Predicate Abstraction based Verification Framework: First results

Edgar Pek and Nikola Bogunović
Department of Electronics, Microelectronics,
Computer and Intelligent Systems
Faculty of Electrical Engineering and Computing
Unska 3, 10000 Zagreb
Phone: (01) 6129-999 int. 548 Fax: (01) 6129-653 Email: edgar.pek@fer.hr

***Abstract*— The Analysis of software artifacts is steadily advancing due to the increase in computational power and new verification methodologies. As a result there is a trend to approach the verification of the implementation level code and systems with unbounded state space rather than verification of abstract hand-built models of code.**

We consider a methodology which enables the verification of systems with unbounded state space. This methodology is called predicate abstraction. The predicate abstraction enable automatic construction of the finite abstraction from the infinite system. The finite abstraction has more behaviours than original system and preserves safety properties.

In this work we present first results in our implementation of the framework for the predicate abstraction.

I. Introduction

Computer programs can be described as formal descriptions constructed by programmers and executed by computers. The behaviour of the program during execution can be observed by its users (e.g. other programs). The programmer attempts to create the program that conforms to the behaviour expected by the user. Any discrepancy between actual and expected program behaviour could happen because the programmer didn't understand the user requirements, the user misunderstood the program's specification, the programmer designed a flawed program or because the computer executed the program incorrectly.

In this work we are concerned with the formal verification, in which program composition has semantics independent of its execution on computer and the program properties are well defined as well as their semantics. In other words, our concern are the flawed programs.

Modern approach to formal verification puts greater emphasis to the analysis of implementation level descriptions, instead of to manually constructed high-level code [1]. Our previous work on protocol verification [2], [3], [4] has been mostly concerned with manually obtained (ad-hoc) abstractions. Thus, we have noted the need for the new direction.

We have recognised that abstraction is a key element of a modern approach to automated verification of transition system properties that combines both deduction and exploration. We have chosen predicate abstraction [5] as the abstraction methodology.

This paper describes first results on the implementation of the predicate abstraction framework. The emphasis is on description of the mapping between a semantic level representation and a representation that is amenable to reasoning using decision procedures. The framework is being implemented in object-oriented setting using C++, and we use well-known open source tools, Lex and Yacc.

Predicate abstraction, the abstraction methodology, is described in the first section. Actual implementation of the methodology, called counter-example driven refinement is discussed in Section III. After that we explicate precise the model of computation (Section IV).

Section V describes the automated theorem prover that provides support for a representation and decision procedures that are used in the process of predicate abstraction. Preliminary results on our implementation of the predicate abstraction framework is given in the Section VI. Concluding observations are presented in the Section VII.

II. Predicate Abstraction

In this section we will describe predicate abstraction, the methodology based on a combination of the two dominant approaches in formal verification: deduction and exploration.

Deductive formal verification is based on the construction of the proof that the program satisfies a property. One of the techniques used in deductive approach is induction. Proof by induction often requires that inductive properties are suitably strengthened. Process of strengthening the invariants is usually achieved through careful manual analysis of failed proof attempts, which is a tedious task.

In the formal verification based on the exploration of the state space, the program is represented as transition among the states. The states of all possible executions are thoroughly examined in order to detect possible violation of the program properties. Since the basic idea of exploration is to analyse all the states from the computation space, it must be achieved in a finite number of steps. This constraint makes this approach less general than deduction.

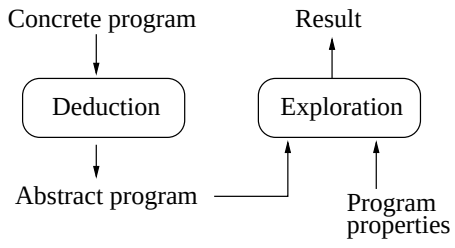

Fig. 1. Predicate abstraction as the synergy of deduction and exploration

Predicate abstraction is a methodology based on the synergy of deduction and exploration as shown in Figure 1. It is based on the observation that specific properties or classes of properties can be checked by examining an abstraction of the program using exploration. Valid abstract program is constructed from the concrete program using deduction. The analysis of abstract programs against properties is based on the exploration. The use of abstraction enables verification of the programs for which deduction and exploration would be individually unfeasible.

The theoretical aspects of predicate abstraction are quite intricate. This methodology is based on the abstract interpretation [6], and it was first described by Graf and Saidi in [5].

Predicate abstraction is a parametric abstraction technique that constructs a finite-state abstraction of an infinite state system. The first step in the predicate abstraction is to provide a set of predicates (defined over an infinite state system). If there are n predicates, then a state of the original program maps to a bit-vector of length n, where each bit has a value equivalent to the value of the corresponding predicate in that state. The obtained finite-state abstraction is amenable to the symbolic model checking [7].

