Symbolic Localization Reduction with Reconstruction Layering and Backtracking

Sharon Barner, Daniel Geist and Anna Gringauze

IBM Haifa Research Lab, Haifa Israel

Abstract. Localization reduction is an abstraction-refinement scheme for model checking which was introduced by Kurshan [12] as a means for tackling state explosion. It is completely automatic, but despite the work that has been done related to this scheme, it still suffers from computational complexity. In this paper we present algorithmic improvements to localization reduction that enabled us to overcome some of these problems. Namely, we present a new symbolic algorithm for path reconstruction including incremental refinement and backtracking. We have implemented these improvements and compared them to previous work on a large number of our industrial examples. In some cases the improvement was dramatic. Using these improvements we were able to verify circuits that we were not previously able to address.

1 Introduction

The directions in which contemporary research is tackling the state explosion problem are quite diversified. Structural model reductions are performed, various optimizations are published, and bounded model checking [4] is gaining momentum. Formal verification activity has also somewhat shifted from verification to falsification or as it is popularly called "bug hunting". This paper concentrates on a method which is called localization reduction or iterative abstraction refinement. The strength of this method is in verification and therefore it is complimentary to the methods such as bounded model checking and partial search whose main strength is in falsification.

Localization reduction with counterexample guided refinement was introduced by Kurshan [12]. Localization reduction is an iterative technique that starts with an abstraction of the model under verification and tries to verify the specification on this abstraction. When a counterexample is found a *reconstruction* process is executed to determine if it is a valid one. If the counterexample is found to be bogus (or spurious), then the abstract model is refined to eliminate the possibility of this counterexample in the next verification iteration. The details are described in Section 2.

Note that the localization technique either leaves a variable unchanged or replaces it by a non-deterministic abstraction. A similar approach is described by Balarin and Sangiovanni-Vincentelli [2]. Another refinement technique has been proposed by Lind-Nelson and Andersen [14]. They use under and over-approximation in order to handle all CTL. Their approximation techniques enable them to avoid rechecking the entire model after each refinement step while guaranteeing completeness.

There are abstraction-refinement techniques that do not use counterexamples to refine the abstraction [13, 16]. A technique proposed by Govindaraju and Dill [10] uses under approximation techniques and counterexamples to verify the specification. The refinement technique used there is different - it randomly chooses a concrete state corresponding to the first spurious state in the abstract counter example and tries to construct a real counterexample starting with the image of this state under the transition relation. Furthermore, the paper is limited to handling of safety properties and non-cyclic counterexamples.

D. Brinksma and K. G. Larsen (Eds.): CAV 2002, LNCS 2404, pp. 65-77, 2002.

A general abstraction method using a counterexample guided refinement was recently proposed by Clarke et al. in [7]. Wang et al. have reported the use of an ATPG solver for reconstruction [19]. Clarke et al. [9] have reported similar work where they used the GRASP [18] SAT solver to perform reconstruction. Gupta and Clarke have used spectral analysis to perform refinement[11]. The algorithms reported in this paper are implemented using BDDs but some of them can be implemented otherwise and can be used in conjunction with SAT or ATPG based implementations.

The steps of the abstraction refinement process described in this paper are no different than those described by Clarke et al. [7]. However an implementation of the methods described in Clarke et al. resulted for the most part with state explosion when they were attempted on our current industrial examples. We therefore improved the algorithms described by Clarke et al. and as result achieved dramatic improvements in some cases of real industrial examples.

Our improvements are described in Section 4. Most of the improvements were on path reconstruction and we also changed the way refinement is done. Since localization reduction is most effective when a property passes (as can be seen in the results in Section 5), it may seem that concentration on the refinement process would be more appropriate. However, since the refinement is guided by the counterexample, the reconstruction phase also calculates information that is crucial for the refinement phase. Therefore, improving reconstruction also improves refinement.

We have implemented the changes to localization reduction and used them on some of our designs. The new algorithms enabled us to verify circuits which we could not handle previously and in some cases the improvement was dramatic.

The rest of the paper is organized as follows. The next section describes localization reduction in detail. Section 3 defines the notation we use. Section 4 describes our algorithms. Section 5 details experimental results. We conclude in Section 6 with some suggestions for future work.

