Detecting Races in Relay Ladder Logic Programs

Alexander Aiken*, Manuel Fähndrich*, and Zhendong Su*

EECS Department
University of California, Berkeley**

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. Our approach is an interesting combination of probabilistic testing and program analysis, and we show that our system is able to detect bugs with high probability, up to the approximations made by the conservative program analysis. We demonstrate that our analysis is useful in detecting some flaws in production RLL programs that are difficult to find by other techniques.

1 Introduction

Programmable logic controllers (PLC's) are used extensively for complex embedded control applications such as factory control in manufacturing industries and for entertainment equipment in amusement parks. 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 programs [18]. Moreover, validation of RLL programs is extremely expensive, often measured in millions of dollars (for factory down-time) or human safety (for rides). One solution is to replace RLL with a higher-level, safer programming language. An alternative is to provide better programming support directly for RLL. Since there are many existing RLL applications, and many more will be written in this language, we consider the 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

* Supported in part by the National Science Foundation, Grant No. CCR-9416973, by NSF Infrastructure Grant No. CDA-9401156, and 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.

** Authors' address: EECS Department, University of California, Berkeley, 387 Soda Hall #1776, Berkeley, CA 94720-1776 Email: {aiken,manuel,zhendong}@cs.berkeley.edu

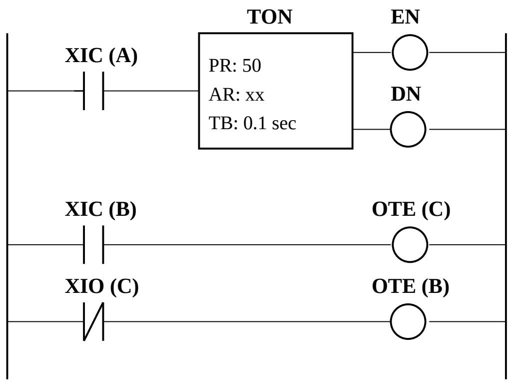

Fig. 1. An example RLL program.

extremely difficult to detect through inspection or testing. 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 costed approximately $750,000 in factory down-time [5].

Our analysis is *constraint-based*, meaning that the information we wish to know about a program is expressed as constraints [16, 2, 3]. The solutions of these constraints yield the desired information. Our analysis is built using a general constraint resolution engine, which allows us to implement the analysis directly in the same natural form it is specified. Constraint-based program analysis is discussed further in Section 2.

Our system has two components: (a) a conservative data and control flow analysis captures information about a program in an initial system of constraints and (b) additional constraints binding program inputs to actual values are added to the initial constraint system, which is then solved to obtain the desired information. Part (a) is done only once, but part (b) is done many times for randomly chosen inputs. Our underlying constraint resolution engine solves and simplifies the initial constraints generated by (a), thereby greatly improving the performance of (b).

Beyond the particular application to RLL programs, this system architecture has properties that may be of independent interest. First, the use of constraints greatly simplifies the engineering needed to factor out the information to be computed once from that which must be reevaluated repeatedly—we simply add new constraints to the initial system. Second, our system is (to the best of our knowledge) a unique blend of conservative program analysis (part (a), which approximates certain aspects of computation) and software testing (part (b), which "executes" the abstraction for concrete inputs). Third, we are able to prove that classes of program errors are detected with high probability, up to the approximations made by the conservative analysis.

We expect that the engineering advantages of using constraints will carry over to other static analysis tools. The latter two results apply directly only if the programming language has a finite domain of values (RLL has only booleans). Thus, our approach is suitable for some other special-purpose languages (e.g., other control languages) but not necessarily for general purpose languages.

1.1 A More Detailed Overview

By any standard RLL is a strange language, combining features of boolean logic (combinatorial circuits), imperative programming (assignment, `goto`, procedures, conditionals), and real-time computation (timers, counters) with an obscure syntax and complex semantics. Although widely used, RLL is not well-known in the research community. We give a brief overview of RLL together with a more detailed, but still high level, description of our analysis system.

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 three 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 if power is supplied to the wire, and the wire is false 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 to 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. Examples of relay instructions are XIC, XIO, and OTE. We briefly describe how these three instructions work for the explanation of our analysis. Let w_1 and w_2 be the wires before 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 for detecting *relay races*. In RLL programs, it is desirable that 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 analysis generalizes traditional data flow analyses [1]. Instead of data flow equations, set constraints [16, 2, 3] are used. Set constraints are more expressive than data flow equations because the constraints can model not only data flow but also control flow of a program.

