# Semantics and Analysis of Instruction List Programs

Ralf Huuck [1]

*National ICT Australia*
*University of New South Wales*
*2052 Sydney, Australia*

**Abstract**

*Instruction List* (IL) is a simple typed assembly language commonly used in embedded control. There is little tool support for IL and, although defined in the IEC 61131-3 standard, there is no formal semantics. In this work we develop a formal operational semantics. Moreover, we present an abstract semantics, which allows approximative program simulation for a (possibly infinte) set of inputs in one simulation run. We also extended this framework to an abstract interpretation based analysis, which is implemented in our tool HOMER. All these analyses can be carried out without knowledge of formal methods, which is typically not present in the IL community.

*Key words:* Instruction List, Programmable Logic Controllers, operational semantics, abstract simulation, abstract interpretation.

## 1 Introduction

Programmable Logic Controllers (PLC) are widely used in automation control. They drive assembly lines, robots, and whole chemical plants. The standard IEC 61131-3 [IEC98] defines a number of programming languages for PLCs. These languages range from high-level, graphical ones with powerful structuring possibilities to low level languages close to circuit design or machine language. One of the low level languages is *Instruction List* (IL).

IL is a simple typed assembly language, frequently used whenever it is necessary to have compact, time-critical code. The IL language itself provides little structuring possibilities, in fact, goto-like jumps are the only ones. This makes IL programs difficult to read and difficult to manually analyze. Furthermore, there are hardly any tools available for algorithmic analyses of IL

[1] Email: rhuuck@cse.unsw.edu.au

*This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

programs. The situation is even worsened by the fact that the standard itself does not provide a formal semantics.

The IL language is by its nature particularly prone to run-time errors: variables exceed their allowed range, code is unreachable or leads to infinite loops, there are typing mistakes, or illegal arithmetic operations.

There have been some approaches to abstract IL programs to automata [MW99,Wil99,CCL+00] and Petri net like formalisms [HTLW97,HM98,Bau98]. The analysis is generally carried out by translating [RK98,KBP+99] the formalism into model checking tools ([OY93,LPY97,HHWT97,McM00]). The disadvantages we see in these approaches are that there is no formal operational semantics for IL itself, the abstract models are sometimes too coarse for the nature of errors, and the analysis process requires substantial background in formal methods. The people programming PLCs, however, are often control engineers whose expertise is rather in the development of the plant itself which is driven by the PLC

In this work we propose analysis approaches which do not require any formal methods knowledge and can often be carried out fully automatically. We first develop a *formal operational semantics* for IL programs. The operational semantics does allow code to be *simulated.* Since PLC are reactive systems, it is tedious and sometimes impossible to simulate all possible runs. One improvement we propose is an *abstract simulation.* This allows to simulate approximatively for possibly infinite sets of inputs in one simulation run. Moreover, we explain the extension of this abstract simulation to standard *abstract interpretation* [Cou78,CC79], for analyzing statically the program code with respect to certain generic properties. This has been implemented into the tool Homer.

The remainder of this work is organized as follows: In Section 2 we give a formal semantics to IL programs. The subsequent Section 3 provides the framework for abstract simulation of IL programs and its extension to abstract interpretation. Section 4 explains the analysis features implemented in the tool Homer. Conclusions and future work are discussed in Section 5.

## 2 Syntax and Semantics of IL Programs

### *2.1 Basics*

PLCs are reactive systems interacting in a cyclic manner with their environment. In each cycle inputs (sensor values) are read, computations take place and outputs are written (to actuators). It is possible that a number of IL programs are called sequentially within one cycle.

Each IL program starts with a declaration part, defining *program variables* and their respective types. IL basically supports Booleans, integers,

and floating point numbers. In this work we consider Booleans and integers only. The extension of our framework to floating point numbers, however, is straightforward. We denote the set of all program variables by $Var$, where we tacitly assume all variables and expressions to be well typed. Some variables are marked as input or output variables or both, and we write $Var_{in}$ and $Var_{out}$ for the corresponding subsets of $Var$. Variables that are neither input nor output variables are called *local variables.* The set of all local variables is denoted by $Var_{loc} \subseteq Var$.

Next to variables IL supports the use of one distinct register call *current result* (CR). Every computation takes place in the CR. E.g., a variable value is loaded into the CR, some operations are performed on it and, then, the current value of the CR is stored back into some variable. Since every variable can be loaded into the CR it is dynamically typed. In contrast to most other assembly languages, IL only supports exactly one distinct register. A distinct variable $cr \notin Var$ is used to denote the CR.