2 Overview of Localization Reduction Process

The process of localization reduction is depicted in Figure 1. Given a model M and an ACTL [8] formula φ where the model checking problem $M \models \varphi$, is too large for a model checker to handle, the localization reduction method works as follows: first a heuristic process is executed in order to obtain an abstract model M' of M such that $M \prec M'$ where $\prec$ is the simulation relation. Next, the model checking problem $M' \models \varphi$, is submitted to a model checker. Note that although M' may contain more behavior than M, it's structure and description are much simpler so the model checker is able to resolve the problem without reaching state explosion. The resolution may result in a "pass" or a "fail". In the case of a "pass" (i.e. $M' \models \varphi$ is true), the process can terminate because $M \prec M'$ and this implies that $M \models \varphi$. However, in the case of a "fail", the counter-example path, π' generated is valid for M' but may not have a corresponding execution path π in M. In this case it is necessary to validate that there is a corresponding path in M. This process is called "*reconstruction*". If a path π is successfully reconstructed then the process terminates. However, if reconstruction is not possible then π' is considered to be "spurious and the next iteration is started by heuristically

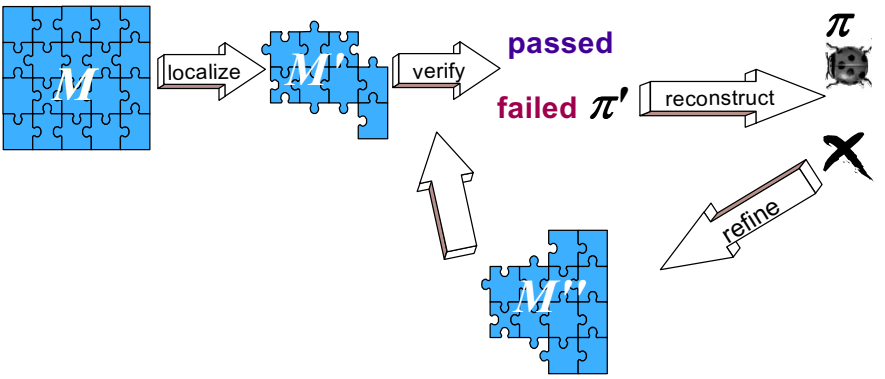

Fig. 1. The localization reduction process

refining and replacing M' with M'' such that $M \prec M'' \prec M'$. This process of iterative refinement continues until a "pass" is returned, or reconstruction of a "fail" is successful, or eventually state explosion is reached during model checking.

2.1 Improvements

Aside from state explosion in model checking, the abstraction refinement has two additional difficulties:

1. The reconstruction operation can in itself explode.
2. When reconstruction fails, the refinement operation is usually guided by trying to determine from the path π', what are the inconsistencies of this path and the model M. This is usually defined as a NP hard problem [7].

The methods of Clarke et al. explore all concrete paths which correspond to a certain abstract path starting from the initial states. In case there is no such concrete path they refine the model. Since these algorithms still suffer from state explosion we improved them in the following ways:

1. The path reconstruction was significantly changed:
   a. Incremental reconstruction: Instead of trying to reconstruct π' directly on model M, our algorithm performs successive reconstructions on model $M_i, i = 1 \ldots k$ where $M = M_k \prec \ldots M_1 \prec M$.
   b. Partial search and backtracking: Instead of trying to find all paths that correspond to π' the algorithm selects a subset of such paths and continues reconstruction. It is therefore possible to backtrack when a dead end is reached. The details are described in Section 4.2.
2. The refinement criterion is essentially the same as in Clarke et al. [7] however the method of computation takes advantage of the fact that we are handling a restricted class of abstractions (as described next).

We chose to restrict the class of abstractions we support to a subset of those supported in Clarke et al. While Clarke et al. support very general abstraction we confined ourselves to variable projection because with the general abstraction you need more refinement iterations and the benefit is small. Overall, our new method is entirely auto-

matic and symbolic. However, some of the improvements we report can also be applied to the more general class of abstractions.

3 Background and Definitions

A model M has finite set of variables $V = \{v_1, \ldots, v_n\}$. Each variable in V is called a *state variable*. Each state variable v_i can be equal to one of a finite set of values D_i. A particular value assignment to the variables $\{v_1, \ldots, v_n\}$ defines a state in M. *Expressions* are built from variables in V, constants from D_i, and function symbols (e.g. $v_1 + 1 + (v_2 / v_5)$ or v_{16}). *Atomic formulas* are constructed from expressions and relation symbols (e.g. $v_1 + 1 + (v_2 / v_5) = 12$ or $v_{16} < 20$). The set of all atomic formulas are called *Atoms(M)*. The *Support* of an atomic formula *f* is the set of state variable $V' \subseteq V$ that explicitly appear in *f* (e.g. $Support(v_1 + 1 + (v_2 / v_5) = 12) = \{v_1, v_2, v_5\}$, $Support(v_{16} < 20) = \{v_{16}\}$).

