CEUR-WS.org/Vol-2571/CSP2019_paper_2.pdf

Modelling Programmable Logic Controllers in Refinement Calculus of Reactive Systems

Viorel Preoteasa, Timo Latvala, and Kimmo Varpaaniemi

Space Systems Finland

Abstract. We present a translation from languages for programmable logic controllers (PLC) into refinement calculus of reactive systems (RCRS). RCRS is a compositional formal framework for modeling and reasoning about reactive systems. RCRS is based on monotonic property transformers (monotonic functions from sets of infinite output traces to infinite input traces) and is implemented in the Isabelle theorem prover. PLCs are industrial digital computers adapted for controlling manufacturing processes. Our translation provides a formal semantics for these systems, and a framework to formally analyze them.

1 Introduction

In this paper we present a translation from languages for programmable logic controllers (PLC) into refinement calculus of reactive systems. PLCs are industrial digital computers designed for controlling manufacturing processes. They provide high reliability and ease of programming as well as fault diagnosis. Originally PLCs were developed to replace hard wired relays, timers and sequencers in the automobile manufacturing industry. Since then PLCs have been adopted as highly reliable automation controllers suitable for harsh environments.

The PLC programming languages standard IEC 61131-3 [14] specifies the graphical languages FBD (Function Block Diagram) and LD (Ladder Diagram), the textual languages IL (Instruction List) and ST (Structured Text), and common elements that consist of SFC (Sequential Function Chart) elements and various elements for data types, variables, resources, access paths, tasks, functions, function blocks and programs.

Refinement calculus of reactive systems (RCRS) is a compositional framework for modeling and reasoning about reactive (see [13]) systems. RCRS has been inspired from interface automata [7] and it has its origin in the theory of relational interfaces [24], but also from classic refinement calculus [1] and action systems formalism [2].

RCRS allows compositional modeling of input-output *non-deterministic* (for a given input, there could be different possible outputs) and *non-input-receptive* systems (some inputs may be illegal). Being able to model systems having these characteristics enables static analysis similar to type checking [24, 25].

The theory of relational interfaces allows also modeling of non-deterministic and non-input-receptive systems, but relational interfaces are limited to safety properties. One of the main motivations of RCRS has been to lift this limitation, and be able to model both safety and liveness properties. This has been achieved by using the powerful semantics of the refinement calculus (RC) [1]. RC is based on *monotonic predicate transformers* [8] and is a compositional modeling and verification formalism for sequential programs. RCRS uses *monotonic property transformers* (monotonic functions

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

from sets of output traces to sets of input traces), which are suitable for expressing dynamic behaviors. The theory of RCRS has been introduced in [22] and is thoroughly described in [20].

RCRS is implemented in the Isabelle/HOL proof assistant [17], and it provides a tool-set [10, 9] for analyzing RCRS models, as well as a translator from Simulink models into RCRS language. The translator has been formally verified [21] in Isabelle/HOL.

This paper presents an embedding of PLC programming languages into RCRS framework. As function block diagrams are closely related to hierarchical block diagrams (HBD) in general, and Simulink diagrams in particular, previous work relating HBDs and Simulink to RCRS is also applicable to PLCs. However, one important feature of PLC programming languages is the modeling of error conditions like overflows and divisions by zero, and the focus of this paper is to show how RCRS combined with the powerful typing system of Isabelle/HOL can be used to model these aspects.

Although, PLCs are deterministic systems, having a formalism capable to express non-determinism is important for modeling the environment and for expressing specifications or properties of the system. Expressing non-input-receptive systems is equally important in the context of PLCs as it enables consistency checking, i.e. checking if the system suffers from runtime errors as overflows, divisions by zero, and others.

The main contributions of this paper are the following:

1. We present an embedding of atomic PLC components (arithmetic, logic, timers) into RCRS and we provide different mechanisms for handling error situations.
2. We show how a concrete example, expressed as a ladder logic diagram, can be translated into RCRS and we show how RCRS can be used to prove properties of this example.

As a consequence of our work, RCRS framework can be used for PLC systems. We introduce a formal mechanized semantics for PLC programming languages, and we can (symbolically) execute these systems, we can check consistency and refinement (verification of properties). We can also generate Python code that can be used to run the PLC program.

The Isabelle theories for the results presented in this paper are available from https://megamart.ssf.fi/rcrs/plc.zip.