Our analysis consists of two steps. In the first step, we generate constraints that describe the data and control flow dependencies of an RLL program. The constraints are generated in a top-down traversal of the program's abstract syntax tree (AST). According to a set of constraint generation rules (see Section 3), appropriate constraints are generated for each AST node. These data and control flow constraints are solved to yield another system of simplified constraints, the *base system*. The base system models where and how a value flows in the program. 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 of the relay race analysis simulates multiple scans and looks for racing outputs. 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 output bits are also inputs, the input assignment of the next scan is updated using the outputs from the previous scan. Again, we add this input assignment to the base system and solve to obtain the minimum solution of the outputs after the second scan. If an output changes 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 first rung, consider these two rungs only. 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 race analysis is conservative in the sense that it cannot detect all of the relay races in a program. However, any relay races the analyzer detects are indeed relay races, and we can prove that a large class of relay races is detected with high probability.

We have implemented the race analysis in Standard ML of New Jersey (SML) [20]. Our analyzer is accurate and fast enough to be practical—production RLL programs can be analyzed. The relay race analysis not only detected a known bug in a program that took an RLL programmer four hours of factory downtime 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 the analysis (Section 2). The rules for generating the base system come next (Section 3), followed by a description of the relay race analysis (Section 4). Finally, we present some experimental results (Section 5), followed by a discussion of related work (Section 6) and the conclusion (Section 7).

2 Constraints

In this section, we describe the set constraint language we use for expressing our analysis. Our expression language consists of set variables, a least value $\perp$, a greatest value $\top$, constants $\mathbf{T}$ and $\mathbf{F}$, intersections, unions, and conditional expressions. The syntax of the expression language is

$$E ::= v \mid \perp \mid \top \mid c \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid E_1 \Rightarrow E_2,$$

where c is a constant (either $\mathbf{T}$ or $\mathbf{F}$) and $v \in V$ is a set variable.

The abstract domain consists of four elements: $\emptyset$ (represented by $\perp$), $\{\mathbf{T}\}$ (represented by $\mathbf{T}$), $\{\mathbf{F}\}$ (represented by $\mathbf{F}$), $\{\mathbf{T}, \mathbf{F}\}$ (represented by $\top$) with set inclusion as the partial order on these elements. 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 2.

Conditional expressions deserve some discussion. Conditional expressions are used for accurately modeling flow-of-control (see e.g., [3]). In the context of RLL, they can be used to express boolean relations very directly. For example, we can express the boolean expression $v_1 \wedge v_2$ with the following conditional expression:

$$((v_1 \cap \mathbf{T}) \Rightarrow (v_2 \cap \mathbf{T}) \Rightarrow \mathbf{T}) \cup ((v_1 \cap \mathbf{F}) \Rightarrow \mathbf{F}) \cup ((v_2 \cap \mathbf{F}) \Rightarrow \mathbf{F})$$

$$\begin{aligned}
\rho(\bot) &= \emptyset \\
\rho(\top) &= \{\mathbf{T}, \mathbf{F}\} \\
\rho(\mathbf{T}) &= \{\mathbf{T}\} \\
\rho(\mathbf{F}) &= \{\mathbf{F}\} \\
\rho(E_1 \cap E_2) &= \rho(E_1) \cap \rho(E_2) \\
\rho(E_1 \cup E_2) &= \rho(E_1) \cup \rho(E_2) \\
\rho(E_1 \Rightarrow E_2) &= \begin{cases} \rho(E_2) & \text{if } \rho(E_1) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 2. Semantics of set expressions.