A model M can be defined by a program written in SMV [15] and translated into a *Kripke structure* $K = (S, I, R, L)$ Where $S = D_1 \times \ldots \times D_n$, is a set of states, $I \subseteq S$, is a set of Initial states $R \subseteq S \times S$, is a transition relation and $L : S \rightarrow 2^{Atoms(M)}$ is a labeling of the states in *S* given by $L(s) = \{ f \in Atoms(M) \mid s \models f \}$. Based on the Kripke structure K of M, formulas of the ACTL temporal logic can be constructed and evaluated (i.e. model checked). For a detailed definition of ACTL see [8].

Localization reduction involves abstraction of the model *M* (and its associated structure K). There is more than one way to obtain an abstract model of M. We now describe the type of abstraction used in this paper.

Definition 1 *(State projection) Given a state* $s = (v_1, \ldots, v_n)$ *and a subset of the state variables* $V' \subseteq V$ *where*

1. $m = |V'|$.
2. *Let* $i_1, \ldots, i_m$ *be the indices in increasing order of the state variables that belong to* V', *then* $v'_1 = v_{i_1}, v'_2 = v_{i_2}, \ldots, v_m = v'_{i_m}$.

The state projection of s on V' *is the m-tuple* $s' = (v_1', \ldots, v_m')$ *which satisfies* $s' = \exists (V / V') s(V)$ *and denoted by* $proj(s, V')$.

Intuitively, the state s is projected onto the coordinates of the variables contained in V'.

Definition 2 *(Set projection) Given a set of states* S *and a set of state variables* V', *The set projection* S' *of the set* S *on* V' *is defined as* $S' = \{proj(s, V') | s \in S\}$.

Definition 3 *(Model projection) Given a Kripke structure* K *which represents a model* M *and a subset of the state variables* $V' \subseteq V$ *we define a model projection* K' *of* K *with respect to* V' *which represents a model* M' *as follows:*

1. V' *is the set of state variables of* M'.
2. $K' = (S', I', R', L')$ *where:*

   $S' = proj(S, V')$.

   $I' = proj(I, V')$.

   $R' = \{(s'_1, s'_2) | s'_1 = proj(s_1, V'), s'_2 = proj(s_2), (s_1, s_2) \in R\}$.

   $L'(s') = \{f \in Atoms(M') \mid s' \models f\}$.

where $Atoms(M')$ is the subset of $Atoms(M)$ where only variables that belong to V' appear.

Note that its easy to see that $M \prec M'$ by definition of model projection, and that model projection defines a family of model abstractions for M that has a partial order with respect to $\prec$.

From here on and throughout the paper, the term *abstraction* will mean projection type of abstraction. The choice of projection as an abstraction is intuitive when working with Binary Decision Diagrams (BDDs). Projection of a set can be calculated by existential quantification which is a standard operation of BDD packages [6].

Definition 4 *(Path projection) The projection of path* $\pi = \{s_0, s_1, \ldots, s_k\}$ *on a set of variables* V' *is a set of paths* $\Pi' = \{S_0', S_1', \ldots, S_k'\}$, *where for all i,* $S_i' = proj(s_i, V')$. *We denote* $\Pi_i' \equiv S_i'$.

Note that a projection of a path has a set of states as path elements, so it corresponds to a set of paths in the original model. We make this distinction by denoting a set of paths (and its elements) with capital letter.

Since the Kripke structures we handle are derived from hardware implementations the transition relations obtained can be partitioned according to the state variables as follows: $R = \bigcap R_i$ where $R_i \subseteq S_i \times V_i'$, S_i is a projection of S onto some $Supp_i \subseteq V$ and V_i' is a projection of S' onto $\{v_i'\}$. We call $Supp_i$ the *support* of variable v_i. Intuitively the next value of each state variable in the model is independent of the next values of other variables of the model and the support is the state variables which appear in the atomic formula that describes the next state behavior (in practice some variables may be optimized out if the formula is equivalent to one that does not contain them). The support induces a graph on the state variables, where each variable is a node and there is an edge between v_i and v_j if $v_j \in Supp_i$.

