2012 IEEE Sixth International Symposium on Theoretical Aspects of Software Engineering

Symbolic Model Checking for Propositional Projection Temporal Logic

Tao Pang, Zhenhua Duan* and Cong Tian
Institute of Computing Theory & Technology and ISN Laboratory
Xidian University
Xi'an 710071, P.R. China
Email: t_pang@126.com, {zhhduan, ctian}@mail.xidian.edu.cn

***Abstract*—This paper presents a symbolic model checking algorithm for Propositional Projection Temporal Logic (PPTL). Within this method, the model of a system is specified by a Kripke structure *M*, and the desired property is specified in a PPTL formula *P*. First, *M* is symbolically represented with boolean functions while $\neg P$ is transformed into its normal form. Then the set of states in *M* that satisfies $\neg P$, namely *Sat*($\neg P$), is computed recursively with respect to the transition relations. Thus, whether the system satisfies the property can be equivalently checked by determining the emptiness of *Sat*($\neg P$). All the operations above can be implemented by a graph algorithm operated on ROBDDs.**

***Keywords*-Propositional Projection Temporal Logic; ROBDD; Symbolic Model Checking; Verification**

I. Introduction

Model checking [1] is widely used in verification of concurrent systems. With model checking, the system is modeled as a state-transition structure while the specifications are expressed in temporal logic formulas [2]. Although a number of temporal logics, such as Computation Tree Logic (CTL) [3] and Linear Temporal Logic (LTL) [4], have been proposed to specify properties of concurrent systems, they are not powerful enough to describe full regular properties [23]. Fortunately, these properties can be specified by Propositional Projection Temporal Logic (PPTL) [5], [6] with chop and projection constructs. In fact, PPTL has been proved to have the power of full regular expressiveness [8].

With PPTL, lots of logic laws are formalized and proved, and a decision procedure for checking satisfiability of PPTL formulas is given in [7]. Further, an automata based model checking algorithm for PPTL is given in [10] and implemented within Spin [9]. Therefore, full regular properties of systems can be automatically verified with Spin. However, as known, automata-based model checking algorithm can easily lead to state space explosion when the number of concurrent components is large. To conquer this problem, several approaches, such as Symbolic Model Checking (SMC) [15], Bounded Model Checking (BMC) [21], Abstract Model Checking (AMC) [19], and Compositional Model Checking [13], have been proposed with success. SMC is a successful mechanism based on Reduced Ordered Binary Decision Diagrams (ROBDDs) [11] for hardware systems verification. With SMC, states and transition relations are all encoded in boolean functions. As ROBDD is a canonical form for representing boolean functions, operations on state space can be implemented as an efficient graph algorithms based on ROBDDs. Much larger practical systems can be verified using this method than systems with explicit state model checking. Symbolic model checking for linear-time and branching-time temporal logics, have been studied in recent years [7], [24]. Corresponding model checkers based on them have also been implemented, such as SMV [26], NuSMV [20] on CTL and NuSMV2 [22] on LTL. Other model checkers including CHESS [27], BLAST [28] and so on are also available.

Therefore, we are motivated to investigate symbolic model checking algorithms and the corresponding verification techniques for PPTL. With our method, the system to be verified is modeled as a Kripke structure $M = (S, R, L)$, while the property of the system to be verified is described by a PPTL formula *P*. First, $\neg P$ is transformed to its normal form [7], and then the sets of states in M that satisfy the sub-formulas of $\neg P$ can be recursively computed with respect to the transition relation *R*. Thus, whether $M \models P$ can be equivalently checked by determining the emptiness of the set of states in M where the $\neg P$ holds. Since *M* is stored symbolically with boolean functions, all the operations above can be implemented as an efficient graph algorithm operated on ROBDDs.

Our method has several advantages. First, since PPTL has the power of full regular expressiveness, it enables us to describe full regular properties which are failed to be specified by CTL and LTL; secondly, due to the reduced storage scheme employed by ROBDDs, we can, to some extent, alleviate the state space explosion problem encountered in automata-based model checking for PPTL; finally, compared with Spin-based model checking, our method offers a unified model checking algorithm for PPTL formulas with both finite and infinite models.

The rest of this paper is organized as follows. The following section briefly presents the preliminaries, involving the syntax, semantics of the underlying logic, the definition of normal form and LNFG as well as the symbolic representation of Kripke structure. The proposed symbolic model checking algorithm for PPTL and a small case study are presented in Section 3. Finally, conclusions are drawn in

978-0-7695-4751-0/12 $26.00 © 2012 IEEE
DOI 10.1109/TASE.2012.35

Section 4.

II. Preliminaries

A. Propositional Projection Temporal Logic

Our underlying logic is Propositional Projection Temporal Logic (PPTL) [5], which is an extension of Propositional Interval Temporal Logic (PITL) [12].

1) Syntax: Let $Prop$ be a countable set of atomic propositions. The syntax of PPTL formula P is given as follows:

$$P ::= p \mid \bigcirc P \mid \neg P \mid P_1 \vee P_2 \mid (P_1, \ldots, P_m)\ prj\ P$$

where $p \in Prop$, $P_1, \ldots, P_m$ and P are all well-formed PPTL formulas. $\bigcirc$ (`next`) and *prj* (`projection`) are basic temporal operators. A PPTL formula is called a state formula if it contains no temporal operators, and a temporal formula otherwise. The abbreviations $true, false, \vee, \rightarrow$ and $\leftrightarrow$ are defined as usual. Moreover, we have the following derived formulas:

ε	$\stackrel{\text{def}}{=}$	$\neg \bigcirc true$	$more$	$\stackrel{\text{def}}{=}$	$\neg\varepsilon$
$\bigcirc^0 P$	$\stackrel{\text{def}}{=}$	P	$\bigcirc^n P$	$\stackrel{\text{def}}{=}$	$\bigcirc(\bigcirc^{n-1} P)$
$\odot P$	$\stackrel{\text{def}}{=}$	$\varepsilon \vee \bigcirc P$	$P;Q$	$\stackrel{\text{def}}{=}$	$(P, Q)\ prj\ \varepsilon$
$\Diamond P$	$\stackrel{\text{def}}{=}$	$true\ ;\ P$	$\Box P$	$\stackrel{\text{def}}{=}$	$\neg\Diamond\neg\ P$
$len\ n$	$\stackrel{\text{def}}{=}$	$\bigcirc^n\ \varepsilon$	$halt(P)$	$\stackrel{\text{def}}{=}$	$\Box(\varepsilon \leftrightarrow P)$

where $\odot$ (`weak next`), $\Box$ (`always`), $\Diamond$ (`sometimes`) and ; (`chop`) are derived temporal operators; ε (`empty`) denotes an interval with zero length, and *more* means the current state is a non-final state of an interval; $halt(P)$ is true over an interval if and only if P is true at the final state.

