Abstraction Techniques for Word-level RTL Designs

MTP Stage I Report

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

by

Sukanya Basu
Roll No : 09305908

under the guidance of

Prof. Supratik Chakraborty

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

October 2011
Abstraction Techniques for Word-level RTL Designs

October 21, 2011
Contents

1 Introduction 2
   1.1 Motivation ........................................... 2
   1.2 An example ........................................... 2

2 Review of literature 5
   2.1 Abstraction of the RTL system ......................... 5
   2.2 Predicate abstraction ................................ 5
      2.2.1 Word-Level Transition Functions .................. 5
      2.2.2 SAT-based Predicate Abstraction ................. 6
      2.2.3 Predicate Clustering .............................. 6
      2.2.4 Support .......................................... 8
   2.3 Abstraction using linear arithmetic functions .......... 8
   2.4 Bitwidth reduction techniques ........................ 10

3 Counterexample guided abstraction refinement 13
   3.0.1 Generating the initial abstraction .................. 13
   3.0.2 Model checking the abstract model .................. 13
   3.0.3 Identification of spurious counterexample .......... 14
   3.0.4 Refining abstraction to eliminate spurious counterex-
         ample ............................................. 14

4 A case study 16

5 Future work and conclusion 17

Appendices 19

A ITC '99 Benchmark example used for case study 19
Chapter 1

Introduction

Most abstraction and refining techniques for verifying hardware designs use model checkers that operate at the gate level. These designs get flattened into the gate level circuit designs and a model checker then checks this model. However, RTL designs describe the system at a level of abstraction higher than the netlist. By flattening out to the netlist level these techniques fail to exploit the information available from the abstract RT level designs.

1.1 Motivation

Designs written in a high level language, like VHDL or Verilog, provide features like bit-vector arithmetic and concurrency. At the word level, data-path elements and data packets are viewed as entities in their own right as opposed to a group of bit-level signals without any special meaning. It would be nice if we can make use of the information available at the abstract level. Using the abstract model, the problem given to the model checker is made simpler. This will speed up the verification process. The focus of this project will be to explore abstraction techniques that work at the word-level.

1.2 An example

We look at the example of an industrial benchmark. This simple example will demonstrate why word-level modelling is beneficial over bit-level. The code is provided in the appendix. The table below shows the 13 different entries and their corresponding bit numbers.

The example is a VHDL program from ITC '99 testbenches. It has a memory array eight entries long, where each entry is 20 bits. The property that we want to check here is as follows. We take an 8 bit bitvector as the input; if this number is present in bit_vector(19 downto 12) or its complement is present in bit_vector(11 downto 4), bit_vector(3 downto 0) of that memory array entry is returned as output. If the input is present in
more than one array entry, bitwise OR operation is performed on the bits
\texttt{bit\_vector\_3\_downto\_0} of each of the array entries that it is present in,
and result of this OR operation is given as output.

Though this is the functionality performed by this code, it does so by
using bitwise operations on individual bits. It uses bitwise or, bitwise and,
and bitwise comparison. Each entry is 20 bits long. This gives rise to a
state space with $2^{20}$ possible states. Bit level analysis would have to reason
about all these $2^{20}$ states, which is a formidable task.

However, on a closer look, we can see that the bits in
\texttt{bit\_vector\_19\_downto\_12} and \texttt{bit\_vector\_11\_downto\_4} are handled together. i.e. the
bits in these chunks are treated uniformly. The entries in the array are
treated as chunks of data as \texttt{bit\_vector\_19\_downto\_12}, \texttt{bit\_vector\_11\_downto\_4} and \texttt{bit\_vector\_3\_downto\_0}.

Now, we are interested in the numbers present in the memory array, the
first chunk as is, the second chunk complemented. There are 13 different
values with some repetitions. Though the 8 bit numbers can have $2^8 = 64$
different values, we are only interested in 13 of those. The remaining 51 are
of no value to us.

These 51 values, each with their 4-bit output would mean $51 \times 2^4$ different
outputs which will never occur. Bit-level reasoning would still allow these
values which is unnecessary.