### *2.2 Syntax*

Apart from a variable declaration part, instruction list programs are sequences of *statements.* A statement consist of an *instruction* (operator) and an *operand* which can either be a variable, a constant or a jump label. Additionally, programs can be augmented by comments. An example is shown below.

| **instruction** | **operand** | **comment** |
|---|---|---|
| `LD` | `x` | (* loads operand's value to CR *) |
| `JMP` | `lab1` | (* jumps to `lab1` *) |

Some instructions can be augmented by *modifiers.* There are two modifiers: N and C. The N modifier changes an operation from the original to an operation with the negated argument, i.e., negated operand value, while an instruction augmented by the C modifier is only executed under the condition that the CR value is *true*. The use of brackets is allowed to force the evaluation of sub-expressions first and, hence, to avoid auxiliary variables or additional load/store operations. However, it does not add to the expressiveness of this language and we omit this feature in the following. Table 1 lists the most prominent IL commands we use throughout this work.

We denote the set of all instructions, possibly augmented by a modifier, by $Ins$ and the set of operands (variables, CR, labels) by $Ops$. Hence, a statement is an element in $Ins \times Ops$. The set of all statements is denoted by $Stms$. For the sake of simplicity we assume in the remainder that the last instruction of every IL program is `RET`.

### *2.3 Semantics*

We formally define the operational semantics of an IL program by the set of all its possible executions.

Table 1
List of basic IL commands

| Instruction | Modifier | Operand | Description |
|---|---|---|---|
| LD | N | variable, constant | loads operand |
| ST | N | variable, constant | stores operand |
| S | | variable | sets operand to $true$ |
| R | | variable | sets operand to $false$ |
| NOT | | | Boolean negation |
| AND | N | variable, constant | Boolean AND |
| OR | N | variable, constant | Boolean OR |
| XOR | N | variable, constant | Boolean XOR |
| ADD | | variable, constant | addition |
| SUB | | variable, constant | subtraction |
| MUL | | variable, constant | multiplication |
| DIV | | variable, constant | integer division |
| GT | | variable, constant | comparison greater than |
| GE | | variable, constant | comparison greater equal |
| LT | | variable, constant | comparison less than |
| LE | | variable, constant | comparison less equal |
| EQ | | variable, constant | comparison equal |
| NE | | variable, constant | comparison unequal |
| JMP | N, C | label | jump to label |
| RET | | | return from function (block) |

A program *location* is just a line number of code. We freely assume that every program location contains exactly one IL statement. The set of all locations of a program $P$ is denoted by $Locs_P$ and the first location by $l_0$. The function $stm : Locs_P \to Stms$ maps each location to its statement. Moreover, let $succ : Locs_P \to 2^{Locs_P}$ denote the function mapping each location to the set of its successors, i.e., the next location and, if the instruction is a jump to the location with the corresponding label.

We define some auxiliary functions *instr* and *op*. The function $instr : Locs_P \to Ins$ maps any location $l \in Locs_P$ to the corresponding instruction $stm(l)_1$. Complementary, the function $op : Locs_P \to Ops$ maps any location $l \in Locs_P$ to the operand of its associated statement, i.e., $stm(l)_2$.

A *state* of a program is a snapshot of all its variable values while a *configuration* also includes the current program location as well as the mode the PLC is currently in. Formally:

**Definition 2.1** [IL State] The global *IL state* contains the values of all variables and is modeled as a mapping $\Sigma : Var \cup \{cr\} \to D$, where $D$ stands for the union of all data domains.

We assume the values in the state to be type-consistent; we use $\sigma$ as typical element of $\Sigma$.

**Definition 2.2** [IL configuration] An *IL configuration* $\gamma : Locs \times \Sigma \times Mode$

of a program is characterized by

- a location $l \in Locs$,
- a state $\sigma \in \Sigma$, and
- a mode of type *Mode*, which can be either $\mathsf{I}$, $\mathsf{O}$ or $\mathsf{C}(ILi)$, where $ILi$ is an IL instruction.

The mode in the configuration is used to control the various phases of the system behavior and $\mathsf{I}$ stands for "input", $\mathsf{C}(ILi)$ for "calculating" a statement $ILi$, and $\mathsf{O}$ for "output".