2) Semantics: In accordance with the definition of Kripke structure [24], we define a state *s* as a mapping from *Prop* to *B* = {*true*, *false*}, *s*: *Prop* $\longrightarrow$ *B*. We will denote the valuation of *p* at the state *s* as *s*[*p*].

An interval σ is a non-empty sequence of states. The length of σ, $|\sigma|$, is the number of states minus 1 if σ is finite, and ω otherwise. To have a uniform notation for both finite and infinite intervals, we unite the set of non-negative integers N_0 with $\{\omega\}$, $N_\omega = N_0 \cup \{\omega\}$, and extend the relational operators, =, <, $\leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0$, $i < \omega$. Moreover, we define $\preceq$ as $\leq -\{(\omega, \omega)\}$. For simplicity, we will denote σ as $< s_0, \ldots, s_{|\sigma|} >$, where $s_{|\sigma|}$ is undefined if σ is infinite. Similarly, $\sigma_{(i\ldots j)}$ ($0 \leq i \preceq j \leq |\sigma|$) and $\sigma^{(k)}$ ($0 \leq k \preceq |\sigma|$) represents the subinterval $< s_i, \ldots, s_j >$ and $< s_k, \ldots, s_{|\sigma|} >$ respectively. The concatenation of a finite interval σ_1 with another interval (or empty string) σ_2 denoted by $\sigma_1 \cdot \sigma_2$ is defined as follows:

$$\sigma_1 \cdot \sigma_2 = \begin{cases} \sigma_1 & \text{if } \sigma_2 = \epsilon \\ < s_0, ..., s_i, s_{i+1}, ... > & \text{if } \sigma_1 =< s_0, ..., s_i > \\ & \text{and } \sigma_2 =< s_{i+1}, ... > \end{cases}$$

Let σ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq r_2 \leq \cdots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the projected interval

$$\sigma \downarrow (r_1, \ldots, r_h) =< s_{t_1}, s_{t_2}, \ldots, s_{t_l} >$$

where $t_1, \ldots, t_l$ is the longest strictly increasing subsequence obtained from $r_1, \ldots, r_h$ by deleting all duplicates. For instance,

$$< s_0, s_1, s_2, s_3, s_4 >\downarrow (0, 2, 2, 3) =< s_0, s_2, s_3 >$$

An interpretation is a triple $\mathcal{I} = (\sigma, k, j)$, where σ is an interval, k is an integer, and j an integer or ω such that $k \preceq j \leq |\sigma|$. The notation $(\sigma, k, j) \models P$ means that formula *P* is interpreted and satisfied over the subinterval $\sigma_{(k\ldots j)}$ with the current state being s_k. The satisfaction relation ($\models$) is inductively defined as follows:

$\mathcal{I} \models p$ iff $s_k[p]$ = *true*, for any $p \in Prop$
$\mathcal{I} \models \neg P$ iff $\mathcal{I} \not\models P$
$\mathcal{I} \models P \vee Q$ iff $\mathcal{I} \models P$ or $\mathcal{I} \models Q$
$\mathcal{I} \models \bigcirc P$ iff $k < j$ and $(\sigma, k+1, j) \models P$
$\mathcal{I} \models (P_1, \ldots, P_m)$ *prj* Q, if there exist integers $k = r_0 \leq r_1 \leq \cdots \leq r_m \leq j$ such that $(\sigma, r_0, r_1) \models P_1$, $(\sigma, r_{l-1}, r_l) \models P_l, 1 < l \leq m$, and $(\sigma', 0, |\sigma'|) \models Q$ for one of the following σ':
(a) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma_{(r_{m+1}\ldots j)}$ or
(b) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_h)$ for some $0 \leq h \leq m$

3) Satisfaction and Validity: A formula *P* is satisfied by an interval σ, denoted by $\sigma \models P$, if $(\sigma, 0, |\sigma|) \models P$. A formula *P* is called satisfiable if $\sigma \models P$ for some σ. A formula *P* is valid, denoted by $\models P$, if $\sigma \models P$ for all σ.

B. Normal Form of PPTL

Normal forms are useful in constructing LNFGs [7], [25] of PPTL formulas. In the following, we briefly present the definition of normal form and the relevant concepts.

Definition 1: Let Q be a PPTL formula and Q_p denotes the set of atomic propositions appearing in Q. The normal form of Q is defined as follows:

$$Q \equiv \bigvee_{j=1}^{m} (Q_{ej} \wedge \varepsilon) \vee \bigvee_{i=1}^{n} (Q_{ci} \wedge \bigcirc Q'_i)$$

where $Q_{ej} \equiv \bigwedge_{k=1}^{m_0} \dot{q_{jk}}$, $Q_{ci} \equiv \bigwedge_{h=1}^{n_0} \dot{q_{ih}}$, $l = |Q_p|$, $1 \leq m$ (also n) $\leq 3^l$, $1 \leq m_0$ (also n_0) $\leq l$, $q_{jk}, q_{ih} \in Q_p$, for any $r \in Q_p$, $\dot{r}$ denotes r or $\neg r$; Q'_i is a general PPTL formula.

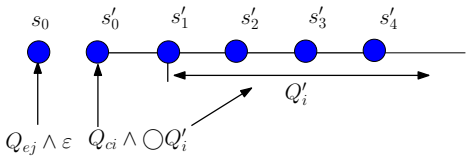


Figure 1. Terminating part and future part of a normal form

Normal forms enable us to rewrite a PPTL formula into two different parts (Fig. 1): the terminating part $Q_{ej} \wedge \varepsilon$ indicates that Q_{ej} is satisfied over an interval with a single state s_0, while the future part $Q_{ci} \wedge \bigcirc Q'_i$ means that Q_{ci} is satisfied by the current state s'_0 of an interval σ ($|\sigma| > 1$) and Q'_i is satisfied over the sub-interval $\sigma^{(1)} =< s'_1, s'_2, s'_3, s'_4, ... >$.

