if `T` is `struct {int`

`info; T* next;}`, the pointer expression `p->next` is assumed to be an alias of every pointer expression of type `T*`.

```
1.#define maxEl 50
2.#define  max_element  3  /*the  number  of  pointers
  declared in the program*/
3.#define key 1
4.#define numbers 2
5.#define index 1
6.#define position 2
7.int Pointers[max_element];
8.active proctype main () {
9.     setAddr(index,numbers+1);
10.    do     ::contAddr(index) < (numbers+maxEl) ->
11.                  set(position,index);
12.           do ::(contAddress(position)>numbers) ->
13.                        may_alias(position,index)
14.                        notAlias;
15.                  contAddr(position)--;
16.                        mayAlias(position,index)
17.                        notAlias;
18.              ::else -> break;
19.           od;
20.           contAddr(index)++;
21.    ::else -> break;
22.    od
23.}
```

Fig. 2. The Promela translation of insSort.c

For a more complete example, consider the simple program `doubleList.c` of Fig. 3, which inserts two integers in a double-linked list built from scratch.

```
1.struct elem {int inf; struct elem next*; struct elem previ-
  ous*;};
2.int main(){
3.  struct elem *root;
4.  root = (struct elem*)malloc(sizeof(struct elem));
5.  scanf("%d", &(root->inf));
6.  root->previous = null;
7.  root->next = (struct elem*)malloc(sizeof(struct elem));
8.  root->next->previous=root;
9.  root->next->next = null;
10. scanf("%d", &(root->next->inf));
11. /* root->next and root->next->previous may be alias here?*/
12.}
```

Fig. 3. The `doubleList.c` program, which inserts two elements in a double-linked list

In a traditional approach, `root->next` and `root->next->previous` are all considered to be aliases, preventing optimizations. We will show that this is not the case in our approach.

To make this presentation simple, here we show how to deal with `malloc` instructions combined with variables of type `struct*`, ignoring statically-declared variables of type `struct` and dynamic allocation of pointers to values of the basic types. Also, we assume that the address of a nonpointer field of a structure is never taken, i.e., there is no statement of type `x = &(root->inf)`. The latter restriction allows us to ignore the non-pointer fields, since in this case no pointer expression may be an alias of a nonpointer field. Notice that the actual translation we implemented does not have any of these limitations, which can be relaxed without any loss in efficiency by using a more complex translation. Another limitation is that we do not allow casting of structure types (as instead done in [15]), since we treat each pointer field as a separate object based on its offset and size: the results would not be portable because the memory layout of structures is implementation-dependent.

The declaration of the `struct` of line 1 corresponds to the following Promela declaration:

```
#define sizeof_elem 2 /*number of pointer fields in elem */
#define offset_elem_next 0 /*offset of "next" field */
#define offset_elem_previous 1 /*offset of "previous" field
*/
```

Number all the `malloc` statements in the source code, starting from 1. We introduce in Promela a malloc statement called `malloc(P,T,N)`, where `P` is a pointer expression of type `T*`, `T` is the type name and `N` is the `malloc` number. Hence, a statement of the form `p = (int*) malloc(sizeof(int));` (corresponding to the 3rd occurrence of a `malloc` in the source code) is translated in Promela into `malloc(p,int,3);`

A `malloc(P,T,N)` statement must generate a new address for a variable of type `T` and store it in the cell of the `Pointers` array whose index is `P`. To be able to define dynamically new addresses, the size of the array `Pointers`, defined via the constant `max_element`, must be greater than the number of pointer variables declared in the program. A new address can be easily generated by declaring a global counter variable, called `current`, to be incremented of `sizeof_T` each time a new address is allocated (and to be checked against `max_element`). The new address may again be used to index the array `Pointers`. Also, there is a global array `count_malloc` of integer counters, which keeps count of the number of actual allocations for each `malloc`. Only a limited number of addresses, denoted by the constant `max_malloc`, is in fact available for each `malloc` (typically, just one): hence, if no new address is available (`count_malloc[N] >= max_malloc`), the `malloc` must return a fictitious address, whose value is conventionally `all_alias +N`, where `all_alias` is a constant denoting a special, fictitious address that is used to signal that an expression address is actually an alias of every other expression. Every address greater or equal to `all_alias` is considered to be fictitious, i.e., it does not refer to an actual memory cell in the Promela program (hence, `all_alias` must be greater than `max_element`). The `malloc` statement in Promela is defined as follows (with the C preprocessor, "\" is used to define a multi-line macro, while "##" denotes string concatenation)

```
int current=total_number_of_pointer_variables+1;
```

```
int Pointers[max_element];
byte count_malloc[total_number_of_malloc];
#define malloc(P,T,n) \
     if \
     :: P >= all_alias -> skip; \
     :: else -> \
            if \
            :: (current + sizeof_##T <= max_element)&& \
                count_malloc[n-1] < max_malloc -> \
                Pointers[P] = current; \
                current=current+sizeof_##T; \
                count_malloc[n-1]++; \
            :: else -> Pointers[P]=all_alias+n; \
            fi; \
     fi
```