The operational semantics for IL programs in our framework is based on labeled transition systems. The nodes of the transitions systems are configurations and the transitions themselves represent the i/o behavior as well as the execution of single IL statements. The transition system is labeled to distinguish between input, output and internal transitions.

**Definition 2.3** [Labeled Transition System of IL Program] With every IL program $P$ we associate a *labeled transition system* $\mathcal{T}_P = (\Gamma, \gamma_0, \rightarrow_\xi)$, where

- $\Gamma$ denotes the set of IL configurations,
- $\gamma_0 \in \Gamma$ is the initial IL configuration and
- $\rightarrow_\xi$ is the transition relation between configurations.

The initial configuration $\gamma_0$ is given by $(l_0, \sigma_0, \mathsf{I})$, where the initial state $\sigma_0$ evaluates all Booleans to *false* and all integers to 0. The operational rules [2] are shown in Figure 1 specifying the labeled transition relation $\rightarrow_\xi$ between system configurations.

The labeled transitions $\rightarrow_{?\boldsymbol{v}}$ and $\rightarrow_{!\boldsymbol{v}}$ in Figure 1 are used to mark reading the input and writing the output variables; all other transitions are unlabeled and internal.

An execution cycle starts by reading the input (cf. rule Input). The state $\sigma$ is updated by assigning values to all input variable as read from the environment and the next mode is activated, the computation. During the computation phase $\mathsf{C}$ the values of the variables or of the CR are updated according to the operations. After performing an operation control moves to the next statement. Note, despite jumps and the final return statement, every statement has only one successor node in the IL graph, i.e., for a node $l$ the successor $l' \in succ(l)$ is unique. Jumps are treated as (possible) branches to nodes with the label statement. They have exactly two successors and we assume that only one of the successors is a label. IL programs are executed until a return statement occurs (cf. rule RET). This statement forces a program to terminate and the mode switches from $\mathsf{C}$ to $\mathsf{O}$ where the output values are written (cf. rule Output). Afterwards, the complete cycle restarts.

[2] Due to space limitations only representative rules are shown. The full set get be found in [Huu03].

$$\frac{\sigma' = \sigma_{[\boldsymbol{x} \mapsto \boldsymbol{v}]} \qquad \boldsymbol{x} = Var_{in}}{(l, \sigma, \mathsf{I}) \rightarrow_{?v} (l, \sigma', \mathsf{C}(instr(l)))} \textbf{Input}$$

$$\frac{instr(l) = RET}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l_0, \sigma, \mathsf{O})} \textbf{RET}$$

$$\frac{\boldsymbol{v} = [\![\boldsymbol{x}]\!](\sigma) \qquad \boldsymbol{x} = Var_{out}}{(l, \sigma, \mathsf{O}) \rightarrow_{!\boldsymbol{v}} (l, \sigma, \mathsf{I})} \textbf{Output}$$

$$\frac{instr(l) = LABEL \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma, \mathsf{C}(instr(l')))} \textbf{LABEL}$$

$$\frac{instr(l) = JMP \qquad l' \in succ(l) \qquad instr(l') = LABEL}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma, \mathsf{C}(instr(l')))} \textbf{JMP}$$

$$\frac{instr(l) = JMPC \qquad l' \in succ(l) cr(\sigma) = false \qquad instr(l') \neq LABEL}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma, \mathsf{C}(instr(l')))} \textbf{JMPCff}$$

$$\frac{instr(l) = JMPC \qquad l' \in succ(l) cr(\sigma) = true \qquad instr(l') = LABEL}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma, \mathsf{C}(instr(l')))} \textbf{JMPCtt}$$

$$\frac{instr(l) = LD \qquad \sigma' = \sigma_{[op(l) \mapsto cr]} \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \textbf{LD}$$

$$\frac{instr(l) = ST \qquad \sigma' = \sigma_{[cr \mapsto op(l)]} \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \textbf{ST}$$

$$\frac{instr(l) = ADD \qquad \sigma' = \sigma_{[cr \mapsto cr + op(l)]} \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \textbf{ADD}$$

$$\frac{instr(l) = MUL \qquad \sigma' = \sigma_{[cr \mapsto cr * op(l)]} \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \textbf{MUL}$$

$$\frac{instr(l) = NOT \qquad \sigma' = \sigma_{[cr \mapsto \neg cr]} \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \textbf{NOT}$$

$$\frac{instr(l) = AND \qquad \sigma' = \sigma_{[cr \mapsto cr \wedge op(l)]} \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \textbf{AND}$$

$$\frac{instr(l) = LT \qquad \sigma' = \sigma_{[cr \mapsto cr < op(l)]} \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \textbf{LT}$$

$$\frac{instr(l) = EQ \qquad \sigma' = \sigma_{[cr \mapsto cr = op(l)]} \qquad l' \in succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \textbf{EQ}$$

Fig. 1. Concrete operational semantics

The semantics of an IL program is defined by the set of all possible execution sequences.