We are interested in 13 values out of the possible 64. It is obvious that
we do not need all 8 bits for distinguishing these 13 values. Hence we try
to uniquely represent the 13 entries with as few bits as possible. The easiest
scheme would be to select a subset of the 8 bits that seperates the 13 values.
We look at the 8 bits representing the 13 values. Careful observation shows
that, if we select bits $b_7, b_6, b_5, b_4, b_2$ and $b_0$ (from MSB to LSB) we can
uniquely identify the 13 entries.

<table>
<thead>
<tr>
<th></th>
<th>$b_7$</th>
<th>$b_6$</th>
<th>$b_5$</th>
<th>$b_4$</th>
<th>$b_3$</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
So, when we get an input, we check bits $b_7, b_6, b_5, b_4, b_2$ and $b_0$ and we can say, whether it is present in the memory array. However, the remaining 2 bits can still differ. So, we may get some false results. In that case we will have to look into the remaining 2 bits.

Selective slicing of 6 out of 8 bits can be seen as a crude way of abstracting the system, for this example. We would like to look into ways of generating abstractions that will take into account the functionality of the particular program. We will take up this example again in a later chapter to discuss the directionality of the project.
Chapter 2

Review of literature

2.1 Abstraction of the RTL system

The following techniques exist for word-level abstraction and refinement and have been studied as a part of this project.

- Predicate abstraction
- Abstraction using linear arithmetic functions
- Signal width reduction or bitwidth reduction techniques

We now look into details of each method.

2.2 Predicate abstraction

Predicate abstraction is a popular software verification technique. The RT level of hardware design languages is very similar to a software program. Predicate abstraction can be applied in verifying RT level HDL programs with little modification. This is the approach taken in [6].

A symbolic variable $b_i$ is associated with each predicate $\pi_i$. Each concrete state $\vec{r} = \langle r_1, ..., r_n \rangle$ maps to an abstract state $\vec{b} = \langle b_1, ..., b_k \rangle$, where $b_i = \pi_i(\vec{r})$. If the concrete machine makes a transition from state $\vec{r}$ to state $\vec{r}' = \langle r_1', ..., r_n' \rangle$, then the abstract machine makes a transition from state $\vec{b}'$ to $\vec{b}' = \langle b_1', ..., b_k' \rangle$, where $b_i' = \pi_i(\vec{r}')$. We refer to $\pi_i(\vec{r})$ as a current-state predicate and $\pi_i(\vec{r}')$ as a next-state predicate. For example, if $x = y$ is a current-state predicate, then the corresponding next-state predicate is $x' = y'$.

2.2.1 Word-Level Transition Functions

Let $R = \{r_1, ..., r_n\}$ denote the set of registers and external inputs in a given Verilog program. Let $S$ denote the set of states for a given Verilog program.
Let $Q \subseteq R$ denote the set of registers. Next-state function of a register $r_i \in Q$ is denoted by $f_i(r_1, ..., r_n)$ or $f_i(\bar{r})$, where $\bar{r} = \langle r_1, ..., r_n \rangle$. The value of $r_i$ in the next state is given by $f_i(\bar{r})$ as the function of the current state. The next-state functions are used to define the transition relation $R(\bar{r}, \bar{r}')$.

$$R(\bar{r}, \bar{r}'): \bigwedge_{r_i \in Q} (r'_i = f_i(\bar{r}))$$

### 2.2.2 SAT-based Predicate Abstraction

Often SAT solvers are used for abstracting ANSI-C programs, including bit-vector operators. Similar technique is used here to compute abstraction of Verilog programs. Abstract transition relation is computed using SAT:

A symbolic variable $b_i$ associated with each predicate $\pi_i$. $\pi_i(\bar{r})$ is referred to as the current-state predicate and $\pi_i(\bar{r}')$ as the next-state predicate. The formula that is passed to the SAT solver directly follows from the definition of the abstract transition relation. Suppose the SAT solver returns $\bar{r}, \bar{r}', \bar{b}, \bar{b}'$ as the satisfying assignment. We project out all variables but $\bar{b}$ and $\bar{b}'$ from this satisfying assignment to obtain one abstract transition $(\bar{b}, \bar{b}')$. Since we want all the abstract transitions, we add a blocking clause to the SAT equation that eliminates all satisfying assignments that assign the same values to $\bar{b}$ and $\bar{b}'$, and re-start the solver. This process is continued until the SAT formula becomes unsatisfiable. The disjunction of the abstract transitions obtained gives us the abstract transition relation.

