Automatic Analysis of Relay Ladder Logic Programs[*]

Zhendong Su
EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776
zhendong@cs.berkeley.edu
(510) 642 - 6509

Abstract

Relay Ladder Logic (RLL) [4] is a programming language widely used for complex embedded control applications such as manufacturing and amusement park rides. The cost of bugs in RLL programs is extremely high, often measured in millions of dollars (for shutting down a factory) or human safety (for rides). In this paper, we describe our experience in applying constraint-based program analysis techniques to analyze production RLL programs. We demonstrate that our analyses are useful in detecting some common programming mistakes and can be easily extended to perform other kinds of analysis for RLL programs such as some of the analyses described in [6].

1 Introduction

Programming logic controllers (PLC's) are control development systems used extensively in manufacturing industries for complex embedded control applications such as factory control and for entertainment equipment such as amusement park rides. Relay Ladder Logic (RLL) is the most widely used PLC programming language; approximately 50% of the manufacturing capacity in the United States is programmed in RLL [5].

RLL has long been criticized for its low level design, which makes it difficult to write correct RLL programs [19]. Moreover, validation of RLL programs is extremely expensive, often measured in millions of dollars (for shutting down a factory) or human safety (for rides). One solution is to replace RLL with a higher-level, safer programming language. An alternative is to provide direct programming support for RLL. Since there are many existing RLL applications, and many more will be written in this language, we consider this latter approach in this paper.

We have designed and implemented a tool for analyzing RLL programs. Our analyzer automatically detects some common programming mistakes that are difficult, if not impossible, to detect manually. The information inferred by the analyzer can be used by RLL programmers to identify and correct these errors.

Our most interesting result is an analysis to detect certain race conditions in RLL programs. Tested on real RLL programs, the analysis found several such races, including one known bug that originally cost several million dollars measured in factory down-time [5].

Our analyses are *constraint-based*, meaning that the information we wish to know about a program is expressed as constraints [17, 2, 3]. The solutions of these constraints yield the desired information. Our analyses are built using a generic constraint resolution engine, which allows our analyses to be expressed very directly. Constraint-based program analysis is discussed further in Section 2.

[*]The research was funded in part by the National Science Foundation, Grant No. CCR-9416973, and by NSF Infrastructure Grant No. CDA-9401156, and supported in part by a gift from Rockwell Corporation. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

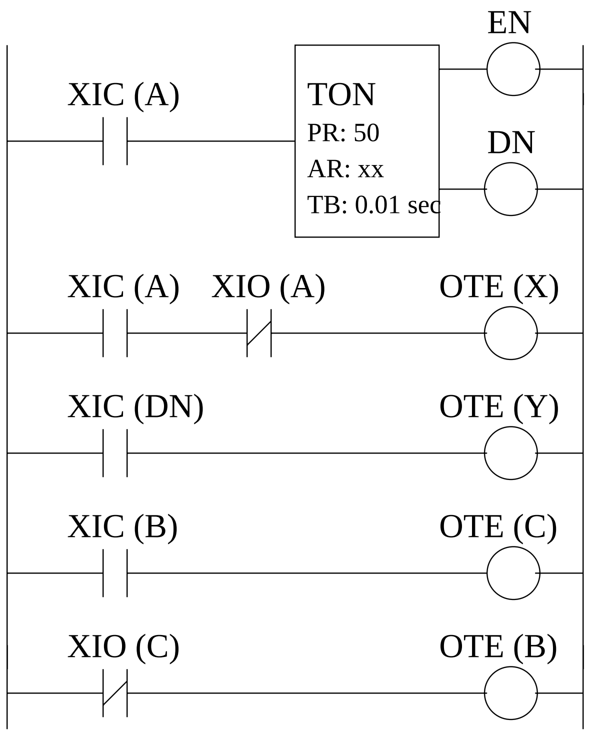


Figure 1: An example RLL program.

RLL programs are represented as *ladder diagrams*, which are a stylized form of a circuit or data flow diagram. A *ladder diagram* consists of a set of *ladder rungs* with each rung having a set of input instructions and output instructions. We explain this terminology in the context of the example RLL program in Figure 1. In the example, there are two vertical rails. The one on the left supplies power to all crossing rungs of the ladder. The five horizontal lines are the ladder rungs of this program. This example has four kinds of RLL instructions: input (two kinds), outputs, and timer instructions. The small vertical parallel bars | | and |/| represent input instructions, which have a single bit associated with them. The bit is named in the instruction. For example, the | | instruction (an XIC for "Normally Closed Contact" instruction) in the upper-left corner of the diagram reads from the bit named A, and the |/| instruction (an XIO for "Normally Opened Contact" instruction) in the lower-left corner of the diagram reads from the bit named C. The small circles represent output instructions that update the value of their labeled bits. The bits named in input and output instructions are classified into *external* bits, which are connected to inputs or outputs external to the program, and *internal* bits, which are local to the program for temporarily storing program states. External inputs are generally connected to sensors, while external outputs are used to control actuators. The rectangular box represents a timer instruction (a TON for "Timer On-Delay" instruction), where PR (preset) is an integer representing a time interval in seconds, AR (accumulator) keeps the accumulated value, and TB (time base) is the step of each increment of the AR. The timer instructions are used to turn an output on or off after the timer has been on for a preset time interval (the PR value). Instructions are connected by wires, the horizontal lines between instructions. We say a wire is true (or on) if power is supplied to the wire, and the wire is false (or off) otherwise.

An RLL program operates by first reading in all the values of the external input bits and executing the rungs in sequence from top to bottom and left to right. Program control instructions may cause portions of the program be skipped or repeatedly executed. After the last rung is evaluated, all the real output devices connected to the external output bits are updated. Such a three step execution (read inputs, evaluate rungs, update outputs) of the program is called a *scan*. Programs are executed scan after scan until interrupted. Between scans, the input bit values might be changed, either because the inputs were modified by the previous scan (bits can be inputs, outputs, or both) or because of state changes in external sensors attached to the inputs. Subsequent scans use the new input values.

RLL has many types of instructions: relay instructions, timer and counter instructions, data transfer instructions, arithmetic operations, data comparison operations, and program control instructions. A grammar for the subset of RLL discussed in this report is in Figure 2.

Examples of relay instructions are XIC, XIO, and OTE. We briefly describe how these three instructions work for the explanation of our analyses. Let w_1 and w_2 be the wires before an instruction and after an instruction respectively. Further, let b be the bit referenced by an instruction.

XIC: if w_1 and b are true, w_2 is true; otherwise, w_2 is false.

XIO: if w_1 is true, and b is false, w_2 is true; otherwise, w_2 is false.

OTE: the bit b is true if and only if w_1 is true.

In this paper, we describe the design and implementation of our RLL program analyzer. Currently the analyzer performs two different analyses. One is *constant wire analysis*, in which the analyzer detects if there is any wire in a given program that is always true or always false during the execution of a program. Constant wires indicate possible programming mistakes, because it is unlikely that a programmer would intentionally write constant-valued circuits. If a wire is always true or always false, there is no reason to put any instructions before these wires. For example, in the program in Figure 1, if we know that the wire after the XIO(A) instruction in the second rung is always false, then the two instructions XIC(A) and XIO(A) are superfluous.

Our second analysis detects *relay races*. In RLL programs, it is desirable if the values of outputs depend solely on the values of inputs and the internal states of timers and counters. If under fixed inputs and timer and counter states, an output x changes from scan to scan, then there is a *relay-race on x*. For example, in the program in Figure 1, we will see later that the bit B changes value each scan regardless of its initial value. Relay races are particularly difficult to detect by traditional testing techniques, as races can depend on the timing of external events and the scan rate.

Our analyses are a generalization of traditional data flow analyses [1]. Instead of data flow equations, set constraints [17, 2, 3] are used. Set constraints are more expressive than data flow equations since the constraints can model not only the data flow but also the control flow of a program.

Our analyses consist of two steps. In the first step, we generate constraints that describe the data and control flow dependences of an RLL program. The constraints are generated in a top-down traversal of the abstract syntax tree (AST) of the program. According to a set of constraint generation rules (see Section 4), appropriate constraints are generated for each AST nodes. These data and control flow constraints are solved to yield another set of simplified constraints. We call the set of resulting constraints the *base system*. The base system models where and how a value flows in the program. For example, the constraints in Figure 3 are produced for the third rung of the example program in Figure 1.

The constraints in Figure 3 are solved and reduce to the constraints shown in Figure 4. The base system is a *conservative approximation* of the program: if during program execution, a wire or a bit can be true (false), then true (false) is in the set that denotes the values of the wire or the bit in the base system; however, false (true) may be a value in that set, too.

The second step is analysis-specific. For constant wire analysis, we use two different approaches. In the first approach, we constrain every input by both true and false and add the corresponding constraints to the base system. The resulting system is then solved, and the minimum solution is extracted. If in the minimum solution a wire x is not both true and false, we are sure that x is constant since the base system is a conservative approximation of the program. In the second approach, we use random sampling of input assignments to detect constant wires. This approach gives a probabilistic guarantee that a wire is constant. The basic idea is to generate random input assignments and add corresponding constraints to the base system and solve. If a wire x takes on different values in different solutions of the respective systems, we consider that wire as "non-constant." If after some number of samples, a wire x still remains single-valued,

```
       program  ::=  ladder_files
  ladder_files  ::=  ladder_files ladder_file
                 |   ladder_file (* at least one ladder file *)
   ladder_file  ::=  rungs
         rungs  ::=  rungs rung
                 |   (* empty *)
          rung  ::=  input_list output_list
    input_list  ::=  instruction input_list
                 |   input_branch input_list
                 |   (* empty *)
  input_branch  ::=  input_level input_list
   input_level  ::=  input_level input_list
                 |   input_list
   output_list  ::=  instruction
                 |   output_branch
 output_branch  ::=  output_branch input_list output_list
                 |   (* empty*)
   instruction  ::=  XIC   (* a partial list*)
                 |   XIO
                 |   OTE
                 |   OTL
                 |   OTU
                 |   TON
                 |   CTU
                 |   MOV
                 |   JSR
                 |   ⋮
```

Figure 2: Grammar of the ladder language.

$$\begin{array}{rcl}
 & & (\mathbf{true} \in w_1) \\
(\mathbf{true} \in w_1) \Rightarrow (\mathbf{true} \in b_{DN}) & \Rightarrow & (\mathbf{true} \in w_2) \\
(\mathbf{false} \in w_1) & \Rightarrow & (\mathbf{false} \in w_2) \\
(\mathbf{false} \in b_{DN}) & \Rightarrow & (\mathbf{false} \in w_2) \\
(\mathbf{true} \in w_3) & \Rightarrow & (\mathbf{true} \in b_Y) \\
(\mathbf{false} \in w_3) & \Rightarrow & (\mathbf{false} \in b_Y) \\
(\mathbf{true} \in w_2) & \Rightarrow & (\mathbf{true} \in w_3) \\
(\mathbf{false} \in w_2) & \Rightarrow & (\mathbf{false} \in w_3)
\end{array}$$

where

w_1	:	set variable denotes the wire before the instruction XIC (DN);
w_2	:	set variable denotes the wire after the instruction XIC (DN);
w_3	:	set variable denotes the wire before the instruction OTE (Y);
b_{DN}	:	set variable denotes the bit DN, a status bit of the TON instruction;
b_Y	:	set variable denotes the bit Y.

