Predicate Abstraction for Murphi

Xiaofang Chen and Ganesh Gopalakrishnan

UUCP-06-002

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

Abstract

Predicate abstraction is a technique used to prove properties in a finite or infinite state system. It employs decision procedures to abstract a concrete state system into a finite state abstraction system, which will then be model checked and refined. In this paper, we present an approach for implementing predicate abstraction for Murphi[1] using CVC Lite[2]. Two cases for each property(*i.e.* SAT and UnSAT), are tried in model checking. When a fixed point is reached finally, the validity of each property is declared. We applied our tool(called `PAM`) on the FLASH[3] and German[4] protocols. The preliminary result on these protocols is encouraging.

1 Introduction

To verify interesting properties in a concurrent system, traditional approaches based on simulation and testing are often not adequate. This is because many concurrent systems, such as cache coherence protocols, are characterized by very large state spaces so that simulation and testing cannot achieve a reasonable coverage. To overcome this limitation, *model checking*[5] is widely used in verification, as it provides full state space coverage.

However, model checking also has the well known problem of state explosion, which makes it unable to verify large scale systems. The idea of *predicate abstraction* was first described by Graf and Saïdi[6]. It is a technique trying to abstract large-scale or infinite-state systems into tractable finite-state systems. The states in the abstract systems correspond to the truth values of a set of predicates in the original concrete systems. The abstraction is $conservative$, meaning that if a property is shown to hold on the abstract system, there is a concrete version of the property that holds on the original system. However, if the property fails to hold on the abstract system, it may or may not hold on the concrete system.

In this paper, we present an implementation of predicate abstraction for Murphi using `CVC Lite`(CVCL). We use it both as a symbolic simulation library and a decision procedure. The result is encouraging as it is able to process cache coherence protocols such as German and FLASH.

2 Related Work

Over-approximation[7] and under-approximation[8] are two types of abstractions used in predicate abstraction. Over-approximation based abstraction is conservative. It usually introduces more concrete states than those in the real system when abstraction is concretized. It also requires that an abstract transition relation be derived from the concrete system. On the other hand, under-approximation based abstraction is often more precise: any property that holds on a concrete system also holds on the abstract system. Over-approximation based abstraction usually generates abstract states directly from the concrete states, without knowing an abstract transition relation.

Because of the over-approximation introduced in abstraction and concretization, refinement is often required to improve the precision of the abstract transition relation in predicate abstraction. Counter-example guided abstraction refinement [9, 6, 10, 11] is one approach to extract information from false negatives ("spurious counterexamples"). Another approach, such as overlapping projections[12] and lazy abstraction[13], uses both forward and backward transition relations to do refinement. Recently, unsat core extraction[14, 15] has been tried in SAT solvers [] as another approach of refinement.

There are other approaches to generate abstract state graphs. For example, [13] abstracts C programs into boolean programs, in which the control flow of the boolean program is the same with the C program, and each boolean variable corresponds a C program predicate. However, this approach has the disadvantage of losing information permanently after abstraction, which makes it unable to do enough refinement in the abstracted program.

Our work is derived from Das and Dill[7], which uses over-approximation based abstraction. However the original implementation of [7] was not publicly available and several technical details cannot be inferred from the paper. As an initial trial, we implemented the predicate abstraction for Murphi (called PAM) using CVCL. It first converts a protocol described in the Murphi modeling language into a symbolic mode, and then use CVCL as a decision procedure library to infer the truth values of predicates.

3 Predicate Abstraction

In this section, we first describe the features of the Murphi model checker, then illustrate the theoretic basis of PAM, and finally present an example to show how the algorithm works.

3.1 The Murphi Model Checker

The Murphi[1] model checker was designed for verification of asynchronous high-level systems. It consists of two components: the description language and the compiler. The description language can be used to model an asynchronous system to be verified, and the compiler compiles the model into a special purpose verifier. This verifier uses an explicit state enumeration algorithm to check the properties of the system, such as error assertions, invariants and deadlocks. If the system fails to observe these properties, a counter-example is generated and reported by the verifier.

The Murphi modeling language describes the transitions of a system using a set of *guarded commands* (also known as $rules$). Each guarded command consists of a boolean expression (called *guard*) and a collection of statements (called *action*). A rule is said to be enabled if the guard expression evaluates to *true*. Given the current state of the system, one of the enabled rules is chosen nondeterministically and the corresponding action is executed to compute the next state of the system. The initial state(s) of a system is described using a rule without guard.

