Automatic Creation of Environment Models via Training

Thomas Ball[1], Vladimir Levin[1], and Fei Xie[2]

[1] Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA
{tball,vladlev}@microsoft.com Fax: +1 (425) 936-7329
[2] Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, USA
feixie@cs.utexas.edu Fax: +1 (512) 471-8885

Abstract. Model checking suffers not only from the state-space explosion problem, but also from the *environment modeling* problem: how can one create an accurate enough model of the environment to enable precise yet efficient model checking? We present a novel approach to the automatic creation of environment models via *training*. The idea of training is to take several programs that use a common API and apply model checking to create abstractions of the API procedures. These abstractions then are reused on subsequent verification runs to model-check different programs (which utilize the same API). This approach has been realized in SLAM, a software model checker for C programs, and applied to the domain of Windows device drivers that utilize the Windows Driver Model API (a set of entry points into the Windows kernel). We show how the boolean abstractions of the kernel routines accessed from a device driver are extracted and merged into a boolean library that can be reused by subsequent model checking runs on new drivers. We show that the merged abstraction is a conservative extension of the boolean abstractions created by training.

1 Introduction

Recently there has been significant progress in model checking [9,21] of software programs. The technique of predicate abstraction [12] has been successfully applied to automatically create boolean program abstractions of software. In the SLAM project [7], software model checking has been applied to the domain of Windows device drivers to check that these programs are good clients of the Windows Driver Model (WDM) application programming interface (API), a direct interface to hundreds of procedures in the Windows kernel. The SLAM engine, packaged with temporal safety properties that define correct usage of the WDM API, comprises a tool called Static Driver Verifier (SDV).

As with all model checking projects, a central question that we have had to address is where a good environment model comes from. In our case, the Windows kernel is the source code we seek to model. Needless to say, this is a non-trivial piece of code, whose complexity and size are much greater than those of the device drivers that use it. We have found that highly demonic and non-deterministic over-simplified models of the driver environment often lead to SDV reporting too many false errors. To date, we have constructed models of the kernel manually and on an "as-needed" basis. These models capture the behaviors of the API that are necessary for the set of properties SDV

K. Jensen and A. Podelski (Eds.): TACAS 2004, LNCS 2988, pp. 93–107, 2004.

checks. This approach eliminates most of the false bugs while enabling model checking of reasonably large programs. However, this approach suffers from the following problems:

- Windows kernel procedures can be very complex. Manual creation of models for these procedures is a time-consuming and error-prone process.
- The Windows API evolves, which makes the maintenance of manually created environment models very difficult.
- The number of properties that SDV checks always is growing, requiring refinement of the kernel models.

Therefore, scalable software model checking requires automating the creation of environment models corresponding to the API as much as possible.

Our solution to these problems is to create environment models via *training*. That is, we leverage predicate abstraction across a large set of example drivers to automatically create models of kernel procedures. While this process can be expensive, it is only done occasionally (by the maintainers of the SDV tool).

Let us describe how training works on a single kernel procedure k (the process generalizes straightforwardly to a set of procedures). A set of drivers $\{d_1, \cdots, d_n\}$ that utilizes k is selected as the training set. SDV is run on each driver linked together with the kernel procedure k. This results in n boolean abstractions of k, which are then extracted and merged together to create a refined abstraction of the procedure, b_k. In future runs of SDV, the boolean abstraction b_k is used in the place of the kernel procedure k. That is, the boolean abstractions of a set of procedures that result from training are packaged as a boolean library which is reused in place of these C procedures in model checking on other drivers.

We have implemented this idea in the context of the SLAM project [7], in which predicate abstraction is applied to model checking of C programs, and methods for creating, extracting, merging and reusing boolean libraries are provided as extensions to the SLAM toolkit. This approach has been applied in the SDV project to automate environment modeling.

While our approach was developed in the context of the SLAM and SDV projects, we believe that the basic idea is generally applicable to programs written in other program languages and to other analysis techniques. Our approach has four basic requirements:

- **Model Creation.** An automatic method for creating abstract models of software, such as predicate abstraction, is required.
- **Model Extraction.** It must be possible to extract models at some well-defined boundary in the program, such as procedures. This means that the model checking process should preserve certain structural elements of the source program (procedures) or provide a way to extract them.
- **Model Merging.** Given a number of models for a procedure, it must be possible to conservatively merge them into a single model.
- **Model Reuse.** Given a model for a procedure, there must be a way to incorporate the model into a subsequent model checking run.

The remainder of this paper is organized as follows. Section 2 presents background on the SLAM abstraction process that creates boolean program abstractions of C programs.