Definition 2: For a normal form,

$$Q \equiv \bigvee_{j=1}^{m} (Q_{ej} \wedge \varepsilon) \vee \bigvee_{i=1}^{n} (Q_{ci} \wedge \bigcirc Q'_i)$$

if $\bigvee_i Q_{ci} \equiv true$ and $\bigvee_{i \neq j} (Q_{ci} \wedge Q_{cj}) \equiv false$, the normal form is called a complete normal form.

Complete normal form plays an important role in transforming the negation of a PPTL formula into its normal form. For instance, if the complete normal form of *P* is: $P \equiv \bigvee_{j=1}^{m} (P_{ej} \wedge \varepsilon) \vee \bigvee_{i=1}^{n} (P_{ci} \wedge \bigcirc P'_i)$ we have, $\neg P \equiv \bigwedge_{j=1}^{m} (\neg P_{ej}) \wedge \varepsilon \vee \bigvee_{i=1}^{n} (P_{ci} \wedge \bigcirc \neg P'_i)$. An important conclusion is that any PPTL formula can be rewritten to its normal form and complete normal form [7].

C. Labeled Normal Form Graph

Labeled Normal Form Graph (LNFG) of a PPTL formula is generated with respect to its normal form. The finiteness of LNFG has been proved in [7].

Definition 3: The LNFG of a PPTL formula *P* is a directed graph, $G = (V(P), E(P), V_0, \mathbb{L} = \{\mathbb{L}_1, \ldots, \mathbb{L}_m\})$, where $V(P)$ denotes the set of vertices and $E(P)$ is the set of directed edges in the graph, $V_0 \subseteq V(P)$ is the set of root vertices, while each $\mathbb{L}_k \subseteq V(P)$, $1 \leq k \leq m$, is the set of vertices with $\mathtt{l_k}^{\sim}$ which means that the finiteness of some chop formula has not been satisfied at this vertex [25]. In $V(P)$, each vertex is specified by a PPTL formula, while in $E(P)$, each edge is a directed arc labeled with a state formula Q_e from vertex Q to vertex Q_1 and is denoted by a triple (Q, Q_e, Q_1). $V(P)$ and $E(P)$ of G can be inductively produced by repeatedly transforming the unmarked vertices into their normal forms:
(a) If P is not in the form of $\bigvee_i P_i$, $P \in V(P)$, $V_0 = \{P\}$, otherwise, for each i, $P_i \in V_0$, $P_i \in V(P)$;
(b) for all $Q \in V(P) \setminus \{\varepsilon, false\}$, if $Q \equiv \bigvee_{j=1}^{m} (Q_{ej} \wedge \varepsilon) \vee \bigvee_{i=1}^{n} (Q_{ci} \wedge \bigcirc Q'_i)$, then $\varepsilon \in V(P)$, $(Q, Q_{ej}, \varepsilon) \in E(P)$ for each $1 \leq j \leq m$; $Q'_i \in V(P)$, $(Q, Q_{ci}, Q'_i) \in E(P)$ for all $1 \leq i \leq n$.

In an LNFG, a finite path, $\pi_f = (v_0, e_0, v_1, e_1, \ldots, \varepsilon)$, is an alternative sequence of vertices and edges obtained by traveling from a root vertex to the ε vertex and keeping vertices and edges encountered in their occurrence order, while an infinite path, $\pi_i = (v_0, e_0, v_1, e_1, \ldots, (v_i, e_i, \ldots, v_j, e_j)^{\omega})$ can also be gained similarly except that the ε vertex is not included in. Note that, in an infinite path, there must exist some vertices, e.g. $v_i, \ldots, v_j$, occurring for infinitely many times. For convenience, sometimes, in a path of LNFG of a PPTL formula P, a vertex v_i can be replaced by a formula $Q_i \in V(P)$ while an edge e_i can be replaced by a state formula $Q_{ie} \in E(P)$. In fact, finite (or infinite) paths in the LNFG of a formula *P* characterize finite (or infinite) models of *P*. This brings the satisfiability of a PPTL formula into contact with paths in its LNFG [7], which will be employed later in our model checking algorithm.

D. Symbolic Representation of System Models

The key idea of symbolic model checking lies in expressing a problem in a form where all the objects are represented as boolean functions, which can then be efficiently manipulated by ROBDDs. In the following, we briefly review ROBDDs and present the method for the symbolic representation of Kripke Structure.

1) Reduced Ordered Binary Decision Diagrams: A Binary Decision Diagram (BDD) [11] represents a boolean function as a rooted, directed acyclic graph, with internal vertices corresponding to the variables over which the function is defined and terminal vertices labeled by 0 and 1. For example, Fig. 2 represents the BDD of the boolean function $f(a, b, c, d) = (a \wedge b) \vee (c \wedge d)$.

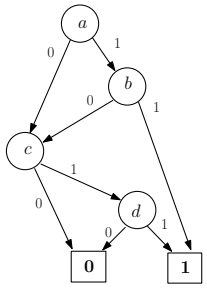


Figure 2. BDD of boolean function $f(a, b, c, d) = (a \wedge b) \vee (c \wedge d)$

A given boolean function can be represented by many different BDDs. To insure each boolean function having a unique representation, two restrictions must be placed on the form of BDDs: (1) A total ordering is given for the variables in the boolean function. For instance, the variable ordering in Fig. 2 is $a < b < c < d$. (2) There should be no isomorphic subtrees or redundant vertices in the BDD. This can be achieved by repeatedly applying three transformation rules in [11].

We use the term "ROBDD" in the following sections to refer to a maximally reduced graph that obeys a given ordering.

2) Symbolic Representation of Kripke Structures: A Kripke structure is basically a graph whose nodes represent the reachable states of the system and whose edges represent state transitions. A labeling function maps each node to a set of properties hold in the corresponding state.