Figure 3: Constraint system for a fragment of the example program in Figure 1.

$$\begin{array}{rcl}
 & & (\mathbf{true} \in w_1) \\
(\mathbf{true} \in b_{DN}) & \Rightarrow & (\mathbf{true} \in w_2) \\
(\mathbf{false} \in b_{DN}) & \Rightarrow & (\mathbf{false} \in w_2) \\
(\mathbf{true} \in w_3) & \Rightarrow & (\mathbf{true} \in b_Y) \\
(\mathbf{false} \in w_3) & \Rightarrow & (\mathbf{false} \in b_Y) \\
(\mathbf{true} \in w_2) & \Rightarrow & (\mathbf{true} \in w_3) \\
(\mathbf{false} \in w_2) & \Rightarrow & (\mathbf{false} \in w_3)
\end{array}$$

Figure 4: Base system for a fragment of the example program in Figure 1.

then x is considered "constant." For example, consider again the example program of Figure 1. Since the second rung does not interfere with the other rungs, we can consider it in isolation. For this rung, whatever the value of the bit A is, the wire after the XIO (A) instruction is always false, since it requires that A to be at the same time both true and false for the wire to be true.

Relay race analysis works by simulating multiple scans and looking for racing outputs. Similar to the constant wire analysis, we choose a random assignment of inputs and add the corresponding constraints to the base system. The resulting system is solved; its minimum solution describes the values of the outputs at the end of the scan. Since some of the output bits are also inputs, in the next scan, the input assignment is updated using the minimum solution from the previous scan. Again, we add the resulting system to the base system and solve to obtain the new minimum solution of the outputs. This process repeats. If an output changes value across scans, a relay race is detected. For example, consider the example program in Figure 1. Since the bottom two rungs do not interfere with the other three, let us consider these two rungs only. Let us assume that B has initial value true. Then C also is true, and so in the last rung, B becomes false. Thus, in the next scan, B is initially false. Thus, C becomes false, which makes B true at the end of this scan. Consequently, we have detected a relay race on B: after the first scan B is false, and after the second scan B is true.

The two analyses are conservative in the sense that they cannot detect all of the constant wires or relay races in a program. However, any constant wire detected by the first constant wire analysis are indeed constant wires, and any constant wire reported by the second constant wire analysis are constant wires with provably high probability. As to the relay race analysis, any relay races the analyzer detects are indeed relay races, and we can prove that a large class of relay races are detected with high probability.

We have implemented the two analyses described in this paper in Standard ML of New Jersey (SML) [21]. Our analyzer is accurate and fast enough to be practical — production RLL programs can be analyzed in a few minutes. The relay race analysis not only detected a known bug in a program that took an RLL programmer four hours of factory down-time to uncover, it also detected many previously unknown relay races in our benchmark programs.

The rest of the paper is structured as follows. First, we describe the constraint language used for our analyses (Section 2). The rules for generating the base system come next (Section 3), followed by a description of our analyses (Section 4). We then discuss some techniques of using constraints to provide support for the analyses (Section 5). Finally, we present some experimental results (Section 6), followed by a discussion of related work (Section 7) and the conclusion (Section 8).

2 Set Constraints

In this section, we describe the constraint language we use for expressing our analyses. We give its syntax and semantics.

Set constraints [17, 2, 3] are inclusion constraints between sets of terms. A set constraint is of the form $x \subseteq y$, where x and y are set expressions. Our expression language consists of set variables, a least value $\bot$, a greatest value $\top$, constant constructors **true** and **false**, intersections, unions, and conditional expressions. The syntax of the expression language is given by the following grammar:

$$E ::= \alpha \mid \bot \mid \top \mid c \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid E_1 \Rightarrow E_2,$$

where c is a constant (either **true** or **false**) and $\alpha \in V$ is a variable.

The abstract domain consists of four elements: $\{\}$ (represented by $\bot$), $\{\mathbf{true}\}$ (represented by **true**), $\{\mathbf{false}\}$ (represented by **false**), $\{\mathbf{true}, \mathbf{false}\}$ (represented by $\top$) with the partial order on these elements given in Figure 5. The domain is a finite lattice with $\cap$ and $\cup$ being the *meet* and *join* respectively. The semantics of the expression language is given in Figure 6.

Conditional expressions deserve some discussion. Conditional expressions are proposed in [3] for accurately modeling of flow-of-control. In the context of RLL, they can be used to express boolean relations very

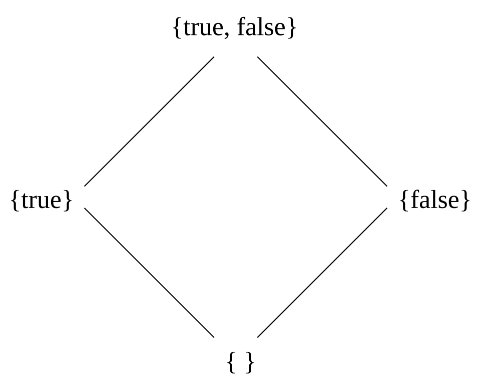


Figure 5: Our working abstract domain.

$$
\begin{array}{rcl}
\rho(\bot) & = & \{\} \\
\rho(\top) & = & \{\mathbf{true}, \mathbf{false}\} \\
\rho(\mathbf{true}) & = & \{\mathbf{true}\} \\
\rho(\mathbf{false}) & = & \{\mathbf{false}\} \\
\rho(E_1 \cap E_2) & = & \rho(E_1) \cap \rho(E_2) \\
\rho(E_1 \cup E_2) & = & \rho(E_1) \cup \rho(E_2) \\
\rho(E_1 \Rightarrow E_2) & = & \begin{cases} \rho(E_2) & \text{if } \rho(E_1) \neq \{\} \\ \{\} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 6: Semantics of set expression in the expression language.

directly. For example, we can express the boolean expression v_1 **and** v_2 with the following conditional expression:

$$((v_1 \cap \mathbf{true}) \Rightarrow (v_2 \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false}) \cup ((v_2 \cap \mathbf{false}) \Rightarrow \mathbf{false})$$

To see this expression does model the operator, notice that if $v_1 = \mathbf{true}$ and $v_2 = \mathbf{true}$, the above expression simplifies to

$$\begin{aligned} ((\mathbf{true} \cap \mathbf{true}) \Rightarrow (\mathbf{true} \cap \mathbf{true}) \Rightarrow \mathbf{true}) &= (\mathbf{true} \Rightarrow \mathbf{true}) \Rightarrow \mathbf{true} \\ &= \mathbf{true}. \end{aligned}$$

One can easily check that the other three cases are also correct.

Most of the RLL instructions can be expressed naturally with boolean expressions. Therefore, the semantics of RLL programs can be very directly modeled by our constraint language. We use set constraints to model RLL programs instead of, for example, boolean logic for the following reason. Although the core of RLL is boolean logic, other instructions such as program control flow and arithmetic instructions also exist. These instructions could, in principle, be expressed using boolean logic. However, the use of boolean logic increases the complexity of modeling RLL programs substantially. On the other hand, set constraints give us the flexibility to model certain instructions less accurately and less expensively than others, thus, making the analysis of RLL programs more manageable.

3 Constraint Generation

In this section, we describe how we use inclusion constraints to model RLL programs. We give the constraint generation rules used to express RLL programs in our constraint language.

Because of the scan evaluation model of RLL programs, it is natural to express the meaning of a program in terms of the meaning of a single scan. The constraint generation rules we present model the meaning of a single scan of RLL programs.

In the constraint generation rules, we use set variables to denote the values of bits and wires in RLL programs. Thus, a bit or wire may be assigned the abstract values $\{\}$ (meaning no value), $\{\mathbf{true}\}$ (definitely true), $\{\mathbf{false}\}$ (definitely false) or $\{\mathbf{true}, \mathbf{false}\}$ (meaning either true or false, i.e., no information). Constraints between these set variables are used to express the data and control flow of a program in one scan.

The rules are of the form

$$E, I \rightarrow E', S, v_1, v_2$$

where:

- E and E' are mappings of bits and wires to their corresponding set variables;
- I is the current instruction;
- S is the set of constraints generated for this instruction;
- v_1 and v_2 are set variables linking the instructions together.

In this section, let w_1 and w_2 denote the wires preceding and following an instruction respectively. Further, let b be the bit referenced by an instruction unless specified otherwise.

Figure 7, Figure 8 and Figure 9 give the rules for generating the constraints describing the data and control flow of RLL programs.

Contacts

The instruction XIC is called "Normally Closed Contact." If w_1 is true, then b is examined. If b is true, then w_2 is true. Otherwise, w_2 is false. In the rule [XIC], we use two fresh set variables v_1 and v_2 to represent the two wires w_1 and w_2. The set variable v_{ct} represents the referenced bit b. The constraints express that w_2 is true if and only if both w_1 and b are true.

The instruction XIO, called "Normally Opened Contact," is the dual of XIC. The wire w_2 is true if and only if w_1 is true and the referenced bit b is false. The rule [XIO] is similar to the rule [XIC].

Energise Coil

The instruction OTE is called "Energise Coil." It is programmed to control either an output connected to the controller or an internal output bit. If the wire w_1 is true, then the referenced bit b is set to true. Otherwise, b is set to false. Rule [OTE] models this instruction. The set variables v_1 and v_2 are the same as in rules [XIC] and [XIO]. The set variable v_{ct} is fresh, representing a new instance of the referenced bit b. Any later references to b use this instance. The constraints express that b is true if and only w_1 is true.