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

$$((\mathbf{T} \cap \mathbf{T}) \Rightarrow (\mathbf{T} \cap \mathbf{T}) \Rightarrow \mathbf{T}) \;=\; ((\mathbf{T} \Rightarrow \mathbf{T}) \Rightarrow \mathbf{T}) \;=\; \mathbf{T}.$$

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

We use set constraints to model RLL programs instead of boolean logic for two reasons. First, although the core of RLL is boolean logic, other instructions (e.g., control flow instructions) are at best difficult to express using boolean logic. Second, RLL programs are large and complex, so approximations are needed for performance reasons. 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. 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 model the meaning of a single scan of RLL programs. In the rules set variables denote the values of bits and wires. Thus, a bit or wire may be assigned the abstract values $\emptyset$ (meaning no value), $\{\mathbf{T}\}$ (definitely true), $\{\mathbf{F}\}$ (definitely false) or $\{\mathbf{T}, \mathbf{F}\}$ (meaning either true or false, i.e., no information). Rules have the form

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

where:

- E and E' are mappings of bits to their corresponding set variables. The operator $+$ extends the mapping such that $(E + \{b, v\})(b') = \begin{cases} v, & \text{if } b' = b \\ E(b'), & \text{otherwise} \end{cases}$
- I is the current instruction;
- S is the set of constraints generated for this instruction;

– v_1 and v_2 are set variables associated with the wires before and after instruction I and are used to link instructions together.

In this section, w_1 and w_2 denote the wires preceding and following an instruction respectively. Furthermore, b denotes the bit referenced by an instruction unless specified otherwise. Figure 3 gives some example inference rules for generating the constraints describing the data and control flow of RLL programs. Below, we explain these rules in more detail.

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], two fresh set variables v_1 and v_2 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 for 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 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 the rule [XIC]. The set variable v_{ct} is fresh, representing a new instance[1] of the referenced bit b. The new instance is recorded in the mapping E'. Later references to b use this instance. The constraints express that b is true if and only if 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. Once an OTL instruction activates its bit b, then b remains true until it is cleared by an unlatch instruction OTU, independently of the wire w_1 which activated the latch. The unlatch coil (OTU) instruction is symmetric. In the rule [OTL], 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. The constraint expresses that b is true if and only the wire w_1 is true or b is true before evaluating this instruction. The rule for OTU is similar.

Timers
Timers (TON) are instructions that activate an output after an elapsed period of time. 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

[1] Due to the sequential evaluation of rungs, a particular bit can take on distinct values in different parts of a program. An instance of a bit captures the state of a bit at a particular program point.

$$\frac{\begin{array}{c} v_1 \text{ and } v_2 \text{ are fresh variables} \\ v_{ct} = E(b) \\ S = \{((v_1 \cap \mathbf{T}) \Rightarrow (v_{ct} \cap \mathbf{T}) \Rightarrow \mathbf{T}) \cup ((v_1 \cap \mathbf{F}) \Rightarrow \mathbf{F}) \cup ((v_{ct} \cap \mathbf{F}) \Rightarrow \mathbf{F}) \ \subseteq v_2\} \end{array}}{E, XIC(b) \mapsto E, S, v_1, v_2} \quad [\text{XIC}]$$

$$\frac{\begin{array}{c} v_1,\ v_2, \text{ and } v_{ct} \text{ are fresh variables} \\ E' = E + \{(b, v_{ct})\} \\ S = \{((v_1 \cap \mathbf{T}) \Rightarrow \mathbf{T}) \cup ((v_1 \cap \mathbf{F}) \Rightarrow \mathbf{F}) \ \subseteq v_{ct}\} \end{array}}{E, OTE(b) \mapsto E', S, v_1, v_2} \quad [\text{OTE}]$$

$$\frac{\begin{array}{c} v_1,\ v_2, \text{ and } v_{ct} \text{ are fresh variables} \\ v'_{ct} = E(b) \\ E' = E + \{(b, v_{ct})\} \\ S = \{((v'_{ct} \cap \mathbf{T}) \Rightarrow \mathbf{T}) \cup ((v_1 \cap \mathbf{T}) \Rightarrow \mathbf{T}) \cup ((v_1 \cap \mathbf{F}) \Rightarrow (v'_{ct} \cap \mathbf{F}) \Rightarrow \mathbf{F}) \ \subseteq v_{ct}\} \end{array}}{E, OTL(b) \mapsto E', S, v_1, v_2} \quad [\text{OTL}]$$