A frame work for encoding finite domains, sets, and k-ary relations is proposed in [14], where all these objects can be

represented as ROBDDs with respect to their characteristic functions. From their method, we can infer that the key point of representing a Kripke structure $M = (S, R, L)$ is to find a proper encoding for S. Since L labels each $s \in S$ with a set of atomic propositions hold in it, we can employ these atomic propositions to encode it. Let $AP = \{a_1, a_2, \ldots, a_n\}$ be the set of atomic propositions over which M is defined. We assume a fixed order on atomic propositions in AP, $a_1 < a_2 < \ldots < a_n$ and assign each atomic proposition a_i a corresponding boolean variable x_i. Then the encoding of each state $s \in S$ could be thought as a valuation of the boolean vector $(x_1, x_2, \ldots, x_n)$, where for each $1 \leq i \leq n$, x_i equals to 1 if $s[a_i] = true$ and x_i is 0 otherwise. Accordingly, *S* and its subsets can be represented with their characteristic functions.

As R is a set of state transitions, each transition in it should be represented at first. We may need a pair of boolean vectors, $((x_1, x_2, \ldots, x_n), (x'_1, x'_2, \ldots, x'_n))$, to encode a transition $s \rightarrow s'$, where the unprimed vector denotes the current state *s* and the primed vector represents the next state s'. Similar to the boolean encoding of S, for each $1 \leq i \leq n$, x_i (or x'_i) equals to 1 if $s[a_i] = true$ (or $s'[a_i] = true$) and x_i (or x'_i) is 0 otherwise. Then we can get *R* represented by its characteristic function over $X \cup X'$, where $X = \{x_1, x_2, \ldots, x_n\}$ and $X' = \{x'_1, x'_2, \ldots, x'_n\}$.

Finally, we consider the labeling function *L*. In [1], *L* is considered as a mapping from atomic propositions to subsets of *S*: $AP \longrightarrow 2^S$ so that it can be conveniently represented with the encoding above. Nevertheless, in our opinion, the explicit encoding of *L* is not necessary, as the representation of *S* implies the encoding of the labeling function.

III. Symbolic Model Checking for PPTL

A. Symbolic Model Checking for PPTL

Similar to SMC for CTL and LTL, our approach employs a Kripke structure $M = (S, R, L)$ to model the system and a PPTL formula *P* to specify the property. The main idea of our algorithm is given as follows.

Definition 4: Let M be the system model, P be a PPTL formula. The notation $M, \Pi \models P$ means that P holds along a path Π in M. We will simply write $\Pi \models P$, when M is unambiguous in the context.

Accordingly, we define the set of states satisfying a PPTL formula P, denoted by $Sat(P)$, as follows:

Definition 5: Let M be the system model, P be a PPTL formula. For each path $\Pi \in M$, if $\Pi \models P$, then the first state of Π, $\Pi(0) \in Sat(P)$.

In the LNFG of P, finite or infinite paths characterize finite or infinite models of P. With the function $P2M(\pi, G)$ formalized in [25], given a path π in the LNFG of P, G, an interval (or a model) σ_π can be obtained. Since a path in system model M can also be treated as an interval, the model checking procedure is equivalent to finding all paths that characterize models of P and then checking

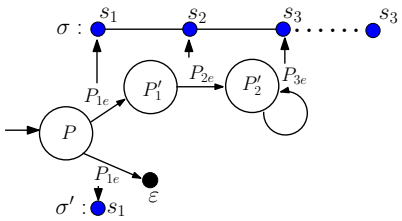


Figure 3. LNFG of *P*

whether or not all the paths in M are models (i.e. the intervals) of P. For instance, there exist an infinite path $\pi_i = (P, P_{1e}, P'_1, P_{2e}, (P'_2, P_{3e})^\omega)$ in LNFG of P as shown in Fig. 3. Intuitively, with respect to Definition 3 and $P2M$, P can be satisfied over an interval $\sigma =< s_1, s_2, {s_3}^\omega >$, where P_{1e}, P_{2e} and P_{3e} hold in s_1, s_2 and s_2 respectively. Similarly, for finite path $\pi_f = (P, P_e, \varepsilon)$, another model $\sigma' =< s_1 >\models P$ can be constructed, where state formula P_{1e} holds in s_1. If $s_1 \rightarrow s_2$, $s_2 \rightarrow s_3$ and $s_3 \rightarrow s_3$ are included in the transition relation R, then $M \models P$. Finally, with definition 5, we have $Sat(P) = \{s_1\}$, $Sat(P_1) = \{s_2\}$ and $Sat(P_2) = \{s_3\}$. Inversely, if $M \models P$, then the number of states in M where $\neg P$ hold is zero, which means that $Sat(\neg P) = \phi$ (see Theorem 1). Moreover, the computation of $Sat(P)$ performed by a backtrace scanning all the paths in the LNFG of P, that is $Sat(P)$ can only be determined after $Sat(P_1)$ and $Sat(P_2)$. This can be done by repeatedly rewrite the unmarked formulas into their normal forms, and compute the state set satisfying the new generated sub-formulas firstly.

Theorem 1: Let $M = (S, R, L)$ be a system model and P be a proepry formula in PPTL. $M \models P$ iff the set of states satisfying $\neg P$ is empty, i.e. $Sat(\neg P) = \phi$.

Proof: "$\Rightarrow$" If $M \models P$, then $Sat(\neg P) = \phi$.

If $M \models P$, then P holds along all paths of M. Consequently, there exists no path Π of M, such that $\Pi \models \neg P$. According to the definition of $Sat(\neg P)$, $Sat(\neg P) = \phi$.

"$\Leftarrow$" If $Sat(\neg P) = \phi$, then $M \models P$.

With the definition of $Sat(\neg P)$, if $Sat(\neg P) = \phi$, then there is no path $\Pi \in M$, such that $\Pi \models \neg P$. That is, P holds along all paths of M. Hence, $M \models P$.

Finally, we can conclude that $M \models P$ iff $Sat(\neg P)$ is empty. □

The idea mentioned above is implemented by a procedure "CheckPPTL" who takes the property formula P and the transition relations R as its arguments, and returns a ROBDD representing $Sat(P)$, namely the set of states in system model M satisfying P. To this end, each object of M is symbolically represented with boolean functions. Then, *P* is transformed to its normal form NF_P, where all the temporal operators occur in *P* are normalized by "$\bigcirc$" (`next`) and state formulas. Further, with the semantic of "$\bigcirc$" operator,

$Sat(P)$ can be computed recursively using a ROBDD-based algorithm with respect to *R* and NF_P.