Latches

The instructions OTL and OTU are similar to OTE. OTL is "Latch Coil," and OTU is "Unlatch Coil." These two instructions appear in pairs. In latch coil, the bit b remains true even though the bits that caused this output to be true have changed (i.e., it is a latch). The bit b is true if w_1 is true or it is true before the instruction executes. Otherwise, b is false. The unlatch coil (OTU) instruction is symmetric. In both the rules [OTL] and [OTU], the set variable v'_{ct} represents the value of the b prior to the instruction, while the variable v_{ct} denotes the new instance of b. In the rule [OTL], the constraints express that b is true if and only the wire w_1 is true or b is true before evaluating this instruction. The rule [OTU] is similar.

Timers

Timers (TON) and counters (CTU) are output instructions that control a device after an elapsed period of time or an expired count. They are normally internal output instructions with some associated status bits that may cause other outputs to be on (true) or off (false).

Three status bits are associated with a timer: the *done bit* (DN), the *timing bit* (TT), and the *on bit* (EN). The DN bit is true if the wire w_1 has remained true for a preset period of time. The bit remains true unless w_1 becomes false. The TT bit is true if the wire w_1 is true and the DN bit is false. It is false otherwise, i.e., it is false if the wire w_1 is false or the DN bit is true. The EN bit is true if and only if the wire w_1 is true. In the rule [TON], v_{dn}, v_{tt} and v_{en} are fresh set variables representing new instances of the corresponding bits. The constraint for the DN bit is

$$((v_1 \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup \ \mathbf{false} \ \subseteq v_{dn}.$$

The constraint says that if the wire w_1 is true, then the DN bit is either true or false, i.e., we do not have any information of whether it is true or of whether it is false. If the wire w_1 is false, then the DN bit is definitely false. Notice that in this constraint, we over-estimate the value of the DN bit, meaning that additional values may be assumed for the bit besides its actual value. The constraints for the TT and EN bits are straight forward.

Counters

A counter instruction has two associated status bits: the *done bit* (DN) as in timers and the *on bit* (CU). The DN bit becomes true if the wire w_1 has made a preset number of false to true transitions across scans. The CU bit is true if and only if the wire w_1 is true. In the rule [CTU], v_{dn} and v_{cu} are fresh set variables representing new instances of the corresponding status bits. The constraint for the CU bit is the same as that for a timer's EN bit. The constraint for the DN bit is

$$((v_1 \cap \mathbf{true}) \Rightarrow (v_1 \cap \mathbf{false}) \Rightarrow \mathbf{true}) \cup \ \mathbf{false} \ \subseteq \ v_{dn}.$$

Notice that for the DN bit to be true, the wire w_1 must have made at least one false to true transition. The variable that models the wire w_1 is v_1. The constraint says that if v_1 has both true and false, the DN bit could be either true or false. If v_1 does not have both true and false, the DN bit is definitely false. Again, we over-estimate the value of the DN bit.

$$\frac{\begin{array}{l} v_1 \text{ and } v_2 \text{ are fresh variables} \\ E' = E + \{(XIC_{wb}, v_1), (XIC_{wa}, v_2)\}) \\ v_{ct} = E(XIC_{ct}) \\ S = \begin{array}{l} \{ ((v_1 \cap \mathbf{true}) \Rightarrow (v_{ct} \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false}) \cup \\ \quad ((v_{ct} \cap \mathbf{false}) \Rightarrow \mathbf{false}) \ \subseteq v_2 \ \} \end{array} \end{array}}{E, XIC \to E', S, v_1, v_2} \qquad \text{[XIC]}$$

$$\frac{\begin{array}{l} v_1 \text{ and } v_2 \text{ are fresh variables} \\ E' = E + \{(XIO_{wb}, v_1), (XIO_{wa}, v_2)\}) \\ v_{ct} = E(XIO_{ct}) \\ S = \begin{array}{l} \{ ((v_1 \cap \mathbf{true}) \Rightarrow (v_{ct} \cap \mathbf{false}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false}) \cup \\ \quad ((v_{ct} \cap \mathbf{true}) \Rightarrow \mathbf{false}) \ \subseteq v_2 \ \} \end{array} \end{array}}{E, XIO \to E', S, v_1, v_2} \qquad \text{[XIO]}$$

$$\frac{\begin{array}{l} v_1,\ v_2, \text{ and } v_{ct} \text{ are fresh variables} \\ E' = E + \{(OTE_{wb}, v_1), (OTE_{wa}, v_2), (OTE_{ct}, v_{ct})\} \\ S = \begin{array}{l} \{ ((v_1 \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false}) \ \subseteq v_{ct} \ \} \end{array} \end{array}}{E, OTE \to E', S, v_1, v_2} \qquad \text{[OTE]}$$

$$\frac{\begin{array}{l} v_1,\ v_2, \text{ and } v_{ct} \text{ are fresh variables} \\ E' = E + \{(OTL_{wb}, v_1), (OTL_{wa}, v_2), (OTL_{ct}, v_{ct})\} \\ v'_{ct} = E(OTL_{ct}) \\ S = \begin{array}{l} \{ ((v'_{ct} \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow (v'_{ct} \cap \mathbf{false}) \Rightarrow \mathbf{false}) \ \subseteq v_{ct} \ \} \end{array} \end{array}}{E, OTL \to E', S, v_1, v_2} \qquad \text{[OTL]}$$

$$\frac{\begin{array}{l} v_1,\ v_2, \text{ and } v_{ct} \text{ are fresh variables} \\ E' = E + \{(OTU_{wb}, v_1), (OTU_{wa}, v_2), (OTU_{ct}, v_{ct})\} \\ v'_{ct} = E(OTU_{ct}) \\ S = \begin{array}{l} \{ ((v'_{ct} \cap \mathbf{false}) \Rightarrow \mathbf{false}) \cup \\ \quad ((v_1 \cap \mathbf{true}) \Rightarrow \mathbf{false}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow (v'_{ct} \cap \mathbf{true}) \Rightarrow \mathbf{true}) \ \subseteq v_{ct} \ \} \end{array} \end{array}}{E, OTU \to E', S, v_1, v_2} \qquad \text{[OTU]}$$

Figure 7: Part one of rules for generating constraints.

Data Transfers

The MOV instruction is used for bit transfers. If the wire w_1 is true, the source (a word of 16 bits) is moved into the destination (also a word of 16 bits). If w_1 is false, no action is taken. The fresh variables $dv_i, 0 \leq i \leq 15$ are new instances for the 16 bits of the destination. dv'_i are the variables that represents the old values of the bits in the destination. The set variables sv_i represent the 16 bits of the source. The constraints are

$$(v_1 \cap \mathbf{true}) \Rightarrow sv_i \ \cup \ (v_1 \cap \mathbf{false}) \Rightarrow dv'_i \ \subseteq \ dv_i, 0 \leq i \leq 15.$$

The constraints simply say that if the wire before is true then the source is moved to the destination, otherwise there is no transfer of bits.

Subroutines

JSR is the subroutine call instruction. If the wire w_1 evaluates to true, the subroutine (a portion of ladder rungs) with the same label as specified in the JSR instruction is evaluated until a return is evaluated, after which execution continues with the rung after the JSR instruction. If w_1 is false, execution continues immediately with the rung after the JSR instruction being evaluated. In the rule [JSR], we let B denote the set of all bits in a program. Further, let S be a set of constraints and τ a set expression. We use

$$\tau \Rightarrow S$$

to represent the set of constraints

$$\{\tau \Rightarrow \tau_0 \ \subseteq \ \tau_1 \mid (\tau_0 \ \subseteq \ \tau_1) \in S\}$$

The fresh variables $nv_b, \forall b \in B$ represent new instances of the bits in B. Constraints S are generated for the subroutine with the same label as specified in the JSR instruction together with a modified mapping E'. The variables nv'_b and nv''_b represent the instances of the bits in E and E' respectively. The constraint

$$(v_1 \cap \mathbf{true}) \Rightarrow nv''_b \ \cup \ (v_1 \cap \mathbf{false}) \Rightarrow nv'_b \ \subseteq \ nv_b, \forall b \in B$$

is similar to the constraint in rule [MOV] for merging the bit variables. It says if the wire w_1 is true, then nv''_b should be the value of the current instance, otherwise, nv'_b is the value of the current instance.

The rules in Figure 9 specify the order of evaluation of RLL programs. Constraints are generated in this same order. The order of generating constraints is important because the correct instances of wires and bits should be used.

The rule [RUNG] specifies that the constraints are generated rung by rung in order. The rule [NORUNG] is straight forward, simply saying that no constraints need to be generated.

The rule [IO] describes the generation of constraints for a single rung. The constraints for the input instructions are generated and then the constraints for the output instructions are generated. Notice that, in the rule, the constraint

$$\mathbf{true} \ \subseteq \ v_1.$$

The constraint says that, in a rung, the wire before the first instruction is always true. The constraint

$$v_2 \ \subseteq \ v'_1$$

is for connecting inputs and outputs.

Rules [INO] and [IBRANCH] are similar to the rule [IO], except that v_1 is not always true. The rule [NOINPUT] is straight forward. Similar to the rule [NORUNG], it says that no constraint is generated.

$$\frac{\begin{array}{l} v_1 \;\; v_2 \;\; v_{dn},\, v_{en}, \text{ and } v_{tt} \text{ are fresh variables} \\ E' = E + \begin{array}{l} \{\ (TON_{wb}, v_1), (TON_{wa}, v_2), \\ \quad (TON_{dn}, v_{dn}), (TON_{en}, v_{en}), \\ \quad (TON_{tt}, v_{tt})\ \} \end{array} \\ S = \begin{array}{l} \{\ ((v_1 \cap \mathbf{true}) \Rightarrow \mathbf{true}\ \cup\ \mathbf{false})\ \subseteq v_{dn}, \\ \quad ((v_1 \cap \mathbf{true}) \Rightarrow (v_{dn} \cap \mathbf{false}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false}) \cup \\ \quad ((v_{dn} \cap \mathbf{true}) \Rightarrow \mathbf{false})\ \subseteq\ v_{tt}, \\ \quad ((v_1 \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false})\ \subseteq\ v_{en}\ \} \end{array} \end{array}}{E, TON \to E', S, v_1, v_2} \quad \text{[TON]}$$