`P` stands for the address (index) of the memory cell where the newly generated address must be stored. Hence, the test `P >= all_alias -> skip;` is introduced in order not to allocate memory to a nonexistent cell. If the test is false, the `malloc` has to check whether there is space enough in the unused portion of the array `Pointers` to allocate `sizeof_T` consecutive cells (`current + sizeof_T <= max_element`) and whether the maximum numbers of addresses available for that single `malloc` has not been exceeded (`count_malloc[n-1] < max_malloc`). If the test is passed, the cell of index `P` is assigned the new address, the current counter is incremented of `sizeof_T`, and `count_malloc[n-1]` is incremented of one. Otherwise, the fictitious address `all_alias+n` is stored in `Pointers[P]`, to denote that the n-th `malloc` has not been able to allocate a new address.

To be able to deal with structures and their fields, we need to introduce Promela equivalents of `root->previous`, `&(root->next)`, etc. We first modify the operator `contAddr` to be able to deal with fictitious addresses:

```
#define contAddr(expr) (expr >= all_alias -> expr : Point-
ers[expr])
```

Hence, if the address of `expr` is fictitious, the array `Pointers` is not accessed and the fictitious value is returned.

The address of a field of a pointer expression `expr` of type `T` is given by the Promela macro `getField`:

```
#define getField(expr,field,T) (contAddr(expr)>=all_alias ->
all_alias :\
          Pointers[expr]+offset_##T##_## field) \
```

Hence, the access of a field returns the `all_alias` value if its address is fictitious, the address of the field otherwise. The latter address is obtained by adding, to the address of the first cell of the structure, the value of the offset of the field inside the structure. Hence, line 7 of the above C program:

```
root->next = (struct elem*)malloc(sizeof(struct elem));
```

is denoted in Promela by:

```
malloc(getField(root,next,T), T,2);
```

When a field is assigned a new value, e.g., `root->previous = null;` it is not possible to use the `setAddr` operator defined above, since now the various pointer expressions involved may also assume fictitious values. Hence, we redefine the operator as follows:

```
#define setAddr(address,expr) if \
          ::address >= all_alias -> skip; \
          ::else -> Pointers[address]=expr; \
          fi
```

When a pointer expression of address `address` is assigned the value `expr`, we first check whether `address` denotes a valid cell, otherwise no assignment can be executed.

Hence, `root->previous = null;` is translated into:

```
setAddr(getField(root, previous), null);
```

where the `null` pointer is simply the constant 0: `#define null 0`

The `mayAlias-notAlias` pair has now to be extended: in fact, two pointer expressions may be alias also because either one evaluates to `all_alias`.

We extend the operators as follows:

```
#define mayAlias(ex1, ex2) if \
:: (contAddr(ex1) == contAddr(ex2) && \
   contAddr(ex1) != null) -> skip; \
:: (contAddr(ex1) == all_alias || \
   contAddr(ex2) == all_alias) -> skip

#define notAlias :: else -> skip; fi
```

The `mayAlias` operator checks whether the two expressions `ex1` and `ex2` reference the same cell, provided that the cell is valid and that the expressions do not refer to the null pointer. Notice that two cells with invalid but different addresses, such as `all_alias+1` and `all_alias+2`, are not considered to be aliases of each other (since they were generated by two distinct `malloc` instructions). Otherwise, the `mayAlias` operator checks whether one of the two expressions is an `all_alias`: in this case, the two expressions are also considered potential aliases. The `notAlias` operator corresponds to the case where neither of the two previous cases occurs: the two expressions cannot be aliases. The complete translation of the `doubleList.c` program is reported at *http://xtese1.elet.polimi.it*.