### 2.2.3 Predicate Clustering

A single abstract transition relation computed using all available predicates has some drawbacks. Number of satisfying assignments can be exponential in number of predicates. The process can be very slow even for small number of predicates.

A probable solution not to aim at most precise abstract transition relation. An over-approximation of the relation is generated by an eager approach.

**Overview** Set of predicates and their next-state versions are grouped into smaller sets (clusters) of related predicates. Let them be denoted them by $C_1, ..., C_l$. An abstract transition relation is computed with respect to each cluster. The total number of abstract transition relations are conjoined to form $\hat{R}$.

$$\hat{R} := \bigwedge_{i=1}^l \hat{R}_i$$
Abstract transition relation with respect to a cluster $C_j$ are denoted as follows:

$$\tilde{R}_j := \exists \bar{r}, \bar{r}': \bigwedge_{\pi_i \in C_j} b_i \iff \pi_i(\bar{r}) \land R(\bar{r}, \bar{r}') \land \bigwedge_{\pi'_i \in C_j} b'_i \iff \pi_i(\bar{r}')$$

Satisfying assignments to the above equation correspond to abstract transition relation $\tilde{R}_j$.

Three different techniques for creating predicate clustering have been mentioned in [6]. These are cone clustering, clustering for lazy abstraction and semantic predicate clustering.

**Abstraction refinement**

When refining the abstract model, distinction is done between two cases of spurious behavior.

1. **Spurious transitions** are abstract transitions that do not have any corresponding concrete transitions. By definition, spurious transitions cannot appear in the most precise abstraction, which is computed by the eager approach. However, computing the most precise abstract model is expensive. Hence various predicate clustering techniques are used. This can result in many spurious transitions.

2. **Spurious prefixes** are prefixes of the abstract counterexample that do not have a corresponding concrete path. This happens when the set of predicates is not rich enough to capture the relevant behaviors of the concrete system, even for the most precise abstraction.

Given a spurious counterexample first it is checked if any transition in the counterexample is spurious. If a spurious transition is found, it is eliminated from the abstract model by adding a constraint to the abstract model. If no transition in the counterexample is spurious, then new predicates are generated in order to eliminate a spurious prefix in the counterexample. The entire spurious counterexample is treated as a spurious prefix and the shortest spurious prefix is not found.

1. **Detecting and Removing Spurious Transitions** An abstract transition from $\bar{s}$ to $\bar{t}$ is a spurious transition if there are no concrete states $\pi, \pi'$ such that $\pi$ is abstracted to $\bar{s}$, $\pi'$ is abstracted to $\bar{t}$, and there is a transition from $\bar{\pi}$ to $\bar{\pi'}$.

Formally, the abstract transition from $\bar{s}$ to $\bar{t}$ is spurious iff the following formula is unsatisfiable:

$$\beta(\bar{s}, \bar{\pi}) \land R(\bar{\pi}, \bar{\pi'}) \land \beta(\bar{t}, \bar{\pi'})$$
2. Detecting and Removing Spurious Prefixes

An abstract counterexample \( \bar{s}(0), \cdots, \bar{s}(l) \) of length \( l \) spurious prefix if there is no concrete execution of \( l \) transitions such that at each step the concrete state is consistent with the corresponding abstract state.

Here an approach is taken use of *weakest preconditions* as done in software verification. New word-level predicates are generated by computing the weakest precondition of the given property with respect to the transition function given by the RT-level circuit.

When the spurious counterexample is long, the weakest precondition computation becomes expensive and the predicates generated can become very complex. This adversely affects the abstraction refinement loop. In order to simplify the weakest preconditions, the guards in the weakest preconditions are substituted with their truth values. Furthermore, only atomic predicates occurring in the weakest preconditions are added as new predicates.

2.2.4 Support

The tool VCEGAR [7] implements this approach for Verilog programs. A user of the tool needs to provide the input program, property to check, and a few options. Given these inputs, the tool performs all the steps of the CEGAR loop automatically.