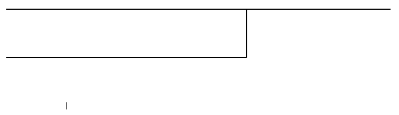
$$\frac{\begin{array}{l} v_1 \;\; v_2 \;\; v_{dn}, \text{ and } v_{cu} \text{ are fresh variables} \\ E' = E + \begin{array}{l} \{\ (CTU_{wb}, v_1), (CTU_{wa}, v_2), \\ \quad (CTU_{dn}, v_{dn}), (CTU_{cu}, v_{en})\ \} \end{array} \\ S = \begin{array}{l} \{\ ((v_1 \cap \mathbf{true}) \Rightarrow (v_1 \cap \mathbf{false}) \Rightarrow \mathbf{true}) \cup \\ \quad \mathbf{false}\ \subseteq\ v_{dn}, \\ \quad ((v_1 \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false})\ \subseteq\ v_{cu} \end{array} \end{array}}{E, CTU \to E', S, v_1, v_2} \quad \text{[CTU]}$$

$$\frac{\begin{array}{l} v_1 \;\; v_2 \;\; dv_i, 0 \le i \le 15, \text{ are fresh variables} \\ E' = E + \begin{array}{l} \{\ (MOV_{wb}, v_1), (MOV_{wa}, v_2), \\ \quad (MOV_{sw_i}, dv_i), 0 \le i \le 15\ \} \end{array} \\ dv'_i = E(MOV_{sw_i}), 0 \le i \le 15 \\ sv_i = E(MOV_{dw_i}), 0 \le i \le 15 \\ S = \begin{array}{l} \{\ ((v_1 \cap \mathbf{true}) \Rightarrow sv_i\ \cup \\ \quad (v_1 \cap \mathbf{false}) \Rightarrow dv'_i)\ \subseteq\ dv_i, 0 \le i \le 15\ \} \end{array} \end{array}}{E, MOV \to E', S, v_1, v_2} \quad \text{[MOV]}$$

$$\frac{\begin{array}{l} B = \text{the set of bits in a program} \\ v_1 \;\; v_2 \;\; nv_b \text{ (where } b \in B) \text{ are fresh variables} \\ E, JSR_{file} \to E', S_0 \\ E'' = E' + \begin{array}{l} \{\ (JSR_{wb}, v_1), (JSR_{wa}, v_2), \\ \quad (b, nv_b), \forall b \in B\ \} \end{array} \\ nv'_b = E(b)\ \forall b \in B \\ nv''_b = E'(b)\ \forall b \in B \\ S = ((v_1 \cap \mathbf{true}) \Rightarrow S_0) \cup \begin{array}{l} \{\ (v_1 \cap \mathbf{true}) \Rightarrow nv''_b \cup \\ \quad (v_1 \cap \mathbf{false}) \Rightarrow nv'_b\ \subseteq\ nv_b, \forall b \in B\ \} \end{array} \end{array}}{E, JSR_{file} \to E', S, v_1, v_2} \quad \text{[JSR]}$$

Figure 8: Part two of rules for generating constraints.

The rule [ILEVEL] describes the generation of constraints for parallel inputs — inputs of the form:



In the rule

$$v_1 = v_1'$$

is an abbreviation for the two constraints

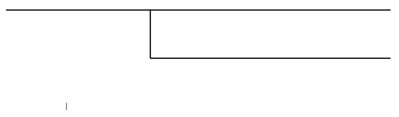
$$v_1 \subseteq v_1' \text{ and } v_1' \subseteq v_1.$$

The fresh variable v is used to model the wire after the parallel wires. The constraint

$$(v_2 \cap \mathbf{true}) \Rightarrow \mathbf{true} \cup (v_2' \cap \mathbf{true}) \Rightarrow \mathbf{true} \cup (v_2 \cap \mathbf{false}) \Rightarrow (v_2' \cap \mathbf{false}) \Rightarrow \mathbf{false} \subseteq v$$

says that the wire after the parallel wires is true if one of the parallel wires is true.

The rule [OBRANCH] describes the generation of constraints for parallel outputs — outputs of the form:



The rule says that the parallel levels of outputs are evaluated from top to bottom. Note that Figure 7 and Figure 8 only give a partial list of all the instructions in RLL. The rules for most other instructions are straightforward. We now present a theorem which states that the constraints generated from an RLL program together with constraints for restricting the inputs has a least solution.

Theorem 1 (Existence of Least Solution) *For any RLL program $\mathcal{P}$, let S be the constraint system generated by the rules given in Figure 7, Figure 8 and Figure 9. Further let c be an input configuration for $\mathcal{P}$. The constraint system S together with the corresponding constraints of c has a least solution, Sol_{least}.*

Theorem 1 is proven in Appendix A.

Next, we state a soundness theorem of our model of RLL programs, namely that our model is a safe approximation of RLL.

Theorem 2 (Soundness) *Let $\mathcal{P}$ be an RLL program and S be the constraint system generated by the rules given in Figure 7, Figure 8 and Figure 9. Further let c be an input configuration for $\mathcal{P}$. The least solution Sol_{least} to the constraint system S together with the constraints restricting the inputs* safely approximates *the values of the wires and bits in one scan, meaning that if an instance of a bit or a wire is true (false), then true (false) is a value in the set representing this instance.*

This theorem is proven in Appendix B.

4 Analyses

In this section, we describe our analyses for detecting constant wires and relay races in RLL programs. The general strategy for each analysis is

1. generate the base system using the constraint generation rules presented in the previous section.
2. add constraints that restrict the inputs to the base system to express the desired information.

In both analyses, we make the assumption that all input assignments are possible. Our analyses can be made more accurate if additional information about the possible input values are available.

rungs

$$\frac{\begin{array}{c} E, rungs \rightarrow E', S_0 \\ E', rung \rightarrow E'', S_1 \end{array}}{E, rungs\ rung \rightarrow E'', S_0\ \cup S_1} \quad \text{[RUNG]}$$

$$\frac{}{E, \epsilon \rightarrow E, \emptyset} \quad \text{[NORUNG]}$$

rung

$$\frac{\begin{array}{c} E, input\_list \rightarrow E', S_0, v_1, v_2 \\ E', output\_list \rightarrow E'', S_1, v_1', v_2' \end{array}}{E, input\_list\ output\_list \rightarrow E'', S_0\ \cup S_1\ \cup \{v_2 \subseteq v_1', \mathbf{true}\ \subseteq\ v_1\}, v_1, v_2'} \quad \text{[IO]}$$

input_list

$$\frac{\begin{array}{c} E, instruction \rightarrow E', S_0, v_1, v_2 \\ E', output\_list \rightarrow E'', S_1, v_1', v_2' \end{array}}{E, instruction\ output\_list \rightarrow E'', S_0\ \cup S_1\ \cup \{v_2 \subseteq v_1'\}, v_1, v_2'} \quad \text{[INO]}$$

$$\frac{\begin{array}{c} E, input\_branch \rightarrow E', S_0, v_1, v_2 \\ E', input\_list \rightarrow E'', S_1, v_1', v_2' \end{array}}{E, input\_branch\ input\_list \rightarrow E'', S_0\ \cup S_1\ \cup \{v_2 \subseteq v_1'\}, v_1, v_2'} \quad \text{[IBRANCH]}$$

$$\frac{v \text{ fresh}}{E, \epsilon \rightarrow E, \emptyset, v, v} \quad \text{[NOINPUT]}$$

input_level

$$\frac{\begin{array}{l} v \text{ is a fresh variable} \\ E, input\_level \rightarrow E', S_0, v_1, v_2 \\ E', input\_list \rightarrow E'', S_1, v_1', v_2' \\ S = \begin{array}{l} \{\ (v_2 \cap \mathbf{true}) \Rightarrow \mathbf{true}\ \cup\ (v_2' \cap \mathbf{true}) \Rightarrow \mathbf{true} \cup \\ \quad (v_2 \cap \mathbf{false}) \Rightarrow (v_2' \cap \mathbf{false}) \Rightarrow \mathbf{false}\ \subseteq\ v\} \end{array} \end{array}}{E, input\_level\ input\_list \rightarrow E'', S \cup\ S_0\ \cup\ S_1\ \cup\ \{v_1 = v_1'\}, v_1, v} \quad \text{[ILEVEL]}$$

output_branch

$$\frac{\begin{array}{l} E, output\_branch \rightarrow E', S_0, v_1, v_2 \\ E', input\_list \rightarrow E'', S_1, v_1', v_2' \\ E'', output\_list \rightarrow E'', S_2, v_1'', v_2'' \\ S = S_0\ \cup S_1\ \cup S_2 \cup \{v_1 = v_1', v_2' \subseteq v_1''\} \end{array}}{E, output\_branch\ input\_list\ output\_list \rightarrow E''', S, v_1, v_2} \quad \text{[OBRANCH]}$$

Figure 9: Part three of rules for generating constraints.

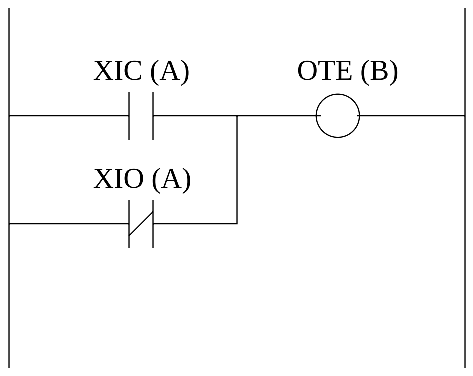


Figure 10: An example RLL program.

4.1 Constant Wire Analysis

We first describe the analysis for detecting constant wires in an RLL program. Recall that the problem is detecting wires that are constant over all possible program executions. Since such a wire contributes nothing to any run of the program, the existence of such a wire usually indicates a programming mistake.

Our approach is to compute both an upper and a lower bound on the set of constant wires. For the lower bound, we constrain every input variable v by

$$\mathbf{true} \cup \mathbf{false} \subseteq v.$$

These constraints are added to the base system. The least solution for the resulting constraint system is then computed. If a variable v is *not* $\mathbf{true} \cup \mathbf{false}$ in the least solution, then we know that the variable *must* only have one value: either **true**, **false** or $\perp$ (undefined). We call this analysis **LB**.

The drawback of **LB** is that it is very inaccurate in the sense that most wires are considered non-constant; in practice, it is a very coarse approximation. Consider the example in Figure 10. It is clear that the wire before the instruction OTE(B) is always true. However, this simple analysis cannot detect this fact. The inaccuracy of **LB** results from its inability to capture interdependencies between quantities, for example between a variable and its negation. The base system for this program is given in Figure 11.

Since bit A is the only input bit, we add the constraint

$$\mathbf{true} \cup \mathbf{false} \subseteq b_A$$

to the base system. The minimum solution of the resulting system is presented in Figure 12. We see that **LB** does not detect the constant wire before OTE(B).