## 3 Analysis

When considering analysis techniques for IL programs it is important to have in mind the users of these techniques. PLCs are foremost programmed by

control engineers more familiar with technical design of the driven plant than, e.g., formal methods. Hence, any proposed analysis should reflect this, i.e., should be able to be carried mostly automatically or reside in the known context.

Moreover, the types of errors occurring in IL programming a likely to be generic run-time errors such as variables exceeding their allowed range, unreachable code, deadlocks, or illegal arithmetic operations.

The developed operational semantics allows to simulate the code for given inputs. A complete coverage is, however, tedious or even impossible. In this section we propose two solutions: One is an abstract simulation of the code. This means, we estimate the range of variables in a simulation run not only for single inputs but (possibly infinite) sets of inputs. Second, we explain how to extend this framework to abstract interpretation which gives us an approximation for all runs at all program locations.

Since we are mostly concerned to find upper and lower bounds for variables, an interval approximation for integer variables seems to be appropriate. Booleans will be extended to carry *don't know* ($\top$) elements, denoting that Boolean variables can be of any Boolean value.

To replace the concrete semantics with an abstract one, we have to replace the concrete domain with the mentioned *abstract domain* and define for any concrete operation a corresponding *abstract semantic operations.* Based on this we define the abstract semantics allowing for abstract simulation. And by enforcing safe termination of the simulation, we extend it to the standard abstract interpretation.

### *3.1 Abstract Domains*

In the previous section the concrete domains have been the set of Booleans and integers. Since we are only interested in the minimum and maximum value of each program variable at each location we introduce as abstract domains the *lattices* [Bri67] of Booleans $\langle \mathcal{B}, \subseteq_{\mathcal{B}} \rangle$ and intervals $\langle \mathcal{I}, \subseteq_{\mathcal{I}} \rangle$ . The lattice of Booleans is depicted in Figure 2. The lattice of intervals is defined by the set $\mathcal{I}$ of all intervals over natural numbers augmented by the top element $[-\infty, +\infty]$. The top element denotes the interval comprising all numbers including infinity. The empty interval [] represents the bottom element $\bot$. The partial ordering relation $\subseteq_{\mathcal{I}}$ is defined by interval inclusion. Moreover, for any any lattice $L$ with a partial ordering relation $\subseteq_L$ we say $p_2$ *approximates* $p_1$ if, and only if, $p_1 \subseteq_L p_2$.

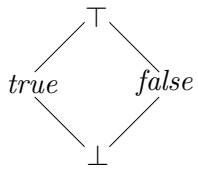


Fig. 2. Lattice of Booleans

### *3.2 Abstract Semantic Operations*

The corresponding abstract operators are defined in Table 2. Note that we consider all operators to be strict, i.e., if any argument is the bottom element of the respective lattice the result yields the bottom element. For the sake of brevity this is not explicitly mentioned in the definitions. Note that in an abstract semantics comparisons and logic operations might result into an unknown, i.e., $\top$, result, e.g., by comparing two overlapping intervals such as $[1,3] < [2,4]$. The operation glb stands for the greatest lower bound and lub for the least upper bound.

Table 2
Abstract operators

| operator | abstract semantics |
|---|---|
| $\neg^\#$ | $\neg^\# b = \begin{cases} \top & \text{if } b = \top \\ \neg b & \text{otherwise} \end{cases}$ |
| $\wedge^\#$ | $b_1 \wedge^\# b_2 = \begin{cases} b_1 \wedge b_2 & \text{if } b_1 \neq \top \text{ and } b_2 \neq \top \\ \top & \text{otherwise} \end{cases}$ |
| $+^\#$ | $i_1 +^\# i_2 = [\mathsf{glb}(i_1 + i_2), \mathsf{lub}(i_1 + i_2)]$ |
| $*^\#$ | $i_1 *^\# i_2 = [\min(\mathit{product}), \max(\mathit{product})]$ where $\mathit{product} = \{\mathsf{glb}(i_1 * i_2), \mathsf{lub}(i_1 * i_2)\}$ |
| $=^\#$ | $i_1 =^\# i_2 = \begin{cases} \mathit{true} & \text{if } i_1 =_{\mathcal{I}} i_2 \\ \mathit{false} & \text{if } i_2 \neq_{\mathcal{I}} i_1 \end{cases}$ |
| $<^\#$ | $i_1 <^\# i_2 = \begin{cases} \mathit{true} & \text{if } i_1 \subset_{\mathcal{I}} i_2 \\ \mathit{false} & \text{if } i_2 \subseteq_{\mathcal{I}} i_1 \\ \top & \text{otherwise} \end{cases}$ |