The result of the analysis with Spin is that the only unreachable part is the `mayAlias` condition of line 16, meaning that, as expected, `root->next` and `root->next->previous` cannot be aliases. Since the program is quite small, the execution time and memory requirements of Spin are negligible.

3 Experimental Results

Table 1 summarizes the experimental results obtained for the example `insSort.c` of Section 2. The running times are given on a PC with a Pentium III 733 MHz processor, with 256 KB cache Ram, 256 MB Ram and the Linux OS (vkernel 2.4.0, glibc

Pointers Elements	Array El.	State Vector (byte)	States	Memory Usage (Mbyte)	Approximated Analysis	Search Depth	Running Time (sec)
4	40	28	5085	2.527	No	5084	0.01
4	80	28	19765	3.192	No	19764	0.06
4	160	28	77925	5.861	No	77924	0.24
4	320	28	309445	16.501	No	309444	1.07
4	640	28	1.23e+06	62.716	No	1.233e+06	4.68
4	1000	28	3.00e+06	151.140	No	3.000e+06	35.6

Table 1. Performance results for `InsSort.c`

2.1). Notice that the running times do not include the compilation time (Spin generates a C program to be compiled with gcc before execution), which usually is fairly independent on the size of the Promela program and takes a few seconds. Memory occupation is often dominated by the stack size, which must be fixed before the execution. Hence, if a stack size too large is chosen, the memory occupation may seem quite large even if only a few states were reachable. On the other hand, the exploration of a large number of states may be very memory consuming, even though often can be completed in a very short time.

As it can be noticed, the number of states explored by Spin with `insSort.c` is quadratic in the size of the static array, but the running time is still small even when the array has hundreds of elements. The quadratic complexity is not surprising, since it results from the time complexity of the original insertion sort algorithm. The memory occupation is large, and it is dominated by the size of the stack used by Spin, while the number of bits in each state (called *state vector size* in Spin) is negligible, since the array size only impacts on the number of reachable states.

We also explored the use of Spin using "critical" examples, to check whether this approach really improves precision of analysis on known benchmarks. We considered, among others, one example taken from [7]. The example was introduced by Landi in order to find a "difficult" case where his new algorithm for alias analysis with dynamic memory allocation performed poorly. Landi's original source code and its Promela translation are reported at *http://xtese1.elet.polimi.it*. This example represents the worst case for Landi's algorithm, which finds $3n^3 + 7n^2+6n+18$ aliases, where the parameter `n` is the size of the array `v`. Our analysis, instead, finds the exact solution, `n+11`, for each fixed `n`. Notice that our analysis is also able to distinguish that `b` and `d` cannot be alias after the execution of the two conditionals inside the `while` loop. Table 2 shows also the performance of the tool for various values of the parameter `n`. The running time is still quadratic, even though the original program runs in linear time. In fact, both the number of states searched and the state vector size increase linearly: the memory occupation and the running time must be a quadratic function of `n`.

Array Elements	State Vector (byte)	States	Memory Usage (Mbyte)	Approximated Analysis	Search Depth	Running Time (sec) (compilation excluded)
40	432	754	2.302	No	387	0.01
80	432	1394	2.404	No	707	0.03
160	832	2674	2.849	No	1347	0.09
320	1632	5234	3.822	No	2627	0.36
1000	4112	16114	11.557	No	8067	2.92
2000	8112	32114	36.957	No	16067	11.43
4000	16112	64114	135.757	No	32067	46.00

Table 2. Performance Results for Landi's example

Another example of a nontrivial program has been translated into Promela and the quality of the alias analysis has been assessed and the performance results have been studied. The example takes in input two series of integers in two separate linked lists, calculates the maximum element of the two lists and then stores their sum in a third list. The source code LinkedLists.c, along with its translation, is reported at *http://xtese1.elet.polimi.it*, where we will collect further experimental results. The example is composed of 116 lines of C code and makes a heavy use of dynamic memory allocation to insert elements in the various lists. The tool is able to distinguish as not being aliases at least the heads of the tree lists, even though the analysis cannot be exact due to the presence of malloc instructions inside unbounded loops. Since each malloc instruction is allowed to generate only a very limited number of new non-fictitious addresses, a very small number of pointers can be allocated dynamically by the Promela version. Hence, the maximum number of dynamically allocated pointers does not affect the performances.