The pseudo code of CheckPPTL is demonstrated in Fig. 4. Note that , "+" and "·" are logic "OR" and "AND" operators respectively. B_P is the boolean encoding of $Sat(P)$. $R^{-1}(B_P)$ denotes the preimage of B_P under the transition relation *R*. NF(*P*) is the procedure for constructing the normal form of *P*.

```
function CheckPPTL (P, R)
/* precondition: P is a PPTL formula, R is the transition relations
   of system model M = (S, R, L) */
/* postcondition: CheckPPTL returns an ROBDD representing Sat(P) */
begin
    boolean B_P = 0, int i;
    for i=1 to n  boolean B_{P_i} = 0;  end for
    NF_P= NF (P);   /* NF(P) = V_{i=1}^{n} P_i where P_i can either be
          the terminating part (Q_e ∧ ε) or the future part Q_ie ∧ ◯Q'_i */
    for i=1 to n
        case
            P_i ≡ Q_e ∧ ε : B_{P_i} = B_{Q_e};
            P_i ≡ Q_ie ∧ ◯Q'_i (Q'_i is not marked): mark Q'_i,
                  B_{P_i} = (B_{Qie} · R^{-1}(CheckPPTL(Q'_i, R)));
            P_i ≡ Q_ie ∧ ◯Q'_i (Q'_i has been marked):
                  B_{P_i} = (B_{Q_ie} · R^{-1}(fixpoint(τ(B_{Q'_i})))) ;
        end case
    end for
    B_P = B_{P_1} + B_{P_2} + ... + B_{P_n} ;
    return B_P;
end
```

Figure 4. Algorithm for Computing the ROBDD representation of $Sat(P)$

The basic idea of the Algorithm CheckPPTL is recursive. We firstly rewrite *P* into its normal form NF_P, and then consider each disjunct P_i of NF_P:

(a) If P_i is $Q_e \wedge \varepsilon$, then according to the definition of normal form, Q_e is a state formula. Thus the computation of B_{P_i} is straightforward.

(b) If P_i is $Q_{ie} \wedge \bigcirc Q'_i$ and Q'_i has never appeared during the transformation of *P* into its normal form. We first compute $B_{Q'_i}$ with the same procedure CheckPPTL, and then $R^{-1}(B_{Q'_i})$ accordingly. B_{P_i} can be obtained by exerting logic "AND" operation on $B_{Q_{ie}}$ and $R^{-1}(B_{Q'_i})$, $B_{P_i} = B_{Q_{ie}} \cdot R^{-1}(B_{Q'_i})$.

(c) If P_i is $Q_{ie} \wedge \bigcirc Q'_i$ and Q'_i has been encountered during the transformation of *P* into its normal form, the procedure of computing B_{P_i} turns to be much more intricate because of the involvement of infinite paths in the LNFG. For instance, (P'_2, P_{3e}, P'_2) occurs infinitely many times in the infinite path π_i in Fig. 3, which means that the new generated formula during the transformation of P'_2 into normal form is P'_2 itself. This property can easily be characterized by fixpoint theory. Without loss of generality, suppose the normal forms of a PPTL formula Q_k and its sub-formulas Q_l and Q_j as follows:

$$Q_k \equiv Q_{ke} \wedge \bigcirc Q_l, \quad Q_l \equiv Q_{le} \wedge \bigcirc Q_j \quad , Q_j \equiv Q_{je} \wedge \bigcirc Q_k$$

With reference to Algorithm CheckPPTL, we have:

$$B_{Q_k} = B_{Q_{ke}} \cdot R^{-1}(B_{Q_l}) \tag{1}$$

$$B_{Q_l} = B_{Q_{le}} \cdot R^{-1}(B_{Q_j}) \tag{2}$$

$$B_{Q_j} = B_{Q_{je}} \cdot R^{-1}(B_{Q_k}) \tag{3}$$

Substituting equation (2), (3) into equation (1)

$$B_{Q_k} = B_{Q_{ke}} \cdot R^{-1}(B_{Q_{le}} \cdot R^{-1}(B_{Q_{je}} \cdot R^{-1}(B_{Q_k}))) \tag{4}$$

From equation (4) we can infer that B_{Q_k} is the fix point of function $\tau(B) : \mathcal{L}(2^S) \longrightarrow \mathcal{L}(2^S)$, where $\mathcal{L}(2^S)$ denotes the boolean encoding of powerset over S

$$\tau(B) = B_{Q_{ke}} \cdot R^{-1}(B_{Q_{le}} \cdot R^{-1}(B_{Q_{je}} \cdot R^{-1}(B))) \tag{5}$$

The computation of B_{Q_k} can be done as follows: B is initially assigned with $B_{Q_{ke}}$ and $B^{'}$ with $\tau(B)$. Then $B := B^{'}$ and $B^{'} := \tau(B)$, where $\tau(B)$ is computed with the updated value of B. This iteration will not terminate until B equals to $\tau(B)$. Actually, B_{Q_k} is the final value of B when $B = \tau(B)$ occurs. Finally, we can figure up B_P by exerting logic "OR" operation on all the B_{P_i}, $B_P = B_{P_1} + B_{P_2} + \ldots + B_{P_n}$.

By Theorem 1, the model checking procedure can be performed in the following way: Firstly, we calculate $B_{\neg P}$, i.e. the boolean encoding of $Sat(\neg P)$, by invoking the procedure CheckPPTL $(\neg P, R)$. Secondly, if $B_{\neg P}$ equals to 0, then we can infer that the number of models in *M* over which $\neg P$ is satisfied is zero. Consequently, we have $M \models P$. Moreover, if $B_{\neg P} \neq 0$, then starts with any state in $Sat(\neg P)$, we can always find a path Π_{CE} of M as a witness to the fact that the system model M violates the property P.