As a remark: It can be shown, that every abstract operation *safely approximates* its concrete counterpart, i.e., the effects of an abstract operation comprise the effect of the corresponding concrete operation.

### *3.3 Abstract Simulation*

As its concrete counter-part in Section 2.3 the interpretation of the abstract semantics is based on labeled transition systems where nodes are configurations and the transitions themselves represent the i/o behavior as well as the abstract execution of single IL statements. Each execution of an IL program is then covered by a run in this transition system. *Abstract states* and *abstract configurations* are defined as follows:

**Definition 3.1** [abstract state] The global *abstract IL state* contains the values of all variables and is modeled as a mapping $\Sigma^\# : \mathit{Var} \cup \{cr\} \rightarrow D^\#$, where $D^\#$ stands for the union of all abstract data domains.

Again, we assume the values in the state to be type consistent and use $\sigma^\#$ as typical element of $\Sigma^\#$.

**Definition 3.2** [abstract configuration] An *IL configuration* $\gamma : Locs \times \Sigma^\# \times Mode$ of a program is characterized by

- a location $l \in Locs$,
- an abstract state $\sigma^\# \in \Sigma^\#$, and
- a configuration of type $Mode$.

The differences between abstract states or abstract configurations to their concrete counterparts are the different data domains. The labeled transition systems are defined accordingly:

**Definition 3.3** [abstract labeled transition system] With every IL program $P$ we associate an *abstract labeled transition system* $\mathcal{T}_P^\# = (\Gamma^\#, \gamma_0^\#, \rightarrow_\xi^\#)$, where

- $\Gamma^\#$ denotes the set of abstract configurations,
- $\gamma_0^\# \in \Gamma^\#$ is the initial configuration and
- $\rightarrow_\xi^\#$ is the transition relation between abstract configurations.

The initial configuration $\gamma_0^\#$ is given by $(l_0, \sigma_0^\#, \mathsf{I})$, where the initial state $\sigma_0^\#$ evaluates all Booleans to $\top$ and all integer intervals to top element of the lattice $[-\infty, +\infty]$. The operational rules are shown in Figure 3 specifying the labeled transition relation $\rightarrow_\xi^\#$ between system configurations.

These initial configurations are abstractions of the initial configuration for the concrete level. The operational rules a very similar to the ones of Section 2.3 and the semantics is again given by the set of all possible executions.

### *3.4 Abstract Interpretation*

While abstract simulation is a way to execute IL programs for set of inputs and tracks program behavior for certain paths, abstract interpretation approximates the program behavior for *all* possible inputs and *all* possible paths. Moreover, unlike abstract simulation it ensures termination of the analysis process. In order to do so, *acceleration techniques* are used to speed-up the convergence of the analysis. These accelerations provided a safe approximation of the program behavior, however, they often come with an additional loss of precision, i.e., can lead to further over-approximation.

More formal, from a fixed point perspective the semantics of any program $P$ is described by its least fixed point $\mu_P$. The abstract semantics $\mu_P^\#$ we developed, safely approximates the concrete one, while adding any acceleration $\nabla$ is a further approximation, i.e., $\mu_P^{\nabla\#}$ approximates $\mu_P^\#$.