$$\frac{\begin{array}{c} v_1,\ v_2,\ v_{dn},\ v_{en}, \text{ and } v_{tt} \text{ are fresh variables} \\ E' = E + \{(DN, v_{dn}), (EN, v_{en}), (TT, v_{tt})\} \\ S = \left\{ \begin{array}{r} ((v_1 \cap \mathbf{T}) \Rightarrow \mathbf{T} \ \cup \ \mathbf{F}) \ \subseteq v_{dn}, \\ ((v_1 \cap \mathbf{T}) \Rightarrow (v_{dn} \cap \mathbf{F}) \Rightarrow \mathbf{T}) \ \cup ((v_1 \cap \mathbf{F}) \Rightarrow \mathbf{F}) \ \cup ((v_{dn} \cap \mathbf{T}) \Rightarrow \mathbf{F}) \ \subseteq \ v_{tt}, \\ ((v_1 \cap \mathbf{T}) \Rightarrow \mathbf{T}) \ \cup ((v_1 \cap \mathbf{F}) \Rightarrow \mathbf{F}) \ \subseteq \ v_{en} \end{array} \right\} \end{array}}{E, TON \mapsto E', S, v_1, v_2} \quad [\text{TON}]$$

$$\frac{\begin{array}{c} B = \text{the set of bits in the program} \\ v_1,\ v_2,\ nv_b \text{ (for all } b \in B\text{) are fresh variables} \\ R_{fname} = \text{the rungs in the file } \mathit{fname} \\ E,\ R_{fname} \mapsto E', S_0 \\ E'' = \{(b, nv_b) \mid b \in B\} \\ S = ((v_1 \cap \mathbf{T}) \Rightarrow S_0) \cup \{(v_1 \cap \mathbf{T}) \Rightarrow E'(b) \ \cup (v_1 \cap \mathbf{F}) \Rightarrow E(b) \ \subseteq \ nv_b \mid b \in B \ \} \end{array}}{E, JSR_{fname} \mapsto E'', S, v_1, v_2} \quad [\text{JSR}]$$

$$\frac{\begin{array}{c} v \text{ is a fresh variable} \\ E, R_1 \mapsto E', S_0, v_1, v_2 \\ E', R_2 \mapsto E'', S_1, v'_1, v'_2 \\ S = \{(v_2 \cap \mathbf{T}) \Rightarrow \mathbf{T} \ \cup (v'_2 \cap \mathbf{T}) \Rightarrow \mathbf{T} \ \cup (v_2 \cap \mathbf{F}) \Rightarrow (v'_2 \cap \mathbf{F}) \Rightarrow \mathbf{F} \ \subseteq \ v\} \end{array}}{E, R_1 || R_2 \mapsto E'', S \cup \ S_0 \ \cup \ S_1 \ \cup \ \{v_1 = v'_1\}, v_1, v} \quad [\text{PAR}]$$

Fig. 3. Some rules for generating constraints.

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

$$((v_1 \cap \mathbf{T}) \Rightarrow \mathbf{T}) \cup \ \mathbf{F} \ \subseteq v_{dn}.$$

The constraint approximates timer operation while ignoring elapsed time. The DN bit can be false (the timer has not reached its preset period), or if the wire w_1 is true, then the DN bit can be true (the timer may have reached its preset period). The constraints for the TT and EN bits are straightforward.

Remark 1. For the relay race analysis, we assume that the DN bit does not change value across scans. This assumption is reasonable since the scan time, compared with the timer increments, is infinitesimal. The DN bit essentially becomes an input bit in the race analysis, and the constraint is accordingly simplified to $E(DN) \ \subseteq v_{dn}$.

Subroutines

JSR is the subroutine call instruction. If the wire w_1 evaluates to true, the subroutine (a portion of ladder rungs with label *fname* as specified in the JSR instruction) is evaluated up to a return instruction, 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. In the rule [JSR], B denotes the set of all bits in a program. IF S is a set of constraints and τ a set expression, then the notation $\tau \Rightarrow S$ abbreviates the set of constraints

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

The fresh variables nv_b represent new instances of all bits $b \in B$. Constraints S_0 are generated for the ladder rungs of the subroutine together with a modified mapping E'. The constraints

$$\{(v_1 \cap \mathbf{T}) \Rightarrow E'(b) \ \cup \ (v_1 \cap \mathbf{F}) \Rightarrow E(b) \ \subseteq \ nv_b \mid b \in B\}$$

merge the two instances of every bit b from the two possible control flows. If the wire w_1 (modeled by v_1) is true, then $E'(b)$ (the instance after evaluating the subroutine) should be the value of the current instance, otherwise, $E(b)$ is the value of the current instance.