3.2 Abstraction and Concretization

In this paper, we regard all the states in a system before abstraction as *concrete states*, and the truth values of a set of predicates in the concrete system as *abstract states*. A concrete state can only have one corresponding abstract state, while an abstract state may represent a set of concrete states. For example, suppose a concrete system is described using variables $\{pc,\ y_1,\ y_2,\ z\}$, and there are two predicates $\{\phi_1,\ \phi_2\}$: $\phi_1 \equiv (pc = 1) \wedge (y_1 = 1);\ \phi_2 \equiv \neg(pc = 1) \vee (y2 = 2)$. If a concrete state has the value of $\{pc = 1,\ y_1 = 1,\ y_2 = 0,\ z = false\}$, then the corresponding abstract state is $(\phi_1, \phi_2) = (true, false)$.

We now formally define the terms used in the rest of this paper. We denote all the concrete states in a system as C, the concrete transition relations in C as R_C, and the set of predicates as $\{\phi_1, \phi_2, \ldots, \phi_n\}$. Thus, a concrete state y is a successor of a concrete state x iff $R_C(x, y) = true$. On the other hand, we denote all the abstract states as A, and the abstraction function as α, which maps a concrete state to an abstract state. The concretization function γ is defined as the inverse of α. It maps an abstract state to a set of concrete states. The above formal definitions are shown as following:

$$\begin{array}{ll} R_C : & C \times C \rightarrow Bool \\ \alpha : & C \rightarrow A \\ & \alpha(x) = (\phi_1(x), \ldots, \phi_n(x)) \\ \gamma : & A \rightarrow 2^C \\ & \gamma(s) = \{x \mid s = (s_1, ..., s_n),\ \phi_i(x) = s_i,\ \forall i \in [1, n]\} \end{array}$$

3.3 Predicate Abstraction Algorithm

3.3.1 Initial Abstract States

To generate all the reachable states in the abstract system, we first compute the initial abstract states. This can be done by first computing the initial concrete state s_0, and then set predicate i to be $false$ if $(s_0 \wedge \phi_i)$ is UnSAT, and to $true$ otherwise. The satisfiability check is done by CVCL, and it is equivalent to compute $\phi_i(s_0)$.

When `ruleset`[16] is used in the rule construction, it can be thought of as syntactic sugar for creating a copy of its component rules for every value of its quantifier. In this case, the satisfiability check over the combination of each predicate, including both ϕ_i and $\neg\phi_i$ is performed. This is similar with the `H()` function in Section 3.3.2. The result will be stored in a BDD structure, and each satisfiable assignment in this BDD will be taken out separately to generate its own reachable abstract state space.

3.3.2 Generating Next Abstract States

Suppose x is a set of concrete states, y is a set of next concrete states corresponding to x, and S is a set of abstract states. To generate a set of abstract successor states of S in at most one step, $i.e.$ generating successor states or keeping current states, the function "`H`()" is used defined as following. It is a recursive function with the initial input of value $\{\psi = \gamma(S)(x) \wedge R_c(x, y),\ i = 1, bdd = bddfalse\}$. Here S is an initial abstract state, and $\gamma(S)(x)$ represents a set of concrete states concretized from S. Concretization is done by replacing the satisfiability value of each predicate in abstract states with the corresponding predicate formula, i.e. $true$ with $\phi_i(x)$ and $false$ with $\neg\phi_i(x)$. bdd is a BDD structure. It is a pointer in `H`(), with the initial value being $true$.

$$H(\psi, i, bdd) = \begin{cases} false & if\ \psi(x,y)\ UnSAT \\ true & if\ i > n \\ \quad H(\psi(x,y) \wedge \phi\_i(y),\ i+1,\ bdd\ \&\ bddvar(i)) & \\ \vee\ H(\psi(x,y) \wedge \neg\phi\_i(y),\ i+1,\ bdd\ \&\ !bddvar(i)) & otherwise \end{cases}$$

When `H`() returns $true$, bdd will contain the set of successor reachable abstract states. These states will be used to check if a fixed point has been reached. The computation of `H`() will continue with a new set of (ψ, i, bdd) if no fixed point has been reached, and exit otherwise. Following is an example illustrating how the predicate abstraction algorithm works.

```
---------------------------------------------------------------------
Concrete System
    ruleset i: 0..3         rule "1"
     p := 0;                 p := p + 1;              phi1 = (p=0)
     q := i;                 q := q + 1;              phi2 = (q=1)

    (init state)            (transition rule)        (predicates)
---------------------------------------------------------------------
```

```
Abstract System
    {(1,0)} --> {(0,0), (0,1)} --> {(0,0), (0,1)}
    {(1,1)} --> {(0,0)}        --> {(0,0), (0,1)} --> {(0,0), (0,1)}

  (init states)     (successor states)     (2 fixed points reached)
----------------------------------------------------------------
```

4 Implementation

We have implemented the predicate abstraction algorithm described in Section 3.3 with the Murphi modeling language called `PAM`. Given a model with user provided predicates, `PAM` first converts data types described in Murphi into CVCL data types, and then creates a record type REC which contains all the global variables in the model. Two CVCL expressions of the type REC are declared: $Expr\ X, Y$, which represent the current and next concrete states in the system.