The Support of a set of state variables is defined to be the union of the supports of the variables belonging to the set.

We are now ready to describe the abstraction refinement algorithms we improved.

4 The Abstraction Refinement Process

As mentioned in Section 1 our abstraction-refinement process follows the same steps as described in Clarke et al. We now describe the differences that make it more practical.

4.1 Initial Abstraction

Given a Model M and a model checking problem $M \models \varphi$, our initial abstraction M' is obtained by projection of M onto the set of state variables that are in the Support of the atomic formulas of φ. Then we utilize a model checker to resolve $M' \models \varphi$. In the case where a false answer is returned, we continue the process with reconstruction.

4.2 Trace Reconstruction

4.2.1 Reconstructing a Finite Path

Given a path Π' in the abstract model M', the purpose of trace reconstruction is to find a path π of M such that Π' is a projection of π.

Let M be the original model and $V = \{v_0, v_1, \ldots, v_n\}$ denote the original set of variables. Assume that we have model M' which is projection of the original model on the

```
reconstruct (Π′, M′, M ) {
    Π^0 = Π′ ;
    i = 1 ;
    U:=variables_of(M′) ;
    while ( (U ≠ ∅) ∧ (Π^i ≠ ∅) ) {
        U := choose_n_variables(support(U)/U) ∪ U ;
        M′′ := project_model(M, U );
        Π^(i+1) := reconstruct_one_layer(Π^i, M′, M′′ );
        if(Π^(i+1) = ∅ ) return ∅ ; /*refinement needed */
        i = i + 1 ;
    }
    return choose_one_counter_example(Π^i );
}
```

Fig. 2. Layered reconstruction

set of variables $V' = \{v'_0, v'_1, \ldots, v'_m\}$, where $m \le n$. Let also $\Pi = \{S_0, S_1, \ldots, S_k\}$ be the counter example for the original formula in the projected model.

Generally, the counter example (path) reconstruction algorithm analyzes the reachable state space for the variables in V/V', where all search steps are performed inside $\bigcup_{0 \le i \le k} S_i$. In [7], the algorithm iteratively performs consequent image computations using the model M, intersecting each *i*-th step with S_i before the next image computation, till no further step is possible or the end is reached. In the former case we have a spurious counter example, and in the latter case the counter example is proved real. However, this algorithm often leads to state space explosion, because in many practical cases $|V/V'|$ is significantly greater than $|V'|$.

In this section we present some techniques to overcome the state space explosion problem. First, we introduce the notion of *layer*

4.2.2 The Layering Reconstruction Algorithm

Definition 5 *layer.*

1. $U_0 = V'$ is a layer.
2. Any set $U_i \subseteq support\left(\bigcup_{0 \le k \le i-1} U_k\right) / \left(\bigcup_{0 \le k \le i-1} U_k\right)$ is a layer.

With the notion of layer we can divide the variable dependency graph into disjoint sets of variables (or layers), such that each variable in the layer is in the direct dependency of some variable(s) in one of the previous layers.

We divide the set V/V' into layers, and perform the layer reconstruction algorithm below iteratively, each time by computing an additional layer of variables and adding it to the abstract model. On the *i*-th iteration, the partially reconstructed counter example $\Pi^i = \{S_0^i, S_1^i, \ldots, S_k^i\}$ is produced.

Let Π be a path in the abstract model. Our algorithm for the path reconstruction is shown on Figure 2. The function **reconstruct** accepts an abstract path Π', an abstract model M' and a concrete model M. The algorithm performs layer by layer reconstruction iteratively, each time reconstructing one more layer, till all the variables are reconstructed ($U = \varnothing$)or no further reconstruction is possible and the refinement is done ($\Pi^i = \varnothing$). For layer computation (**choose_n_variables**), the next layer can be chosen