2 Related Work

Halang, Krämer and Völker [12] have introduced a run-time environment for high integrity software represented by functional logic diagrams, and have developed a formal correctness proof of a functional block occurring in the design of *emergency shutdown systems* using Isabelle/HOL. Krämer and Völker [15, 26] have extended this research with methods for verification and validation of behavioral correctness and functional safety of PLC programs, with support to the languages FBD, SFC and ST. The semantics employed by these approaches allows specifying non-deterministic systems, however it does not allow non-input-receptive systems. It is also unclear if the approach is compositional and if it can be used for symbolic execution and code generation.

Newell, Pang, Tremaine, Wassyng and Lawford [16] have presented a translation from function block diagrams to the PVS theorem prover. This formalism seems also to allow non-determinism, but it does not allow modeling non-input-receptive systems. The translation from [16] cannot be used for symbolic execution or for code generation.

Barbosa and Dérb [3] use the B method for formal verification of PLC programs. In this approach. it is possible to specify and verify some specific properties expressed in LTL, but there is no support for LTL properties in general.

For each of the PLC programming languages FBD, IL, LD, SFC and ST standardized by IEC 61131-3 [14], the model checking survey by Ovatman et al. [18] identifies several tools that proceed via translation from that language. Interestingly, the model checking approach by Darvas et al. [6] uses IL as an intermediate language, whereas Pavlović and Ehrich [19] consider intermediate use of IL as a scalability problem.

3 Refinement calculus of reactive systems

In this section we introduce the RCRS language that we use for modeling the PLC systems. Since RCRS is implemented in Isabelle/HOL, we use a mathematical language very close to the language supported by Isabelle/HOL. Isabelle/HOL is a general purpose interactive theorem prover, implementing higher order logic (simple typed lambda calculus). In Isabelle/HOL we have type variables `'a`, type constants `nat`, `int`, `real`, function types `'a ⇒ 'b`, and abstract data types. The basic Isabelle terms are constructed from constant and variables using function application `(f x y) = ((f x) y)` (function `f` applied to `x` applied to `y`), and lambda abstraction `(λ x . Suc (Suc x))` – the function mapping `x` into `Suc (Suc x)`. Typing of terms can be specified `(f::nat ⇒ 'a)`, or it can be inferred. The type inference for a term produces the most general type such that the term is well typed. For example the inferred types for the term `(f x (g f))` are `(f::'a ⇒ 'b ⇒ 'c) (x::'a) (g::('a ⇒ 'b ⇒ 'c)⇒ 'b)`. Binary operators `(+, -, ∧, ...)` are functions with two arguments, and they have an infix syntax `(x + y)`.

In RCRS we model reactive systems that take as input infinite traces of values and produce as output infinite traces of values as monotonic property transformers. These are monotonic functions from sets of output traces to sets on input traces, and they have a weakest precondition interpretation [8]. A property transformer S applied to a set of output traces q returns the set of all input traces from which the execution is guaranteed to produce a trace from q. This formalization allows modeling of *non-deterministic* systems as well as systems that are *non-input-receptive* (there are inputs that cannot be handled by the system). The complete treatment of these concepts is available in [20, 22]. Here we introduce some concepts that we use in this paper.

We may use *linear temporal logic* (LTL), or other formalisms on infinite traces to define specification of reactive systems, but concrete systems, that can be implemented, are working in steps, and they maintain a state that is continuously updated. We call these *state transitions systems* (STS), and we model them as *monotonic predicate transformers*, mapping sets of output values to sets on input values with a similar weakest precondition interpretation.

For example a summation system that at step n outputs the sum of all inputs up to step $n - 1$ can be defined in RCRS as

```
definition summation = [- x, s ⇝ y: s, s': x + s -]
```

This system has input `x` and initial `s`, and produces output `y` and next state `s'`. Asumming that the initial state is $s_0 = 0$, and the input trace is $x_0, x_1, x_2, \ldots$, the state trace is $s_0 = 0, s_1 = s_0 + x_0 = x_0, x_2 = s_1 + x_1 = x_0 + x_1, \ldots$, and the output trace is $y_0 = s_0 = 0, y_1 = s_1 = x_0, y_2 = s_2 = x_0 + x_1, \ldots$.

