Noname manuscript No.
(will be inserted by the editor)

The Refinement Calculus of Reactive Systems Toolset

Iulia Dragomir · Viorel Preoteasa · Stavros Tripakis

Received: date / Accepted: date

Abstract We present the Refinement Calculus of Reactive Systems Toolset, an environment for compositional formal modeling and reasoning about reactive systems, built around Isabelle, Simulink, and Python. The toolset implements the Refinement Calculus of Reactive Systems (RCRS), a contract-based refinement framework inspired by the classic Refinement Calculus and interface theories. The toolset formalizes the entire RCRS theory in about 30000 lines of Isabelle code. The toolset also contains a translator of Simulink diagrams, and a formal analyzer implemented on top of Isabelle. We present the main functionalities of the RCRS Toolset via a series of pedagogical examples, and also describe a larger case study from the automotive domain.

Keywords Block diagrams · Compositionality · Refinement · Contract synthesis · Formal verification · Theorem proving · Isabelle · Simulink

1 Introduction

The *Refinement Calculus of Reactive Systems* (RCRS) is a compositional framework for modeling and reasoning about reactive systems. One of the motivations behind RCRS has been to model and formally reason about industrial-strength systems, for instance, those modeled in the widespread Simulink environment by the Mathworks. RCRS has been inspired by component-based frameworks such as interface automata [9] and has its origins in the theory of relational interfaces [41], but also borrows from the classic *refinement calculus* [4].

RCRS allows compositional modeling of input-output systems which are both *non-deterministic* (meaning that for a given input, there could be many possible outputs) and *non-input-receptive* (meaning that some inputs are illegal). Being able to model systems that have both these characteristics in turn enables static analysis similar to type checking [41,43].

The ability to model systems which are both non-deterministic and non-input-receptive is also present in the theory of relational interfaces. But relational interfaces are limited to safety properties. The main motivation behind RCRS has been to lift this limitation, and to be able to describe both safety and liveness properties. This has been achieved by abandoning the relational semantics of relational interfaces and adopting the much more powerful semantics of the refinement calculus (RC) [4]. RC is a compositional modeling and verification framework for sequential programs. RCRS extends RC to reactive systems. One of the main ideas is to adapt the semantics of RC, which is based on *predicate transformers* [11], to *property transformers* which are suitable for expressing dynamic behaviors. The theory of RCRS has been introduced in [33] and is thoroughly described in [30].

This paper does not focus on the theory, but rather on the implementation of RCRS, called the *RCRS Toolset*. The RCRS Toolset, illustrated in Fig. 1, consists of the following:

– A full formalization of the RCRS theory in the Isabelle proof assistant [28].

Iulia Dragomir
GMV Aerospace and Defence SAU, Madrid, Spain
E-mail: idragomir@gmv.com

Viorel Preoteasa
Huld, Finland
E-mail: viorel.preoteasa@gmail.com

Stavros Tripakis
Northeastern University, Boston, MA, USA
E-mail: stavros@northeastern.edu

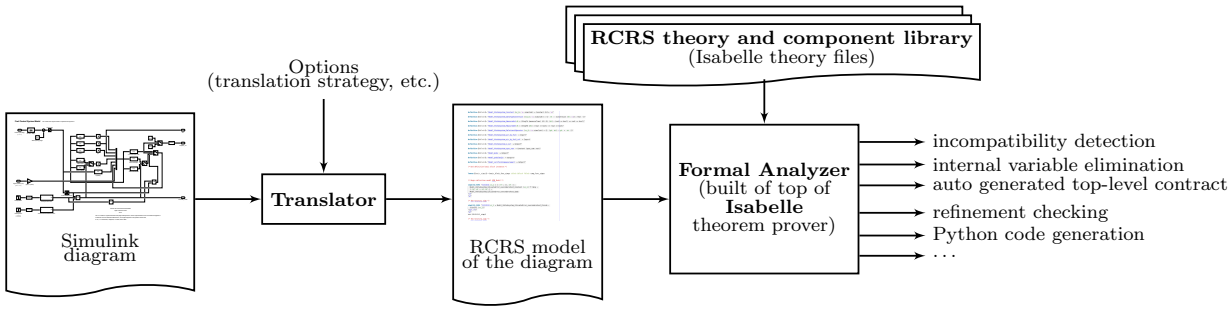


Fig. 1: The RCRS Toolset.