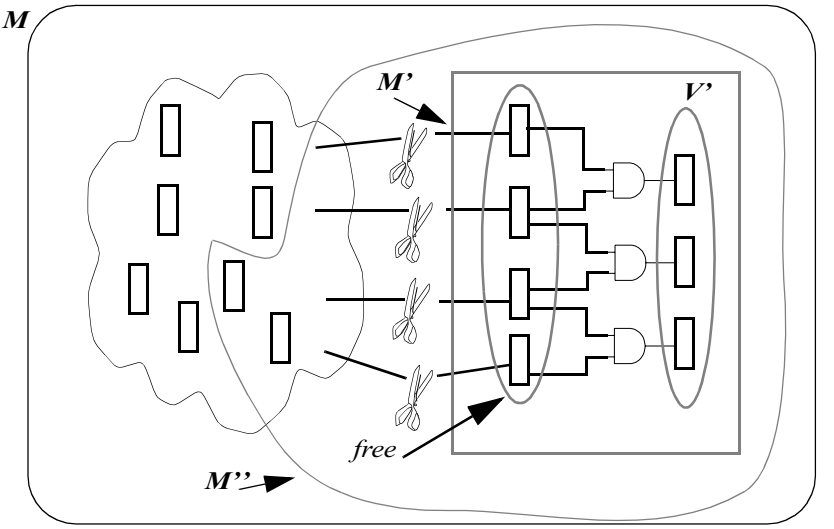

Fig. 3. The Abstract Model Structure

to be the entire support of the previous layer. However, on the first iteration of the loop we give priority to the support of the variables that were added to the model in the last refinement stage since they are the most suspect as ones that will force another refinement. In further iterations we take the support of the entire set of variables in the current model.

The advantage of layered reconstruction is twofold: first, we only reconstruct a few variables at a time while the rest are restricted to a very small subset. This maintains the state space we work on very small and avoid state explosion. Secondly, we can detect that a refinement is required early, many iterations before we actually obtain the entire concrete model.

4.2.3 Reconstructing one Layer

Reconstruction of one layer is different depending if Π' contains a loop or not. In the case of a simple path (no loop) we have implemented two algorithms. As explained above, the straightforward algorithm described in [7] suffers from state explosion. Both our algorithms try to alleviate that problem. The first one is faster and in usually good enough but when it fails due to the fact that too many variables are added back to the model, we apply our second algorithm which employs backtracking.

4.2.3.1 Reconstructing without a Loop (Algorithm 1)

The function **reconstruct_one_layer_no_loop1** accepts an abstract path Π' an abstract model M' and an intermediate model M''. We first try to reconstruct the path for model M''. Note that we do not care what the values of those new variables will be at each point of the reconstructed path except that the path has to be valid in M'' and its projection on the abstract model should be Π'. We therefore iteratively perform forward steps starting from I'', the initial set of M'' and conjunct each step with the corresponding step in Π'. This is no different so far than what is done in [7]. However, if we reach a dead end (i.e. the conjunction becomes empty) instead of proceeding to refine the model we try to modify Π' to be consistent with M''.

```
reconstruct_one_layer_no_loop1 (Π′, M′, M′′ ) {
    last = |Π′| – 1 ;
    s := S_0 ;
    i := 0;
    V′ := variables_of(M′) ;
    while (i < last) {
        next_s := S_{i+1} ;
        new := image(s, M′′) ∩ next_s ;
        if( new = ∅ ) {
            /* try to see if the selection of values in V′′/V′ can be changed */
            suspect := preimage(project(next_s, V′ ), M′ ) ;
            if ( (suspect ∩ s) ≠ ∅ ) {
                /* check whether we can replace S_{i+1} */
                new := image(s, M′′) ∩ project(next_s, V′) ;
                if( new = ∅ )
                    return ∅ ; /*refinement needed */
                else /* replace S_{i+1} */
                    S_{i+1} := new;
            }
            else
                return ∅ ; /*refinement needed */
        }
        i := i+1;
    }
    return Π ;
}
```

Fig. 4. The first reconstruction algorithm (without a loop)

The choice of the specific Π' is arbitrary and is done mainly to avoid state explosion. Thus, we can modify it during reconstruction. Note that in addition to values of variables in V', Π' also contains values for the Support of V'. These variables are "cut" from their behavior logic and have completely free (nondetermimistic) behavior as shown in Figure 3. One can change the values of these free variables in Π' as long as this change is consistent with M'' (and therefore, also M'). The resulting path will still be a valid counter-example of the formula φ. Figure 4 details the algorithm: whenever Π' is found to be inconsistent with M'', we project the offending state onto V' and perform a preimage computation. That is, we preserve the values of V' and discard the other values.We try to find other values for V''/V' in order to make the state consistent with the model. Notice that we avoid preimage computations of M'' which tends to explode.