The notation `[- x, y, s, t ⇝ z: x + y * x, s′: s + 1, t′: t + y -]` is the *deterministic update* statement and it introduces a system with inputs `x, y` and current state `s, t`, output `z = x + y * x` and next state `s′ = s + 1, t′ = t + y`. Non-deterministic systems are defined using the *non-deterministic update statement* `[: x, y, (s::nat) ⇝ z, s′ . z > x + y ∧ s < s′ < s + 5 :]`. The output `z` is chosen such that is greater than `x + y`, and next state `s′` is chosen between `s + 1` and `s + 4`. Systems that are not input receptive are defined using assert statements: `{. x, s . 0 ≤ x ≤ s .}`. If input `x` is between 0 and current state `s`, then this system behaves as skip, otherwise it fails (the input in this case is not valid).

In addition to the basic statements `[- -], [: :], {. .}`, we introduce also *serial*, *parallel* and *feedback* compositions, represented graphically in Figure 1.

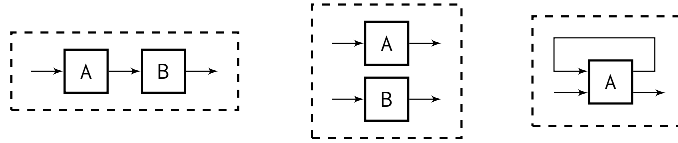

Fig. 1. Graphical representation of composition operators.

The serial composition of `A` and `B` is denoted by `A o B`. In this composition, the output of `A` becomes the input of `B`, and output type of `A` must match input type of `B`.

We model a square root component with input `x` as a non-input-receptive system that fails when `x < 0`. For this we use the serial composition:

```
definition Sqrt = {.x . x ≥ 0.} o [- x ⇝ y: sqrt x -]
```

If `x < 0`, then `Sqrt` fails, otherwise it produces output `y = sqrt x`, where `sqrt::real ⇒ real` is Isabelle's square root function. An important consequence of this modeling is that if the input of `Sqrt` is provided by another component, for example a non-deterministic system `A = [:u ⇝ x. x ≥ u + 1:]`, then the condition on the input of `Sqrt` imposes a condition on the input of the composition:

```
A o Sqrt = {.u. u + 1 ≥ 0.} o [: u ⇝ y . y ≥ sqrt (u + 1) :]
```

A detailed discution of this can be found in [20, 22].

The parallel composition of `A` and `B` is denoted by `A ** B`. The input of the parallel composition is the pair `(x,y)` where `x` and `y` are the inputs of `A` and `B`, respectively, and the output is the pair `(u,v)`, where `u` is the output of `A` for input `x`, and `v` is the output of `B` for input `y`. For example we have:

```
({.x. x > 0.} o [-x ⇝ y: 2 + x-]) ** ({.u. u < 0.} o [:u⇝ v. v > u:])
  = {.x, u. x > 0 ∧ u < 0.} o [:x, u ⇝ y, v. y = 2 + x ∧ v > u:]
```

The result of composing in parallel a deterministic component with a non-deterministic one is overall non-deterministic.

The feedback composition of a system `A` is denoted `feedback A`, and it connects its first output in feedback to its first input.

```
feedback [- x, y, z ⇝ u: y + z, v: x + y, w: 2 * x * z -]
  = [- y, z ⇝ v: (y + z) + y, w: 2 * (y + z) * z -]
```

In this example, the output `u = y + z` does not contain variable `x` and the feedback is simply the substitution of `x` by `y + z` in outputs `v` and `w`. In practical situations in PLCs this is always the case, and as we will see in Section 5 sometimes we need to add

delays in order to enforce this property. Designing a feedback operation in the context of non-deterministic and non-input-receptive systems is a non-trivial problem, and a treatment of this subject can be found in [23].

The summation example introduced earlier can be expressed using the RCRS operators applied to some simpler components as shown in Figure 2. This system can be expressed in RCRS using the declaration `simplify_RCRS`:

```
simplify_RCRS "summation′ = feedback([- u, x, s ⇝ (u, x), s -] o
  (ADD ** SKIP) o DELAY o (SPLIT ** SKIP) o [- (v, y), s' ⇝ v, y, s′ -])"
```

This declaration, in addition to defining `summation′`, also simplifies the result automatically producing a lemma stating the equality between `summation′` and its simplified version:

```
summation′ = [- x, s ⇝ y: s, s′: x + s -]
```

The simplification is based on equality of predicate transformers. In the definition of `summation′` we use the following atomic components:

```
definition "ADD = [- a, b ⇝ c: a + b -]"
definition "DELAY = [- a, s ⇝ b: s, s′: a -]"
definition "SPLIT = [- a ⇝ b: a, c: a -]"
```

Where `ADD` and `SPLIT` are stateless components, performing addition and splitting of the input, respectively, and `DELAY` is a stateful component working in steps, similarly to the `summation`. The component `DEALY` delays its input with one step. Please note

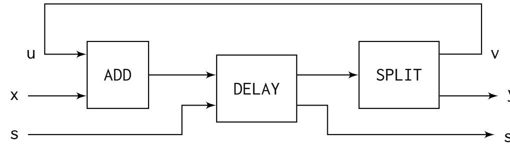

Fig. 2. Summation as composition of basic components.

that the names of the variables in the atomic components `[- -]`, `[: :]`, and `{. .}` are bound, and serial and feedback compositions are not performed based on names but require matching types.