Parallel Wires

The rule [PAR] describes the generation of constraints for parallel wires. Parallel wires behave the same as the disjunction of two boolean variables, i.e., the wire after the parallel wires is true if any one of the two input wires is true. In the rule $v_1 = v_1'$ is an abbreviation for the two constraints $v_1 \ \subseteq \ v_1'$ and $v_1' \ \subseteq \ v_1$. The fresh variable v is used to model the wire after the parallel wires. The constraint

$$(v_2 \cap \mathbf{T}) \Rightarrow \mathbf{T} \ \cup (v_2' \cap \mathbf{T}) \Rightarrow \mathbf{T} \ \cup (v_2 \cap \mathbf{F}) \Rightarrow (v_2' \cap \mathbf{F}) \Rightarrow \mathbf{F} \ \subseteq \ v$$

says that the wire after the parallel wires is true if one of the parallel wires is true. There are other rules for linking instructions together. These rules are similar to [PAR] and are also straightforward.

All solutions of the generated constraints conservatively approximate the evaluation of RLL programs. However, the best approximation is the least solution (in terms of set sizes). We now present a theorem which states that the constraints generated from an RLL program together with constraints for restricting the inputs have 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 3. 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}.*

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 3. 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) in an actual scan, then true (false) is a value in the set representing this instance.*

Theorem 1 and Theorem 2 are proven in [21].

4 Relay Race Analysis

In this section, we describe our analysis for detecting relay races in RLL programs. 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 b changes from scan to scan, then there is a relay race on b.

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[2]. Let C be the set of all possible input configurations. Further, let $\Psi_i : \mathbf{OUT} \rightarrow \{\mathbf{T}, \mathbf{F}\}$ be the mapping from the set of outputs to their corresponding values at the end of the ith scan.

Definition 1. *An RLL program P is* race free *if for any input configurations $c \in C$, by fixing c, it holds that for all $i \geq 1, \Psi_i = \Psi_1$. Otherwise, we say the program has a race.*

[2] Note that **IN** = set of external inputs + internal bits, and **OUT** = set of external outputs + internal bits.

Definition 1 states under what conditions a program exhibits a race. Note that this definition assumes that outputs should stabilize after a single scan.

Definition 2. *Let P be an RLL program. An* approximation *A of P is an abstraction of P such that, for any configuration c and bit b of P, at the end of any scan, the following condition holds: $P_c(b)$ (the value of b in the program P) is contained in $A_c(b)$ (the value of b in the abstraction A), i.e., $P_c(b) \in A_c(b)$.*

Let A be an approximation of P. Let $\Phi_i : \mathbf{OUT} \rightarrow \wp(\{\mathbf{T}, \mathbf{F}\})$ be the mapping from the set of outputs to their corresponding values at the end of the ith scan in A, where $\wp(\{\mathbf{T}, \mathbf{F}\})$ denotes the powerset of $\{\mathbf{T}, \mathbf{F}\}$.

Definition 3. *An approximation A of an RLL program P is* race free *if for any fixed initial input configuration $c \in C$, and the resulting infinite sequence of abstract scans $S_1, S_2, S_3, \ldots$, there exists $\Psi^* : \mathbf{OUT} \rightarrow \{\mathbf{T}, \mathbf{F}\}$ such that $\Psi^*(b) \in \Phi_i(b)$, for all $b \in \mathbf{OUT}$ and $i \geq 1$.*

Lemma 1. *Let P be an RLL program and A 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 $\Psi_i = \Psi_1$ for all $i \geq 1$. Since A is an approximation of P, by Definition 2, $\Psi_i(b) \in \Phi_i(b)$ for all $i \geq 1$. Thus, $\Psi_1(b) \in \Phi_i(b)$ for all $i \geq 1$, and by Definition 3, the approximation A is also race free.

Lemma 1 states that if our analysis detects a race under some input c, then the program will exhibit a race under input c. We now deal with the problem of detecting races in our approximation of RLL programs.

Theorem 3. *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 $b \in \mathbf{OUT}$ such that $\bigcap_{i \geq 1} \Phi_i(b) = \emptyset$.*

Proof. Let $b \in \mathbf{OUT}$ be an output such that $\bigcap_{i \geq 1} \Phi_i(b) = \emptyset$. Since A is an approximation of the program P, we have $\Phi_i(b) \neq \emptyset$. Thus, there exist positive integers $i \neq j$ such that $\Phi_i(b) = \{\mathbf{T}\}$ and $\Phi_j(b) = \{\mathbf{F}\}$. Therefore, there does not exist a $\Psi^* : \mathbf{OUT} \rightarrow \{\mathbf{T}, \mathbf{F}\}$ such that $\Psi^*(b) \in \Phi_i(b)$ for all $b \in \mathbf{OUT}$ and for all $i \geq 1$. Hence, A has a race under c.