VCEGAR was used to check safety properties of Instruction Cache Unit and Instruction Cache RAM (ICRAM) of Sun PicoJava II microprocessor. It has also been applied to examples from the opencores (www.opencores.org), and the Texas97 and VIS benchmark suites.

2.3 Abstraction using linear arithmetic functions

The focus here is to abstract the word-level functionality of a component from its bit-level specification [5]. An algorithm is presented for abstraction of word-level linear functions from bit-level component descriptions.

The *BMD Data Structure

A data structure called Multiplicative Binary Moment Diagrams (*BMDs) [2] is used here. *BMDs represent functions having Boolean variables as arguments and numeric values as results. Their structure is similar to that of Ordered BDDs, but incorporate two novel features: they are based on a decomposition of a linear function in terms of its “moments”, and they have weights associated with their edges which are combined multiplicatively.

*BMDs provide a concise representation of functions defined over words of data.
The algorithm works in two phases.

- A *BMD representation of the component is created and a word-level function is hypothesized from it.
- The hypothesis is verified.

**Algorithm 1** Hypothesize(node: $n$)

Let $x[i]$ be the variable which labels node $n$. Let $l$ and $r$ denote the left and right children of $n$. Let $w_l$ and $w_r$ be the weights of the left and right edges respectively. $f_n$ denotes the word-level hypothesis at $n$. $\text{visited}[n]$ is a flag which is set when the node $n$ is first visited

1. If $\text{visited}[n]$ is true, return $f_n$.

2. Determine $f_r$, the word-level hypothesis at $r$:
   (a) If $r$ is a terminal node, then $f_r$ is the constant labeling $r$.
   (b) Otherwise, determine $f_r$ using Hypothesize($r$)

3. If $f_r$ contains any term involving $x$, then report that no word-level hypothesis exists and Exit.

4. Create a word-level function, $f'_r$, from $f_r$ by multiplying each term of $f_r$ by $2^{-i}xw_r$.

5. Determine $f_l$, the word-level hypothesis at $l$:
   (a) If $l$ is a terminal node, then $f_l$ is the constant labeling $l$.
   (b) Otherwise, determine $f_l$ using Hypothesize($l$)

6. Create a word-level function, $f'_l$, from $f_l$ by multiplying each term of $f_l$ by $w_l$.

7. If any term of $f'_r$ is inconsistent with any term of $f'_l$, report that no word-level hypothesis exists and Exit.

8. Create $f_n$ as the sum of each distinct product term from the set of terms of $f'_r$ and $f'_l$.

9. Set $\text{visited}[n]$ to True and Return $f_n$.

End.

**Phase 1: Hypothesis Creation** In the first phase, a *BMD representation for the output word is constructed for the output word of the component
and a word-level hypothesis is created from it. Intuitively, a hypothesis is a
guess of the word-level function.

It is shown that if the output word represents a word-level function, then
the hypothesis is the same function (that is, the guess is correct). However,
if the output word does not represent any word-level function, a hypothesis
may still be obtained. In this case the second phase of the algorithm is
required.

**Algorithm 2** Word-Level-Abstract(component: \( C \))

For each output word \( v \) of \( C \):

1. Create a *BMD for \( v \)
2. Use Algorithm Hypothesize to create a word-level hypothesis for \( v \)
3. If such a hypothesis, \( f \), exists for \( v \), then
   
   (a) Create a *BMD for \( f \).
   
   (b) Verify whether the *BMD for \( f \) is identical to the *BMD for \( v \).
       
       If they are identical then return the word-level hypothesis, \( f \).

4. Otherwise, report that \( v \) is not word-level linear.

End.

**Phase 2: Hypothesis Verification** In the second phase, it is verified
whether the hypothesis is correct. To do this, the *BMD for the hypothesis
is created and its equivalence is tested with the *BMD of the component.

### 2.4 Bitwidth reduction techniques

BooStER \([8]\) implements a new word-level abstraction technique. It is a
tool which operates as a pre and postprocessor for RTL property checking.
In a preprocessing step prior to the property checking, it takes the RTL
netlist and computes a scaled down RTL model of the design. In doing so,
each word-level signal \( x \) is replaced by a corresponding shrunken signal of
width \( m_x \leq n \), where \( n \) is the original width of \( x \), while still guaranteeing
that the property holds for the reduced RTL if and only if it holds for the
original RTL. This reduced RTL is given to the property checker instead of
the original RTL.