The last construct that is needed for modeling PLCs is an operator `DelayFeedbackInit` which applied to an STS (a transition from input and current state to output and next state) returns a reactive system working on infinite sequences on input values, and producing infinite sequences of output values. For the `summation` system we have:

```
simplify_RCRS "summation_iter = DelayFeedbackInit 0
    ([-s, x ⇝ x, s-] o summation o [-s', y ⇝ y, s'-])"
```

where `0` is the initial value for the state `s`, and the two *switches* `[-s, x ⇝ x, s-]` and `[-s’, y ⇝ y, s’-]` are needed because `DelayFeedbackInit` requires the state variables to be first . The operator `DelayFeedbackInit` is formally defined in [23, 20].

To symbolically execute the summation system all we need is to use Isabelle’s `value` construct:

```
value "((func summation_iter) x 4)::nat"
```

which returns `x 0 + x 1 + x 2 + x 3`, where `func` returns the function of the statement `[- -]` (`func [- x, y ⇝ x + 2 * y -] (a, b) = a + 2 * b`).

4 Modeling basic PLC functions in RCRS

In this section we show how to model basic PLC functions as STS components in RCRS. We model these components as defined by the PLC standard IEC 61131-3 [14], but we also model a retentive timer (RTO). The standard IEC 61131-3 is very precise about the data types available in PLC programs, and describes in details the behavior of PLC functions in the case of error conditions (overflows, divisions by zero, ...).

4.1 PLC data types

In this subsection we show how to model the numeric data types for storing Boolean, integer and floating point values. In addition to the Boolean type (`BOOL`), PLC standard defines a collection of different sized integer types, both signed (`SINT` - short integer) and unsigned (`USINT` - unsigned short integer), as well as single and double sized floating point numbers (`REAL` and `LREAL`). Additionally, PLC standard introduces some generic data types. For example `ANY_NUM` is a generic data type subsuming any numeric types and `ANY_REAL` subsuming the real types and `ANY_INT` subsuming the integer types.

In Isabelle all numerical operators are polymorphic and they are defined using constructive type classes [11]. A type class on an arbitrary type variable `'a` introduces a series of abstract operations and their assumptions as well as properties of these operations based on the assumptions. Next Isabelle specification introduces the class of a semigroup with an operation plus (+) satisfying the associativity property.

```
class semigroup =
   fixes plus:: "'a ⇒ 'a ⇒ 'a" (infixl "+" 65)
   assumes add_assoc: "(a + b) + c = a + (b + c)"
```

Concrete operations on concrete types like integer, real and natural numbers are introduced as instantiations of these classes where each operation has a concrete definition, and each assumption must be proved. For example the concrete type `nat` of natural numbers is made a semigroup instance by providing a definition for the plus operation and by proving the semigroup assumption:

```
instantiation nat :: semigroup begin
  fun plus_nat where "0 + n = n" | "Suc m + n = Suc (m + n)"
    instance proof ... end
```

The type classes provide a very powerful mechanism for reusability. Many properties common to different numeric types are proved at the abstract class level, and they become available for concrete types via the instantiations.

The fixed size integers and their operations are implemented in the Isabelle library and some additional operations are implemented in the AFP entry [4]. The type of integers that can be represented on n bits is implemented by the Isabelle type (n `word`). This type implements both signed and unsigned integers. Signed and unsigned arithmetic operations are also implemented on (n `word`). Some of these operations (addition, subtraction, multiplication) are the same for signed and unsigned integer, while other operations like comparisons and division are different. Overflow conditions are also different for signed vs unsigned integers. Because some operations are different for signed and unsigned integers we introduce a new type of signed integers isomorphic with the type (n `word`):

```
typedef (overloaded) 'n sword = "UNIV::'n word set"
```

and we also lift all operations and properties from (n word) to (n sword).