The Murphi data types that `PAM` currently supports include the following. Complex data types can be declared by nesting "record" or "array".

$$boolean \quad array \quad record \quad enum \quad subrange$$

In the following sections, we will illustrate how `PAM` deals with "rule", "startstate", "statement" and "expression". Because the body of a "predicate" is just an "expression", it will be covered in Section 4.3.

4.1 Rules

There are three types of rules in the Murphi modeling language: "startstate", "rule" and "invariant". As the names indicate, "startstate" specifies an initialization of concrete states, "rule" defines a concrete transition relation, and "invariant" specifies a user-provided predicate.

To support the nondeterministic syntax of multiple startstates, `PAM` selects one startstate each time, computes its initial abstract state and then generates all possible successor states until it finally reaches a fixed point. For each startstate rule, `PAM` implements a method "`void get_startstate_expr(Expr *start_expr)`", in which $start\_expr$ is an output CVCL expression. This method symbolically executes each statement in the start-state rule sequentially. The satisfiability check described in Section 3.3.1 will be performed on $start\_expr$ when computing the initial abstract states.

4.1.1 The Concrete Transition Relation

The concrete transition relation $Rc()$ is computed by logically disjuncting all the "rule" transition rules defined in a model. In the Murphi modeling language, a "rule" is represented by a "guard-action" pair, in which $guard$ is a boolean expression working as the prerequisite to execute the action, and $action$ is a collection of statements to compute the next concrete state. When multiple rules are enabled (with their guards being satisfied), one of them is selected to execute nondeterministically.

`PAM` computes $Rc()$ by having each transition rule implement a method "`void get_rule_expr (Expr &rc)`", in which rc_expr is an output CVCL expression. This method first obtains the guard expression by calling the method "`generate_expr()`" defined in Section 4.3. It then symbolically executes each statement in the action body sequentially. The logical conjunction of the guard and action expressions is the output of rc_expr.

As an example, considering a rule with the guard being $(p = 0)$ and the action being $\{p := p + 1;\ q := p;\}$, the method "`get_rule_expr()`" will work as following:

```
(1)  void get_rule_expr(Expr &rc) {
(2)     Expr guard = vc->eqExpr(vc->recSelctExpr(X, "p"), vc->ratExpr(0);
(3)     Expr Z = X;
(4)     Z = vc->recUpdateExpr(Z, "p",
(5)                  vc->plusExpr(vc->recSelectExpr(Z, "p"), vc->ratExpr(1)));
(6)     Z = vc->recUpdateExpr(Z, "q", vc->recSelectExpr(Z, "p"));
(7)     rc = vc->andExpr(guard, vc->eqExpr(Y,Z));
(8)  }
```

Finally, $Rc()$ is the logical disjunction of all the rc expressions corresponding to all the "rules" defined in a model, representing a one-step transition relation. Any of the enabled rules can be fired and the corresponding action will be executed to compute the next state. When none of the guards is enabled, the next state Y will have the same value as the current state X. The following formula illustrates Rc (assuming there are t transition rules)

$$
\begin{aligned}
Rc() = \quad & ((\text{rule\_no=1}) \ \wedge \ \text{rc\_1} \\
\vee \ & \ldots \\
\vee \ & ((\text{rule\_no=t}) \ \wedge \ \text{rc\_t} \\
\vee \ & ((\text{rule\_no=0}) \ \wedge \ (\text{Y=X}) \ \wedge \ (\neg\text{guard\_1} \wedge \ldots \wedge \neg\text{guard\_t}))
\end{aligned}
$$

4.2 Statements

`PAM` currently supports five types of statements in the Murphi modeling language, including *assignment, ifstmt, forstmt, proccall* and *returnstmt*. It implements the method "`generate_action`()" shown as following for each type of statement.

```
virtual void generate_action( char *state,
                              map<char *, char *>& locals,
                              map<char *, char *>& proc_params,
                              char *cond,
                              vector<ReturnStmtClass *>& retn_stmts)
```

In this method, $state$ is a string representing whether the method is processing the action of the current state "X", the next state "Y", or a temporary state "Z". Note this method is part of the `PAM` compiler, which will generate a C++ file for a model described in the Murphi modeling language.