The internal bit-level representation computed from the reduced RTL
contains less variables than the one computed using the non-reduced RTL.
Depending on the degree of reduction, this difference might be significant.
If the property does not hold, the counterexample returned by the property checker is taken, which is a counterexample relating to the signals of the reduced RTL. A corresponding counterexample for the original RTL is computed, using information about the applied reduction, gathered during the preprocess.

Flowchart showing the working of BooStER [8].

**Approach**

BooStER reads an RTL representation of a design and a property and generates a system $E$ of equations over a theory of fixed-size bitvectors (extension of [4]). This theory features high-level operators like bitwise Boolean operations, arithmetics and if-then-else, and allows complete RTL designs to be modeled. $E$ is satisfiable if and only if the property does not hold for the RTL.

Word-level signals in the RTL correspond to bitvector variables in $E$. Thus the information, which bits belong to the same signal, is preserved. A
satisfying solution of $E$ yields a counterexample for the RTL. For each bitve-
tor variable occurring in $E$ the smallest possible number of bits is computed,
such that a second system $E'$ of bitvector equations, which differs from $E$
solely in the manner that variable widths are shrunken to these computed
numbers, is satisfiable if and only if $E$ is satisfiable. $E'$ is generated using
these minimum signal widths and then retranslated into a netlist, which is
output by the tool and represents a scaled down version of the original RTL.

The process of scaling down signal widths is separated into two phases.
Structural and functional dependencies are imposed on the bitvector vari-
ables by the high-level operators occurring in the equations of $E$. Thereby,
variables typically have non-uniform data dependencies. Different depen-
dencies exist for different chunks of a signal. The method here analyzes
such dependencies.

Another approach taken for scaling down is given in [1].

For each variable, contiguous parts are determined in which all bits are
treated uniformly in the exact same manner with respect to data dependen-
cies. Such decomposition of a variable into a sequence of chunks is called a
granularity. For each such chunk of a signal, the necessary minimum width is
computed, as required by dynamical data dependencies. According to these
computed minimum chunk widths, the reduced width for the corresponding
shrunken signal is reassembled.
Chapter 3

Counterexample guided abstraction refinement

The basic idea of abstraction refinement techniques is to create a new abstract model that contains more details in effect to prevent the spurious counterexample. This process is repeated until the property is either proved or disproved. It is known as the Counterexample Guided Abstraction Refinement framework, or CEGAR for short [3].

The framework used here is that of existential abstraction. Existential abstraction computes an over-approximation of the original model. Thus when a specification of the temporal logic ACTL* is true in the abstract model, it will also be true for the concrete model.

3.0.1 Generating the initial abstraction

Generating the initial abstraction consists of the following basic steps:

- Extracting the atomic formulas
- Grouping formulas into formula clusters
- Generating abstraction for each cluster

3.0.2 Model checking the abstract model

Given a generated abstraction function $h$, $M'h$ is built by using existential abstraction. If $M'h$ satisfies specification $\varphi$, then the original model also satisfies $\varphi$. If not, then the model checker generates a counterexample trace $T'h$. Current model checkers generate paths or loops as counterexamples. Next these counterexamples have to checked for spuriousness.
3.0.3 Identification of spurious counterexample

Path counterexample

The following algorithm identifies if the path counterexample is spurious

**Algorithm 3** SplitPATH(T)

\[
\begin{align*}
S &:= h^{-1}(s_1) \cap I \\
j &:= 1 \\
\text{while } (S \neq \emptyset \text{ and } j < n) \{ & \text{ do } \{ \\
& j := j + 1 \\
& S_{prev} := S \\
& S := Img(S, R) \cap h^{-1}(s_j) \} \\
\text{if } S \neq \emptyset \text{ then output } \text{“counterexample exists” } & \text{ else output } j, S_{prev}
\end{align*}
\]

Loop counterexample

The case for a spurious loop counterexample also has to be considered. When the counterexample T includes a loop, it is written as \(s_1, \cdots, s_is_{i+1}, \cdots, s_n\). The loop starts at the abstract state \(s_{i+1}\) and ends at \(s_n\).