The design of an appropriate way of acceleration is, e.g., discussed in [Cou78], [Bou92], and [Sch95]. Our approach is based on these investiga-

$$\frac{\sigma^{\#\prime} = \sigma^{\#}[\,\boldsymbol{x} \mapsto^{\#} \boldsymbol{v}\,] \qquad \boldsymbol{x}^{\#} = Var_{in}}{(l, \sigma^{\#}, \mathsf{I}) \rightarrow^{\#}_{?v} (l, \sigma'^{\#}, \mathsf{C}(instr(l)))} \quad \textbf{Input}$$

$$\frac{instr(l) = RET}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l, \sigma^{\#}, \mathsf{O})} \quad \textbf{RET}$$

$$\frac{\boldsymbol{v}^{\#} = [\![\boldsymbol{x}]\!]^{\#}(\sigma^{\#}) \qquad \boldsymbol{x}^{\#} = Var_{out}}{(l, \sigma^{\#}, \mathsf{O}) \rightarrow^{\#}_{!\boldsymbol{v}} (l_0, \sigma^{\#}, \mathsf{I})} \quad \textbf{Output}$$

$$\frac{instr(l) = LABEL \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#}, \mathsf{C}(instr(l')))} \quad \textbf{LABEL}$$

$$\frac{instr(l) = JMP \qquad l' \in Succ(l) \qquad instr(l') = LABEL}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#}, \mathsf{C}(instr(l')))} \quad \textbf{JMP}$$

$$\frac{instr(l) = JMPC \qquad l' \in Succ(l) cr^{\#}(\sigma^{\#}) = false \vee cr^{\#}(\sigma^{\#}) = \top \qquad instr(l') \neq LABEL}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#}, \mathsf{C}(instr(l')))} \quad \textbf{JMPCff}$$

$$\frac{instr(l) = JMPC \qquad l' \in Succ(l) cr^{\#}(\sigma^{\#}) = true \vee cr^{\#}(\sigma^{\#}) = \top \qquad instr(l') = LABEL}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#}, \mathsf{C}(instr(l')))} \quad \textbf{JMPCtt}$$

$$\frac{instr(l) = LD \qquad \sigma^{\#\prime} = \sigma^{\#}[\,op(l)^{\#} \mapsto^{\#} cr^{\#}\,] \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))} \quad \textbf{LD}$$

$$\frac{instr(l) = ST \qquad \sigma^{\#\prime} = \sigma^{\#}[\,cr^{\#} \mapsto^{\#} op^{\#}(l)\,] \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))} \quad \textbf{ST}$$

$$\frac{instr(l) = ADD \qquad \sigma^{\#\prime} = \sigma^{\#}[\,cr^{\#} \mapsto^{\#} cr^{\#} +^{\#} op^{\#}(l)\,] \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))} \quad \textbf{ADD}$$

$$\frac{instr(l) = MUL \qquad \sigma^{\#\prime} = \sigma^{\#}[\,cr^{\#} \mapsto^{\#} cr^{\#} *^{\#} op^{\#}(l)\,] \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))} \quad \textbf{MUL}$$

$$\frac{instr(l) = NOT \qquad \sigma^{\#\prime} = \sigma^{\#}[\,cr^{\#} \mapsto^{\#} \neg^{\#} cr^{\#}\,] \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))} \quad \textbf{NOT}$$

$$\frac{instr(l) = AND \qquad \sigma^{\#\prime} = \sigma^{\#}[\,cr^{\#} \mapsto^{\#} cr^{\#} \wedge^{\#} op^{\#}(l)\,] \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))} \quad \textbf{AND}$$

$$\frac{instr(l) = LT \qquad \sigma^{\#\prime} = \sigma^{\#}[\,cr^{\#} \mapsto^{\#} cr^{\#} <^{\#} op^{\#}(l)\,] \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))} \quad \textbf{LT}$$

$$\frac{instr(l) = EQ \qquad \sigma^{\#\prime} = \sigma^{\#}[\,cr^{\#} \mapsto^{\#} cr^{\#} =^{\#} op^{\#}(l)\,] \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))} \quad \textbf{EQ}$$

Fig. 3. Abstract operational semantics

tions, it uses the abstract semantics as introduced in the previous section and just adds an acceleration as described in [Huu03]. Due to space limitations we do not go into detail here.

Instead, consider the example of Figure 4. It shows an IL program with a single input variable `x`. It works as follows: In the beginning `x` is set to 1 and, then, within a loop successively incremented to 10. Once it reaches 10 the loop is left and the program terminated.

The two columns to the very right show the abstract interpretation result for the abstract values of *cr* and `x`. Note, since we do not have any information about the initial input value the possible value of `x` at line 0 is within $[-\infty, +\infty]$. At lines 7 and 8 the value of `x` is compared to 10. If strictly less than the loop is entered once more. Therefore, at line 3 we have the information that the value of `x` can be anywhere between $[1, 9]$. Moreover, we know at line 9 that `x` must have the value 10 and the *cr* is equal to *false*.