III. Counterexample-driven refinement

Counterexample-driven refinement is one of the approaches in practical implementation of predicate abstraction. The process of predicate abstraction can be described in the following terms:

1) Definition of the predicates
2) Construction of an abstract state graph
3) Abstract state space exploration

Definition of the predicates is not well defined procedure, and it usually depends on an application. Most common approach is to provide predicates that appear in desired system properties. We will address this issue later in this section.

The construction of an abstract state graph provides an over-approximation of the set of reachable states. This is sufficient for the analysis of an invariant system properties.

An abstract state space exploration is usually accomplished by model checking a boolean abstraction of the concrete program.

In the process of verification one of the first concerns is to determine properties of the system one wants to check. Therefore, we now provide a formal treatment of an invariant system properties.

Definition 1. *An **invariant** for a transition system is a predicate, P, on a states, such that if P(q) is true, and $q \xrightarrow{a} r$ for some state, r, then P(r) is also true.*

Using an induction we can formulate the following theorem:

Theorem 1 (Invariant theorem). *Let P be invariant for a transition system. If P holds for the initial state, then P holds for all reachable states.*

Proof: The proof of the theorem can be done by induction on length of finite executions.

When checking an invariant property it is sufficient to check an abstraction that has more behaviours than original program. In other words, an abstraction over-approximates the behaviours of the original program (Fig. 2). This is a basic concept of abstract interpretation [6].

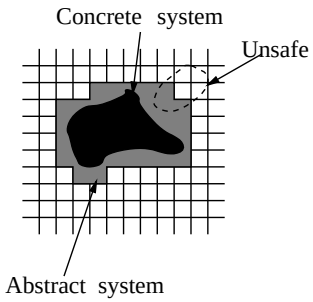

Fig. 2. Example of an abstraction and a possible relation with the unsafe states

Note that in the Figure 2 the unsafe properties are not abstracted for the sake of simplicity. If the invariant property holds of the over-approximation then it holds for the concrete system (original program). However, if the invariant does not hold for the abstraction, it may or may not hold for the original program. In that case, the verification process yields a "false negative". In the context of model checking, this means that the result is the spurious counterexample. One can determine if the counterexample is spurious using an automated theorem prover. The basic idea, as stated in [8], is to build a formula $f(t)$ from the trace t such that $f(t)$ is satisfiable if and only if the trace t is a concrete (feasible) execution trace. An automatic theorem prover is used to establish whether $f(t)$ is satisfiable. If the $f(t)$ is not satisfiable (the counterexample is spurious), then it is necessary to refine the abstraction by adding new predicates (which eliminate the spurious counterexample).

Previously mentioned verification process is actually symbolic model checking [7] procedure. It is being repeated after each refinement until all false counterexamples are eliminated.

The process of predicate abstraction with counterexample-driver refinement is depicted in Figure 3.

IV. Transition systems

In this section we will discuss our choice of system representation and how it was implemented. We have decided that system should be expressive enough but still not too complicated. Thus, we have opted for the Labelled Transition System (LTS) representation [9].

Definition 2 (Labelled Transition System). *A labelled transition systems (LTS) is a triple $T = (S, Act, \rightarrow)$, where S is a set of states, Act is a set of actions, and $\rightarrow \subseteq S \times Act \times S$ is a transition relation. A transition $(s, a, s') \in \rightarrow$, can be described more intuitively as $s \xrightarrow{a} s'$, and it states that system can evolve from state s to*

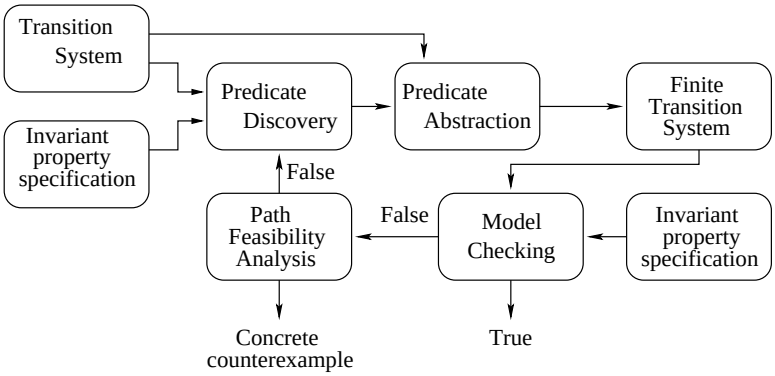

Fig. 3. A diagram of the counterexample-driven refinement

*state s' exchanging the action a with it environment in the process. Notation $s \xrightarrow{a} s'$ describes **transition** from s to s' labelled by action a, and s' is an $a - successor$ of s.*

Labelled transition systems originate from concurrency theory, where they are used as on operational model of process behaviour.