- A set of analysis procedures for RCRS models, implemented on top of Isabelle and collectively called the *Analyzer*.
- A *Translator* of Simulink diagrams into RCRS code.
- A *library* of basic RCRS components, including a set of basic Simulink blocks modeled in RCRS.

This paper revises and extends a shorter version presented previously in [14]. An archived and open-source, but out of date version of the RCRS Toolset is available in a figshare repository [15]. The latest version of the RCRS Toolset is distributed under the MIT license and can be downloaded from the RCRS web page: http://rcrs.gitlab.io/. The distribution provides all data (code and models) required to reproduce all the results of this paper.

To use the RCRS Toolset one needs the Isabelle 2018 proof assistant, available from https://isabelle.in.tum.de/. If one wishes to use the Simulink Translator, then one also needs Python 2.7. Python is also needed to execute any simulation code generated automatically by the Analyzer. Note that Simulink from the Mathworks is not required, unless one wants to build Simulink models. However, RCRS can be used independently from Simulink, since models can be written directly in RCRS as illustrated in the following sections.

2 Modeling Systems in RCRS

RCRS provides a language of *components* to model systems in a modular fashion. Components can be either *atomic* or *composite*.

2.1 Atomic Components

Atomic components can be described as input-output functions, input-output relations, symbolic transition systems with inputs and outputs, or property specifications (e.g., in linear temporal logic) with inputs and outputs. We give examples of all these types of atomic components in what follows. The semantics of atomic components are defined in terms of *monotonic predicate transformers* [11] or *monotonic property transformers* [33,30]. We do not define these semantic objects in this paper, and refer the reader to [30] instead.

Let us begin with some simple examples of components. These are written in the RCRS Isabelle syntax, so this is legal RCRS code (in what follows, blue text denotes RCRS/Isabelle keywords)[1]:

```
definition "Id = [: x ⇝ y . y = x :]"
definition "Add = [: (x, y) ⇝ z . z = x + y :]"
definition "Constant c = [: x::unit ⇝ y . y = c :]"
```

`Id` models the identity function: it takes input x and returns y such that $y = x$. `Add` takes a pair of inputs x and y and returns as output z the sum of x and y. `Constant` is parameterized by `c`, takes no input (equivalent to specifying its input variable to be of type `unit`, meaning the type that contains a single element), and returns an output which is always equal to `c`. The constraints between input and output variables, such as $y = x$, $z = x+y$, etc., are also referred to as the *contract* of the component, for reasons that will become clear in what follows.

All three components, `Id`, `Add`, and `Constant`, are *deterministic* (meaning that their output is unique, given the input) and *input-receptive* (meaning that all inputs are legal – modulo typing restrictions). Because these components are deterministic, we can also use a specialized *functional* syntax that RCRS provides to describe them, as follows:

```
definition "Id' = [- x ⇝ x -]"
definition "Add' = [- (x, y) ⇝ x + y -]"
definition "Constant' c = [- x::unit ⇝ c -]"
```

Each of the primed components above are semantically equivalent to their unprimed versions. More gen-

[1] In order to write special characters in Isabelle such as ⇝, one has to type the TeX corresponding command (e.g., \leadsto) and press the Tab key. Then the special character will be typed in the Isabelle jEdit interface as shown in Fig. 5.

erally, the component `[- x ⇝ f(x) -]`, where f is a function, is equivalent to `[: x ⇝ y . y = f(x) :]`. The reason for having the syntax `[: _ :]` is that it can be used also for relations (i.e., non-deterministic systems), as explained below. Using `[- _ -]` instead of `[: _:]` to describe deterministic systems aids the Analyzer to perform simplifications – see Section 5.