Section 3 introduces the concept of boolean libraries and the algorithm for merging boolean libraries. Section 4 discusses the formal properties of our merge algorithm. Section 5 reports our experiences on applying our technique to Windows device drivers. Section 6 discusses related work. Section 7 presents future work and concludes.

2 Background

This section first gives a brief overview of the SLAM abstraction and refinement process, then introduces the basic concept of boolean programs, and finally reviews how SLAM performs predicate abstraction of C code. Detailed discussions about these topics can be found in [7,5,3,4,6].

2.1 SLAM Abstraction and Refinement Process

Given a safety property to check on a C program, the SLAM process has the following three phases: (1) abstraction, (2) model checking, and (3) refinement (predicate discovery). Three tools have been developed to support each of these phases:

- C2BP, a tool that transforms a C program P into a boolean program $\mathcal{BP}(P, E)$ with respect to a set of predicates E over the state space of P [3];
- BEBOP, a tool for model checking boolean programs [5];
- NEWTON, a tool that discovers additional predicates to refine the boolean program, by analyzing the feasibility of paths in the C program [6].

The SLAM toolkit provides a fully automatic way of checking temporal safety properties of C programs. Violations are reported by the SLAM toolkit as paths over the program P. It never reports spurious error paths. Instead, it detects spurious error paths and uses them to automatically refine the abstraction (to eliminate these paths from consideration).

2.2 Boolean Programs

Given a C program P and a set E of predicates (pure C boolean expressions containing no procedure calls), C2BP automatically creates a boolean program $\mathcal{BP}(P, E)$, which is an abstraction of P. A boolean program is essentially a C program in which the only type available is boolean. The boolean program contains $|E|$ boolean variables, each representing a predicate in E. For example, if the predicate $(x < y)$ is in E, where x and y are integer variables in P, then there is a boolean variable in $\mathcal{BP}(P, E)$ whose truth at a program point p implies that $(x < y)$ is true at p in P. For each statement s of P, C2BP automatically constructs a boolean transfer function that conservatively represents the effect of s on the predicates in E.

The syntax of boolean programs is given in Figure 1. In boolean programs, control-flow constructs such as *if-then-else* conditionals and *while* loops are represented using the non-deterministic *goto* statement and *assume* statements. For example, the statement "if (P) A else B" in a C program can be represented by:

```
goto L1, L2;
L1: assume(P); A; goto L3;
L2: assume(!P); B;
L3: skip;
```

Expressions:	e	::=	$T \mid F \mid x \mid !e \mid e_1\ op\ e_2 \mid choose(e, f)$
Binary operators:	op	::=	$\&\& \mid \|\|$
Declaration:	d	::=	**bool** $x_1, \cdots, x_n$;
Statements:	s	::=	**goto** $L_1, \cdots, L_n$
		$\mid$	L: s
		$\mid$	**assume**(e)
		$\mid$	**assert**(e)
		$\mid$	**return** $x_1, \cdots, x_n$
		$\mid$	$x_1, \cdots x_n := e_1, \cdots, e_n$
		$\mid$	$x_1, \cdots, x_m := f(e_1, \cdots, e_n)$
Statement sequence:	$\overline{s}$	::=	$s_1; \cdots; s_n;$
Procedure:	p	::=	**bool** $id\,(f_1, \cdots, f_n)$ **begin** $d\ \overline{s}$ **end**
Program:	g	::=	$d\ p_1 \cdots p_n$

Fig. 1. Syntax of boolean programs.

in a boolean program. The *assume* statement silently terminates execution if its expression evaluates to false, and is the dual of the *assert* statement [10]. The *assert* statement formulates a safety property. The property is violated if there exists a path in which the asserted expression evaluates to false when the *assert* statement becomes executable.

In boolean programs, variable names can be specified as an arbitrary string within curly braces, which allows us to name the boolean variable corresponding to a predicate p by "{p}". In addition to the usual boolean connectives, boolean expressions in boolean programs have a built-in function called *choose*. The semantics of the *choose(p,n)* is as follows. If p is true, then *choose* returns true. Otherwise, if n is true, then *choose* returns false. If neither p nor n is true, then *choose* non-deterministically returns true or false.

2.3 Predicate Abstraction of C Code

The SLAM abstraction algorithm handles arbitrary C programs. For each C procedure in program P, its abstraction in $\mathcal{BP}(P, E)$ is a boolean procedure. A key feature of this algorithm is modularity: each C procedure can be abstracted by C2BP given only the *signatures* of procedures that it calls. The signature of a procedure can be determined in isolation from the rest of the program given the set of predicates that are local to the procedure. C2BP operates in two passes. In the first pass, it determines the signature of each procedure. In the second pass, it uses these signatures to abstract procedure calls (along with other statements). These aspects of SLAM enable us to create, extract and reuse the boolean program models of procedures (we will deal with merging later).