The type of an arithmetic expression (a + b) * 12 is 'a::{plus,times,numeral}, where 'a is a type variable that belongs to classes plus (introducing the + operator), times (introducing the * operator), and numeral (introducing the numeral constants – 0,1,2,...). However, if we restrict some sub-term to be of some specific type, then the entire expression will be of this type. Moreover, the operations will be the arithmetic operations as defined for the specific type. For example if we specify that the expression 21 + 11 has type int ((21::int) + 11), then + is the expected addition operation for integers, and (21::int) + 11 = 32 ≠ 0 as expected. If we specify that this expression is of type 4 word (unsigned integers represented on 4 bits), then (21::4 word) + 11 = 0.

Unlike regular programming languages, PLC standard specify the possibility for functional blocks to have Boolean outputs that are true when the arithmetic operations overflow. To capture this we introduce a new class for the overflow operations:

```
class overflow =
  fixes overflow_add:: "'a ⇒ 'a ⇒ bool"
  fixes overflow_sub:: "'a ⇒ 'a ⇒ bool"
```

together with the instantiations to signed integers:

```
instantiation word :: (len) overflow
    definition "overflow_add a b = (uint a + uint b ≠ uint (a + b))"
    definition "overflow_sub a b = (uint a - uint b ≠ uint (a - b))"
```

where uint is the mapping from unsigned integers to unbounded integers.The addition operation overflows for unsigned integers a and b if the result of the addition of a and b as bounded unsigned integers is different from the addition of a and b as unbounded integers. Similarly it works for signed integers, and for the other operations.

4.2 Arithmetic Functions

We show how to model the arithmetic ADD function depicted in Figure 3. All other functions are modeled in a similar manner. In general a function may have in addition to the proper inputs and outputs, an input EN (Enable) and an output ENO (Enable Out). If input EN is true, then the function output should be computed, and ENO should be true if there are no errors (overflows, conversion errors, ...). If EN is false, then ENO must be false, and the function output is not specified. Different manufacturers may chose different implementations in this case. In case when EN is false we model the output to be non-deterministic.

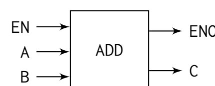

Fig. 3. Addition function.

```
definition "ADD = [: EN, A, B ⇝ ENO, C .
  if EN then ENO = ¬ overflow_add A B ∧ C = A + B else ENO = False :]"
```

In this definition, if EN is true, then the output of the function is A + B and ENO is true if there is no overflow when adding A and B. If EN is false then ENO is false, and the output of the function can be any value.

The advantage of this non-deterministic definition, is that any manufacturer specific implementation is a refinement of our definition, and as a consequence, a system that is correct using our non-deterministic definition would be correct when using any deterministic implementation. However, if we know that we work with a specific implementation, then we could use a deterministic model. For example the `ADD` function which outputs the default value 0 when `EN` is false is defined by:

```
definition "ADD′ = [- EN, A, B ⇝
     EN ∧ ¬ overflow_add A B, if EN then A + B else 0 -]"
```

The version of the `ADD` function with the output `ENO` is useful in practical situations when overflows can occur and the system needs to respond appropriately. However, in situations when the system is designed such that no overflows occur, then we want to make sure that this is the case. For these cases we can use a non-input-receptive version of the addition function:

```
definition "ADDₙ = {. EN, A, B . ¬ EN ∨ ¬ overflow_add A B .}
    o [: EN, A, B ⇝ C . ¬ EN ∨ C = A + B :]"
definition "ADD′ₙ = {. A,B . ¬ overflow_add A B .} o [- A,B ⇝ A + B -]"
```

If these functions occur in complex systems, then the local overflow conditions imposes global conditions on the inputs of the overall system as discussed in the `Sqrt` example in Section 3. If the global input conditions are satisfied, then we will not encounter overflow in the addition function.

4.3 Timers

We introduce the definition of a retentive timer (RTO) that we will use later in our example. As compared to more standard PLC timers, the RTO retains the accumulated time when the timer is disabled, and the accumulated time is set to zero only by a reset input signal.

```
definition "RTO = [- Enable, Reset, Preset, Accum ⇝
    Enabled: Enable, Done: Enable ∧ Preset ≤ Accum,
    Timing: Enable ∧ Accum < Preset, Accum′: if Reset then 0 else
        (if Enable ∧ Accum < Preset then nxt Accum else Accum) -]"
```

Intuitively, `nxt t` is adding to `t` the time duration of a PLC execution step. The RTO is a stateful component, where the accumulated time is the state of the component: `Accum` is the current state, and `Accum′` is the next state.