In our work labelled transition system have been implemented using the Fair Transition System (FTS) representation as described by [10] and implemented in Stanford Temporal Prover (STeP) [11]. However, currently we are using only a subset of the STeP's FTS definition that is necessary for the first phase of development. In the sequel we will give details about Fair Transition System specification.

A. Fair Transition System

Specification of the fair transition system [11] consists of:

- Declaration of types and variables.
- An optional **initial condition**, which is assumed to hold at the initial state of every computation.
- A set of transitions.

The transitions are expressed in terms of **transition relations**. A transition relation describes how the system can change from one state to the next. The previous intuitive description can be stated formally: the transition relation is expressed by a formula over the set of primed and unprimed variables. Transition relation in STeP can be specified by a combination of five different fields. In our implementation we have restricted to the two fields which are necessary to describe the transition relation. The two fields are:

- The `enable` field contains the **enabling condition**. The enabling condition is a formula that may not contain any primed variables. The transition can be executed only if the enabling condition is satisfied.
- The `assign` field lists the assignments for those variables whose next-state values can be described as a function of the current-state (unprimed) variables. The assignments listed in the `assign` field are made in parallel. E.g. `assign x:=y, y:=x` actually swaps the values of `x` and `y` and is not just an assignment of `y` to `x`.

V. CVC LITE

The automated theorem prover, which have been used in the implementation of the predicate abstraction framework, is described in this section.

CVC Lite (CVCL) [12] is a tool used for establishing the validity (or satisfiability) of a first order formula over a group of specific theories. It has been developed to replace the Cooperating Validity Checker (CVC) [1], which was successor of the Stanford Validity Checker (SVC).

CVCL accepts as input one or more assertional formulas and a query formula. It then checks whether assertional formula imply the query formula. Former intuitive definition amounts to following:

$$\Gamma \vdash_L^? \omega \tag{1}$$

Where, Γ is a set of assertion formulas, L represent the set of specific theories (implemented in CVCL), while ω stands for the formula which validity is being established. Each formula belongs to first-order formulas whose parameters (non-logical symbols) must be among the union of theories.

The CVCL theories are:

- **Equality with uninterpreted functions**
  This is the simplest supported theory, it contains an arbitrary number of functions and predicates. The functions and predicates are "uninterpreted" which means that theory doesn't provide any additional information about them, other that they are functions and predicates.
- **Arrays**
  CVCL consists of a theory of abstract arrays with two operations: *read* and *write*. The first operation is used to read from a location in an abstract array, while with the second operation one can create a new array by writing a new value to a location in a existing array.
- **Records and tuples**
  In CVCL simple aggregate types, such as a record and a tuple can be described. Simple decision procedures is used to handle set of operations to create, read from, write to these new datatypes.

[1]The name, CVC Lite, doesn't imply that new tool is less powerful. It means that this tool is easier to use, maintain and extended without sacrificing functionality and performance

- **Arithmetic**
  Decision procedures for arithmetic in CVCL support theory of linear arithmetic over both reals and integers. Furthermore, any combination of real and integer variables can be handled with CVCL. However, support for the nonlinear arithmetic is very limited.

All this decision procedures are very useful for wide variety of verification tasks. One important novelty, which is very useful in th protocol verification, is native support for quantifiers. The quantifiers make the logic undecidable, but in many applications simple heuristics for the quantifier instantiation can be sufficient.

VI. Results

This section contains brief report about our implementation of the predicate abstraction framework. The emphasis is on the parser for the fair transition system (see Section IV-A). The result of the parsing process are date structures expressed in terms of the CVCL structure (expression, types, etc.).

Fair Transition System (FTS) parser inputs are: a fair transition system model, a specification of invariants and a definition of abstraction predicates. Input file is parsed using well-known open source tools: Lex and Yacc. The first part of the grammar for the Fair Transition System is shown on Figure 4.

```
fts : TOK_TS declarations initExpn
       transitions opt1
    ;

declarations : typeDecls varDecls
             | varDecls
             ;

opt1 : {}
     | invarSpecs
     | absPreds
     | invarSpecs absPreds
     | absPreds invarSpecs
     ;

typeDecls  : typeDecl
           | typeDecls typeDecl
           ;

varDecls  : varDecl
          | varDecls varDecl
          ;

invarSpecs : invarSpec
           | invarSpecs invarSpec
           ;

absPreds  : absPred
          | absPreds absPred
          ;

initExpn  : TOK_INIT expn
          ;
transitions : transition
            | transition transitions
            ;
...
```

Fig. 4. Part of the FTS grammar

One of the ideas in the implementation is to provide interfaces on two different levels of abstraction. The first interface provides access to the data structures that naturally follow from the process of abstraction: the symbol table, an initial expression, the transition representation. For each transition it is necessary to describe enable (guard) expressions and assign expressions. The second interface is defined using CVCL primitives (expressions, type, etc.). The main usage of that interface is application of the decision procedures as provided by the CVCL theorem prover (Section V).