We explain SLAM's modular abstraction process through an example. Figure 2(a) and (b) show two C procedures. The procedure *inc_dec* returns the increment/decrement of its first argument (x) depending on the value of its second argument (op). The procedure *main* calls *inc_dec* to increment the value of i from 1 to 2. Figure 2(c) and (d) show the boolean procedures constructed by SLAM in order to prove that the *assert* statement always succeeds. SLAM generates five predicates to prove this: in *main*, the

```
int inc_dec (int x, int op) {
 int t;

 if (op == 0)
   t = x+1;
 else
   t = x-1;

 return t;
}
```

(a)

```
main () {
 int i = 1;
 i = inc_dec(i, 0);
 assert(i == 2);
}
```

(b)

```
bool inc_dec ({x==X},{op==0}) begin
 bool {t==X+1};
 goto L1, L2;

 L1: assume({op==0});
 {t==X+1} := choose({x==X},F); goto L3;

 L2: assume(!{op==0});
 {t==X+1} := choose(F,{x==X});

 L3: return {t==X+1};
end
```

(c)

```
main () begin
 bool {i==1}, {i==2}, ret;
 {i==1} := T;
 ret := inc_dec(T,T);
 {i==1},{i==2} := {i==1}&&!ret, {i==1}&&ret;
 assert({i==2});
end
```

(d)

Fig. 2. Example C program, (a) and (b), and boolean program produced by SLAM to prove assertion, (c) and (d).

predicates `i==1` and `i==2`; in *inc_dec*, the predicates `x==X`, `t==X+1`, and `op==0`, where `X` is a *symbolic constant* representing the value of variable `x` at the entry point of *inc_dec*.

Abstraction of assignments and assumes. The abstraction of assignments and *assumes* in this example is simple. Consider the assignment statement `t=x+1` in the procedure *inc_dec*. If the predicate `x==X` is true before the execution of this statement then the predicate `t==X+1` is true after. This is captured in the boolean program by the corresponding assignment statement

```
{t==X+1} := choose({x==X},F);
```

On the other hand, the assignment statement `t=x-1` translates to the boolean program statement

```
{t==X+1} := choose(F,{x==X});
```

because it can only make the predicate `t==X+1` false (when `x==X` is true before).

The *assume* statements in the boolean program reflect the abstraction of the *if* conditional precisely because of the presence of the predicate `op==0`.

Procedure signatures. Let Q be a procedure in P, Q' be the abstraction of Q in $\mathcal{BP}(P, E)$, and E_Q be the set of predicates in E that are local to Q. The signature of procedure Q is a four-tuple (I_Q, F_Q, R_Q, M_Q), where

- I_Q is the list of formal parameters of Q.
- F_Q is the set of formal parameter predicates of Q', defined as the set of those predicates in E_Q that refer to a formal parameter of Q but do not refer to any local variable of Q.
- R_Q is the set of return predicates of Q'. They provide information to callers about the effect of Q. R_Q contains those predicates in E_Q that mention return variables of Q as well as those predicates in F_Q that reference a global variable or dereference a formal parameter of Q.
- M_Q is the modification set of procedure Q, a conservative approximation of the set of locations that Q could potentially modify.

In our example, the signature of the procedure *inc_dec* as Q is

- $I_Q = [x, op]$.
- $F_Q = \{$ `x==X`, `op==0` $\}$. These predicates become the formal parameters of the boolean procedure corresponding to *inc_dec*.
- $R_Q = \{$ `t==X+1` $\}$. This predicate becomes the return value of the boolean procedure.
- $M_Q = \{$ `t` $\}$.

The predicates `x==X` and `t==X+1` deserve special mention as they contain the symbolic constant (an artificial variable) `X`, which represents the initial value of variable `x` upon entry to the procedure *inc_dec*. These predicates are said to be *polymorphic* because they do not refer to a particular value of `x` like 1, 4 or 12. As a result, the predicates are reusable in many different contexts.

Abstraction of procedure calls. Consider a call $y = Q(a_1 \ldots a_n)$ to procedure Q at label ℓ in a procedure R (in program P). The abstraction $\mathcal{BP}(P, E)$ contains a call to the boolean procedure Q' at label ℓ in the boolean procedure R'. Let the signature of Q be (I_Q, F_Q, R_Q, M_Q). For each formal parameter predicate $e \in F_Q$, C2BP computes a boolean expression over (boolean) variables in R' that yields the actual value of the predicate to be passed into the call.

