Abstracting C with abC

Dennis Dams[*] William Hesse[†] Gerard Holzmann[*]

Abstract

A conceptually simple and practically very useful form of data abstraction in model checking is *variable hiding*, which amounts to suppressing all information about a given set of variables. The abC tool automates this for programs written in the C programming language. It features an integrated demand-driven pointer analysis, and has been implemented as an extension of GCC.

1 Introduction

The starting point of a model checker [2] is a *verification model*. The construction of a suitable such model comprises a major part of the overall verification effort. A well-chosen model may indeed eliminate some or even all sources of the notorious state-explosion problem. Tools for *automated abstraction* assist in the extraction of verification models from system descriptions that are otherwise too detailed. We consider here the checking of C source code, using the FeaVer tool [1] which is based on an extended version of the explicit-state model checker Spin [1, 5]. Case studies with FeaVer have indicated that *data hiding* and *narrowing* are promising candidates for automated abstraction. In data hiding, variables are removed from the program and their value is assumed to be unknown. In narrowing, the types of variables are replaced by smaller types. In both cases, nondeterminism is introduced at the level of the model checker so that the model's behaviour overapproximates the original program, which is needed to ensure preservation of universal temporal properties. Data hiding and narrowing were the most effective types of abstractions used in the verification of Lucent's PathStar call-processing software [6]. In this case study, about 30 so-called “class 5” telephony features were checked mechanically with a method based on model extraction from the call-processing source code. Over a period of roughly two years, FeaVer caught 70 errors in the code. The types of abstraction used, and hiding in particular, are simple enough to justify elaborate tool support, and also to be readily understood and applied by non-expert users.

These findings motivated the development of abC, a tool that automates most of the process of hiding data in C code. Given an initial set of variables to be abstracted, abC performs an analysis to find dependent variables that must additionally be hidden,

[*] Bell Labs, Lucent Technologies, 600 Mountain Ave, Murray Hill, NJ 07974.

[†] Dept. of Computer Science, Univ. of Massachusetts, Amherst, MA 01003.

and then performs the actual program transformation in which all such variables are removed from the program. While both tasks are simple in concept, implementing them for the full (ANSI) C language is a non-trivial effort. The variable-dependency analysis is complicated by the presence of pointers and arrays. In their presence, a *pointer analysis* is necessary in order to still be able to (over)approximate variable dependencies with reasonable precision. abC integrates such an analysis into the algorithm for inference of abstraction, in a demand-driven fashion: Points-to information is only computed when needed for the abstraction inference. The program transformation can in general not just remove entire assignment statements, because a single expression may manipulate both abstract and concrete variables—any concrete side-effects will have to be filtered out and retained in such a case.

The tool has been implemented as an extension to the front end of the GNU Compiler Collection (GCC version 3.0.1, [1]). GCC's representation of the parsed C code is relatively well documented and comes with an extensive API offering many access and utility functions. Furthermore, this representation is shared between compilers of various other languages (including an extension of C that has become commonly used among many programmers), thus facilitating any extension of abC to such languages.

Various other verification platforms offer facilities for data abstraction. Bandera ([1]) comes with a library of abstractions for Java data types which includes the point abstraction, which is similar to hiding. Reference types are currently not handled but work on this extension is underway ([4]). Also the PET tool ([3]) implements a limited form of variable hiding for Pascal programs that does not handle pointers. Other techniques for abstraction, employed in Bandera, Java Pathfinder (JPF, [1]), and the BeBop tool set ([1]) are *slicing* and *predicate abstraction*. Whereas hiding starts from a given set of variables to be abstracted, in slicing and predicate abstraction one specifies variables (or, more generally, conditions) that need to be retained, which are then propagated backwards through the program in order to find all code (generally: predicates) that they depend on. Being dual, both approaches may be used in a complementary fashion.

2 Overview of the Tool

In abC, the objects of abstraction[1] are *memory locations as specified by variable names*. The intuition is that neither the value, nor the address of such a variable is known. This design decision determines the granularity of abstraction. Aggregate variables like arrays and structures can only be abstracted as a whole, not per element or member; in these cases the location of such a variable will be understood to comprise that whole region of memory. Given a set of abstract variables, we call an expression (and also its value) abstract if it contains and abstract variable. When an expression is abstract, its value is unknown. Ultimately, in the *transformation phase*, this will lead to that expression being replaced by the special token `NONDET`, that signifies a nondeterministic choice to the model checker. In order to avoid the introduction of too much nondeterminism, which could easily cancel any positive effect of the abstraction on the state space, an *abstraction inference phase* takes place first.

[1] We refer to objects that are not abstract as *concrete*.