Conversely, suppose for all $b \in \mathbf{OUT}$, we have $\bigcap_{i \geq 1} \Phi_i(b) \neq \emptyset$. Then, let $\Phi(b) = \bigcap_{i \geq 1} \Phi_i(b)$ for all $b \in \mathbf{OUT}$. Clearly there exists a $\Psi^* : \mathbf{OUT} \rightarrow \{\mathbf{T}, \mathbf{F}\}$ such that $\Psi^*(b) \in \Phi(b)$ for all $b \in \mathbf{OUT}$. Therefore, A does not race under input c.

In principle, for any given input assignment, it is necessary to simulate scans until a repeating sequence of output configurations is detected, which may require a number of scans exponential in the number of inputs. However, the following lemma shows that two scans are sufficient to uncover the common case.

for every output b
    $B_{sum}(b) := \{\mathbf{T}, \mathbf{F}\}$;
S_{input} := random assignment;
for $Scan := 1$ **to** 2
    $B_{current} := Sol_{least}(S_{base} \cup S_{input})$;
    $S_{input} := GetInput(B_{current})$;
    $B_{sum} := B_{sum} \cap B_{current}$;
    if $B_{sum}(b) = \emptyset$ for some output b
        then output b is racing;

Fig. 4. Algorithm for detecting races.

Lemma 2. *Let A be an approximation of a program P. If A has a race of bit b under input configuration c, such that $\Phi_i(b) \cap \Phi_{i+1}(b) = \emptyset$ for some scan i, then there exists another input configuration c' such that $\Phi_1(b) \cap \Phi_2(b) = \emptyset$ under c', i.e., it is sufficient to use two scans on every input configuration to uncover the race on b.*

Proof. Let $\Phi_i^c(b)$ denote the value of b at the end of the ith scan starting with input configuration c. Without loss of generality, assume $\Phi_i^c(b) = \{\mathbf{T}\}$ and $\Phi_{i+1}^c(b) = \{\mathbf{F}\}$. Consider the values of the inputs c_i prior to scan i. Now choose any configuration c', s.t. $c'(b) \subseteq c_i(b)$ for all b. Since our analysis is monotone in the input (Theorem 1), we have $\Phi_1^{c'}(b) = \{\mathbf{T}\}$ and $\Phi_2^{c'}(b) = \{\mathbf{F}\}$. Hence, the race on bit b can be detected within two scans, starting from a configuration c'.

We have verified experimentally that performing only two scans works well; an experiment in which we performed ten scans per initial input configuration detected no additional races. Theorem 3 and Lemma 2 thus lead naturally to the algorithm in Figure 4 for detecting relay races. The general strategy for the analysis is:

1. Generate the base system using the constraint generation rules presented in Section 3.
2. Add constraints that assign random bits to the inputs.
3. Check whether the program races under this input assignment.
4. Repeat 2.

We make the assumption that all input assignments are possible. In practice, there may be dependencies between inputs that make some input configurations unrealizable. Our analysis can be made more accurate if information about these dependencies is available.

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 the top of Figure 5. Assume the bit B is initially true. Adding the constraint $\mathbf{T} \subseteq b_{B_0}$ to the base system and solving the resulting system, we obtain its least solution at the end

$$\begin{aligned}
\mathbf{T} &\subseteq w_0 \\
((\mathbf{T} \cap b_{B_0}) \Rightarrow \mathbf{T}) \cup ((\mathbf{F} \cap b_{B_0}) \Rightarrow \mathbf{F}) &\subseteq w_1 \\
((\mathbf{T} \cap w_1) \Rightarrow \mathbf{T}) \cup ((\mathbf{F} \cap w_1) \Rightarrow \mathbf{F}) &\subseteq w_2 \\
((\mathbf{T} \cap w_2) \Rightarrow \mathbf{T}) \cup ((\mathbf{F} \cap w_2) \Rightarrow \mathbf{F}) &\subseteq b_C \\
\mathbf{T} &\subseteq w_3 \\
((\mathbf{T} \cap b_C) \Rightarrow \mathbf{F}) \cup ((\mathbf{F} \cap b_C) \Rightarrow \mathbf{T}) &\subseteq w_4 \\
((\mathbf{T} \cap w_4) \Rightarrow \mathbf{T}) \cup ((\mathbf{F} \cap w_4) \Rightarrow \mathbf{F}) &\subseteq w_5 \\
((\mathbf{T} \cap w_5) \Rightarrow \mathbf{T}) \cup ((\mathbf{F} \cap w_5) \Rightarrow \mathbf{F}) &\subseteq b_{B_1}
\end{aligned}$$