$locals$ in this method is a map from the names of local variables (including `ruleset`) to names of unique CVCL expressions; $proc\_params$ is a map from procedure/function parameters to names of vector variables which specify how to do the get and update operations on each parameter, as the Murphi modeling language supports the "call-by-reference" syntax. $cond$ is a CVCL expression which stores the path constraints up to the current statement, i.e. the condition expressions in nested "if-then-else" statements. Finally, $retn\_stmts$ will record all the return statements in a $rule,\ startstate,\ funcdecl$ or $procdecl$. Each `ReturnStmtClass` object in the vector contains the path constraint of a returnstmt, its call type (from a rule/startstate or funcdecl/procdecl), the output parameters values, and the returned value. Because `PAM` does symbolic execution on Murphi models, when there are multiple returnstmts in a code block, it has to execute to the end of the block, gather all the returnstmts information and return the combined value.

4.2.1 assignment

Assignment is defined as "*target := src*". It is the most commonly used statements in the Murphi modeling language. Based on the class of $target$ and whether its top level name is in the $locals$ or $proc\_params$ mapping, `PAM` will perform the update operation recursively on the corresponding local variable, procedure/function parameter, or a global variable.

4.2.2 proccall

The syntaxes of procedure calls and function calls are similar. In the Murphi modeling language, procedure calls are regarded as statements, while function calls are regarded as expressions in Section 4.3. Because `call-by-reference` is allowed in procedure calls, `PAM` converts each parameter in a procedure call into "`vector<ProcParamClass *> vec`" or a pair "`vector< ProcParamClass *> vec, Expr &ex`" depending on whether the parameter is updatable or not ($i.e.$ declared as `var` in the model). Here, vec specifies how to read and update the value of the parameter, and ex does the real operation for that CVCL expression. As an example, if the parameter is "$var\ a$" and the value to be passed is "$X.p.f[e]$", When a is updated in the procedure, X should be updated by recursively updating its field "p", subfield "f" and array element "e".

Procedure calls correspond to procedure declarations. In `PAM`, procedure declarations are implemented as functions to support conditional mutual calls (e.g. $A()$ calls $B()$) and recursive calls. Because recursions depending on symbolic values may not necessarily terminate, `PAM` currently only supports recursions based on concrete values, and the same strategy is used for the $forstmt$s.

4.3 Expressions

`PAM` currently supports 13 types of expressions in the Murphi modeling language shown as following. Complex expressions can be constructed by nesting, $i.e.$ `binaryexpr` and `designator`.

$unaryexpr \quad binaryexpr \quad boolexpr \quad notexpr \quad equalexpr \quad compexpr$
$arithexpr \quad unexpr \quad mulexpr \quad quantexpr \quad condexpr \quad designator \quad funccall$

For each type of expression, `PAM` converts it to a CVCL expression by implementing the following method.

```
virtual char* generate_expr(   map<char *, char *>& locals,
                               map<char *, char *>& proc_params,
                               char *state);
```

Here $locals$, $proc\_params$ and $state$ have the same functionality as in Section 4.2. Take the expression type `designator` as an example, this method needs to consider 18 cases of situations, depending on whether there are path constraints over the expression, whether the top level name is a local variable, a procedure/function parameter or a global variable, and whether its type is `Base`, `FieldRef`, or `ArrayRef` as defined in the Murphi modeling language.

4.4 Obtaining the Next Abstract State

`PAM` uses a BDD object to store the abstract states and uses the method $H()$ described in Section 3.3 to generate the successor abstract states. The following pseudo code illustrates the basic control flow to reach a fixed point in the abstract state space. Note that *vc->assertFormula()* is to add a formula as a fact expression into the CVCL database. Following queries of satisfiability will be affected by this database.

```
main() {
    expr_rc = Rc()
    vc->assertFormula(expr_rc)
    for (each startstate)
        generate_abstract_state(bdd)
        do
            bdd_old = bdd
            for (each SAT asgn in bdd)
                ψ = concretization(asgn)
                ret = H(ψ, 1, bdd_tmp)
                if (ret)
                    bdd = bdd | bdd_tmp
        while (bdd_old != bdd)
}
```

```
bool H(ψ, i, bdd) {
    unsat1 = true, unsat2 = true
    for (j = 0; j < rule_num; j ++)
        if (! unsat(ψ ∧ preds[i] ∧ rules[j]))
            unsat1 = false
        if (! unsat(ψ ∧ ¬preds[i] ∧ rules[j]))
            unsat2 = false
    if (unsat1)
        return H(ψ ∧ ¬preds[i], i+1, bdd & !bddvar(i))
    if (unsat2)
        return H(ψ ∧ preds[i], i+1, bdd & bddvar(i))
    return H((ψ ∧ preds[i], i+1, bdd & bddvar(i))
            ∨ H(ψ ∧ ¬preds[i], i+1, bdd & !bddvar(i))
}
```