All components presented so far are *stateless*, i.e., "memory-less". RCRS can also describe *stateful* components, i.e., components with state. The way stateful components are modeled in RCRS is by specifying constraints on the input variables, output variables, *current state* and *next state* variables. An example is the `UnitDelay` component:

```
definition "UnitDelay =
       [: (x,s) ⇝ (y,s') . y = s ∧ s' = x :]"
```

In the above description, `s` is the current-state variable and `s'` is the next-state variable. What the specification of `UnitDelay` says is that the output y is always equal to the current state, and that the input x is equal to the next state. This effectively defines that the output at discrete time step $t + 1$ is equal to the input at time t. Note that this specification only defines the transition function and output function of `UnitDelay`. It does *not* define its initial state, neither the set of reachable states (which needs to somehow be computed iteratively, might be infinite, etc.). `UnitDelay` is deterministic, so it could more appropriately be described as:

```
definition "UnitDelay' = [- (x,s) ⇝ (s,x) -]"
```

We note that parentheses are not needed in the listing of input and output variables, so that the above could be also written as:

```
definition "UnitDelay' = [- x, s ⇝ s, x -]"
```

RCRS is a discrete-time framework. Continuous-time systems can be modeled by discretizing time. We do this in particular for all continuous-time Simulink blocks. We handle continuous-time blocks with a fixed-step forward Euler integration scheme. For example, Simulink's integrator block (which is deterministic) can be defined in two equivalent ways as follows:

```
definition "Integrator dt =
       [: x, s ⇝ y, s'. y = s ∧ s' = s + x * dt :]"
definition "Integrator' dt =
       [- x, s ⇝ s, s + x * dt -]"
```

All the components presented so far are input-receptive. As mentioned earlier, a powerful feature of RCRS is its ability to specify non-input-receptive components, i.e., components which reject some inputs as illegal. For example, consider the following alternative definitions of a square-root component, where `sqrt` defines the square root function for floats in Isabelle.

```
definition "ReceptiveSqrt =
         [: x ⇝ y . x ≥ 0 ⟶ y = sqrt x :]"
definition "SqrRoot =
         {. x . x ≥ 0 .}  o  [- x ⇝ sqrt x -]"
definition "NonDetSqrt =
         {. x . x ≥ 0 .}  o  [: x ⇝ y . y ≥ 0 :]"
```

`ReceptiveSqrt` is input-receptive and non-deterministic. This component accepts all inputs. If the input `x` is non-negative, then the output is the (unique) square root of `x`. If, however, `x` is negative, then the output can be anything, since the implication `x ≥ 0 ⟶ y = sqrt x` is satisfied for any `y` when `x < 0`.

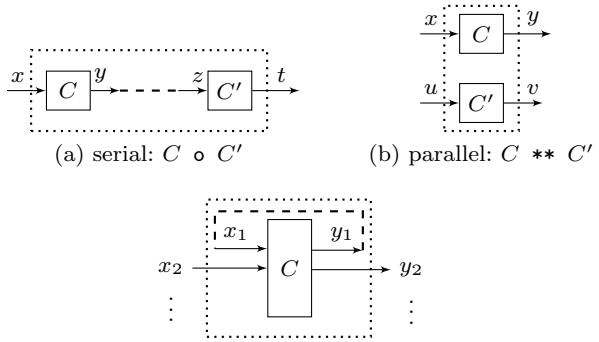
`SqrRoot` is non-input-receptive: its input `x` is required to satisfy `x≥0`. This is specified by the syntax `{. x . x ≥ 0 .}`. Formally, syntax of the form `{. _ .}` denotes a *monotonic predicate/property transformer* (MPT) of type *assert*, whereas `[: _ :]` and `[- _ -]` denote *demonic update* MPTs [30]. The combination of such MPTs can be seen as resulting in contracts that constrain the input variables. For example, the contract of `SqrRoot` can be seen as being the formula `x ≥ 0 ∧ y = sqrt x`, which constrains `x` to be non-negative. In contrast, the contract of `ReceptiveSqrt` is `x ≥ 0 ⟶ y = sqrt x`, which does not constrain `x` at all.