bit or wire	variable	value after the first scan	value after the second scan
wire preceding XIC(B)	w_0	**T**	**T**
wire following XIC(B)	w_1	**T**	**F**
wire preceding OTE(C)	w_2	**T**	**F**
wire preceding XIO(C)	w_3	**T**	**T**
wire following XIO(C)	w_4	**F**	**T**
wire preceding OTE(B)	w_5	**F**	**T**
first instance of B	b_{B_0}	**T**	**F**
last instance of B	b_{B_1}	**F**	**T**
the bit C	b_C	**T**	**F**

Fig. 5. Base system for the last two rungs of the example program in Figure 1 with the least solutions at the end of the first and the second scans given in the table.

of the first scan (column 3 in Figure 5). We see that at the end of the first scan, the bit B is false. In the second scan, we add the constraint $\mathbf{F} \subseteq b_{B_0}$ to the base system. The resulting system is solved, and its least solution is shown in column 4 of Figure 5. We intersect the values of the output bits, i.e., bits B (the last instance) and C, in the least solutions from the first two scans. Since the intersections are empty, we have detected a race.

If our analysis finds a race, then the program does indeed exhibit a race. The absence of races cannot be proven by our analysis due to approximations and due to the finite subspace of input assignments we sample. However, we can analyze the coverage of our random sampling approach using the well-known Coupon Collector's Problem: Consider a hat containing n distinct coupons. In a trial a coupon is drawn at random from the hat, examined, and then placed back in the hat. We are interested in the expected number of trials needed to select all n coupons at least once. One can show that the expected number of trials is $n \ln n + \mathcal{O}(n)$, and that the actual number of trials is sharply concentrated around this expected value (for any constant $c > 0$, the probability that after $n(\ln n + c)$ trials there are still coupons not selected is approximately $1 - e^{-e^{-c}}$). Notice that $1 - e^{-e^{-c}} \approx 0.05$ when $c = 3$, and this probability is independent of n.

Program	Size	#Vars.	Secs/Scan	Ext. Races	Int. Races	#Samples	Time (s)
Mini Factory	9,267	4,227	0.4	55	186	1000	844
Big Bak	32,005	21,596	4	4	6	1000	7466
Wdsdflt(1)	58,561	22,860	3	8	163	1000	7285
Wdsdflt(2)	58,561	22,860	3	7	156	1000	7075

Fig. 6. Benchmark programs for evaluating our analysis.

Theorem 4. *Using the Coupon Collector's problem, after approximately* $2^k(\ln 2^k + 3)$ *random samples, any race depending on a fixed set of* k *or fewer inputs has been detected with high probability (95%), up to the approximations due to conservative analysis and performing only two scans.*

Note that the expected number of trials depends only on the number of inputs participating in the race, not on the total number of inputs. For example, the number of trials required to find races involving 5 inputs with 95% probability is 200 whether there are 100, 1000, or 10,000 inputs to the program.

5 Experimental Results

We have implemented our analysis using a general constraint solver [13]. Inputs to our analysis 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 3. The resulting constraints are solved and simplified to obtain the base system.

5.1 Benchmarks

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

- **Mini Factory**
  This is an example program written and used by RLL programmers and researchers working on tools for RLL programming.
- **Big Bak**
  This is a production RLL program.
- **Wdsdflt(1)**
  Another production application, this program has a known race.
- **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 6 gives a table showing the size of each program as number of lines in abstract syntax tree form, number of set variables in the base system, and the time to analyze one scan. All measurements reported here were done on a Sun Enterprise-5000 with 512MB of main memory (using only one of the eight processors).