Some explanations of our approach are given as follows: Firstly, as commencing with the transformation of the property formula into its normal form and then addressing each disjunct, our algorithm is linear in the size of normal form of the given PPTL formula. Secondly, with the semantic of "`next`" operation, the set of states where a PPTL formula preceded by "$\bigcirc$" (say $\bigcirc f$) holds can be thought as the preimage of that satisfying f via the transition relation. Therefore, our algorithm is sound. Finally, our method offers a viable solution to the problem (mentioned in (c)) which always occurs when considering a PPTL formula with infinite models. Hence, we can do model checking for PPTL formulas with both finite and infinite models.

B. An Example

As known, symbolic model checking is well suitable to the verification of digital system designs. In this subsection, we will employ a logic circuit implemented in Verilog [16] to illustrate how our method works.

The Verilog program in Fig. 5(a) describes a logical circuit for $3x + 1$ conjecture. Let x be an integer and the value of

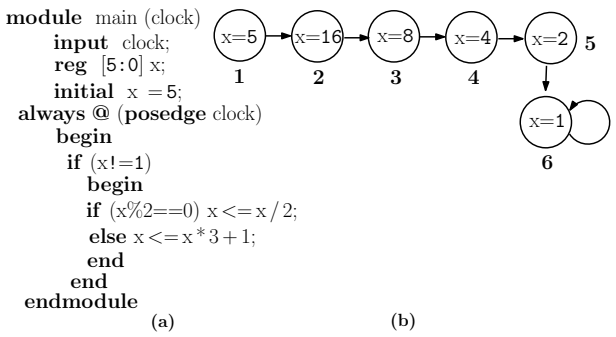
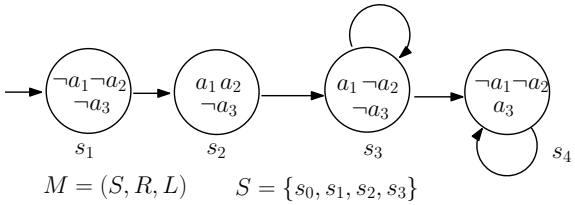


Figure 5. A Verilog program and its concrete model

x in the following clock period equals to $3x+1$ if x is odd and $x/2$ if x is even. With the semantics of synthesizable Verilog [18], "`reg`" could be thought as state variables while "`initial`" and "`always`" blocks define the initial state and the next state function respectively. Hence, the above Verilog program can be modeled as a state transition system in Fig. 5(b). Here, we define three propositions a_1, a_2 and a_3, where "a_1: the value of x in the current state is even", "a_2: the value of x in the current state is that of the previous state multiplies 3 and plus 1" and "a_3 : the value of x in the current state is 1". Intuitively, in the concrete model "$\neg a_1 \wedge \neg a_2 \wedge \neg a_3$" holds in states "1", "$a_1 \wedge a_2$" holds in state "2", "$a_1$" holds in state "$3, 4, 5$" and "$a_3$" holds in state "6". Though failed to be specified by CTL and LTL, this property can easily be expressed in PPTL formula $P_1; P_2; (P_3)^+; P_4$, namely P, by considering $P_1 = \neg a_1 \wedge \neg a_2 \wedge \neg a_3$, $P_2 = a_1 \wedge a_2 \wedge \neg a_3$, $P_3 = a_1 \wedge \neg a_2 \wedge \neg a_3$ and $P_4 = \neg a_1 \wedge \neg a_2 \wedge a_3$. We wish to formally prove that P dose not hold along all paths in the concrete model. To this end, a corresponding abstract model (Fig .6) defined over $\{a_1, a_2, a_3\}$ is given by applying the encoding method mention in [14].



$M = (S, R, L) \qquad S = \{s_0, s_1, s_2, s_3\}$
$R = \{(s_1, s_2), (s_2, s_3), (s_3, s_3), (s_3, s_4), (s_4, s_4)\}$
$L(s_1) = \phi \quad L(s_2) = \{a_1, a_2\} \; L(s_3) = \{a_1\} \; L(s_4) = \{a_3\}$

Figure 6. Abstract model of the Verilog program

Let $\mathcal{L}$ denote the encoding function mapping each state $s \in S$ to a distinct 3-bit boolean vector, $\mathcal{L} : S \longrightarrow \{0,1\}^3$. With the method mentioned in the last section for symbolically representing Kripke structure, we have $\mathcal{L}(s_1) = 000$, $\mathcal{L}(s_2) = 110$, $\mathcal{L}(s_3) = 100$ and $\mathcal{L}(s_4) = 001$. Then the characteristic function of R over $\{x_1, x_2, x_3\} \cup \{x'_1, x'_2, x'_3\}$ can be derived accordingly:

$$\mathcal{C}_R = \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,x'_1\,x'_2\,\overline{x'_3} + x_1\,x_2\,\overline{x_3}\,x'_1\,\overline{x'_2}\,\overline{x'_3} + x_1\,\overline{x_2}\,\overline{x_3}\,x'_1\,\overline{x'_2}\,\overline{x'_3} + \overline{x_1}\,\overline{x_2}\,x_3\,x'_1\,\overline{x'_2}\,x'_3$$

For the purpose of determining the set of states satisfying $P \equiv P_1; P_2; (P_3)^+; P_4$, $\neg P$ is rewritten to its normal form:

$$\neg P \equiv \neg(P_1 \wedge P_2 \wedge P_3 \wedge P_4) \wedge \varepsilon \bigvee P_3 \wedge P_2 \wedge P_1 \wedge \neg P_4 \wedge \bigcirc f_3 \bigvee ((P_2 \wedge P_1 \wedge \neg P_3) \vee (P_2 \wedge P_1 \wedge \neg P_4 \wedge \neg P_3)) \wedge \bigcirc f_2 \bigvee ((P_1 \wedge \neg P_2) \vee (P_1 \wedge \neg P_4 \wedge \neg P_2) \vee (P_1 \wedge \neg P_3 \wedge \neg P_2) \vee (P_1 \wedge \neg P_4 \wedge \neg P_3 \wedge \neg P_2)) \wedge \bigcirc f_1 \bigvee ((\neg P_4 \wedge \neg P_1) \vee (\neg P_4 \wedge \neg P_3 \wedge \neg P_1) \vee (\neg P_3 \wedge \neg P_1) \vee (\neg P_4 \wedge \neg P_2 \wedge \neg P_1) \vee (\neg P_4 \wedge \neg P_3 \wedge \neg P_2 \wedge \neg P_1) \vee (\neg P_3 \wedge \neg P_2 \wedge \neg P_1) \vee (\neg P_2 \wedge \neg P_1) \vee \neg P_1) \wedge \bigcirc true$$

where

$$f_1 \equiv \Box\neg(P_2; P_3^+; P_4) \qquad f_2 \equiv f_1 \wedge \Box\neg(P_3^+; P_4)$$
$$f_3 \equiv \neg(P_3^*; P_4) \wedge \Box\neg(P_3^*; P_4) \wedge f_2 \qquad f_4 \equiv \Box\neg(P_3^*; P_4) \wedge f_2$$