We also remark that `SqrRoot` may be considered non-atomic as it is defined as the serial composition (denoted `o`) of two MPTs. However, we include `SqrRoot` in the list of atomic components, because the pattern $P \circ Q$, where P is an assert MPT and Q is a demonic update MPT, is the standard way to define non-input-receptive atomic components in RCRS. We finally note that `SqrRoot` is deterministic (in essence, `SqrRoot` can be viewed as a partial function).

`NonDetSqrt` is a non-deterministic version of `SqrRoot`. Like `SqrRoot`, `NonDetSqrt` also requires its input to be non-negative. But contrary to `SqrRoot`, `NonDetSqrt` returns an arbitrary (although non-negative) output `y`, and not necessarily the square-root of input `x`. So `NonDetSqrt` is both non-input-receptive and non-deterministic.

As we shall see in Section 3, a refinement relationship exists between the above three square-root components.

One of the design goals of RCRS has been to be able to capture liveness properties. For this and other purposes, RCRS allows to describe components using the temporal logic QLTL, an extension of LTL with quantifiers [40,30]. For example, consider a component `A` which takes as input an infinite sequence of `x`'s and returns as output an infinite sequence of `y`'s. `A` re-



(a) serial: C `o` C' (b) parallel: C `**` C'

(c) feedback: `feedback`(C)

Fig. 2: The three composition operators of RCRS.

quires that its input `x` is infinitely often true, and in turn guarantees that its output `y` is infinitely often true. Therefore, `A` is both non-input-receptive and non-deterministic. In RCRS, `A` can be defined as follows:

```
definition "A = {. □◇ (λ x . x 0) .}
                o [: □◇ (λ x y . y 0) :]"
```

Let `X = (λ x . x 0)` and `Y = (λ x y . y 0)`. Then, `X` is a property (predicate) on a Boolean trace `x` which is true when `x` is true at time 0. `X` models the LTL formula `x` over a single atomic proposition `x`. Similarly, `Y` models the LTL formula `y`, but over the atomic propositions `x,y`. Property `Y` is true for traces `x, y` if `y` is true at time 0.

To be able to express LTL formulas without temporal operators like `X` and `Y` in a more compact form, we also introduced in Isabelle the binding `(ltl x y z . expr)`. Then, a state formula such as `x + y < 5 ∧ z` can be turned into an LTL formula by simply writing `(ltl x y z . x + y < 5 ∧ z)`. Using this binding, component `A` above can be rewritten as:

```
definition "A = {. □◇ (ltl x . x) .}
                o [: □◇ (ltl x y . y) :]"
```

2.2 Composite Components

Composite components are formed by composing other (atomic or composite) components using three composition operators, as illustrated in Fig. 2: C `o` C' (composition in series) connects outputs of C to inputs of C'; C `**` C' (composition in parallel) "stacks" C and C' "on top of each other"; and `feedback`(C) connects the first output of C to its first input. These three operators are the only composition primitives available in RCRS. They are sufficient to express any block diagram, as described in Section 6.

Parallel composition is a total operator, meaning that C `**` C' is always defined, no matter what C and C' are. But serial composition and feedback are partial operators.

C `o` C' is only defined when the type of the output of C matches the type of the input of C'. For example, `Add o SqrRoot` is legal, but `SqrRoot o Add` is not, because `SqrRoot` has one output whereas `Add` has two inputs. On the other hand, `(SqrRoot ** SqrRoot) o Add` is legal. `Id o Add` is also legal because `Id` is polymorphic.

Beyond the above typing requirements, there are also *semantic compatibility* requirements stemming from the receptiveness constraints that components impose on their input values. For example, consider the component

```
definition "Syst = (Constant -1) o SqrRoot"
```

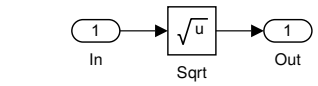
where we instantiate a constant with value -1 and connect it in series with the `SqrRoot` component defined above. This composition is a-priori legal, since `(Constant -1)` has a single output, `SqrRoot` has a single input, and their types match. But `Syst` should be semantically invalid, because `SqrRoot` requires its inputs to be non-negative and `(Constant -1)` outputs a negative number. Indeed, as we shall see in Section 3, RCRS can automatically simplify `Syst` and show that it is semantically equivalent to the MPT $\perp$ which captures all invalid components.