This is a simple example without any over-approximation. However, if we increment `x` by 2 instead of 1 within the loop, our analysis would not be able to reveal that even numbers never occur. Further over-approximations occur when the jump condition cannot be used to give an upper approximation of the variable values.

| location | program | | $cr^{\#}$ | $x^{\#}$ |
|---|---|---|---|---|
| | `VAR_INPUT` | | | |
| | | `x:INT;` | | |
| | `END_VAR` | | | |
| 0 | | | $\langle\top,$ | $[-\infty, +\infty]\rangle$ |
| 1 | | `LD 1` | $\langle[1, 1],$ | $[-\infty, +\infty]\rangle$ |
| 2 | | `ST x` | $\langle[1, 1],$ | $[1, 1]\rangle$ |
| 3 | `label:` | | $\langle\bot,$ | $[1, 9]\rangle$ |
| 4 | | `LD x` | $\langle[1, 9],$ | $[1, 9]\rangle$ |
| 5 | | `ADD 1` | $\langle[2, 10],$ | $[1, 9]\rangle$ |
| 6 | | `ST x` | $\langle[2, 10],$ | $[2, 10]\rangle$ |
| 7 | | `LT 10` | $\langle\top,$ | $[2, 10]\rangle$ |
| 8 | | `JMPC label` | $\langle\top,$ | $[2, 10]\rangle$ |
| 9 | | `RET` | $\langle false,$ | $[10, 10]\rangle$ |

Fig. 4. IL example with abstract interpretation result

## 4 Homer – a Checker for IL Programs

We implemented the abstract interpretation framework for IL into a prototype tool called HOMER. The abstract domains are as introduced and the used abstract semantics is as described before. In this section we present a number of generic properties that can be checked for IL programs. If not otherwise mentioned the checking is done on the abstract interpretation results.

*Range violation*
HOMER checks whether an operation violates maximal integer bounds. Violating means that, e.g., a subtraction with a positive value takes place on variables already approximated by $-\infty$ to their lower bound or addition to an upper bound of $+\infty$. Such an error would occur at the first `ADD` in Figure 4 if the input variable would not be set to 1 in the beginning.

*Invariant conditional jumps*
A conditional jump is called *invariant* if its jump condition is either always *true* or always $false$. This means, one alternative is never taken which might exhibit a flaw in the program. Replacing `LT 10` by `GE 1` in Figure 4 would provoke this error.

*Unreachable code*
Code is unreachable if there is no program execution ever executing it. In terms of IL language, this means, there are (conditional) jumps that prevent the control flow reaching every line of code and instead always skip some lines. Hence, these code fragments will never be executed.

There are two possibilities for unreachable code: One, there is simply a combination of `JMP` operators such that some lines are excluded from program execution and two, there are some invariant `JMPC` or `JMPCN` operations producing the same effect. This can be uncovered by a simple reachability analysis once the abstract interpretation is completed.

Replacing `LT 10` with `GE 1` in Figure 4 makes line 9 unreachable, since control would loop forever. This example is also a particular instance of the next property.

*Infinite loops*
To detect infinite loops it is helpful to analyze the topological structure of loops in the program. If we take into account the results of the abstract interpretation process, we have to search for strongly connected components which cannot be left.

*Type mismatched*
Type checking IL programs is a special case of abstract interpretation where the abstract domain is given by the possible types and abstract operations describe the changes.

*Redundant jumps*
A jump statement (`JMP`, `JMPC`, `JMPCN`) is redundant if the jump target is the next statement in the control flow.

*Redundant statements*
There are various combinations of redundant statements. In particular, each load statement (`LD`, `LDN`) should be preceded by a store statement (`ST`, `STN`, `S`, `R`) or a conditional jump (`JMPC`, `JMPCN`); if it is not, the code before the load statement is unused, since the old value of *cr* is discarded without having influenced variables or the program flow. Moreover, between two store statements to the same variable there should be some operations modifying *cr*.

These are just some examples of properties that can be checked automatically modulo some abstraction. It is part of future work to investigate on further ones.

The prototype is implemented in OCaml [CMP02] and primarily aims at testing the proposed methods and analyses. It is not optimized for speed, and memory consumption is high, since every program location still stores the information of all abstract values at that location. However, a case study of roughly 2000 lines of code with about 100 variables takes nearly 20 seconds to be analyzed, which is promising when having the potential for speed-up in mind.