Abstraction inference phase This is a flow-insensitive, context-insensite interprocedural static analysis that finds additional variables to be abstracted, for example because abstract values are stored to concrete locations. Due to the presence of reference types, locations may be denoted by expressions whose value is not determined at compile time (like in `*p=...`). In order not to abstract too many additional variables due to this, the abstraction inference is combined with a pointer analysis. The algorithm is based on 4 inference rules, presented below, that are applied whenever the value of an expression `e2` is stored in a location denoted by `e1`—the typical case being an assignment `e1=e2`. With every expression that denotes a location[2], we associate a *key*—intuitively this a variable in the expression that needs to be abstract if that location must be abstract. For example, the keys of `p`, `p->f`, `p++`, and `q=*p` are all `p`, and the key of `a[i]` is `a`. The definition follows the inductive scheme defining valid expressions in C and is beyond the scope of this paper[3].

1. The key of `e1` must be abstract if (a) the value of `e2` is abstract, or (b) the value of `e1` is abstract.

2. In addition, the key of `e2` must be abstract if `e2` is of pointer type and (a) the value of `e1` is abstract, or (b) the value of `e2` is abstract.

Rules 1 reflect pure abstraction inference. 1(a) says that an abstract value may only be assigned to an abstract location. 1(b) deals with cases such as `a[i]=3` when `i` is abstract. Even when `a` is concrete, the value `3` is stored to an unknown element of the array. The rule then prescribes to also make `a` abstract[4]. Rules 2 combine abstraction inference with pointer analysis. For example, if the program contains a pointer assignment `q=p` then further operations on `q` may influence `p`, like `*q=x`. Rule 2(a) captures such aliasing by requiring that `p` must be abstract whenever `q` is assignment). Finally, rule 2(b) captures the combined effect of 1(a) and 2(a) and is thus merely a shortcut.

Pointer arithmetic (including certain casts) and out-of-bounds array indexing invalidate the rules. Occurrences of the former are flagged by the tool so that they can be manually processed. Wrong array indexing is harder to catch—a separate tool like UNO ([7]) or `lclint` ([1]) may be used. abC also flags calls through function pointers as these disable the interprocedural inference; abstraction of function pointers is disallowed. An example run is given in the appendix.

Transformation phase If the application of the inference rules does not abstract additional variables, the set of abstract variables is closed under dependencies in the sense that in no execution of the program, an abstract value will be stored to a concrete loca-

[2] Examples of expressions that denote a location are `x`, `*p`, `p->f`, and `&y`. Expressions that do not denote a location are e.g. `3`, `x+y`, `p=y`, `a==b`, and `y++` when `x` and `y` are not pointers.

[3] The definition actually associates a *set* of keys with every expression `e`; if `e` does not denote a location then it has no keys, and it may also have multiple keys, e.g. if it contains a conditional expression.

[4] If the array is short, then an alternative would be to leave `a` concrete and replace this assignment by `a[NONDET]=3` signifying that `3` has to be stored to a nondeterministic element in the array. This option could be easily added to the tool if deemed useful.

tion. At this point[5], the program transformation will remove all code that stores values to abstract variables (“abstract assignments”), as well as declarations of abstract variables (unless they contain initializers). Furthermore, abstract expressions that occur as tests in `if`, `while`, etc. are replaced by the token `NONDET`.

This transformation has to take into account the fact that abstract assignments may contain concrete side effects[6]. The transformation of an expression is therefore a recursive procedure. For example, denoting by `tr(e)` the result of transforming expression `e`, `tr(e1+e2)` is defined as `e1+e2` if both operands are concrete, and as the comma expression `((tr(e1),tr(e2)),NONDET)` when either one is abstract. Intuitively, in the latter case `tr` first evaluates any side effects contained inside `e1` and `e2` before returning `NONDET`. The full inductive definition of `tr` is again beyond the scope of this paper.

Example: If the variables `a`, `m`, and `n` are abstract, then the code fragment

```
if (k<m) { int n; a[i++] = b[j++] = ( n = k++, k ); }
```

is transformed into

```
if (NONDET) { ( i++, b[j++] = ( k++, k ) ); }.
```

3 In conclusion

While the usefulness of the functionality offered by abC has been established, in order to quantify its efficacy it needs to be used in case studies, in tandem with a model checker. This is what we plan to do next, and the first results should be available by the time of the conference.

Acknowledgements We thank Ilya Shlyakhter for many discussions on the topic and Kedar Namjoshi for his help with the implementation.