`feedback`(C) is defined when C has at least two inputs and at least two outputs. Morever, the types of the first input and of the first output must match. The semantics of feedback for non-deterministic and non-input-receptive systems is a non-trivial topic [34, 30] and beyond the scope of this paper. In the current implementation of the RCRS Toolset, feedback is generally well-defined only when the first output does not depend on the first input. This condition is satisfied when there are no *algebraic loops* in the system, as explained in Section 6.

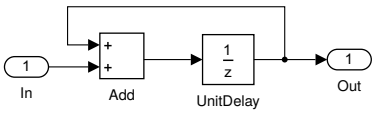
2.3 Running Examples

Consider the Simulink diagrams of Fig. 3. The diagram of Fig. 3a models a system which computes the square root of its input. The diagram in Fig. 3b models a system which computes the sum of all previously received inputs. That is, the output at step i is equal to the sum of all inputs received until and including step $i-1$, plus the initial value of the `UnitDelay`.

These diagrams can be captured in RCRS as shown in Fig. 4 (we will see in Section 6 how such RCRS code can be generated automatically using our Translator –



(a) Square root.



(b) Summation.

Fig. 3: Two Simulink diagrams.

for now we simply want to illustrate how such systems can be modeled in RCRS). The first four lines in each of the RCRS models define a theory, import the relevant files from the RCRS implementation and declare the results to be used during the analysis. Next, the atomic components of each model are defined: `Sqrt` in the first case, and `Add` and `Delay` in the second. Since the output of the `UnitDelay` must be fed to both `Add` and `Out`, we define the `Split` atomic component which duplicates its input value into two outputs (*fan-out*). Finally, the RCRS models for the two diagrams are defined by composing the atomic components into a top-level composite components called `SqrtSyst` and `Summation`, respectively.

Since the first diagram is made only of one atomic component (`Sqrt`) its top-level model `SqrtSyst` is defined simply as `Sqrt`. For the second diagram, `Add` is first composed in parallel with `Id` (i.e., `Add ** Id`) in order to match the two inputs of `UnitDelay`, and then composed in series with the latter, yielding `(Add ** Id) o UnitDelay`. The last expression represents a component C with two outputs, since `UnitDelay` has two outputs.[2] We want to compose C in series with a `Split` component, which would model the fact that the output wires of the UnitDelay Simulink block in Fig. 4b is split into two wires. But we cannot directly compose C with `Split` because C has two outputs whereas `Split` has a single input. Therefore, we first put `Split` in parallel with `Id`, and then compose C in series with the result, yielding `(Add ** Id) o UnitDelay o (Split ** Id)`. The latter expression captures a composite component D with 3 inputs and 3 outputs, and we want to connect its first output to its first input, thereby capturing the feedback look in the diagram of Fig. 4b. But we cannot apply the `feedback` operator directly, because the inputs and outputs of D are structured in the form `(a,b),c`. We therefore use two *switches*, which in that case "flatten" the inputs and outputs of D to `a,b,c`. Once this is done, we can apply `feedback` and get the final top-level composite component representing the entire diagram.

Then, the `simplify_RCRS` keyword is used to invoke the Analyzer as described in Section 5.

As we shall see in Section 6, the same block diagram can be captured as (translated into) many different but semantically equivalent RCRS models. Fig. 4 presents only one of the possible RCRS models for each of the diagrams of Fig. 3. Alternative representations are discussed in Section 6.

[2] Note that `UnitDelay` is the RCRS component, and not the UnitDelay Simulink block. The latter has a single output wire, but in RCRS it is modeled as a stateful block which has an extra input modeling the current state and an extra output modeling the next state.

```
theory SqrtSyst imports ...
begin
named_theorems basic_simps
lemmas basic_simps = simulink_simps
definition [basic_simps]: "Sqrt = {. x . x≥ 0}
      o [- x ⇝ s_sqrt x -]"
simplify_RCRS "SqrtSyst = Sqrt" "b" "d"
end
```