The idea is to unroll the loop. A naive algorithm may have exponential time complexity due to an exponential number of loop unwindings. However, here it shows that for \(T = s_1, \cdots, s_is_{i+1}, \cdots, s_n\), the number of unwindings can be bounded by \(\min = \min_{i+1} h^1(s_j)\), i.e., the number of unwindings is at most the number of concrete states for any abstract state in the loop.

Say, \(T_{unwind}\) denotes this unwinded loop counterexample, i.e., we get the finite abstract path \(\langle s_1, \cdots, s_i \rangle \langle s_{i+1}, \cdots, s_n \rangle^{min+1}\). This can then be treated in the same way as the path counterexample.

3.0.4 Refining abstraction to eliminate spurious counterexample

Now, once the failure state is identified, it can then be refined to eliminate the spurious counterexample. There are three types of concrete states in a failure state:

- Deadend states \(S_D\): reachable but no outgoing edge
• Bad states $S_B$: not reachable but has outgoing edge

• Irrelevant states $S_I$: not reachable and has not outgoing edge

Now the aim is to refine the abstraction such that the deadend states and the bad states do not fall in the same abstract state. The following algorithm achieves this refinement.

**Algorithm 4 PolyRefine**

for $j := 1$ to $m$

$\equiv'_j := \equiv_j$

for every $a, b \in E_j$

if $\text{proj}(S_D, j, a) \neq \text{proj}(S_D, j, b)$

then $\equiv'_j := \equiv'_j \setminus \{(a, b)\}$

where $\text{proj}(X, j, a)$ is given by:

$\text{proj}(X, j, a) = (d_1, ..., d_{j-1}, d_{j+1}, ..., d_m) \mid (d_1, ..., d_{j-1}, a, d_{j+1}, ..., d_m) \in X$.

The refinement procedure continues to refine the abstraction function by partitioning equivalence classes until a real counterexample is found, or the ACTL* property is verified. The partitioning procedure is guaranteed to terminate since each equivalence class must contain at least one element. Thus, this method is complete.
Chapter 4

A case study

The selective slicing in the example of section 1 is generated after careful observation. It can be seen as a special case of parity checking, where each word is 1 bit long. We generate the even parity of this 1 bit data, hence get back the same data as the parity bit. We look at these parity bits as the abstraction generated for the respective 1 bit data words.

Now, we can generalize this further. It is not necessary to generate the abstraction of 1 bit data words. We can take larger words and apply similar abstraction methods that give us a compact signature of large bitvectors. One way of performing this abstraction is to use different error-detection and error-correction techniques. There are many such well known techniques like parity checking, checksumming, etc.

Another aspect we need to keep in mind is that we should abstract the programs appropriately for the property we want to check. For this, we need to take into account the functional behavior of the system. This knowledge is used to abstract the program in a meaningful way. Once an abstract model is obtained we will focus on only those parts of the model that may violate the safety property. Hence we can restrict our subsequent refinements on this section only. The parts which are seen to be safe need not be refined further.
Chapter 5

Future work and conclusion

The case study has provided some insights we would like to build on. To start with we will focus on checking safety properties of programs. For this we need to distinguish between the error and non-error states such that they do not fall in the same state after we perform our abstraction. For this we draw our attention to the following.

- The question now is how to design the hash functions as to suitably represent bit-vectors. We could start from scratch and generate our own hash functions. However there are already many popular hashing techniques present that have been proved to be very effective in their respective domains. But, to the best of our knowledge, they have not yet been used in the context of data abstraction. We would be looking into details of some of these techniques.

**Error detection and correction codes**  
*Simple parity check* is a technique where a redundant bit, called a *parity bit*, is added to every data unit so that the total number of 1s in the unit becomes even (or odd). The parity generator counts the 1s in the data unit and appends a parity bit to the end. The total number of 1s now become even.

In the example taken in section 1, selective slicing of the data words can be seen as a special case of this parity checking technique, where each segment is of 1 bit and so the parity bit has the same value as the 1 bit data.

Another error detection method often used is the *checksum*. The *checksum* generator subdivides the data units into equal segments of $n$ bits (usually 16). These segments are added using ones complement arithmetic in such a way that the total is also $n$ bits long. The total (sum) is then complemented and appended to the end of the original data unit, called the *checksum* field. For checking the original data we need to subdivide the data units as above, add all segments and
complement the result. The total value found by adding the data
segments and the checksum field should be zero. If the result is not
zero, it is detected as an error in the data.