In most cases the algorithm presented in this section gives good results. However, in some of the hard cases we add backtracking to reconstruction.

4.2.3.2 Reconstructing without a Loop (Algorithm 2)

Layering is not sufficient to avoid explosion because at each iteration, the Π^i counter example is getting larger (because all the possible counter examples which comply with Π^{i-1} are searched). However, only one such counter example would suffice, and may be found in less effort than all of them. In order to exploit this, we combine under-approximation of a partially reconstructed counter example with backtracking. The

```
reconstruct_one_layer_no_loop2( Π′, M′, M′′ ) {
    last = |Π′| – 1 ;
    for(i := 0 to last -1) S_i^all = ∅ ;
    V′ := variables_of(M′) ;
    S_0 := I′′ ∩ project(S_0, V′) ;
    i = 0 ;
    while ( i ≥ 0 and i < last ){
        prev := project(S_i, V′ );
        new := project(S_{i+1}, V′ );
        step := preimage(new, M′′) ;
        if (step ∩ S_i ≠ ∅ ) /* step forward */ {
            S_{i+1} = image(step, M′′) ∩ S_{i+1} ;
            S_{i+1} = subset(S_{i+1}) ;
            i = i + 1 ;
        }
        else if ((step ∧ prev) ≠ ∅ ) /* backtracking */ {
            S_i = (preimage(S_{i+1}, M′′) ∩ prev)/S_i^all ;
            if (S_i = ∅ ) return ∅ ;
            S_i = subset(S_i) ;
            S_i^all = S_i^all ∪ S_i ;
            i = i – 1 ;
        }
        else return ∅ ; / *refinement is needed */
    }
    return Π ;
}
```

Fig. 5. Reconstruction with backtracking

basic idea is, when reconstructing one layer of variables, to use subsets of real forward steps (image computations), as long as possible, and try backtracking when a dead end is reached.

Figure 5 shows the function **reconstruct_one_layer_no_loop2** for reconstruction with backtracking. The algorithm performs forward and backward steps (image and preimage computations) as long as possible. The forward step is done if the next step of the path is consistent with the forward computation using the model M'', till the trace is reconstructed and the bad states reached, or no forward step intersects the next state. In the latter case, backtracking is done - the algorithm performs backward steps and chooses different behavior for V''/V' till the forward step is again possible or a backward step cannot be performed anymore. This can happen if we checked all the possible values of V''/V' for the current state, or there is no backward step from the current state that intersects with the previous state. Note that we also employ underapproximation by doing subsetting [17] to avoid state explosion. In our experiments, we used subsetting to reduce the BDD size down to 5000 BDD nodes.

4.2.3.3 Reconstruction of a path with a Loop

To reconstruct an abstract path containing a loop (due to a liveness formula) we have to additionally ensure that the concrete path contains one as well. The concrete model sometimes contains some variable which acts like a “counter”. That is, it changes cyclically independent of the abstract model and it is the root cause of our failure to

```
reconstruct_a_loop( Π′, M′, M′′ ) {
    old := TRUE;
    new := S_0 ;
    last = |Π′| – 1 ;
    while(((new ∩ old) ≠ ∅) ∧ (new ≠ old)) {
        i := 1;
        new := new ∩ old ;
        old := new;
        step := new;
        while(i <= last) {
            step := image(step, M′) ∩ S_i ;
            if(step = ∅)
                refine;
            else {
                S_i := step;
                i := i + 1 ;
            }
            new := step;
        }
    }
    if((new ∩ old) ≠ ∅ ) {
       S_0 := new;
       return extract_loop_trace( Π′, M′ );
    }
    else
       refine;
}
```

Fig. 6. Reconstruction with loop

find a loop using the naive approach. However, if we look for a concrete path that includes traversal of the abstract loop a few times, then we may succeed in finding one which contains a concrete loop. It is possible to try a naive approach using one of the algorithms in Section 4.2.3.1 or 4.2.3.2 and then test if the reconstructed path contains a loop and if not proceed to refinement. However, in some cases we have found that this leads to refinement right away although it is possible to find a loop using the abstract path. In [7], the approach taken was to unwind the loop a sufficient number of times and then reconstruct it. Our approach was to implement a fixed point algorithm.