(a) Square root

```
theory Summation imports ...
begin
named_theorems basic_simps
lemmas basic_simps = simulink_simps
definition [basic_simps]: "Add = [- f, g ⇝ f+g -]"
definition [basic_simps]: "UnitDelay =
    [- d, s ⇝ s, d -]"
definition [basic_simps]: "Split = [- a ⇝ a, a -]"
simplify_RCRS "Summation =
    feedback([- f, g, s ⇝ (f, g), s -]
       o (Add ** Id) o UnitDelay o (Split ** Id)
       o [- (f, h), s' ⇝ f, h, s' -])"
    "(g, s)" "(h, s')"
end
```

(b) Summation

Fig. 4: Capturing the Simulink diagrams of Fig. 3 in RCRS.

3 Basic Reasoning in RCRS: Demonstration

Let us illustrate the most important capabilities of the RCRS reasoning engine in the form of a demo. The demo can be found in the RCRS distribution at http://rcrs.gitlab.io/ under the `Isabelle` folder, file `RCRS_Demo.thy`. The file can be processed with Isabelle version 2018.

The initial skeleton of the file is as shown below:

```
theory RCRS_Demo imports "Simulink/SimplifyRCRS"
```

```
begin
  named_theorems basic_simps
  lemmas [basic_simps] = comp_func_simps
end
```

This code imports the RCRS Isabelle theories and declares the collection of theorems and lemmas that will be used later for simplification.

3.1 Simplification

We next define three RCRS components:

```
definition [basic_simps]: "SqrRoot = {. x. x≥0 .}
      o  [- x ⇝ sqrt x -]"
definition [basic_simps]: "Cst1 = [- u::unit ⇝ 4 -]"
simplify_RCRS "Syst1 = Cst1  o  SqrRoot" "u" "y"
```

`SqrRoot` models the square root function as we described in Section 2. `Cst1` models the constant 4. The composite component `Syst1` is formed by composing `Cst1` and `SqrRoot` in series. To define `Syst1` we use the construct `simplify_RCRS` which performs a number of things as explained in Section 5. First, it defines `Syst1 = Cst1  o  SqrRoot`. Second, it gives names to the external inputs and outputs of `Syst1`: `"u"` and `"y"` in this case. Third, it calls the RCRS Analyzer. The Analyzer performs expansion and simplification and finds that `Syst1` simplifies to `[- u ⇝ 2 -]`. To actually see the result of this simplification, we need to type:

```
thm Syst1_simp
```

The construct `thm name` instructs Isabelle to display the theorem/lemma specified by `name` in Isabelle's jEdit output window (Fig. 5). In our case, by placing the cursor on the line `thm Syst1_simp`, we can see the simplified version of `Syst1` in the Isabelle's output window as shown in Fig. 5:

```
Syst1 = [- u ⇝ 2 -]
```

This result is to be expected, as the whole system outputs the square root of 4, which is 2.

3.2 Contract Checking

Let us now see what happens if we change the constant to -1:

```
definition [basic_simps] : "Cst2 = [- u ⇝ -1 -]"
simplify_RCRS "Syst2 = Cst2  o  SqrRoot" "u" "y"
thm Syst2_simp
```

As before, by placing the cursor on the line `thm Syst2_simp`, we can see in the Isabelle's output window the simplified version of `Syst2`:

```
Syst2 = ⊥
```

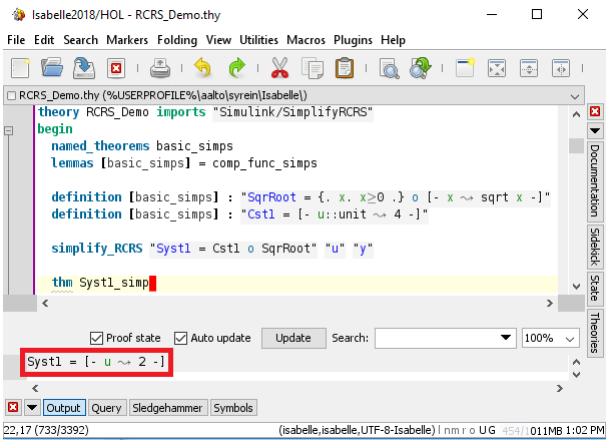


Fig. 5: Simplification in RCRS. The result is shown outlined in red in Isabelle's output window.