In our example, the formal parameter predicates are `x==X` and `op==0`. The call to the procedure *inc_dec* in main is `inc_dec(i,0);`. Its abstraction in the boolean program is `inc_dec(T,T);` because the predicate `x==X` always is initially true (regardless of the value passed in for x at the call-site) and because the call assigns the value 0 to the formal parameter *op*, which makes the predicate `op==0` true.

The return values from a call to Q' are handled as follows. Assume $R_Q = \{e_1 \ldots e_k\}$. C2BP creates k fresh local boolean variables $T = \{t_1 \ldots t_k\}$ in R' and assigns to them, in parallel, the return values of Q'. The final step is to update each local predicate of R whose value may have changed as a result of the call. Any predicate in E_R (the set of local predicates of R) that mentions variable y must be updated. In addition, we must update any predicate in E_R that could potentially be updated by Q. The modification set M_Q provides sufficient information to determine a conservative over-approximation of the set of predicates to update.

In our example, the return variable ret in the procedure *main* in the boolean program represents the predicate `i==X+1`. If the predicate `i==1` is true before the call then `X`, the initial value of x in *inc_dec*, is equal to 1, so if `i==X+1` is true on return then `i==2` is true on return as well. This is captured by the assignment

```
Signature = (
    I_Q = [ x, op ],
    F_Q = { x==X, op==0 },
    R_Q = { t==X+1 },
    M_Q = { t }
)

bool inc_dec ({x==X},{op==0}) begin
 bool {t==X+1};
 goto L1, L2;

 L1: assume({op==0});
 {t==X+1} := choose({x==X},F); goto L3;

 L2: assume(!{op==0});
 {t==X+1} := choose(F,{x==X});

 L3: return {t==X+1};
end
```

(a)

```
Signature = (
    I_Q = [ x, op ],
    F_Q = { x==X, op==0 },
    R_Q = { t==X-1 },
    M_Q = { t }
)

bool inc_dec ({x==X},{op==0}) begin
 bool {t==X-1};
 goto L1, L2;

 L1: assume({op==0});
 {t==X-1} := choose(F,{x==X}); goto L3;

 L2: assume(!{op==0});
 {t==X-1} := choose({x==X},F);

 L3: return {t==X-1};
end
```

(b)

Fig. 3. Two different boolean libraries of *inc_dec*

```
{i==2} := {i==1}&&ret;
```

in the boolean program (part of the parallel assignment to predicates `i==1` and `i==2`).

3 Boolean Libraries

This section defines the structure of boolean libraries and shows how to merge two boolean libraries (of the same set of C procedures) into one.

3.1 Boolean Libraries

For a set of C procedures, a corresponding boolean library consists of two parts: the header and the body. The header contains a signature of each C procedure in the form specified in Section 2. The body contains the boolean procedures abstracted from these C procedures.

Using the procedure *inc_dec* in Figure 2(b) as an example, two different boolean libraries are shown in Figure 3. The first would be generated by running SLAM on the C program comprised of the procedure *inc_dec* and the procedure *main* in Figure 4(a). The second would be generated by running SLAM on the C program comprised of the procedure *inc_dec* and the procedure *main* in Figure 4(b).

The boolean library shown in Figure 3(a) or (b) can then be used in place of *inc_dec* when model checking a program that utilizes *inc_dec*. The SLAM abstraction process uses the signature of *inc_dec* provided in the boolean library to abstract C procedures that call *inc_dec*. SLAM directly incorporates the abstraction of *inc_dec* provided in the boolean library into the abstraction of the program.

main () {
int i = 1;
i = inc_dec(i, 0);
assert(i == 2);
}
	main () {
	int i = 1;
	i = inc_dec(i, 1);
	assert(i == 0);
	}
		main () {
		int i = 1;
		i = inc_dec(i, 0);
		i = inc_dec(i, 1);
		assert(i == 1);
		}

(a)	(b)	(c)

Fig. 4. Three different usages of the *inc_dec* procedure.

3.2 Merge of Boolean Libraries

For a set of C procedures, different boolean libraries may be constructed by using different sets of predicates in the abstraction process. A boolean library that contains better abstractions of these C procedures can be constructed by merging these boolean libraries. Merging boolean libraries involves construction of a new signature and a new boolean abstraction for each C procedure by merging the signatures and boolean abstractions of the procedure from these boolean libraries.

In this section, we discuss the algorithms for merging two signatures, *sig'* and *sig"*, and their corresponding boolean abstractions, *Q'* and *Q"*, of a single C procedure, Q. The algorithms can be readily generalized to the merge of an arbitrary number of boolean libraries of a C procedure or an arbitrary number of C procedures.