The algorithm depicted in Figure 6 describes how this is done. The input path Π' is assumed to be the abstract loop without the tail leading to it from the initial state. The algorithm performs a forward fixed point algorithm to find a loop with length that is a product of $|\Pi'| - 1$. On termination of the fixed point, if $(\text{new} \cap \text{old}) \neq \varnothing$, a concrete loop exists. However, its possible that not all states in S_0 are on a loop and therefore **extract_loop_trace** chooses an arbitrary state from S_0 and iteratively performs backward steps from it until some state in S_0 is encountered more than once. This is similar to the algorithm described in [5] by Biere et al. that proves a tableau construction by showing how a concrete path can be constructed from path with a loop that satisfies the tableau.

```
refine( S_i, S_{i+1}, M', M'' ) {
    V' := variables_of(M') ;
    prev := project(S_i, V') ;
    D := differ(image(prev, M''), S_{i+1}) ;
    if( D ⊆ V ) {
        new := project(S_{i+1}, V') ;
        D := differ(preimage(new, M''), S_i) ;
    }
    return add_to_model(M' ,D );
}
```

Fig. 7. Refinement

4.3 Refinement

When reconstruction fails, it's necessary to refine the model. In the family of abstractions we use, this means adding back state variables from the original model that were eliminated in the previous abstraction. The decision which subset of variables to add back can be formulated as an NP-complete problem [7]. We need to find a small set of state variables that don't belong to the set V', for which values cannot be found such that a path π can be reconstructed from Π'. We use the first element of Π' that cannot be reconstructed (S_{i+1}) and its preceding element (S_i) and try to find for which variables in the set V''/V' we could not find valid values consistent with M''. We use a heuristic function $differ(A, B)$ where $(A \cap B = \varnothing)$ The function tries to find the minimal set of variables P s.t. $proj(A, P) \cap proj(B, P) = \varnothing$. The actual implementation of differ is heuristic. At first, $P = V''$. We randomly choose variables from V''/V' to eliminate until we no longer have $proj(A, P) \cap proj(B, P) = \varnothing$. Since the result of this algorithm depends on the order that we choose variables, we attempt this algorithm with different variable sequences according to a predefined number of attempts.

5 Results

The experimentation with the new algorithms was conducted using hard cases that we accumulated from industrial design groups that use our model checker for the purpose of their verification work. The cases were diverse: they were from different designs from different design groups, and had a significantly different number of state variables. Comparing to easy cases did not seem meaningful as obviously localization will not perform better on them because of the extra overhead. For example, we ran the texas 97 benchmarks [1] but most of them completed in less than a second without localization reduction. The tests were all done on a 375Mhz IBM pSereis 640 with a PowerPC3-2 processor and a 4MB L2 cache and 1G of memory.

The results are divided into two tables of safety and liveness. The first column describes the type of design the example was taken from. The second column gives the number of state variables in the examples. The third column details wether the property passed or failed. The rest of the columns detail the results of the different algorithms run on the example - giving run time (sec) and memory requirement. A "Memory" entry means that the run reached the limit of 1G. Some of these examples were ones that we could not verify even with 2G of memory which is the current limit

of a 32 bit application on the IBM pSereis 640. We run all algorithms using dynamic BDD reordering.

In Table 1 we compared 4 algorithms. All of the algorithms performed On-The-Fly model checking [3]: without localization, with Clarke et al.'s algorithm, with Layering and the algorithm in Section 4.2.3.1, and with layering and the algorithm in Section 4.2.3.2.

Table 1: Results for safety formulas

Name	No. Vars	Verif. Result	Without Local.	Clarke et al.	Layer+ Alg1	Layer+ Alg2
Infiniband 1	396	passed	Memory	Memory	54s/43M	134s/47M
Infiniband 2	377	passed	Memory	0.95s/33M	4.32s/33M	4.19s/33M
Ethernet 1	86	passed	1601s/87M	657s/189M	243s/88M	287s/88M
CPU 1	123	passed	599s/92M	Memory	85s/99M	335s/93M
Queue CRM	79	passed	148s/45M	75s/42M	34s/41M	28s/41M
Ethernet 2	156	passed	Memory	14211s/185M	Memory	9.4s/31M
CPU 2	105	failed	595s/62M	N/A	405s/50M	229s/50M
CPU 3	167	failed	1943s/96M	Memory	28963s/ 192M	1096s/ 103M

In Table 2 we compared 3 algorithms: without localization. With localization and no fixed point, and with localization and the algorithm in Section 4.2.3.3.