Any constant wires that are computed by **LB** are guaranteed to be constant. Thus, it gives a *lower bound* on the number of constant wires in an RLL program. To get more accurate information, we must model concrete inputs as closely as possible. One possibility is to exhaustively test each possible input configuration, which is just a **true** or **false** assignment for each input variable. Since the number of input variables are usually large, and there are 2^n input configurations of n inputs, exhaustive testing is impractical. However, exhaustive testing is not necessary because we are interested not in what the system computes but whether there are any constant wires. Thus, we can choose input configurations uniformly at random, compute the value for each wire under this input configuration, and union the values of the same wire over all configurations. If the union for a wire turns to be $\mathbf{true} \cup \mathbf{false}$, the wire is not constant.

The intuition behind this analysis is that after a relatively small number of samples, there are few single-valued wires remaining, and they are likely to be constant wires. Since there are only a small number of them, a programmer should be able to check each individual wire. The constraint solver can compute a backward slice [22] for a wire to tell what inputs affect it, along with a boolean function of the wire in terms of these inputs. This information can help the programmer to determine whether a wire is constant and, if it is, the reason it is constant. We call this analysis **UB**.

$$\begin{aligned}
\mathbf{true} &\subseteq w_0 \\
\mathbf{true} &\subseteq w_1 \\
((\mathbf{true} \cap b_A) \Rightarrow \mathbf{true}) \cup ((\mathbf{false} \cap b_A) \Rightarrow \mathbf{false}) &\subseteq w_2 \\
((\mathbf{true} \cap b_A) \Rightarrow \mathbf{false}) \cup ((\mathbf{false} \cap b_A) \Rightarrow \mathbf{true}) &\subseteq w_3 \\
((\mathbf{true} \cap w_2) \Rightarrow \mathbf{true}) \cup ((\mathbf{true} \cap w_3) \Rightarrow \mathbf{true}) \cup (((\mathbf{false} \cap w_2) \Rightarrow (\mathbf{false} \cap w_3)) \Rightarrow \mathbf{false}) &\subseteq w_4 \\
w_4 &\subseteq w_5 \\
((\mathbf{true} \cap w_5) \Rightarrow \mathbf{true}) \cup ((\mathbf{false} \cap w_5) \Rightarrow \mathbf{false}) &\subseteq b_B
\end{aligned}$$

where

w_0	:	the wire before XIC(A)
w_1	:	the wire before XIO(A)
w_2	:	the wire after XIC(A)
w_3	:	the wire after XIO(A)
w_4	:	the wire after the joint
w_5	:	the wire before OTE(B)
b_A	:	the bit A
b_B	:	the bit B

Figure 11: Base system for the example in Figure 10.

$$\begin{aligned}
w_0 &= \mathbf{true} \\
w_1 &= \mathbf{true} \\
w_2 &= \mathbf{true} \cup \mathbf{false} \\
w_3 &= \mathbf{true} \cup \mathbf{false} \\
w_4 &= \mathbf{true} \cup \mathbf{false} \\
w_5 &= \mathbf{true} \cup \mathbf{false} \\
b_A &= \mathbf{true} \cup \mathbf{false} \\
b_B &= \mathbf{true} \cup \mathbf{false}
\end{aligned}$$

Figure 12: Results from **LB** for program in Figure 10.

For the example in Figure 10, the analysis **UB** will include the wire before the instruction OTE(B) as possibly constant, since whatever value (either **true** or **false**) the bit A assumes, the wire before OTE(B) is always **true**. The base system is the same as that for **LB**. The bit A is the only input bit. There are two input configurations: A is true, or A is false. For the input configuration that A is true, we add the following constraint to the base system:

$$\mathbf{true} \subseteq b_A$$

In the minimum solution of these constraints, we know that w_5 is **true**. For the input configuration that A is false, the following constraint is added to the base system:

$$\mathbf{false} \subseteq b_A$$

We now see that w_5 is **true** in the new minimum solution, too. Therefore, the wire before OTE(B) is considered constant by **UB**.

The number of wires that are considered possibly constant by **UB** gives an *upper bound* on the number of constant wires under our model of RLL programs.

4.2 The Effectiveness of Random Sampling

In RLL programs, a bit or a wire usually only depends on a small number of inputs, typically around 10 [1]. This fact makes random sampling in **UB** more effective than one might expect. After a relatively small number of samples of input assignments, we are confident that almost all possible input assignments affecting each input are covered.

To be more precise, assume N is the number of inputs and

$$M = \max_{v \in \mathbf{VAR}} |\mathbf{DEP}(v)|,$$

where **VAR** is the set of variables and $\mathbf{DEP}(v)$ of a variable v is the set of inputs that v depends on. In other words, for all variable v, it depends on no more than M variables. Let $k = 2^M$.

Theorem 3 *For any variable v, the expected number of samples to draw to get all the possible truth assignments of the inputs in $\mathbf{DEP}(v)$ is no more than $k \ln k + \mathcal{O}(k)$.*

Proof: Notice this problem is just a variation of the Coupon Collector's Problem (See Appendix C). ∎

We know from the analysis of the Coupon Collector's Problem that the actual value is sharply concentrated around this expected value.

Theorem 4 *For any variable v and $c > 0$, the probability that after $k(\ln k + c)$ random samples that there are truth assignments missing from the samples is approximately $1 - e^{-e^{-c}}$.*

We also present some empirical measurements of the effectiveness of random sampling in Section 6.2.

4.3 Relay Race Analysis

Our second analysis detects relay races. In RLL programs, it is desirable if the values of outputs depend solely on the values of inputs and the internal states of timers and counters. If under fixed inputs and timer and counter states, an output x changes from scan to scan, then there is a relay-race on x.

Before describing our analysis, we give a more formal definition of the problem. Consider an RLL program P. Let **IN** denote the set of inputs, and let **OUT** denote the set of outputs. Let C be the set of all possible input configurations. Further, let

$$\mathcal{V}_i \; : \mathbf{OUT} \; \rightarrow \; \{\mathbf{true}, \mathbf{false}\}$$

be the mapping from the set of outputs to their corresponding values at the end of the ith scan.

[1] This information is obtained from experiments with a few production size RLL programs.

Definition 1 *An RLL program P is* **race-free** *if for all input configurations $c \in C$, by fixing c, the following holds:*

$$\mathcal{V}_i = \mathcal{V}_j, \quad \forall j \geq i \geq 1.$$

Otherwise, the program has a race.

Definition 1 states under what conditions a program exhibits a race.

Definition 2 *Let P be an RLL program. An* approximation *A of P is an abstraction of the RLL program which satisfies that for any input configuration c and any quantity v of P, $P_c(v)$ (the value of v in the program P with input c) at the end of one scan is contained in $A_c(v)$ (the value of v in the abstraction A with input c), i.e., $P_c(v) \subseteq A_c(v)$.*

Let A be an approximation of S. Let

$$\mathcal{V}'_i \ : \mathbf{OUT} \ \rightarrow \ \{\perp, \mathbf{true}, \mathbf{false}, \top\}$$

be the mapping from the set of outputs to their corresponding values at the end of the ith scan.

Definition 3 *An approximation A of an RLL program S is* **race-free** *if for all input configurations $c \in C$, by fixing c, for the infinite sequence of scans $S_1, S_2, S_3, \ldots$, there exists $\mathcal{V}^* : \mathbf{OUT} \ \rightarrow \ \{\mathbf{true}, \mathbf{false}\}$ such that*

$$\mathcal{V}^* \subseteq \mathcal{V}'_j, \quad \forall j \geq 1,$$

where $\mathcal{V} \subseteq \mathcal{V}'$ iff

$$\mathcal{V}(v) \sqsubseteq \mathcal{V}'(v), \quad \forall v \in \mathbf{OUT}.$$

Claim 1 *Let P be an RLL program and A be an approximation of P. If P is race free, then so is A. In other words, if A exhibits a race, so does P.*

Proof:
Since P is race free, by Definition 1, we have

$$\mathcal{V}_i = \mathcal{V}_j, \ \forall j \geq i \geq 1.$$

Since A is an approximation of P, by Definition 2,

$$\mathcal{V}_i \subseteq \mathcal{V}'_i \ \forall i \geq 1.$$

Let $\mathcal{V}^* = \mathcal{V}(1)$. We have

$$\mathcal{V}^* \subseteq \mathcal{V}'_i, \ \forall i \geq 1.$$

Now, by Definition 3, we see that A is also race free. ■

From Claim 1, we see that if our analysis detects a race under some input, then the program also races under the same input. We now need to deal with the problem of detecting races in our approximation of RLL programs.

Theorem 5 *For any approximation A of an RLL program P and input $c \in C$, the approximation A races under c if and only if there exists $v \in \mathbf{OUT}$ such that:*

$$\bigcap_{i>0} \mathcal{V}'_i(v) \subseteq \perp.$$

Proof:
($\Leftarrow$): Let $v \in \mathbf{OUT}$ be an output such that

$$\bigcap_{i>0} \mathcal{V}'_i(v) \subseteq \bot.$$

Since A is an approximation of the program P, we have that

$$\mathcal{V}'_i(v) \neq \bot.$$

Thus, there exists positive integers $i \neq j$ such that

$$\begin{aligned} \mathcal{V}'_i(v) &= \mathbf{true} \quad \text{and} \\ \mathcal{V}'_j(v) &= \mathbf{false}. \end{aligned}$$

Therefore, there is no $\mathcal{V}^* : \mathbf{OUT} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ such that

$$\mathcal{V}^* \subseteq \mathcal{V}'_j, \quad \forall j \geq 1.$$

Hence, A has a race under c.
($\Rightarrow$): Suppose for all $v \in \mathbf{OUT}$, we have

$$\bigcap_{i>0} \mathcal{V}'_i(v) \not\subseteq \bot.$$

Then, let

$$\mathcal{V} = \bigcap_{i>0} \mathcal{V}'_i.$$

It is easy to see that there exists a $\mathcal{V}^*$ $: \mathbf{OUT} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ such that

$$\mathcal{V}^* \subset \mathcal{V}.$$

Also we have:

$$\mathcal{V}^* \subseteq \mathcal{V}'_i$$

for all $i > 0$. Therefore, A does not race under input c. ■

Theorem 5 leads naturally to the algorithm in Figure 13 for detecting relay races.

We use the example in Figure 1 to demonstrate how the race detection algorithm works. Consider the last two rungs in the example RLL program in isolation. The base system for these two rungs is given in Figure 14.

Assume the bit B is initially true. Adding the constraint

$$\mathbf{true} \subseteq b_{B_0}$$

to the base system and solving the resulting system, we obtain its least solution in Figure 15.