We will demonstrate the merge algorithms by merging the signatures and the boolean abstractions of *inc_dec* in Figure 3(a) Figure 3(b). Figure 5 shows the result of the merge.

Merge of signatures. Given the two signatures of Q, $sig' = ({I_Q}', {F_Q}', {R_Q}', {M_Q}')$ and $sig'' = ({I_Q}'', {F_Q}'', {R_Q}'', {M_Q}'')$, a new signature, $sig''' = ({I_Q}''', {F_Q}''', {R_Q}''', {M_Q}''')$, can be constructed by merging sig' and sig'' as follows:

- ${I_Q}''' = {I_Q}' = {I_Q}''$;
- ${F_Q}''' = {F_Q}' \cup {F_Q}''$;
- ${R_Q}''' = {R_Q}' \cup {R_Q}''$;
- ${M_Q}''' = {M_Q}' = {M_Q}''$.

The merge of the two signatures in Figure 3(a) and Figure 3(b) is shown in Figure 5. The basic change is that the new signature for the procedure *inc_dec* now has two return predicates (Figure 3(a) contributes the predicate `t==X+1` and Figure 3(b) contributes the predicate `t==X-1`).

Merge of boolean procedures. A boolean procedure has a control flow structure formed by *goto* statements, *assume* statements, assignments, and procedure calls. Two boolean procedures, Q' and Q'', abstracted from Q are guaranteed to have the same control flow structure. What differentiates Q' and Q'' are the conditionals in the *assume* statements, and the variables and the expressions in the assignments and procedure calls.

Therefore, the control flow structure of the new boolean abstraction Q''' from merging Q' and Q'' is constructed by copying the control structure of either Q' or Q''. The main challenge is how to merge the expressions and variables of the corresponding *assume*, *assert*, assignment and procedure call statements in Q' and Q''.

```
Signature = (
    I_Q = [ x, op ],
    F_Q = { x==X, op==0 },
    R_Q = { t==X+1, t==X-1 },
    M_Q = { t }
)

bool inc_dec ({x==X},{op==0}) begin
 bool {t==X+1}, {t==X-1};
 goto L1, L2;

 L1: assume({op==0});
 {t==X+1},{t==X-1} := choose({x==X},F), choose(F,{x==X}); goto L3;

 L2: assume(!{op==0});
 {t==X-1},{t==X+1} := choose({x==X},F), choose(F,{x==X});

 L3: return {t==X-1}, {t==X+1};
end
```

Fig. 5. The merged boolean library of *inc_dec*.

Merge of assignment statements. Consider an assignment statement `x=e;` at label ℓ in Q. Q' and Q'' will contain at label ℓ a parallel assignment to the boolean variables in scope at ℓ. The parallel assignment is of the following form:

$$b_1, \ldots, b_n := choose(pos_1, neg_1), \ldots, choose(pos_n, neg_n)$$

Suppose the following two parallel assignments appear at label ℓ in Q' and Q'':

$$\{p_1\}', \ldots, \{p_m\}' := choose(pos_1', neg_1'), \ldots, choose(pos_m', neg_m')$$
$$\{q_1\}'', \ldots, \{q_n\}'' := choose(pos_1'', neg_1''), \ldots, choose(pos_n'', neg_n'')$$

If for $\{p_i\}'$ there is no q_j such that $p_i = q_j$ then the assignment to $\{p_i\}$ is simply copied over from Q' to Q'''. (Symmetrically, if for $\{q_i\}''$ there is no p_j such that $q_i = p_j$ then the assignment to $\{q_i\}$ is simply copied over from Q'' to Q'''). On the other hand, if there is a q_j such that $p_i = q_j$ then $\{p_i\}'''$ is assigned in Q''' as follows:

$$\{p_i\}''' := choose(pos_i' || pos_j'', neg_i' || neg_j'')$$

since if either pos_i' or pos_j'' holds, $\{p_i\}$ is **true** after ℓ and if either neg_i' or neg_j'' holds, $\{p_i\}$ is **false** after ℓ. (The SLAM abstraction algorithm guarantees that pos_i' and pos_j'' do not conflict and that neg_i' and neg_j'' do not conflict.)

The boolean procedure merged from the two boolean procedures in Figure 3 is shown in Figure 5 and demonstrates how the corresponding statements are merged. The basic change is that the merged boolean procedure now updates both the predicates `t==X+1` and `t==X-1`, based on how the predicates were updated in each input boolean procedure.

Merge of assume and assert statements. The *assume* statement is the control-flow gate statement of boolean programs. If the expression in the *assume* evaluates to true then execution proceeds past the statement. However, if the expression evaluates to false then execution (silently) halts. C2BP abstracts the *assume* statement in a sound fashion: if an *assume* in the boolean program evaluates to false then the corresponding conditional in the C program will evaluate to false.