This and the above results show that the running time and memory occupation do seem to depend just on the sheer size of the program to be analyzed, but especially on the number of variables. Notice that the memory occupation could be considerably reduced in most examples, by reducing the size of the available addresses (now, 32-bit integers).

4 Conclusions and Related Works

In this paper, we presented a prototype tool, based on the model checker Spin, to study and experiment with alias analysis. There are many known algorithms and tools that can be used to determine may-alias relations, with varying degrees of accuracy (for papers including broad references to alias analysis see for instance [16]). At the best of our knowledge, we ignore of any previous application of model checker technology to this goal. However, there are various studies about the application of model checking to flow analysis, e.g. [17]. In these works, it has been shown how many problems usually solved by means of data-flow techniques can be solved more simply by model checking techniques. In particular, our work is consistent with the methodology proposed in [18], which uses abstract interpretation [19] to abstract and analyze programs by combining model checking and data flow techniques. The links among abstract interpretation and data-flow analysis are also well-known (e.g., [20]). There

is also some relation with works, such as [21], that transform program analysis (e.g., flow-insensitive point-to analysis), into graph reachability problems (on context-free languages), since in our approach the may-alias relation is computed by using the related concept of state reachability (on finite-state automata). Our approach has also connections with works rephrasing static analysis questions in terms of systems of constraints (such as the application of the Omega-library in [22]). There are different approaches that try to deal with dynamic memory allocation. For instance, [23] performs very precise or even exact interprocedural alias analysis for certain programs that manipulate lists, using a symbolic representation. However, the result only has theoretical relevance, since it fails to scale up to real cases. Another method is shape analysis [24], that gives very good results in dealing with acyclic dynamic structures, but does not appear to deal with the cyclic case, such as the double linked lists studied in this paper.

By using our tool, we were able to experiment with more precise may-alias relations than are usually considered in intraprocedural analysis, allowing the study of programs including features such as pointer arithmetics, structures, dynamic allocation, multi-level pointers and arrays. Preliminary experiments with the tool show that it could have better performances even than specialized tools for precise analysis, but further experiments are required to assess this claim. Future work will apply the tool to standard benchmarks and C program, after having completed a translation tool from C to Promela. We are going to apply the tool to object oriented languages such as Java and C++ and to study other applications to static analysis of programs (such as detecting bad pointer initialization). An advantage of our approach is that all the experiments can be easily performed without writing any algorithm, but only defining a few operators and leaving all the processing job to the highly optimized Spin model checker.

The computation of the may-alias relation with a model checker is consistent with an approach to alias analysis and program optimization called *alias on demand* [11]. The approach is based on the consideration that in the optimization phase not all elements in the may-alias relation have the same importance: first, only those parts of the code that are executed frequently (such as loops) may deserve optimization, and hence the may-alias relation for the other parts is not useful; second, many parts of the may-alias relation do not enable any optimization. Therefore, in this approach the may-alias relation is not computed entirely before the optimizations are performed, but only after a preliminary optimization analysis has determined that a certain set of instructions could be optimized if there is no aliasing among certain pointer expressions. For instance, an optimization such as code parallelization for ILP processors must rely on limited parallel resources: most of the may-alias relation is completely irrelevant, since we cannot parallelize many instructions anyway. Therefore, it is possible to save a great deal of computational power that is otherwise needed for alias analysis.

For the moment, interprocedural analysis with our tool must be done by inlining the procedure calls in the main. Hence, precise analyses of the kind described here are probably infeasible for large programs. However, our approach could be easily tailored to deal with the precision that is required for the problem at hand. For instance, loops could be transformed in conditional statements and further details of a program could be omitted, leading to a less precise, but more efficient, analysis when and

where precision is not essential. Again, this could be obtained with almost no cost, leaving the user of the tool the possibility of customizing the analysis to the level of precision that is required. An alternative is to use infinite-state model checking, which however is still a subject of intense research.

A major goal of our group at Politecnico di Milano is to analyze the benefits that can be obtained by using alias information in the various compilation phases (register allocation, instruction scheduling, parallelization) and to examine the connections among the accuracy of the required alias analysis and the intended optimizations [1], also in view of the machine architecture. We believe that our tool could easily be tailored to support this kind of analysis.