To accommodate different vendor specific timings, we introduce a class `nxt` with one operation `nxt`:

```
class nxt = fixes nxt: "′a ⇒ ′a"
```

For a time `t`, `nxt t` is new time that corresponds to adding the duration of one step to the time `t`. If we want to use a discrete time where one unit of time corresponds to one step of the system, then we instantiate the class `nxt` as nat where `nxt` is the successor function:

```
instantiation nat :: nxt definition "nxt = Suc"
```

For times based on real values with step increments of `1/n` we introduce the type

```
typedef ′n::len time = "UNIV::real set"
```

as a copy of the type real and we define the instantiation:

```
instantiation time :: (len) nxt
  definition "nxt (x::'n time) = x + 1 / real_to_time (len_of TYPE('n))"
```

This enables for example types of the form `(2 time)` where `nxt t = t + 1/2` or `(10 time)` where the `nxt t = t + 1/10`. With this approach, we can have a single definition, vendor independent, for timers, and by instantiating it for different types we obtain different implementations.

5 Control lights in sequence example

In this section we show on an example how to translate a PLC system defined by ladder logic diagram into RCRS. We use a simplified version of the system for controlling lights in sequence from [5]. The example is given in Figure 4. After a push to the Start button it starts a cycle of turning on Light 1 for 5 time units, followed by turning on Light 2 for 5 time units, and so on. When pressing Stop button, the active light turns off, but the current internal state is preserved, and a new start will continue from the current state.

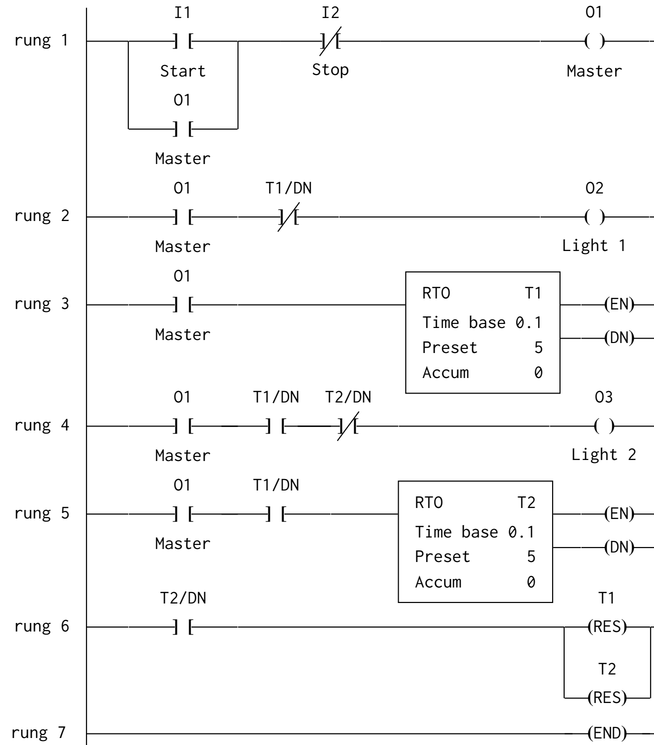

Fig. 4. Control lights in sequence.

The ladder logic diagram is organized in rungs, which have Boolean inputs (–] [–, –]/[–) and Boolean outputs (–( )–), as well as basic functions (timers, arithmetic functions, ...). The list of inputs and outputs for the lights system are: `I1` - Start input, `I2` - Stop input, `01` - Master coil output, `02` - Light 1 output, `03` - Light 2 output, `T1` - RTO for output `02`, `T2` - RTO for output `03`, and `RES` - Reset coil output for timers.

To model this system we use the definition of the RTO introduced before, and we use all tree operations of RCRS (serial, parallel, and feedback). The idea is to model all rungs as RCRS components A_1, A_2, ..., A_n and then connect them all in parallel and use feedback to connect the outputs to the corresponding inputs. In fact we do this construction incrementally. First we construct B_1 = FEEDBAK(A_1 ** A_2), next construct B_2 = FEEDBAK(B_1 ** A_3), and so on until we connect all components A_i. FEEDBAK in this context means connecting the outputs to the corresponding inputs based on their names in the ladder logic diagram. When we construct the feedback we check if there are instantaneous dependencies, and if there are, then we introduce unit delays.

For our example we start with the first rung:

```
definition "R1 = [-Start,Stop,Master⇝Master′: (Start ∨ Master) ∨ ¬ Stop -]"
```

This is the common industrial latching start/stop logic. When Start button is pushed it turns on the Master coil, which turns on also the input Master in parallel with Start. After this Master remains activated even if Start button is released. A push of the Stop button will deactivate the Master coil.