References

[1] **Bandera**: http://www.cis.ksu.edu/santos/bandera **BeBop**: http://www.research.microsoft.com/projects/slam **FeaVer**: http://cm.bell-labs.com/cm/cs/what/feaver **GCC**: http://gcc.gnu.org **JPF**: http://ase.arc.nasa.gov/visser/jpf **lclint**: http://lclint.cs.virginia.edu **Spin**: http://netlib.bell-labs.com/netlib/spin/whatispin.html

[2] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[3] Elsa Gunter, Robert Kurshan, and Doron Peled. PET: An interactive software testing tool. In *Proc. CAV2000*, Springer Verlag, LNCS 1855, pp. 552–556.

[4] J. Hatcliff. Personal communication.

[5] G.J. Holzmann. Logic Verification of ANSI-C Code with Spin. *Proc. SPIN2000*, Springer Verlag, LNCS 1885, pp. 131–147.

[6] G.J. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, Vol. 5, No. 2, April-June 2000, pp. 72–87.

[7] G.J. Holzmann. Static source code checking for user-defined properties. Proc. IDPT June 2002, Pasadena, CA, USA.

[5] The transformation may also be invoked with a non-closed set of abstract variables—this will not be further discussed here.

[6] Two common sources of this are the fact that the C assignment statement is really an expression and hence has a value (equal to the value of the right-hand side), and the presence of the comma expression `(e1,e2)` that evaluates and returns the value of `e2`, but only after evaluating `e1` including any possible side effects.

Appendix

The abstraction inference phase of abC is demonstrated on the following (contrived) C program (line numbers added for convenience).

```
 1 char *nametab[2][2] = { { "(none)", "(none)" }, { "pi2", "xxxxxxxx" } };
 2 typedef struct { float x; float y; char *name; } NamedPoint;
 3 short i = 1;
 4
 5 NamedPoint move(NamedPoint q, short *d)
 6 { float xx = q.x+*d, yy = q.y+*d;
 7   NamedPoint r = { xx, yy };
 8   r.name = nametab[i][i];
 9   return r;
10 }
11
12 int main(void)
13 { char *nullstr = "";
14   NamedPoint newp, p = { 3.1415F, 3.1415F, nametab[i][0]  };
15   nametab[i][i] = "movedpi2";
16   newp = move(p,&i);
17   p.name = nullstr;
18   printf("%s: (%f,%f)\n", newp.name, newp.x, newp.y);
19 }
```

Besides this program, the input to abC consists of a file specifying the initial set of variables to be abstracted. Each variable occupies one line and is preceded by its scope, i.e. the name of the function to which it is local, or the specifier `(global)`. We let this file contain the single line `move d`, indicating that the formal parameter `d` of function `move` is to be abstracted. Instructing abC to perform one abstraction inference step yields a new file containing, besides `move d`, the lines `(global) i`, `move xx`, and `move yy`. A log file explaining the newly inferred abstractions is also produced:

```
line    6: move xx      <--- move d
line    6: move yy      <--- move d
line   16: (global) i   <--- move d
```

The first line says that `xx` has to be abstracted because it depends on `d` in line 6—it is the initializer of `xx` that causes rule 1(a) to apply. The inference given in the third line is because the call to `move` on line 16 causes the argument `&i` to be stored in formal parameter `d`; rule 2(a) applies here.

The next iteration takes the file giving the three currently abstracted variables and produces again a longer list. The log file now contains the following inferences:

```
line    7: move r       <--- move xx, move yy
line    8: move r       <--- (global) i, (global) i
line    8: (global) nametab     <--- (global) i, (global) i
line   14: main p       <--- (global) i
line   14: (global) nametab     <--- (global) i
line   15: (global) nametab     <--- (global) i, (global) i
```

Several variables occur on the right-hand side of an inference in those cases where a single expression (whose storing causes a new abstraction) contains multiple variables, like the (incomplete) initializer on line 7 and the double indexes on lines 8 and 15. Note that `nametab` must be abstracted for three reasons: rule 2(b) applies to the pointer assignment on line 8 and to the initialization in line 14, and rule 1(b) applies to the assignment in line 15.

In the next iteration we get:

```
line    9: (global) move        <--- move r
line   16: move q       <--- main p
line   17: main nullstr         <--- main p
```

Here, the return on line 9 causes the function name `move` to be abstracted. One can think of the return statement as assigning the returned expression, `r`, to the function name, `move`. The ensuing abstraction of `move` will then flow back into the calling context, causing (in this case) also the variable `newp`, to which the return value is assigned, to become abstract. This happens in the next iteration:

```
line   16: main newp    <--- (global) move
```

Another iteration does not produce any inferences, so the current list of abstract variables is closed under dependencies. Some care has to be taken; abC also produces the following warning:

```
*** abC: WARNING (line 18): call to external (or builtin) function;
cannot analyze definition - assuming concrete result returned unless
specified in abstraction list
```

It is easily checked however that the call to printf cannot cause any dependencies.