Corresponding *assume* statements in Q' and Q'' with expressions e' and e'' are merged into the *assume* statement *assume(*e'*&&*e''*)* in Q'''. That is, control in Q''' can only proceed past the *assume* if control in both Q' and Q'' can proceed past the *assume*. Conjunction of e' and e'' provides the most precise boolean abstraction of the C program possible from the merge (disjunction is sound but not as precise as conjunction).

The *assert* statement is the dual of the *assume*. The C2BP tool guarantees that if an *assert* passes in the boolean program then it passes in the C program. Thus, the merge of corresponding *assert* statements in Q' and Q'' with expressions e' and e'' is *assert(*$e'||e''$*)* in Q'''.

Merge of procedural calls. Procedure calls in a boolean procedure are of the form:

$$t_1, \ldots, t_p := foo(e_1, \ldots, e_n);$$
$$\{\text{Assignment of variables in calling context according to } t_1, \ldots, t_p\}.$$

$e_1, \ldots, e_n$ are the actual expressions that are assigned (implicitly) to the boolean program formals (F_Q) in the signature of *foo* and $t_1, \ldots, t_p$ are temporaries corresponding to return predicates (R_Q) in the signature of *foo*.

Suppose in Q, another C procedure, R, is called. In Q' and Q'', the call to R is abstracted as shown in Figure 6. R' and R'' are two boolean procedures abstracted from

$t_1', \ldots, t_p' := R'(e_1', \ldots, e_m');$ {Assignment of variables in Q'}	$t_1'', \ldots, t_q'' := R''(e_1'', \ldots, e_n'');$ {Assignment of variables in Q''}

Fig. 6. Abstractions of the precedure call to R in Q' and Q''

R and may be different. Suppose the merge of R' and R'' is R'''. The merged procedure call in Q''' is as follows:

$$t_1''', \ldots, t_y''' := R'''(e_1''', \ldots, e_x''');$$
$$\{\text{Assignment of variables in } Q''' \text{ according to } t_1''', \ldots, t_y'''\}.$$

The expressions, $e_1''', \ldots, e_x'''$, and the temporaries, $t_1''', \ldots, t_y'''$, are created corresponding to the formal predicates and return predicates in the signature of R'''. Using the merge algorithm for assignments presented above, the expressions $e_1''', \ldots, e_x'''$ in the (implicit) assignment to the formals F_R''' are derived from the assignment of $e_1', \ldots, e_m'$ to the formals F_R' and the assignment of $e_1'', \ldots, e_n''$ to the formals F_R''. The assignment of variables in Q''' is derived from the assignment of variables in Q' and the assignment of variables in Q''.

3.3 An Open Issue: Function Pointers

The method presented above does not support construction of boolean libraries for API procedures that accept function pointers as formal parameters. Function pointers are commonly used by procedures in an API to invoke procedures defined in the client programs that utilize the API. There are several problems that function pointers introduce. First, the behavior of the API is parameterized by the client code. This makes it difficult to create a boolean library abstraction that is reusable in different contexts. Second, the modification information for the API procedure now depends on what the client code modifies (through the call to the function pointer). Thus, to support function pointers in boolean program requires a great deal of parameterization for which we do not yet have good technical solutions.

4 Properties of Merge Algorithms: Discussion and Theorems

Our approach to merging boolean libraries is sound, in other words, the boolean procedure, Q''', constructed by merging Q' and Q'' is a conservative abstraction of Q. Our approach is highly modularized and the merge of two corresponding statements from Q' and Q'' does not involve the analysis of other statements in the two procedures. Therefore, the soundness of our approach is based on the soundness of the merge algorithms for each statement. From the discussions above, it is easy to observe that for each statement in Q, our algorithms construct a conservative abstraction of the statement.

The merged abstraction Q''' also is more refined than Q' and Q'' since it combines information from the statements in both Q' and Q''. The boolean abstraction Q' of the procedure Q has only the predicates generated for the contexts in which Q was model checked. For example, the boolean library in Figure 3(a), precisely abstracts the *then* branch through the procedure, *inc_dec*, however, it abstracts the *else* branch imprecisely, because of the lack of the predicate `t==X-1`. If this boolean library is used in place of *inc_dec* in model checking the *main* procedure in Figure 4(b) then SLAM fails to verify the correctness of the assertion and reports a "give up case".