Already in this first rung the output Master′ is also used as input. We use the feedback operation to connect the output Master′ to the input Master, but we need to use also a unit delay since Master′ depends on Master.

```
simplify_RCRS "R1′ = feedback(
 [- Master, Start, Stop, S ⇝ (Start, Stop), (Master, S) -] o (SKIP ** DELAY)
   o [- (Start, Stop), (Master, S′) ⇝ (Start, Stop, Master), S′ -]
   o (R1 ** SKIP) o [- Master, S′ ⇝ Master, Master, S′ -])"
   "(Start, Stop, S)" "(Master, S′)" use (R1_def)
```

The result of simplify_RCRS is the definition of R1′ plus its simplification lemma:

```
   R1′ = [- (Start, Stop, S) ⇝ Master: (Start ∨ S) ∧ ¬ Stop,
        S′: (Start ∨ S) ∧ ¬ Stop -]
```

The definition of R1′ may seem complicated compared to its simplified version, by it provides a systematic and mechanical way of handling arbitrary rung definitions, and our tool reduces it automatically to its simplified form.

Next we define rung 2 of the system as R2, we connect R1′ with R2 in parallel, and we use feedback to connect the output Master of R1′ to the corresponding input of R2. Additionally we need also to split the output of Master of R1′ such that we can use it again in rungs 3, 4 and 5.

```
definition "R2 = [- Master, T1_DN ⇝ O2: Master ∧ ¬ T1_DN -]"
simplify_RCRS "B1 = feedback (
 [-Master, Start, Stop, T1_DN, S ⇝ (Start, Stop, S), (Master, T1_DN) -]
 o(R1′ ** R2) o [- (Master, S'), Light1 ⇝ Master, Master, Light1, S′ -])"
  "(Start, Stop, T1_DN, S)" "(Master, Light1, S′)" use (R1′_simp)
```

This gives us the simplified version of B1:

```
B1 = [- (Start, Stop, T1_DN, S) ⇝ Master: (Start ∨ S) ∧ ¬ Stop,
   Light1: (Start ∨ S) ∧ ¬ Stop ∧ ¬ T1_DN, S': (Start ∨ S) ∧ ¬ Stop -]
```

Here we can see already that Light 1 is on if the system is started, Stop button is off, and if the timer T1 is not done.

Continuing this approach for rungs 3, 4, 5, and 6 we obtain in the end the system:

```
LightTrs = [- (T1_Acc, T2_Acc, S), (Start, Stop) ⇝
  let Master = (Start ∨ S) ∧ ¬ Stop in ((
     T1_Acc': if Master ∧ T1_Acc ≥ T ∧ T2_Acc ≥ T' then 0 else
        if Master ∧ T1_Acc < T then nxt T1_Acc else T1_Acc,
     T2_Acc': if Master ∧ ¬ T1_Acc < T ∧ T2_Acc ≥ T' then 0 else
        if Master ∧ T1_Acc ≥ T ∧ T2_Acc < T′ then nxt T2_Acc else T2_Acc,
     S': Master), (
     Light1: Master ∧ T1_Acc<T, Light2: Master ∧ T1_Acc≥T ∧ (T2_Acc<T'),
     T1_EN: Master, T2_EN: Master ∧ T1_Acc ≥ T ))-]
```

and also the final reactive system:

```
  simplify_RCRS "Lights = DelayFeedbackInit (0,0,0,False,False) LightTrs"
      "(x)" "(y)" use(LightTrs_simp iter_simps)
```

In this final system, input x is an infinite trace with pairs of values for the start and stop inputs, and otput y is an infinite trace of tuples of Light1, Light2, T1_EN, and T2_EN.

For input x, with `x 0 = (True,False)` and `x i = (False,False), i > 0`, we can evaluate `Lights x 0`, `Lights x 1`, ... and we can observe the intended behavior. We can also prove that after a stop, the lights are off, but the state of the system is preserved:

```
    lemma "func LightTrs ((T1_Acc, T2_Acc, S), Start, True)
          = ((T1_Acc, T2_Acc, False), False, False, False, False)"
       by (simp add: LightTrs_simp func_update)
```

6 Conclusions

We have presented an embedding of PLC programming languages into RCRS framework, and we have applied it to a ladder logic diagram. We also have shown how RCRS, together with the class mechanism of Isabelle/HOL, can be used to efficiently model arithmetical operations that can overflow or generate run-time errors. This enables using all features of RCRS (refinement, consistency checking, symbolic execution, code generation) to PLC programs. Our work is mechanically verified in Isabelle/HOL.