We see that at the end of the first scan, the bit B is false. In the second scan, we add the constraint

$$\mathbf{false} \subseteq b_{B_0}$$

to the base system in Figure 4. The resulting system is solved, and its least solution is shown in Figure 16.

We intersect the values of the output bits, i.e., bits B and C, in the least solutions from the first two scans. The intersection is shown in Figure 17. Since the intersections are both $\bot$, we have detected a race.

The algorithm in Figure 13 detects whether an output races or not under a given input. To help the RLL programmers to find the cause of a race, it is important also to report the relevant inputs. For each input v, we add the constraint

$$\mathbf{true} \cup \mathbf{false} \subseteq v$$

```
$S_{total}$:      the number of scans to perform
$C_{base}$:       the base system
$C_{input}$:      the input configuration as constraints
$S_{current}$:    the current number of scans
$B_{sum}$:        the cumulative intersection of the bounds
$B_{current}$:    the current bound

$S_{current} = 0$;

(* set every output to be true or false *)
for every output $v$
do
    $B_{sum}(v) = \{\mathbf{true}, \mathbf{false}\}$;
od

while $S_{current} \leq S_{total}$
do
    $B_{current} = Sol_{least}(C_{base} \cup C_{input})$;
    $C_{input} = GetInput(B_{current})$;
    $B_{sum} = B_{sum} \cap B_{current}$;
    if $B_{sum}(v) = \{\}$ for some output $v$
        The output $v$ is racing.
    fi
od
```

Figure 13: Algorithm for detecting races.

$$
\begin{array}{rcl}
\mathbf{true} & \subseteq & w_0 \\
((\mathbf{true} \cap b_{B_0}) \Rightarrow \mathbf{true}) \cup ((\mathbf{false} \cap b_{B_0}) \Rightarrow \mathbf{false}) & \subseteq & w_1 \\
((\mathbf{true} \cap w_1) \Rightarrow \mathbf{true}) \cup ((\mathbf{false} \cap w_1) \Rightarrow \mathbf{false}) & \subseteq & w_2 \\
((\mathbf{true} \cap w_2) \Rightarrow \mathbf{true}) \cup ((\mathbf{false} \cap w_2) \Rightarrow \mathbf{false}) & \subseteq & b_C \\
\mathbf{true} & \subseteq & w_3 \\
((\mathbf{true} \cap b_{B_0}) \Rightarrow \mathbf{false}) \cup ((\mathbf{false} \cap b_{B_0}) \Rightarrow \mathbf{true}) & \subseteq & w_4 \\
((\mathbf{true} \cap w_4) \Rightarrow \mathbf{true}) \cup ((\mathbf{false} \cap w_4) \Rightarrow \mathbf{false}) & \subseteq & w_5 \\
((\mathbf{true} \cap w_5) \Rightarrow \mathbf{true}) \cup ((\mathbf{false} \cap w_5) \Rightarrow \mathbf{false}) & \subseteq & b_{B_1}
\end{array}
$$

where

w_0	:	the wire before XIC(B)
w_1	:	the wire after XIC(B)
w_2	:	the wire before OTE(C)
w_3	:	the wire before XIO(C)
w_4	:	the wire after XIO(C)
w_5	:	the wire before OTE(B)
b_{B_0}	:	the initial value of bit B
b_{B_1}	:	the final value of bit B
b_C	:	the bit C

Figure 14: Base system for the last two rungs of the example program in Figure 1.

$$
\begin{array}{rcl}
w_0 & = & \mathbf{true} \\
w_1 & = & \mathbf{true} \\
w_2 & = & \mathbf{true} \\
w_3 & = & \mathbf{true} \\
w_4 & = & \mathbf{false} \\
w_5 & = & \mathbf{false} \\
b_{B_0} & = & \mathbf{true} \\
b_{B_1} & = & \mathbf{false} \\
b_C & = & \mathbf{true}
\end{array}
$$

Figure 15: Least solution at the end of first scan.

$$\begin{aligned}
w_0 &= \textbf{true} \\
w_1 &= \textbf{false} \\
w_2 &= \textbf{false} \\
w_3 &= \textbf{true} \\
w_4 &= \textbf{true} \\
w_5 &= \textbf{true} \\
b_{B_0} &= \textbf{false} \\
b_{B_1} &= \textbf{true} \\
b_C &= \textbf{false}
\end{aligned}$$

Figure 16: Least solution at the end of second scan.

$$\begin{aligned}
\bigcap b_{B_1} &= \perp \\
\bigcap b_C &= \perp
\end{aligned}$$

Figure 17: Intersection of least solutions from the first two scans.

```
$R_{input}$:      an input configuration under which the program races
$I_{current}$:    the current input as constraints

$I_{current} = R_{input}$;

**while** $\exists$ input $v$ that is not checked
**do**
    $I = I_{current} \cup \{(\textbf{true} \cup \textbf{false}) \subseteq v\}$;
    run relay race analysis with $I$ as the input;
    **if** the same races are observed
        $I_{current} = I$;
    **else**
        The input $v$ contributes to the races;
    **fi**
**od**
```

Figure 18: Algorithm for computing the set of inputs causing a race.

```
$B_{racing}$:    the set of racing bits
$C_{base}$:      the base system
$I$:             the set of inputs that affect the bits in $B_{racing}$ transitively

(* compute the set of inputs that affect the bits in $B_{racing}$ in a scan *)
$I = SLICE_{backward}(B_{racing}, C_{base})$

**repeat**
    (* $C$ is the set of the last instances of the bits in $B$ *)
    $C = LAST(I)$;
    $I = I \cup SLICE_{backward}(C, C_{base})$;
**until** $I$ does not change
```

Figure 19: A more efficient algorithm for computing the inputs that cause a race.

to the base constraint system and leave the other inputs unchanged. We run the algorithm in Figure 13 with this modified input configuration. If the same race is observed, we know that v is not one of the inputs causing the race. Otherwise, the input v does contribute to the race. This process repeats until all inputs have been checked. The algorithm is given in Figure 18.

While simple, the algorithm in Figure 18 is an expensive way to compute the inputs that cause a race. Another way of getting the information is presented in Figure 19. The input to the algorithm is the base constraint system and a set of bits that are racing. The algorithm outputs a set of inputs that affect the set of racing bits. The algorithm first computes the inputs that affect B_{racing} in one scan using the facility provided by the constraint solver. Since some of the inputs might be internal, these bits may be affected by other inputs from previous scans. We need to compute what inputs affect these bits by another backward slice. This process repeats until the set I does not grow.

For the relay race analysis, we need to modify the rules [TON] since the status bits of the timers are assumed to be the same for all scans under a given input. This assumption is reasonable since the scan time, compared with the timer increments, is infinitesimal. Figure 20 gives the new rule.

By the analysis of the Coupon Collector's Problem, after approximately $2^k \ln(2^k) = 2^k \cdot k \ln 2 < k \cdot 2^k$ scans, we have detected, in our approximation, all races of k inputs with high probability. These are actual races in the original RLL program.

5 Implementation Techniques

In this section, we discuss some ways in which we use constraints either to limit the size of the information one needs to examine or to obtain useful information from the constraint system. This illustrates that constraints are useful for providing programming support not directly related to the analyses, such as freeing programmers from examining irrelevant information and providing explanation for the causes of certain behaviors of the programs.

$$\frac{\begin{array}{l} v_1\ \ v_2\ \ v_{en}, \text{ and } v_{tt} \text{ are fresh variables} \\ E' = E + \begin{array}{l} \{ (TON_{wb}, v_1), (TON_{wa}, v_2), \\ \quad (TON_{en}, v_{en}), (TON_{tt}, v_{tt}) \ \} \end{array} \\ v_{dn} = E(TON_{dn}) \\ S = \begin{array}{l} \{ \ ((v_1 \cap \mathbf{true}) \Rightarrow (v_{dn} \cap \mathbf{false}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false}) \cup \\ \quad ((v_{dn} \cap \mathbf{true}) \Rightarrow \mathbf{false}) \ \subseteq \ v_{tt}, \\ \quad ((v_1 \cap \mathbf{true}) \Rightarrow \mathbf{true}) \cup \\ \quad ((v_1 \cap \mathbf{false}) \Rightarrow \mathbf{false}) \ \subseteq \ v_{en} \ \} \end{array} \end{array}}{E, TON \to E', S, v_1, v_2} \quad [\text{TON}]$$

Figure 20: Modified rule for timers.

5.1 Filter Values

Recall in the constant wire analysis, after the least solution is computed, we need to determine which wires or bits have only values either $\{\}$, $\{\mathbf{true}\}$, or $\{\mathbf{false}\}$. In order to obtain this information, we test whether

$$(\mathbf{true} \cup \mathbf{false}) \subseteq \bigcup Sol_{least}(v),$$

where v ranges over the instances of a wire w or a bit b. If the subset relation holds, we know that w or b can be both true and false. On the other hand, if the relation does not hold, w or b has one of the other three possible values. With the simple test above, some irrelevant wires or bits may be left for inspection by the programmer. These wires or bits consist of two kinds: the inputs and the left-most wire of each rung.

With random sampling, each input bit is either true or false. To avoid examining these bits, we add a special set constructor **input** to our expression language with semantic value $\{\mathbf{input}\}$. Each input bit has the value $\mathbf{input} \cup \mathbf{true}$ or $\mathbf{input} \cup \mathbf{false}$. Similarly for the beginning wires, we add another special set constructor **initial** to our expression language with semantic value $\{\mathbf{initial}\}$. Each start wire has the value $\mathbf{initial} \cup \mathbf{true}$. Again to determine the wires and bits to inspect, we perform the following test:

$$\begin{array}{rcll} \mathbf{true} \cup \mathbf{false} & \subseteq & V & \mathbf{or} \\ \mathbf{input} & \subseteq & V & \mathbf{or} \\ \mathbf{initial} & \subseteq & V, & \end{array}$$

where V denotes $\bigcup Sol_{least}(v)$. If the test fails, we need to inspect the corresponding bit or wire. Since in the constraint generation rules **input** or **initial** are not propagated from the inputs or the beginning wires, only the inputs have the value **input** and the beginning wires have the value **initial**. Thus, if an input has the value **input**, we know it must be an input, and if a wire has the value **initial**, it must be a beginning wire.

5.2 Counter Wires

In this section, we describe another method to reduce the number of irrelevant wires to be inspected by a programmer.