The above observations are supported by two formal properties of the merge algorithms. The first property, *soundness*, states that the boolean procedure, Q''', constructed by merging two boolean procedures, Q' and Q'', each being an abstraction of a C procedure, Q, is also an abstraction of Q, which means that every (feasible) execution path in Q is a (feasible) execution path in Q'''. The second property, *precision*, states that the set of execution paths allowed by Q''' is not larger than the set of execution paths shared by Q' and Q''.

Suppose E', E'', and E''' are the corresponding set of predicates for Q', Q'', and Q''', respectively. Therefore, $E''' = E' \cup E''$. Suppose e''' is a predicate where $e''' \in E'''$ and b''' is the corresponding boolean variable of e''' in Q'''. Suppose p is a feasible path in Q and Ω is the state of Q after executing p.

Theorem 1. *For any feasible p in Q, it is guaranteed that p is a feasible path in Q''' as well. Furthermore, there exists an execution of p in Q''' ending in a state Γ such that for every e''' in E''', e''' holds in Ω iff b''' is true in Γ.*

The proof of Theorem 1 is similar to the soundness proof [4] of the SLAM abstraction algorithm, where we first prove that one execution step in Q has a corresponding sequence of execution steps in its abstraction constructed by SLAM and then we prove

the correctness of the algorithm using induction over execution steps. In the proof of Theorem 1, we use the same induction. The only difference is that when we prove that one execution step in Q has a corresponding sequence of execution steps in Q''', we, in addition, utilize the fact that each assignment, *assume*, *assert* or procedure call statement in Q''' is the merge of the corresponding statements in Q' and Q''.

Theorem 2. *Q''' is, at least, as precise as Q' and Q'': any execution path allowed in Q''' is also allowed by both Q' and Q''.*

From the merge algorithms, it is easy to observe that Q''' only allows a path that is feasible in both Q' and Q''.

5 Experiences with Windows Device Drivers

Windows device drivers are tightly coupled with the Windows OS. They interact with it through the Windows Driver Model (WDM) API, composed of about 1000 procedures and macros. We use Static Driver Verifier (SDV) to check the source code of a Windows device driver for possible violations of safety properties. These properties formally express WDM safety rules that describe what it means for a device driver to be a good client of the Windows kernel. The verification environment into which SDV places a device driver for model checking has two types of models: (i) scenario models (*harnesses*) that produce sequences of requests reaching a device driver through Windows OS and (ii) operation models (*stubs*) of OS procedures through which the device driver utilizes OS resources to perform requested actions with the device it controls. Presently, models of the both types are written manually in C: the scenario models have a total of 800 lines and the operation models a total of 3000 lines.

The boolean library method we propose in this paper can be used to automatically construct (train) all models of the second type, i.e. operation models. We have implemented extensions to the SLAM toolkit to realize the method. The `-genlib` Q option instructs SLAM to generate a boolean library for the C procedure Q after performing a verification run. The tool `bpmerge` takes two boolean program libraries (of the same set of procedures) and merges them as described previously. The `-uselib` L option instructs SLAM to perform model checking using the boolean library L in place of the corresponding OS functions whose abstractions L contains.

Using these tools, we experimented with training operation models for two WDM functions, namely, *IoAttachDeviceToDeviceStack* and *IoSetDeviceInterfaceState* [19]. The *IoAttachDeviceToDeviceStack* function whose declaration is shown in Figure 7 has 102 lines of code. It attaches the *SourceDevice* object on the top of the chain of device objects already attached over the *TargetDevice* object and returns a pointer to the device object to which the *SourceDevice* object was attached. Note that the returned device object differs from the *TargetDevice* object if there indeed exist additional drivers layered on top of the driver that controls the *TargetDevice* object. This function returns NULL if it could not perform the above action because, for example, the target device driver has been unloaded. The *IoSetDeviceInterfaceState* function whose declaration is shown in Figure 8 has 26 lines of code. It enables or disables an instance of previously registered device interface class. The *SymbolicLinkName* parameter is a pointer to a string identifying the device interface to be enabled or disabled. The *Enable* parameter

```
PDEVICE_OBJECT IoAttachDeviceToDeviceStack(IN PDEVICE_OBJECT SourceDevice,
                                           IN PDEVICE_OBJECT TargetDevice)
```

Fig. 7. Sample WDM procedure: IoAttachDeviceToDeviceStack

```
NTSTATUS IoSetDeviceInterfaceState(IN PUNICODE_STRING SymbolicLinkName,
                                   IN BOOLEAN Enable)
```

Fig. 8. Sample WDM procedure: IoSetDeviceInterfaceState

indicates whether the device interface is to be enabled or disabled. Depending on the value of *Enable*, the function performs different operations and returns different values.