In future work we plan to extend RCRS tool-set with new features for automatically proving consistency properties and properties specified using LTL. We will also use the RCRS tool-set in PLC projects at Space Systems Finland.

References

1. R.-J. Back and J. von Wright. *Refinement Calculus*. Springer, 1998.
2. R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, Jun 1989.
3. H. Barbosa and D. Déharbe. An approach using the b method to formal verification of plc programs in an industrial setting. In R. Gheyi and D. Naumann, editors, *Formal Methods: Foundations and Applications*, pages 19–34. Springer, 2012.
4. J. Beeren, M. Fernandez, X. Gao, G. Klein, R. Kolanski, J. Lim, C. Lewis, D. Matichuk, and T. Sewell. Finite machine word library. *Archive of Formal Proofs*, June 2016. http://isa-afp.org/entries/Word_Lib.html, Formal proof development.
5. M. K. Bhojasia. Plc program to control lights in a sequence (1), January 2017.
6. D. Darvas, I. Majzik, and E. B. Viñuela. Formal verification of safety PLC based control software. In E. Ábrahám and M. Huisman, editors, *Integrated Formal Methods, 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1–5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2016.

7. L. de Alfaro and T. Henzinger. Interface automata. In *Foundations of Software Engineering (FSE)*. ACM Press, 2001.
8. E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.
9. I. Dragomir, V. Preoteasa, and S. Tripakis. Compositional Semantics and Analysis of Hierarchical Block Diagrams. In *SPIN*, pages 38–56. Springer, 2016.
10. I. Dragomir, V. Preoteasa, and S. Tripakis. The Refinement Calculus of Reactive Systems Toolset. In *TACAS*, 2018.
11. F. Haftmann and M. Wenzel. Constructive type classes in isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, pages 160–174. Springer, 2007.
12. W. A. Halang, B. Krämer, and N. Völker. Formally verified building blocks in functional logic diagrams for emergency shutdown system design. *High Integrity Systems*, 1(3):277–286, 1995.
13. D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1985.
14. IEC. *Programmable controllers — Part 3: Programming languages*. International Electrotechnical Commission, Geneva, Switzerland, International standard IEC 61131-3, Second edition, January 2003.
15. B. Krämer and N. Völker. A highly dependable computing architecture for safety-critical control applications. *Real Time Systems*, 13(3):237–251, November 1997.
16. J. Newell, L. Pang, D. Tremaine, A. Wassyng, and M. Lawford. Translation of IEC 61131-3 function block diagrams to PVS for formal verification with real-time nuclear application. *Journal of Automated Reasoning*, 60(1):63–84, January 2018.
17. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
18. T. Ovatman, A. Aral, D. Polat, and A. O. Ünver. An overview of model checking practices on verification of PLC software. *Software & Systems Modeling*, 15(4):937–960, October 2016.
19. O. Pavlović and H.-D. Ehrich. Model checking PLC software written in function block diagram. In *Proceedings of Third International Conference on Software Testing, Verification, and Validation, ICST 2010, 7–9 April 2010, Paris, France*, pages 439–448. IEEE, 2010.
20. V. Preoteasa, I. Dragomir, and S. Tripakis. The Refinement Calculus of Reactive Systems. *CoRR*, abs/1710.03979, 2018.
21. V. Preoteasa, I. Dragomir, and S. Tripakis. Mechanically proving determinacy of hierarchical block diagram translations. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, pages 577–600, 2019.
22. V. Preoteasa and S. Tripakis. Refinement calculus of reactive systems. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10, Oct 2014.
23. V. Preoteasa and S. Tripakis. Towards Compositional Feedback in Non-Deterministic and Non-Input-Receptive Systems. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016.
24. S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A Theory of Synchronous Relational Interfaces. *ACM TOPLAS*, 33(4):14:1–14:41, July 2011.
25. S. Tripakis, C. Stergiou, M. Broy, and E. A. Lee. Error-Completion in Interface Theories. In *International SPIN Symposium on Model Checking of Software – SPIN 2013*, volume 7976 of *LNCS*, pages 358–375. Springer, 2013.
26. N. Völker and B. Krämer. Automated verification of function block-based industrial control systems. *Science of Computer Programming*, 42(1):101–113, January 2002.