The results in Table 1 indicate that when a safety property passes the improvement in time and memory requirements can be 2 orders of magnitude. This is due to the fact that the examples are verified using a much smaller model. When the property fails, it usually requires much more iterations of refinement and the results are either comparable to the result without localization or worse than it. There was only one case (Infiniband 2) where the algorithm of Clarke et al. did better - in this case there were no refinement iterations required, thus our improvements were unnecessary but even with the overhead they added the example ran much faster than without localization.

Table 2: Results for liveness formulas

Name	No. Vars	Verif. Result	Without Local.	Layer+ Naive	Layer+ Fixed Point
Infiniband 3	366	passed	Memory	7.87/33M	7.7s/33M
Ethernet 3	41	failed	1.57s/27M	10.9s/27M	8.9s/27M
CPU 4	66	passed	28s/132M	22s/99M	23s/98M
CPU 5	66	failed	35s/132M	534s/131M	349s/128M

For liveness (Table 2), we also observed an order of magnitude improvement in some cases (Infiniband 3 is an example) but the results were not as consistent as for safety. There were cases where the result was significantly better than without localization. Note also that the fixed point algorithm was not better than the naive approach (except for example CPU 5). This indicates that perhaps doing more refinement can be easier than trying to locate a concrete loop which traverses the abstract loop more than once.

6 Conclusions

We presented improvements to symbolic localization reduction that gave us dramatic improvements in the verification of some hard industrial examples. The improvements were mainly in the reconstruction process. The algorithms we presented were found to be most effective in the cases where the property is a safety property and passes. However, we have also shown that it can give orders of magnitude improvement on liveness formulas that pass. Thus this method is complimentary to the bounded model checking methods which generally work better when the property fails.

For future work we intend to investigate ways to improve counterexample reconstruction of liveness properties since our results in these cases are not as consistent as with safety. We also intend to combine some of our algorithms such as layering with the satisfiability based reconstruction techniques reported in [9,19] which we believe can further speed-up the results described here.

References

[1] The texas97 verification benchmarks. http://vlsi.colorado.edu/ vis/texas-97/.
[2] F. Balarin and A. Sangiovanni-Vincentelli. An iterative approach to language containment. In Computer Aided Verification, pages 29–40, 1993.
[3]I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of rctl formulas. In Computer Aided Verification, pages 184–194, 1998.
[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Proc. of TACAS, 1999.
[5] A. Biere, E. M. Clarke, and Y. Zhu. Multiple state and single state tableaux for combining local and global model checking. In Correct System Design, pages 163–179, 1999.
[6] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implentation of a BDD Package. In 27th ACM/IEEE Design Automation Conference, pages 40–45. ACM/IEEE, 1990.
[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu., H. Veith. Counterexample-guided abstraction refinement. In Computer Aided VerificationI, pages 154–169, 2000.
[8] E. M. Clarke, O. Grumberg, and D. Peled. MIT Press, 2000.
[9] E. M. Clarke, Y. Lu, P. Chauhan, and A. Gupta. Automatic abstraction by counterexample-guided refinement. Private Communication.
[10] S. G. Govindaraju and D. L. Dill. Verification by approximate forward and backward reachability. In Inter. Conf.on Computer Aided Design, 1998.
[11] A. Gupta and E. M. Clarke. Using fourier analysis for abstraction-refinement in model checking. Private Communication.
[12] R. P. Kurshan. Computer-Aided-Verification of Coordinating Processes. Princeton University Press, 1994.
[13] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for ctl model checking. In Inter. Conf.on Computer Aided Design, pages 76–81, 1999.
[14] J. Lind-Nielsen and H. Andersen. Stepwise ctl model checking of state/event systems. In Computer Aided Verification, pages 316–327, 1999.
[15] K. L. McMillan. The SMV System DRAFT. Carnegie Mellon University, Pittsburgh, PA, 1992.
[16] A. Pardo and G. Hachtel. Incremental ctl model checking using bdd subsetting. In IEEE DAC,, 1998.
[17] K. Ravi and F. Somenzi. High-density reachability analysis. In ICCAD, 1995.
[18] G. P. M. Silva and K. A. Sakallah. GRASP – a search algorithm for propositional satisfiability. IEEE Trans. on Computers, 44:506–516, 1999.
[19] D. Wang, P. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In 38th IEEE DAC, pages 35–40, 2001.