They can also be transformed into their normal forms as follows:

$$f_1 \equiv \neg(P_2 \wedge P_3 \wedge P_4) \wedge \varepsilon \bigvee P_3 \wedge P_2 \wedge \neg P_4 \wedge \bigcirc f_3 \bigvee ((P_2 \wedge \neg P_3) \vee (P_2 \wedge \neg P_4 \wedge \neg P_3)) \wedge \bigcirc f_2 \bigvee ((\neg P_4 \wedge \neg P_2) \vee (\neg P_4 \wedge \neg P_3 \wedge \neg P_2) \vee (\neg P_3 \wedge \neg P_2) \vee \neg P_2)) \wedge \bigcirc f_1$$

$$f_2 \equiv (\neg P_4 \vee (\neg P_4 \wedge \neg P_3) \vee \neg P_3 \vee (\neg P_4 \wedge \neg P_2) \vee (\neg P_3 \wedge \neg P_2)) \wedge \varepsilon \bigvee ((P_2 \wedge \neg P_3) \vee (P_2 \wedge \neg P_3 \wedge \neg P_4) \vee (\neg P_2 \wedge \neg P_3 \wedge \neg P_4) \vee (\neg P_2 \wedge \neg P_3)) \wedge \bigcirc f_2 \bigvee ((P_3 \wedge P_2 \wedge \neg P_4) \vee (P_3 \wedge \neg P_4 \wedge \neg P_2)) \wedge \bigcirc f_3$$

$$f_3 \equiv \neg(P_4 \vee (\neg P_4 \wedge \neg P_3) \vee (\neg P_4 \wedge \neg P_2) \vee (\neg P_4 \wedge \neg P_3 \wedge \neg P_2)) \wedge \varepsilon \bigvee ((P_3 \wedge P_2 \wedge \neg P_4) \vee (P_3 \wedge \neg P_4 \wedge \neg P_2)) \bigcirc f_3 \bigvee ((P_2 \wedge \neg P_3 \wedge \neg P_4) \vee (\neg P_2 \wedge \neg P_3 \wedge \neg P_4)) \wedge \bigcirc f_4$$

$$f_4 \equiv \neg(P_4 \vee (\neg P_4 \wedge \neg P_3) \vee (\neg P_4 \wedge \neg P_2) \vee (\neg P_4 \wedge \neg P_3 \wedge \neg P_2)) \wedge \varepsilon \bigvee ((P_3 \wedge P_2 \wedge \neg P_4) \vee (P_3 \wedge \neg P_4 \wedge \neg P_2)) \bigcirc f_3 \bigvee ((P_2 \wedge \neg P_3 \wedge \neg P_4) \vee (\neg P_2 \wedge \neg P_3 \wedge \neg P_4)) \wedge \bigcirc f_4$$

Whether or not $M \models P$ can be checked as follows:

(1) With respect to Algorithm CheckPPTL, we have

$$B_{\neg P} = B_{\neg(P_1 \wedge P_2 \wedge P_3 \wedge P_4)} + B_{\neg(P_1 \wedge P_2 \wedge P_3 \wedge \neg P_4)} \cdot R^{-1}(B_{f_3}) + B_{P_1 \wedge P_2 \wedge \neg P_3} \cdot R^{-1}(B_{f_2}) + B_{P_1 \wedge \neg P_2} \cdot R^{-1}(B_{f_1}) + B_{\neg P_1} \cdot R^{-1}(1)$$

(2) Similarly, we have:

$$B_{f_1} = B_{\neg(P_2 \wedge P_3 \wedge P_4)} + B_{P_2 \wedge P_3 \wedge \neg P_4} \cdot R^{-1}(B_{f_3}) + B_{P_2 \wedge \neg P_3} \cdot R^{-1}(B_{f_2}) + B_{\neg P_2} \cdot R^{-1}(B_{f_1})$$

$$B_{f_2} = B_{\neg(P_4 \wedge P_3)} + B_{\neg P_3} \cdot R^{-1}(B_{f_2}) + B_{P_3 \wedge \neg P_4} \cdot R^{-1}(B_{f_3})$$
$$B_{f_3} = B_{\neg P_4} + B_{P_3 \wedge \neg P_4} \cdot R^{-1}(B_{f_3}) + B_{\neg P_3 \wedge \neg P_4} \cdot R^{-1}(B_{f_4})$$

$$B_{f_4} = B_{\neg P_4} + B_{P_3 \wedge \neg P_4} \cdot R^{-1}(B_{f_3}) + B_{\neg P_3 \wedge \neg P_4} \cdot R^{-1}(B_{f_4})$$

(3) Apparently, with reference to case (c) in CheckPPTL, B_{f_3} is the fixpoint of function $\tau(B)$, where

$$\tau(B) = B_{\neg P_4} + B_{P_3 \wedge \neg P_4} \cdot R^{-1}(B) + B_{\neg P_3 \wedge \neg P_4} \cdot R^{-1}(B)$$

(4) By initially setting B as $B_{\neg P_4} = \overline{x_2}\,\overline{x_3} + x_1\,x_2\,\overline{x_3}$,

we can obtain $R^{-1}(B) = \overline{x_2}\,\overline{x_3} + x_1\,x_2\,\overline{x_3}$. Then with the procedure for computing fixpoints, we have $B_{f_3} = B_{f_4} = \overline{x_2}\,\overline{x_3} + x_1\,x_2\,\overline{x_3}$.

(5) Similarly, we have $B_{f_1} = B_{f_2} = \overline{x_1}\,\overline{x_2} + x_1\,\overline{x_3}$.