Our analysis discovered many relay races in these programs. The results are presented in Figure 6. For each program, we show the number of external racing bits (bits connected to external outputs), the number of internal racing bits (bits internal to the program), the number of samples, and the total analysis time in seconds. By Theorem 4, 1000 trials are sufficient to uncover races involving 7 or fewer inputs.

No relay races were known for the Mini Factory program. Our analysis detected 55 external races, some of which were subsequently verified by running a model factory under the corresponding inputs. Fewer races were found in Big Bak, even though it is a much larger program. Two likely reasons for this situation are that Big Bak uses arithmetic operations heavily (which our analysis approximates rather coarsely) and that Big Bak is a production program and has been more thoroughly debugged than Mini Factory. Our analysis discovered the known relay race in Wdsdflt(1) (fixed in Wdsdflt(2)) among 8 external and 163 internal races. Note that some of the reported races may be unrealizable if they depend on input configurations that cannot occur in practice.

6 Related Work

In this section, we discuss the relationship of our work to work in data flow analysis, model checking, and testing.

Data Flow Analysis Data flow analysis is used primarily in optimizing compilers to collect variable usage information for optimizations such as dead code elimination and register allocation [1]. It has also been applied for ensuring software reliability [14, 15]. Our approach differs from classical data flow analysis in two points. First, we use conditional constraints [3], which are essential for modeling both the boolean instructions and control flow instructions. Second, the use of constraints gives us the flexibility to analyze many input configurations by adding constraints to a base system, instead of performing a global dataflow analysis repeatedly. Our approach is more efficient because the base system can be solved and simplified once and then used repeatedly on different input configurations.

Model Checking Model checking [9, 10] 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 [6, 7, 12, 17, 11]. 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. These abstract systems are symbolically executed to obtain information about the actual systems. Our analysis for RLL programs is similar to model checking in that our abstract models are finite, whereas RLL programs are in general infinite state systems. Similar to model checking, we make the tradeoffs between modeling accuracy and efficiency. Our abstraction approximates timers, counters, and arithmetic. It is through these approximations that we obtain a simpler analysis that is practical for production codes. On the other hand, due to these approximations our analysis cannot guarantee the absence of errors. However, our approach differs from model checking in the way abstract models are obtained. In model checking, abstract models are often obtained manually, while our analysis automatically generates the model.

Testing Testing is one of the most commonly used methods for assuring hardware and software quality. The I/O behaviors of the system on input instances are used to deduce whether the given system is faulty or not [19]. Testing is non-exhaustive in most cases due to a large or infinite number of test cases. One distinction of our approach from testing is that we work with an abstract model of the actual system. There are advantages and disadvantages to using an abstract model. A disadvantage is that there is loss of information due to abstraction. As a result, the detection of an error may be impossible, whereas testing the actual system would show the incorrect I/O behavior. Abstract models have the advantage that a much larger space of possible inputs can be covered, which is important if the set of inputs exhibiting a problem is a tiny fraction of all possible inputs. An abstract model is also advantageous when it is very difficult or very expensive to test the actual system. Both of these advantages of abstract modeling apply in the case of detecting relay races in RLL programs. [8] discusses some other tradeoffs of using the actual system and abstract models of the system for testing.

7 Conclusion

In this paper, we have described a relay race analysis for RLL programs to help RLL programmers detect some common programming mistakes. We have demonstrated that the analysis is useful in statically catching such programming errors. Our implementation of the analysis is accurate and fast enough to be practical — production RLL programs can be analyzed. 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

We would like to thank Jim Martin for bringing RLL to our attention and for making this work possible. We would also like to thank Anthony Barrett for information on RLL, providing us with abstract syntax trees of RLL programs, and running some experiments to validate our results. Finally, we thank the anonymous referees for the helpful comments.

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. A. Barrett. Private communication.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. M. Fahndrich and A. Aiken. Making set-constraint based program analyses scale. Technical Report UCB/CSD-96-917, University of California at Berkeley, 1996.
14. L.D. Fosdick and L.J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.
15. M.J. Harrold. Using data flow analysis for testing. Technical Report 93-112, Department of Computer Science, Clemson University, 1993.
16. N. Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.
17. G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editions, 1991.
18. A. Krigman. Relay ladder diagrams: we love them, we love them not. In *Tech*, pages 39–47, October 1985.
19. 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.
20. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
21. Z. Su. Automatic analysis of relay ladder logic programs. Technical Report UCB/CSD-97-969, University of California at Berkeley, 1997.