- However, the above mentioned techniques for hashing would not be
the best suited for our purpose. This is because they are completely
oblivious about the behavior or functionality of the program it is ab-
stracting. Hence, we strive to go a step further than merely following
any hashing algorithm. We plan to have a two level hashing, where the
first step hashing is done as discussed above. If this is not sufficient
to completely distinguish the error states from non-error states, we
would move ahead with the next level of hashing. This second level
hashing need only be done on those abstract states that do not satisfy
the safety property. The states that satisfy the safety property need
not be refined further.

We aim at separating the error states from the non-error ones at
a certain abstract state. We also want to do this partitioning over
the states just preceding these abstract states, the ones preceding
those and so on up to a certain level $k$. Thus we intend to obtain a
pre-image of the execution trace up to $k$ levels, where at every level
$k$ our abstraction scheme is able to differentiate the error states from
the non-error states.

- The main objective here is to use such techniques to generate a com-
pact signature of the original data word. This compact signature can
then be used as the hashed value. We start with the hash function
as obtained from the first bullet and modulate them to take into ac-
count aspects from the second bullet point. Both of these together
should account for the compact signature we obtain to appropriately
represent the original data word.

If however, this compacted signature fails to distinguish between two
error and non-error states, we would need to refine this signature by
adding more bits to it. This approach is similar to the counterexample
guided abstraction refinement framework.

The focus of the project would be on implementing the above two level
hashing method and device some reasonable abstraction and refinement
technique thereon.
Appendix A

ITC ’99 Benchmark example used for case study

description of entity and architecture

entity b08 is
  port (  
    CLOCK: in bit;  
    RESET: in bit;  
    START: in bit;  
    I: in bit_vector (7 downto 0);  
    O: out bit_vector (3 downto 0)  
  );
end b08;

architecture BEHAV of b08 is

  type mem is array (0 to 7) of bit_vector (19 downto 0);  
  constant ROM: mem :=  
    ("0111111111001110110",  
    "001110011110101100010",  
    "10100000000000000000",  
    "0101010011111101111111",  
    "11010111011010111010",  
    "111111111110110101000",  
    "11101011010011011010",  
    "00101111111111110100");

  constant start_st :integer:=0;  
  constant init :integer:=1;  
  constant loop_st :integer:=2;  
  constant the_end :integer:=3;
signal IN_R: bit_vector (7 downto 0);
signal OUT_R: bit_vector (3 downto 0);
signal MAR: integer range 7 downto 0;

begin
process (CLOCK,RESET)
variable STATO: integer range 3 downto 0;
variable ROM_1: bit_vector (7 downto 0);
variable ROM_2: bit_vector (7 downto 0);
variable ROM_OR: bit_vector (3 downto 0);

begin
if RESET = '1' then
  stato := start_st;
  ROM_1 := "00000000";
  ROM_2 := "00000000";
  ROM_OR := "0000";
  MAR <= 0;
  IN_R <= "00000000";
  OUT_R <= "0000";
  O <= "0000";
elsif CLOCK'event and CLOCK = '1' then
  case STATO is
  when start_st =>
    if (START = '1') then
      STATO := init;
    end if;

  when init =>
    IN_R <= I;
    OUT_R <= "0000";
    MAR <= 0;
    STATO := loop_st;

  when loop_st =>
    ROM_1 := ROM(MAR)(19 downto 12);
    ROM_2 := ROM(MAR)(11 downto 4);
    if ((ROM_2 and not IN_R) or (ROM_1 and IN_R)
      or (ROM_2 and ROM_1)) = "11111111" then
      ROM_OR := ROM(MAR)(3 downto 0);
    end if;
    OUT_R <= OUT_R or ROM_OR;
  end case;
end if;
end process;
end;

end if;
STATO := the_end;

when the_end =>
if (MAR /= 7) then
  MAR <= MAR+1;
  STATO := loop_st;
elsif (START = '0') then
  O <= OUT_R;
  STATO := start_st;
end if;

end case;
end if;
end process;
end BEHAV;
Bibliography