(6) Then with respect to (1) and the normal form of $\neg P$,

$$B_{\neg P} = \overline{x_1}\,\overline{x_2} + x_1\,\overline{x_3}.$$

(7) As the characteristic function for set $\{s_1, s_2, s_3, s_4\}$ is $\mathcal{C}_{\{s_1,s_2,s_3,s_4\}} = \overline{x_1}\,\overline{x_2} + x_1\,\overline{x_3}$, we can infer that $\neg P \equiv \neg(P_1; P_2; (P_3)^+; P_4)$ is true in initial state s_1, which means that there exits a path Π in the abstract model such that Π starts from s_1 and $\Pi \models \neg(P_1; P_2; (P_3)^+; P_4)$. For instance, $\Pi = \langle s_1, s_2, s_3^\omega \rangle \models \neg(P_1; P_2; (P_3)^+; P_4)$. Consequently, we have $M \not\models P$.

Moreover, P can only be satisfied by the paths starting from s_1, traveling through s_2 and s_3 in turn and ending with s_4, where the state transition $s_3 \rightarrow s_3$ occurs for finite times. For instance, $\Pi_f = \langle s_1, s_2, s_3, s_3, s_3, s_3 \rangle \models P_1; P_2; (P_3)^+; P_4$.

IV. Conclusion

In this paper, we have presented a symbolic model checking algorithm for PPTL. This enables us to verify properties of concurrent systems with PPTL, and alleviate the state space explosion problem encountered in automata-based model checking. In the future, we will implement our algorithm within SMV to support the verification with PPTL. Besides the symbolic model checker, it is necessary for us to investigate the complexity of the model checking algorithm for PPTL so that improvement for the algorithms could be made. Finally, as symbolic model checking is well suitable to the verification of digital system designs, we are also motivated to develop a practical environment to ensure the correctness of a system specified by Hardware Description Language (HDL), such as Verilog and VHDL [17], before behavior synthesis in the classical FPGA design flow.

Acknowledgement

The paper is supported by the NSFC Grant No. 61003078, 61133001, 91018010, 60873018, and 60910004, 973 Program Grant No. 2010CB328102, and ISN Lab Grant No. ISN1102001.

References

[1] Clarke, E. M., Grumberg, J. O., Peled, D. A.: Model Checking. MIT Press, 1999.

[2] Pnueli, A.: A temporal logic of concurrent programs. Theoritical Computer Science, pp. 45-60, 1981.

[3] Ben-Ari, M., Manna, Z., Pnueli, A.: The temporal logic of branching time. In *ACM Symp. Principles of Programming Languages*, pp. 164-176, 1981.

[4] Pnueli, A.: The temporal logic of programs. In: Proceedings of 18th IEEE symposium on foundations of computer science, pp. 46-57, 1977.

[5] Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, 2006.

[6] Duan, Z.: An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD thesis, University of Newcastle Upon Tyne, 1996.

[7] Duan, Z., Tian, C., Zhang, L.: A Decision Procedure for Propositional Projection Temporal Logic with Infinite Models. Acta Informatica 45(1), pp. 43-78, 2008.

[8] Tian, C., Duan, Z.: Propositional Projection Temporal Logic, Büchi Automata and ω-Regular Expressions. In proceedings of TAMC'08. pp. 47-48, Springer, 2008.

[9] Holzmann, G. J.: The Model Checker Spin. IEEE Trans. on Software Engineering 23(5), pp. 279-295, 1997.

[10] Tian, C., Duan, Z.: Model Checking Propositional Projection Temporal Logic Based on SPIN. In: Butler, M., Hinchey, M. G., Larrondo-Petrie, M. M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 246-265, Springer, 2007.

[11] Bryant, R. E.: Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 6(Aug), pp. 677-691, 1986.

[12] Moszkowski, B. C.: Reasoning about digital circuits. PhD Thesis, Department of Computer Science, Stanford University. TRSTAN-CS-83-970, 1983.

[13] Clarke, E. M., Long, D, E., McMillian, K. L.: Compositional model checking. In proceedings of the 4th Annual Symposium on Logic in Computer Science. IEEE Computer Society Press, Los Alamitos, Calif., pp. 46-51, 1989.

[14] Bryant, R. E.: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys (CSUR), vol.24, pp. 293-318, 1992.

[15] Burch, J. R., Clarke, E. M., McMillian, K. L.: Symbolic Model Checking: 10^{20} States and Beyond. Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 428-439, 1990.

[16] IEEE.: IEEE standard 1364-2001, Verilog HDL Reference Manual, 2001.

[17] IEEE.: IEEE standard 1076-1993, VHDL language Reference Manual, 1994.

[18] Clarke, E. M., Jain, H., Kroening, D.: Predicate Abstraction and Refinement Techniques for Verifying Verilog. Technical report, Carnegie Mellon University, CMU-CS-04-139, 2004

[19] Clarke, E. M., Grumberg, O., Long, D. E.: Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5): 1512-1542, 1994.

[20] Cimatti, A., Clarke, E. : NUSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer (STTT) Vol. 2, no. 5, pp. 410-425, 2000.

[21] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O. and Zue. Y.: Bounded Model Checking, volume 58 of Advances in computers. Academic Press, 2003.

[22] Cimatti, A., Clarke, E. M.: NuSMV2: An Open Source Tool for Symbolic Model Checking. Volume 2404-2002 of LNCS, pp 241-268, 2002.

[23] Wolper, P. L.: Temporal logic can be more expressive. Inf. Control 56, 72-99, 1983.

[24] Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y.: Symbolic model checking without BDDs. In TACAS99, volume 1579 of LNCS, pp 193-207. Springer, 1999.

[25] Duan, Z., Tian, C.: An Improved Decision Procedure for Propositional Projection Temporal Logic. ICFEM 2010. pp. 90-105, 2010.

[26] McMillan, K. L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht, 1993.

[27] Musuvathi, M., Qadeer, S.: CHESS: A Systematic Testing Tool for Concurrent Software. Microsoft Research Technical Report MSR-TR-2007-149, 2007.

[28] Beyer, D., Henzinger, T. A.: The Software Model Checker Blast: Applications to Software Engineering. Int. Journal on Software Tools for Technology Transfer, 9(5-6):505-525, 2007.