While speed for interval abstraction appears to be a minor issue, a high number of false alarms due to over-approximation is more a concern. To reduce false alarms we suggested a solution based on selective constraint solving in [Huu03], this is, however, not yet implemented.

## 5 Conclusions

In this work we presented a formal operational semantics for IL programs. Moreover, we developed an abstract counterpart of this semantics which allows approximating program simulation for possibly infinite sets of inputs within one simulation run. We also extended this framework to an abstract interpretation analysis, as implemented in our tool Homer. The advantage of the proposed methods is that they can be used by PLC programmers not familiar with formal methods.

One direction for future work is to develop a tool for guided abstract simulation. Up to now we explore path non-deterministicly whenever there is more than one branching possibility. However, often it is of interest in following particular paths and exploiting jump conditions to constrain variable values for these paths.

Moreover, more work should be put in exploring different abstract domains for the analysis of IL code. The interval based abstraction proposed and implemented right now is good for range checking, but lacks precision for other common error such as division by zero. Moreover, the current abstraction does not take any relations between different variables into account. On the other hand, structures such as octagons or, more general, polyhedra [CH78] approximate the concrete space incorporating relationships between variables. Sophisticated methods take also linear [Gra91] or trapezoid linear congruences [Mas92] into account. It remains to explore which is the most suitable one for IL analysis. Moreover, this effort should be driven by the investigation on further generic properties. Hopefully this will also lead to advances in static analysis methods.

## References

[Bau98] N. Bauer. Übersetzung von Steuerungsprogrammen in formale Modelle. Master's thesis, University of Dortmund, 1998.

[Bou92] François Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite.* PhD thesis, École Polytechnique, 1992.

[Bri67] G. Brinkhoff. *Lattice Theory.* American Mathematics Society, Providence, RI, 3rd edition, 1967.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CCL+00] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and Ph. Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. In *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'2000)*, pages 2449–2454, 2000.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[CMP02] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml.* O'Reilly, Paris, April 2002.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* PhD thesis, Université scientifique et médicale de Grenoble, France, 1978.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 493 of *LNCS*, pages 169–192, 1991.

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1:110–122, 1997.

[HM98] M. Heiner and T. Menzel. A Petri net semantics for the PLC language Instruction List. In *Proceedings of the International Workshop on Discrete Event Systems (WoDES)*, pages 161–166. IEE Control, 1998.

[HTLW97] H.-M. Hanisch, J. Thieme, A. Lüder, and O. Wienhold. Modeling of PLC behaviour by means of timed net condition/event systems. In *Proc. of IEEE Int. Symposium on Emerging Technologies and Factory Automation (EFTA '97)*, pages 361–369, 1997.

[Huu03] Ralf Huuck. *Software Verification for Programmable Logic Controllers*. PhD thesis, University of Kiel, April 2003.

[IEC98] International Electrotechnical Commission, Technical Committee No. 65. *Programmable Controllers – Programming Languages, IEC 61131-3*, second edition, November 1998. Committee draft.

[KBP+99] S. Kowalewski, N. Bauer, J. Preußig, O. Stursberg, and H. Treseler. An environment for model-checking of logic control systems with hybrid dynamics. In *Proc. IEEE Int. Symp. On Computer Aided Control System Design*, 1999.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[Mas92] François Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *ICS '92: Proceedings of the 6th ACM International Conference on Supercomputing*, ACM, pages 226–235, 1992.

[McM00] Kenneth L. McMillan. *The SMV system*. Carnegie Mellon University, November 2000. Manual for SMV version 2.5.4.

[MW99] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *Proceedings of the 11th Euromicro Conference on Real Time Systems*, pages 114–122. IEEE Computer Society, 1999.

[OY93] A. Olivero and S. Yovine. *KRONOS: A Tool for Verifying Real-Time Systems. User's Guide and Reference Manual*. Verimag, Grenoble, France, 1993.

[RK98] M. Rausch and B. Krogh. Formal verification of PLC programs. In *American Control Conference*, pages 234–238, June 1998.

[Sch95] Erik Schön. On the computation of fixpoints in static program analysis with an application to analysis of AKL. Master's thesis, School of Engineering Physics, Royal Institut of Technology, Stockholm, October 1995.

[Wil99] H.X. Willems. Compact timed automata for PLC programs. Technical Report CSI-R9925, University of Nijmegen, Computing Science Institute, 1999.