Figure 1: Algorithm of the optimization

4.5 Optimization and Results

Although CVCL is an efficient and a state-of-the-art decision procedure, its performance is not so satisfactory when it deals with big expressions. Our optimization applies similar ideas in [17] to split big expressions into smaller ones. The bottom line is that at any time, only one of the enabled rules can be executed and the next state is updated accordingly. As a result, instead of combining all the transition relations rules defined the a model into one big $Rc()$ formula and using *vc->assertFormula()* to add it into the fact database, we use the divide-and-conquer technique to keep each rule separate. `PAM` implements this by logically conjuncting the concretized current state into the CVL expression of each rule, together with the predicates, to check if it is satisfiable or not. Figure 4.5 describes the basic idea.

By splitting big expressions into smaller ones cannot always guarantee improvement on performance, because the number of decision procedure calls is also increased at the same time. Our experiment on the FLASH protocol with 33 transition relation rules showed that by splitting the $Rc()$ expression into 33 smaller expressions can reduce the computation time pretty much. However, by removing `if-then-else` and `ruleset` in each rule and introducing subrules cannot gain much in computation time.

We have applied `PAM` on the 4 protocols shown in Table 1, and compared its performance before and after the optimization in Figure 4.5. All the experiments were performed on a machine with Intel Xeon 2.80GHz and 1.3G memory. The result is shown as in Table 1[1].

PAM	Bakery	AlternatingBit	German	FLASH
Before Opt.	8sec	240sec	450sec	>24hour
After Opt.	5sec	175sec	150sec	45min

Table 1: Experiment results before and after optimization

5 Conclusion

We present an implementation of predicate abstraction in Murphi using CVC Lite. According to our experiments, we believe that predicate abstraction is an effective verification technique. However, it needs to be carefully implemented to achieve both efficiency and precision. For example, exact symbolic simulation of each statement in the system can be very expensive, so a light-weighted abstract image computation is needed. Also refinement, esp. local refinement, is necessary to improve the precision. Overlapping approximations[12] and lazy abstraction[18] seem to fit well. Our future work will includes adding lazy abstraction and local refinement into `PAM`.

References

[1] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.

[2] Clark W. Barrett and Sergey Berezin. Cvc lite: A new implementation of the cooperating validity. In *CAV*, 2004.

[3] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[1] In the process of doing these experiments, we found a bug (`forallExpr`) in the CVCL C++ interface. We hope this bug will be eliminated in the next version of CVCL, and the computation time be roughly the same as in this experiment.

[4] Steven M. German. Formal design of cache memory protocols in ibm. *Form. Methods Syst. Des.*, 22(2):133–141, 2003.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[6] S. Graf and H.Saïdi. Construction of abstract state graphs with pvs. In *Conference on Computer Aided Verification*, 1997.

[7] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *CAV*, 1999.

[8] Corina Pasareanu, Radek Pelnek, and Willem Visser. Concrete model checking with abstract matching and refinement. In *CAV*, 2005.

[9] R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Prentice University Press, 1994.

[10] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, 2000.

[11] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD*, 2002.

[12] Shankar G. Govindaraju and David L. Dill. Approximate symbolic model checking using overlapping projections. In *First International Workshop on Symbolic Model Checking at Federated Logic Conference*, 1999.

[13] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, 2001.

[14] Anubhav Gupta and Ofer Strichman. Abstraction refinement for bounded model checking. In *Computer Aided Verification*, 2005.

[15] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided undeapproximation-widening for multi-process systems. In *POPL*, 2005.

[16] David L. Dill. The murϕ verification system. In *CAV*, 1996.

[17] In-Ho Moon, James H. Kukula, Kavita Ravi, and Fabio Somenzi. To split or to conjoin: The question in image computation. In *DAC '00: Proceedings of the 37th Conference on Design Automation*, 2000.

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL*, 2002.