Acknowledgements. We thank Marco Garatti and Giampaolo Agosta for their help and their comments. Special thanks to Stefano Crespi Reghizzi for making this research possible by providing us with ideas, encouragement and suggestions.

References

1. M. Shapiro and S. Horwitz, *The effects of the precision of pointer analysis*, In P. Van Hentenryck, (ed.), 4th Intern. Static Analysis Symp., 1997, LNCS 1302, pp. 16-34. Springer-Verlag.
2. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
3. D. Bacon S. Graham and O. Sharp, *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, Vol. 26, n. 4, 1994, 345-419.
4. Joseph A. Fisher, *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Transactions on Computers, 1981, Vol. 30, n. 7, 478—490.
5. G. Ramalingam, *The Undecibility of Aliasing*, ACM TOPLAS, Vol. 16, n. 5, 1994, 1467-1471.
6. S. Horwitz, *Precise Flow-Insensitive May-Alias Analysis is NP-Hard*, ACM TOPLAS, Vol. 19, n. 1, 1997, 1-6.
7. W. A. Landi, *Interprocedural Aliasing in the Presence of Pointers*, Rutgers Univ., PhD Thesis, 1997.
8. G. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
9. W. Landi and B. Ryder, *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing*, Prog. Lang. Design and Impl. ACM SIGPLAN Not., 1992, 235-248.
10. D. Liang and M.J. Harrold, *Equivalence Analysis: A General Technique to Improve the Efficiency of Data-flow Analyses in the Presence of Pointers*, ACM TOPLAS, Vol. 24, n. 5, 1999, 39-46.
11. M. Garatti, S. Crespi Reghizzi, *Backward Alias Analysis*, Tech.Report, Dip. di Elettronica E Informazione, Poilitecnico di Milano, Sept. 2000.
12. B. Steensgaard, *Points-to Analysis in Almost Linear Time*, in Proc. 23rd SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Jan,, 1996.pp. 32--41, ACM Press.
13. D. Liang and M.J. Harrold, *Efficient Points-to Analysis for Whole-Program Analysis*, Lectures Notes in Computer Science, 1687, 1999.
14. K. L. McMillan, *Symbolic model checking - an approach to the state explosion problem*, PhD thesis, Carnegie Mellon University, 1992.
15. Yong, S.H., Horwitz, S., and Reps, T., *Pointer analysis for programs with structures* and *casting*, Proc. of the ACM Conf. on Programming Language Design and Implementation, (Atlanta, GA, May 1-4, 1999), in ACM SIGPLAN Notices 34, 5 (May 1999), pp. 91-103.

16. M. Hind, M. Burke, P. Carini and J. Choi, *Interprocedural Pointer Alias Analysis*, ACM TOPLAS, Vol. 21, 4, 1999, 848-894.
17. B. Steffen, *Data Flow Analysis as Model Checking*, Proc. Int. Conf. on Theoretical Aspects of Computer Software,(TACS 1991), Sendai (Japan), September 1991 , pp. 346-365.
18. D.A. Schmidt and B.Steffen, *Program analysis as model checking of abstract interpretations*, G. Levi. (ed.), pages 351--380. Proc. 5th Static Analysis Symp., Pisa, September, 1998. Berlin: Springer-Verlag, 1998. Springer LNCS 1503.
19. Cousot, P. and Cousot, R., *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,* POPL 1977, pp. 238--252.
20. Cousot, P. and Cousot, R., *Temporal Abstract Interpretation*, POPL 2000, pp.12-25.
21. Reps, T., *Program analysis via graph reachability*, Information and Software Technology 40, 11-12 (Nov./Dec. 1998), pp. 701-726.
22. W. Pugh and D. Wonnacott, *Constraint-Based Array Dependence Analysis*, ACM TOPLAS, Vol. 20, n. 3, 1998, 635-678.
23. Alain Deutsch, *Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting*. In PLDI 1994, pp. 230-241. Jun 1994.
24. Wilhelm, R., Sagiv, M., and Reps, T., *Shape analysis*. In Proc. of CC 2000: 9th Int. Conf. on Compiler Construction, (Berlin, Ger., Mar. 27 - Apr. 2, 2000).