Recall that a counter (CTU) counts how many times the wire preceding the instruction makes false to true transitions. The done bit (DN) associated with a counter becomes true if the preceding wire has made a preset number of false to true transitions across scans. The constraint for the done bit is given by

$$((v_1 \cap \mathbf{true}) \Rightarrow (v_1 \cap \mathbf{false}) \Rightarrow \mathbf{true}) \cup \mathbf{false} \subseteq v_{dn},$$

where v_1 and v_{dn} are the set variables for the wire preceding the counter instruction and the done bit respectively.

Notice that for v_{dn} to have the value **true**, v_1 must be both true and false in some samples. Suppose in the program, the wire corresponds to v_1 can be true and false, then the done bit can be true in some execution sequence of the program. Assume, however, in our approximation of the program, for all samples, v_1 is always true or false, but not both, then v_{dn} only has the value false. Thus, the done bit is considered constant. In addition, many wires and bits affected by this done bit may be considered constant as well because the done bit is always false. To remove these irrelevant wires and bits, we keep a record of *counter wires*, wires that immediately precede counter instructions. We add not only the constraints corresponding to a sample configuration to the base system, but also the constraints

$$V_{union}(w) \subseteq w, \text{ for all counter wires } w$$

where $V_{union}(w)$ gives the union of the values of w up to the current sample. With the addition of these constraints, the problem with the done bit is readily solved.

5.3 Backward Slicing

Let v be a given variable. It is desirable to know the set of inputs that affect v. This set of inputs is called a *backward slice* for v [22]. The constraint solver we are using can provide us with this information by computing a backward slice. The solver not only provides us with the set of inputs that affect v, but also a boolean formula that describes how v depends on these inputs. This information can help an RLL programmer to determine whether a wire is indeed constant, if the wire is constant, possible causes of the problem. The slice of a variable v can be simply computed by recursively replacing the intermediate variables by their lower bounds until all the variables in the lower bound of v are inputs. This lower bound can be simplified, and the inputs left are the slice of v and the simplified lower bound is effectively a boolean formula describing how these inputs affect v.

6 Experimental Results

We have implemented our analyses using a general constraint solver [14]. The analyses are implemented in SML. Inputs to our analyses are abstract syntax tree (AST) representations of RLL programs. The ASTs are parsed into internal representations, and constraints are generated using the rules in Figure 7, Figure 8, and Figure 9. The resulting constraints are solved to obtain the base system.

6.1 Benchmarks

Four RLL programs were made available to us in AST form for evaluating our analyses.

- **Mini Factory**
  This program is an example program that has been studied and tested by RLL programmers and researchers working on program analysis for RLL programs.
- **Big Bak**
  This is a production RLL program.
- **Wdsdflt(1)**
  Another production application, this program has a known race.

Program	Size	Num. of Vars.	Secs / Scan
Mini Factory	9,267	4,227	0.5
Big Bak	32,005	21,596	4
Wdsdflt(1)	58,561	22,860	3
Wdsdflt(2)	58,561	22,860	3

Figure 21: Benchmark programs for evaluating our analyses.

Program	Lower Bound	Upper Bound	Number of samples
Mini Factory	0	0	500
Big Bak	0	0	30
Wdsdflt(1)	32	868	1000
Wdsdflt(2)	32	868	1000

Figure 22: Results from the constant wire analysis.

- **Wdsdflt(2)**
  This program is a modified version of Wdsdflt(1) with the known race eliminated. The program is included for comparing its results with the results from the original program.

Figure 21 gives a table showing the size of each program in terms of number of lines in abstract syntax tree form, number of variables that are in its base system, and the time it takes our analyses to analyze one scan. All measurements reported here were done on a Ultra Sparc with 512MB of main memory.

6.2 Constant Wire Analysis

We performed the two kinds of constant wire analyses on the four benchmark programs. The results from the analyses are given in Figure 22. In the table, we give, for each program, the number of constant wires from **LB** the number of constant wires from **UB** and the number of samples that **UB** used.

For Mini Factory and Big Bak, both **LB** and **UB** do not detect any constant wires. In one run of Mini Factory, after around 500 samples, there were no "constant" wires left. In one run of Big Bak, after 30 random samples, there were no "constant" wires left. This is because there are many arithmetic instructions in Big Bak, which are not easily modeled accurately without drastically increasing the number of constraints. As a result, the inaccurate modeling of arithmetic operations resulted in most wires being inferred to be both true and false rather quickly. Thus, **UB** terminated much earlier on Big Bak than on Mini Factory. For the two Wdsdflt programs, **LB** detected some constant wires. However, these were not bugs, but rather an artifact of some debugging code in the program that is normally turned off. Because of this debugging code, **UB** reported many wires as possibly constant, as shown in the table.

Figure 23 shows the effectiveness of the idea of random sampling in reducing the number of wires to examine in Mini Factory. The x-axis is the number of random samples. The y-axis shows the number of wires that are still possibly constant. After about 200 samples, the number of possibly constant wires drops to 20. Initially there are approximately 2600 wires.

6.3 Relay Race Analysis

We also performed our relay race analysis on the four benchmarks. This analysis produced more interesting results than the constant wire analysis. It discovered many relay races in our benchmark programs. The results from the analysis are presented in Figure 24. In the table, for each program, we show the number of external racing bits — bits that are connected to external outputs, and the number of internal racing bits — bits that are internal to the program, and the number of total samples run. The analysis were run for 1000

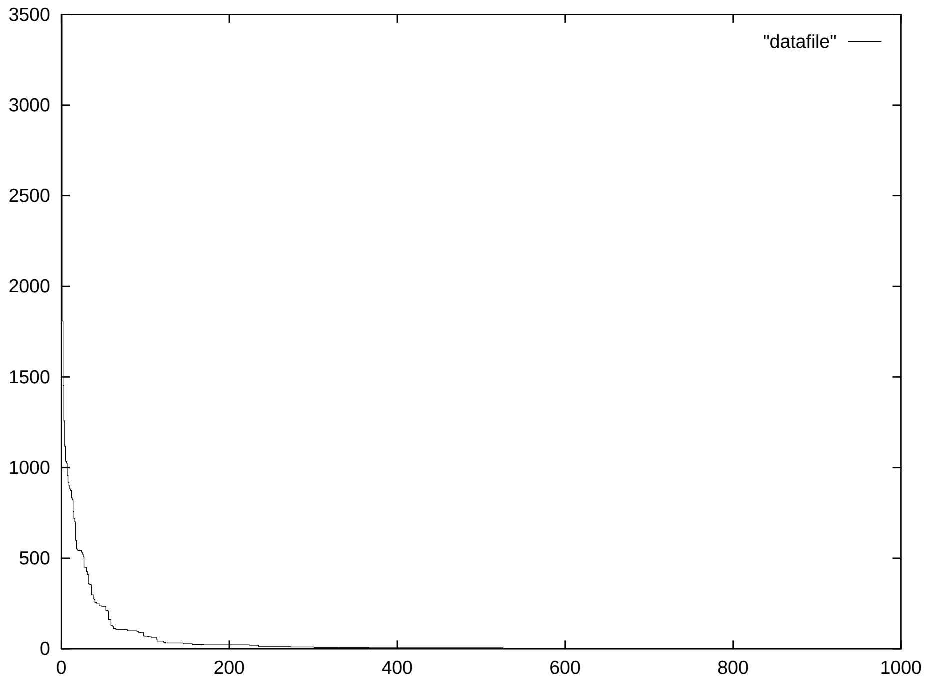


Figure 23: Effectiveness of random sampling.

Program	External Races	Internal Races	Number of samples
Mini Factory	55	186	1000
Big Bak	4	6	1000
Wdsdflt(1)	7	156	1000
Wdsdflt(2)	8	163	1000

Figure 24: Results from the relay race analysis.

samples for all the programs. By the analysis of the Coupon Collector's Problem, 1000 trials are sufficient to uncover all races involves 7 or fewer inputs.

For the Mini Factory program, there were no known relay races in the program, but our analysis detected many such races. Some of the races were subsequently verified by running the program. From the 1000 samples, 55 external races and 186 internal races were reported. For Big Bak, 4 external races and 6 internal races were reported. Although Big Bak is a much bigger program than Mini Factory, the inaccuracy in modeling of arithmetic operations may be one reason why fewer races were found. For Wdsdflt(1), there were 7 external races and 156 internal races reported. The Wdsdflt(2) program has a known relay race, which took the programmer who developed this program four hours to find [5]. Our analysis discovered this bug among 8 external and 163 internal races. Notice that some reported races may be unrealizable if the corresponding input configuration cannot be realized. There is no way without additional information about the possible inputs to characterize which relay races may actually happen.

7 Related Work

In this section, we discuss the similarity and differences of our analyses from work in data flow analysis, model checking, and testing.

Data Flow Analysis Data flow analysis has been traditionally used in optimizing compilers to collect variables usage information for optimizations such as dead code elimination and efficient register allocation [1]. It has also been applied for ensuring software reliability [15, 16]. There are two main distinctions of our approach from data flow analysis. One is the use of conditional constraints [3], which are essential for modeling both the boolean instructions and control flow instructions. The other one is the flexibility of our analyses to add additional constraints to the base system to get desired information, instead of solving the whole constraint system repeatedly. Our approach is more efficient because we work with an initially simplified constraint system.

Model Checking Model checking [10, 11] is a branch of formal verification that can be fully automated. Model checking has been used successfully for verifying finite state systems such as hardware and communication protocols [7, 8, 13, 18, 12]. Model checkers exploit the finite nature of these systems by performing exhaustive state space searches. Because even these finite state spaces may be huge, model checking is usually applied to some abstract models of the actual system. Our analyses for RLL programs use similar techniques. Although RLL programs in general are infinite state systems, our abstract models of RLL programs are finite-state. These abstract systems are symbolically executed to obtain information about the actual systems. In this sense, our analyses are similar to model checking. However, there are some differences. The main difference of our analyses from model checking lies in the way abstract models are obtained and how accurate these systems correspond to the actual system. In model checking, an abstract model of a concrete system is often obtained manually, while our analyses automatically generate the model. With respect to the modeling accuracy, model checking strives to produce an model which has no observable difference from the concrete system from the point of the properties to be checked, i.e., the model is a complete characterization of the actual system. However set constraints (because the use of sets) give us the flexibility to model certain parts of the system more accurately than others for analyzing large scale systems.