In our experiments with these two functions, we used two real device drivers, *fdc* (9209 lines of code) and *flpydisk* (6601 lines of code), as a training set. For *IoAttachDeviceToDeviceStack*, we used an existing SDV rule from the SDV package that checks that the pointer returned by this function indeed points to the correct device object in the chain of objects. For *IoSetDeviceInterfaceState*, we constructed a new rule to check that the parameters and the return value of the function are correctly correlated.

Our experiments with the two functions described above demonstrated that boolean models of OS functions can indeed be built, trained and used with real device drivers. These two functions and the corresponding rules are rather simple. The complexity of their boolean models we automatically built appeared similar to the complexity of the original C code and therefore we did not observe radical improvement in performance when we verified drivers with boolean models instead of the original functions. We, however, observed performance improvement (about 30%) on slightly more complicated versions of these two functions we artificially constructed by introducing additional branches in their implementation bodies. A similar improvement can be observed even on the toy example we used in Sections 2 and 3. Namely, the verification of the program in Figure 4(a) takes 6 iterations when we use the C code of procedure inc_dec given in Figure 2(a) and 4 iterations when we use the merged boolean library of this procedure (in Figure 5). These experiments support our expectation that on a larger library of OS functions, our method will benefit not only from automation, but also from performance improvement. Further experiments with our method are in progress.

6 Related Work

Environment modeling is an indispensable part of software model checking. Many representations have been used to specify environment models. The most commonly used are program languages such as C or Java (with extensions such as non-determinism), design level specification languages such as Executable UML [18], and input languages to model checkers such as Promela [14], SMV [16], and S/R [13]. Temporal logics have also been used to specify environment models, mainly in the context of compositional reasoning [20,2,1,17]. The representations we propose for environment models are boolean libraries, which are especially suitable for environment modeling in model checking of software implementations through predicate abstraction.

Scalable application of model checking requires automation of environment modeling since manual environment modeling are time-consuming and error-prone. The

importance of automating environment modeling has also been discussed in several other software model checking projects, such as Feaver [15] and Java PathFinder [8, 22]. However, there has been very limited effort on this problem. Our approach based on boolean libraries enables partial automation of environment modeling which intrinsically requires some user interactions. Our goal is to automate environment modeling as much as possible.

This paper explored the training approach for construction of boolean libraries, which, in essence, selects predicates for abstracting C procedures by selecting a typical set of programs that utilize these C procedures. There are other methods for selecting and generating predicates, such as static and dynamic analysis. For example, Ernst et al. [11] use dynamic analysis to discover "likely" program invariants that could be used support predicate selection for the purpose of constructing boolean libraries.

7 Conclusions and Future Work

Our approach has two major contributions: (1) a new representation for environment models, boolean libraries, and (2) a method to automation of environment modeling based on boolean libraries. Although the concept of boolean libraries is proposed in the context of model checking C programs through predicate abstraction, it has broader applications in other approaches to model checking such as compositional reasoning.

Our approach currently does not handle function pointers in the C procedures from which boolean libraries are to be generated. Use of function pointers is common in C libraries. Extensions that handle function pointers even in some limited fashion could broaden the application of our approach significantly. Another important future research direction is how to combine different predicate selection techniques, such as training and static analysis, to facilitate quick construction of boolean libraries that are precise enough for checking certain properties and also of minimal complexity.

Acknowledgements. We gratefully acknowledge Sriram K. Rajamani, Byron Cook, and all other members of the SLAM/SDV team for their contribution, support and feedback. We also sincerely thank James C. Browne for his help.

References

1. M. Abadi and Leslie Lamport. Conjoining specifications. *TOPLAS: Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.
2. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
3. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.
4. T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, 2001.
5. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

6. T. Ball and S. K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.
7. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
8. G. Brat, K. Havelund, S. Park, and W Visser. Java PathFinder - a second generation of a Java model checker. In *Workshop on Advances in Verification*, 2000.
9. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
10. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
11. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering*, 27(2):1–25, February 2001.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
13. R. H. Hardin, Z. Harel, and R. P. Kurshan. COSPAN. In *CAV 96: Computer-Aided Verification*, LNCS 1102, pages 421–427, 1996.
14. G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
15. G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.
16. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
17. K. L. McMillan. A methodology for hardware verification using compositional model checking. *Cadence TR*, 1999.
18. S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, 2002.
19. W. Oney. *Programming the Microsoft Windows Driver Model*. Microsoft Press, 2003.
20. A. Pnueli. In transition from global to modular reasoning about programs. In *Logics and Models of Concurrent Systems*. NATO ASI Series, 1985.
21. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1982.
22. W. Visser, K. Havelund, G. Brat, and S. J. Park. Model checking programs. In *ASE 00: Automated Software Engineering*, pages 3–12. IEEE, 2000.