The second interface is currently used to experiment with the predicate discovery and the abstraction procedure. Initially, we have put emphasis on the abstraction procedure. The process of model checking is carried on with the open-source model checker - NuSMV [13].

In the future work the main emphasis will be on construction of the support for the custom data structure representation in CVCL and handling of the quantifiers. After that the procedure for an abstraction can be appropriately refined. The refined version should provide new ways of handling a parameterised system by providing new heuristics. The heuristic procedures are necessary since the logic with quantifiers is undecidable.

VII. Conclusion

In this work we have presented initial results obtained in the implementation of the predicate abstraction framework. The predicate abstraction is a powerful methodology for the formal verification since it provides various new means to tackle the notorious state explosion problem. It combines well known formal verification methodologies, automated theorem proving and model checking. The main components of predicate abstraction implementation are predicate discovery, abstraction construction and exploration of the abstraction. Each of these components can be approached by various means.

In the implementation of the framework we have decided to use simple, yet flexible internal representation in terms of fair transition system. The fair transition system is mapped to the CVCL primitives which is useful in the process of predicate discovery.

The process of predicate discovery is sill to be defined, but there are two main directions which we are going to pursue. The first one is to infer the predicate from the invariant system specifications. While, the second one is to provide new predicates by examining counterexamples. The second one forms the prominent approach in an implementation of the predicate abstraction which has been described in the Section III. The process of abstraction is currently done as described in [5].

The exploration of the state space produced by the process of abstraction is achieved by the symbolic model checker NuSMV.

In the continuation of this project we should finish the complete support for the representation of the quantifiers. The main reason is in the fact that parametrisation problem is still largely unexplored. Other directions naturally follow from the process of counterexample-driven refinement and include predicate discovery, abstraction process and exploration techniques.

REFERENCES

[1] G. J. Holzmann, "Trends in software verification," in *FME 2003: Formal Methods*, ser. Lecture Notes in Computer Science, D. M. Keijiro Araki, Stefania Gnesi, Ed., vol. 2805. Springer-Verlag, 2003, pp. 40–50.

[2] N. Bogunović and E. Pek, "Verification of mutual exclusion algorithms with SMV system," in *Proceedings of IEEE Region 8 Eurocon 2003: Computer as a Tool*, B. Zajc and M. Tkalčić, Eds., vol. II. IEEE, September 2003., pp. 12–25.

[3] E. Pek and N. Bogunović, "Formal verification of communication protocols in distributed systems," in *MIPRO 2003, Proceedings of the Joint Conferences Computers in technical systems and Intelligent systems*, B. Leo and R. Slobodan, Eds., May 2003., pp. 44–49.

[4] ——, "Formal verification of logical link control and adaptation protocol," in *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference, MELECON 200*, M. Maja, P. Branimir, T. Željko, and B. Željko, Eds., May 2004., pp. 583–586.

[5] S. Graf and H. Saïdi, "Construction of abstract state graphs with pvs," in *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*. Springer-Verlag, 1997, pp. 72–83.

[6] P. Cousot and R. Cousot, "Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, Jan. 1977, pp. 238–252.

[7] K.L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992, CMU-CS-92-131.

[8] T. Ball, "Formalizing Counterexample-driven Refinement with Weakest Preconditions," Microsoft Research, One Microsoft Way Redmond, Tech. Rep. MSR-TR-2004-134, December 2004.

[9] M. Müller-Olm, D. Schmidt, and B. Steffen, "Model-checking: A tutorial introduction," in *Static Analysis*, ser. Lecture Notes in Computer Science, A. Cortesi and G. Filé, Eds., vol. 1694. Springer, 1999, pp. 330–354.

[10] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[11] Z. Manna, "STeP: The Stanford Temporal Prover," STeP group, Computer Science Department, Stanford University, Tech. Rep. STAN-CS-TR-94-1518, July 1995.

[12] C. Barrett and S. Berezin, "CVC Lite: A new implementation of the cooperating validity checker," in *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, ser. Lecture Notes in Computer Science, R. Alur and D. A. Peled, Eds., vol. 3114. Springer-Verlag, July 2004, pp. 515–518, boston, Massachusetts.

[13] R. Cavada, A. Cimatti, E. Olivetti, M. Pistore, and M. Roveri, *NuSMV 2.1 User Manual*, 2002. [Online]. Available: http://nusmv.irst.itc.it/NuSMV/userman/v21/nusmv.pdf