Testing Testing is one of the most commonly used methods for assuring hardware and software quality. It is the process of running instance experiments on the system to be checked. The I/O behaviors of the system on these instances are used to deduce whether the given system is faulty or not [20]. Testing is non-exhaustive in most cases due to the infinite or large number of test cases. In addition, testing assumes some kind of distribution of the test cases such as uniform, which is often non-realistic. One distinction of our approach from testing is that we work with an abstract model of the actual system. There are advantages and disadvantages of using an abstract model. An advantage of using the actual system is that there is no loss of information, meaning that if the system shows incorrect I/O behavior under a given input, we can detect this error by running the test with this input; if an abstract model is used, under this same input, we might not detect any error due to the approximation. An advantage of using an abstract model in our system is that we can guarantee that all cases are covered with high probability, while testing fails to cover all the test cases. [9] discusses some other tradeoffs of using the actual system and abstract models of the system for testing.

8 Conclusions

In this paper, we have described two analyses — the constant wire and relay race analyses — for RLL programs using set constraints to help RLL programmers to detect some common programming mistakes. We have demonstrated that these analyses are useful in statically catching some kinds of programming errors. Our implementation of the analyses is accurate and fast enough to be practical — production RLL programs can be analyzed in a few minutes. The relay race analysis not only detected a known bug in a program that took an RLL programmer four hours of factory down-time to uncover, it also detected many previously unknown relay races in our benchmark programs.

Acknowledgments

First and foremost, I would like to thank my advisor Alex Aiken for carefully reading and commenting on drafts of this paper. Discussions with Alex Aiken and Manuel Fahndrich resulted in several ideas in the paper. I thank them. Finally, thanks go to Tony Barrett with Allen-Bradley, Rockwell Automation for information on RLL, providing us with abstract syntax trees of RLL programs, and running some experiments to validate our results.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[3] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[4] Allen–Bradley, Rockwell Automation. *SLC 500 and MicroLogix 1000 Instruction Set.*

[5] T. Barrett. Private communication.

[6] T. Barrett. Ladder logic analysis survey. Unpublished manuscript.

[7] M. Browne, E.M. Clarke, and D. Dill. Checking the correctness of sequential circuits. In *Proc. IEEE Internat. Conf. on Computer Design*, pages 545–548, 1985.

[8] M. Browne, E.M. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Trans. Comput.*, 35(12):1035–1044, 1986.

[9] R.H. Carver and R. Durham. Integrating formal methods and testing for concurrent programs. In *Proceedings of the Tenth Annual Conference on Computer Assurance*, pages 25–33, New York, NY, USA, June 1995.

[10] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, volume 131, pages 52–71, Berlin, 1981. Springer.

[11] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[12] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, North-Holland, April 1993.

[13] D. Dill and E.M. Clarke. Automatic verification of asynchronous circuits using temporal logic. In *Proceedings of the IEEE*, volume 133, pages 276–282, 1986.

[14] M. Fahndrich and A. Aiken. Making set-constraint based program analyses scale. Technical Report UCB/CSD-96-917, University of California at Berkeley, 1996.

[15] L.D. Fosdick and L.J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[16] M.J. Harrold. Using data flow analysis for testing. Technical Report 93-112, Department of Computer Science, Clemson University, 1993.

[17] N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.

[18] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.

[19] A. Krigman. Relay ladder diagrams: we love them, we love them not. In *Tech*, pages 39–47, October 1985.

[20] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. In *Proceedings of the IEEE*, pages 1090–1123, August 1996.

[21] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[22] M. Weiser. Program slicing. *IEEE Transaction on Software Engineering*, SE-10(4):352–357, July 1984.

A Existence of the Least Solution

In this section, we prove Theorem 1.

Proof: Notice that every constraint is of the form $e \subseteq v$, where e is a set expression and v is a variable. Thus we can obtain a solution of the constraint system by assigning each variable $\{\mathbf{true}, \mathbf{false}\}$. To see that there is a least solution, we show that if S_1 and S_2 are two solutions, then $S_1 \cap S_2$ is also a solution, where $S_1 \cap S_2 = \{S_1(v) \cap S_2(v) \mid \text{for all variable } v\}$. First we show by induction that for any set expression e and any two variable assignments S_1 and S_2 the following holds:

$$(S_1 \cap S_2)(e) \subseteq (S_1(e) \cap S_2(e)).$$

- Base cases:
  - $e = \bot$: straight forward.
  - $e = \top$: straight forward.
  - $e = \mathbf{true}$: straight forward.
  - $e = \mathbf{false}$: straight forward.
  - $e = v'$, where v' is a variable:
    $(S_1 \cap S_2)(v') = S_1(v') \cap S_2(v')$ by the definition of $S_1 \cap S_2$.
- Inductive cases:
  - $e = (e_1 \cap e_2)$:
    We have
    $$(S_1 \cap S_2)(e_1) \subseteq (S_1(e_1) \cap S_2(e_1))$$
    and
    $$(S_1 \cap S_2)(e_2) \subseteq (S_1(e_2) \cap S_2(e_2)).$$
    Thus, we have
    $$\begin{aligned}(S_1 \cap S_2)(e_1 \cap e_2) &= (S_1 \cap S_2)(e_1) \cap (S_1 \cap S_2)(e_2) \\ &\subseteq (S_1(e_1) \cap S_2(e_1)) \cap (S_1(e_2) \cap S_2(e_2)) \\ &= (S_1(e_1 \cap e_2) \cap S_2(e_1 \cap e_2)).\end{aligned}$$
  - $e = (e_1 \cup e_2)$:
    We have
    $$(S_1 \cap S_2)(e_1) \subseteq (S_1(e_1) \cap S_2(e_1))$$
    and
    $$(S_1 \cap S_2)(e_2) \subseteq (S_1(e_2) \cap S_2(e_2)).$$
    Thus, we have
    $$\begin{aligned}(S_1 \cap S_2)(e_1 \cup e_2) &= (S_1 \cap S_2)(e_1) \cup (S_1 \cap S_2)(e_2) \\ &\subseteq (S_1(e_1) \cap S_2(e_1)) \cup (S_1(e_2) \cap S_2(e_2)) \\ &\subseteq (S_1(e_1 \cup e_2) \cap S_2(e_1 \cup e_2)).\end{aligned}$$
  - $e = (e_1 \Rightarrow e_2)$:
    We have
    $$(S_1 \cap S_2)(e_1) \subseteq (S_1(e_1) \cap S_2(e_1))$$
    and
    $$(S_1 \cap S_2)(e_2) \subseteq (S_1(e_2) \cap S_2(e_2)).$$

Thus, we have

$$\begin{aligned}
(S_1 \cap S_2)(e_1 \Rightarrow e_2) &= (S_1 \cap S_2)(e_1) \Rightarrow (S_1 \cap S_2)(e_2) \\
&\subseteq (S_1(e_1) \cap S_2(e_1)) \Rightarrow (S_1(e_2) \cap S_2(e_2)) \\
&\subseteq (S_1(e_1) \Rightarrow S_1(e_2)) \cap (S_2(e_1) \Rightarrow S_2(e_2)) \\
&= (S_1(e_1 \Rightarrow e_2) \cap S_2(e_1 \Rightarrow e_2)).
\end{aligned}$$

Now, let S_1 and S_2 be two solutions to the constraint system $S \cup c$. For each constraint $e \subseteq v$, we have

$$(S_1 \cap S_2)(e) \subseteq (S_1(e) \cap S_2(e)) \subseteq (S_1(v) \cap S_2(v)) = (S_1 \cap S_2)(v).$$

Thus, $S_1 \cap S_2$ is also a solution to the constraint system $S \cup c$. Therefore there exists a least solution, namely the intersection of all solutions. ■

B Soundness

In this section, we prove Theorem 2.

Proof: Notice that the constraint system can be represented as a directed, acyclic constraint graph [2]. Thus we can prove the theorem with an induction on this graph from its sources to its sinks.

- Base case:
  The input variables have the same values as the wires or the bits that they model.
- Inductive case:
  Consider the constraint $e \subseteq v$, assuming all the variables in e approximate their corresponding instances of bits or wires. Suppose the constraint $e \subseteq v$ is generated by an application of the rule [XIC]. The proof for the other rules is similar. We thus have v_1 and v_{ct} approximate the values of XIC_{wb} and XIC_{ct}. There are four cases:
  - If $XIC_{wb} = true$ and $XIC_{ct} = true$, then $true \in v_1$ and $true \in v_{ct}$. Thus, simplifying the set expression that restricts v_2, we have $true \in v_2$.
  - If $XIC_{wb} = true$ and $XIC_{ct} = false$, then $true \in v_1$ and $false \in v_{ct}$. Thus, simplifying the set expression that restricts v_2, we have $false \in v_2$.
  - If $XIC_{wb} = false$ and $XIC_{ct} = true$, then $false \in v_1$ and $true \in v_{ct}$. Thus, simplifying the set expression that restricts v_2, we have $false \in v_2$.
  - If $XIC_{wb} = false$ and $XIC_{ct} = false$, then $false \in v_1$ and $false \in v_{ct}$. Thus, simplifying the set expression that restricts v_2, we have $false \in v_2$.

■

C Coupon Collector's Problem

In the Coupon Collector's Problem, there are n different coupons. At each trial a coupon is drawn uniformly at random. The selected coupon is put back with the rest of the coupons after it has been examined. We are interested in the expected number of trials needed to select all of the n coupons.

Theorem 6 *The expected number trials to select all the n coupons is $n \ln n + \mathcal{O}(n)$.*

[2] This is not true if there are backward jump instructions in an RLL program. In that case, we can do a similar induction on the strongly connected component graph of the constraint graph representing the constraint system.

Proof: Let X be a random variable defined to be the number of trials needed to collect all of the n coupons. Define a *success* to be a trial in which a new coupon is collected. Define the random variables X_i, for $0 \le i \le n-1$, to be the number of trials that follows the i-th success and ends on the trial that collects the $(i+1)$-th coupon. Thus, we have

$$X = \sum_{i=0}^{n-1} X_i.$$

Let p_i be the probability of success on any trial after the i-th coupon has been collected. This is the probability of drawing one of $n-i$ coupons from a pool of n coupons, so that

$$p_i = \frac{n-i}{n}.$$

The random variable X_i is geometrically distributed with parameter p_i. Thus, its expectation

$$E[X_i] = \frac{1}{p_i} = \frac{n}{n-i}.$$

By linearity of expectation, we have that

$$E[X] = E[\sum_{i=0}^{n-1} X_i] = \sum_{i=0}^{n-1} E[X_i] = \sum_{i=0}^{n-1} \frac{n}{n-i} = n \sum_{i=1}^{n} \frac{1}{i} = nH_n,$$

where H_n is the n-th Harmonic number. Since $H_n = \ln n + \Theta(1)$, we have

$$E[X] = n \ln n + \mathcal{O}(n).$$

■