Now the Analyzer simplifies `Syst2` to $\bot$, which models the invalid component as stated in Section 2.2. This result indicates an inconsistency in our model. The inconsistency here is obviously that -1 violates the input condition of `SqrRoot`, meaning that the components `Const2` and `SqrRoot` are *incompatible*. This incompatibility detection can be seen as advanced static type checking.

So far, all our components were deterministic. Let us continue our demo by showing what happens if we try to connect `SqrRoot` to the non-deterministic component `true` which can output any value:

```
definition [basic_simps] : "true =
      [: u ⇝ y. True :]"
```

Component `true` can be seen as modeling a "black-box" system for which we have no information (e.g., no available source code) or which we are unable to analyze. Obviously, in such a case, it is difficult to guarantee anything. Therefore, connecting `true` to `SqrRoot` should result in an incompatibility. Let us see what RCRS does. First, we instruct the Analyzer to use also the results for the relational monotonic predicate transformers.

```
lemmas [basic_simps] = comp_func_simps
    comp_rel_simps
```

Next, we invoke the Analyzer for the desired system.

```
simplify_RCRS "Syst3 = true o SqrRoot" "u" "y"
```

Finally, we check the Analyzer results.

```
thm Syst3_simp
```

We see that the Analyzer simplifies `Syst3` to

```
{.u. ∀y. 0 ≤ y.} o [: u ⇝ y. ∃z. y = sqrt z:]
```

The formula $\forall y.0 \leq y$ is unsatisfiable which means that `Syst3` is invalid. But why wasn't `Syst3` simplified to $\bot$?

This is because of Isabelle's limitations in simplifying expressions with quantifiers such as $\forall y.0 \leq y$. In this case we have to "help" Isabelle, by recognizing that the formula $\forall y.0 \leq y$ is unsatisfiable. We state this as a lemma:

```
lemma aux1: "(∀y::real. 0 ≤ y) = False"
  (* sledgehammer *)
  using le_minus_one_simps(3) by blast
```

and we use Isabelle's "sledgehammer" mechanism to prove it. The "sledgehammer" keyword of Isabelle transforms the goal to prove in a format understood by different SMT solvers. Then it calls some of them (e.g. cvc4 [5] or Z3 [10]) and if a proof is found it is displayed to the user. Then the user can just copy the proof. In the case above, sledgehammer manages to find the one line proof of the above lemma.

Having proved this lemma, we can call the simplification procedure again on `Syst4` which is a copy of `Syst3`. The only difference is that this time we ask the simplification procedure to use the lemma:

```
simplify_RCRS "Syst4 = true o SqrRoot" "u" "y"
   use (aux1)
thm Syst4_simp
```

This time simplification succeeds and produces:

```
Syst4 = ⊥
```

Let us end this section by illustrating also contract checking of QLTL components. We first define the following two components:

```
definition "Syst7 = {. □ ◇ (ltl x . x) .}  o  [:
    □ ◇ (ltl x y . y) :]"
definition "Syst8 = [: ◇ □ (ltl x y . ¬ y) :]"
```

Component `Syst7` is identical to component `A` defined at the end of Section 2.1. `Syst7` requires that its input `x` is infinitely often true and guarantees that its output `y` is also infinitely often true. Component `Syst8` also takes as input an infinite sequence `x` and returns as output an infinite sequence `y`. `Syst8` imposes no requirements on its input (i.e., it is input-receptive) and guarantees that its output `y` satisfies $\Diamond\Box\neg$`y`, meaning that `y` is after some point always false. Clearly, the serial composition `Syst8 o Syst7` should be invalid, since the output guarantees of `Syst8` contradict the input requirements of `Syst7`.

As with some of the examples above, in order to have RCRS automatically prove that `Syst8` and `Syst7` are incompatible, we need to help Isabelle by providing the following lemma:

```
lemma simp_ltl: "(∀ y . (◇ □ (ltl x y. ¬ y)) x y
    → (□ ◇ (ltl x. x)) y) = False"
  apply (simp add: LTL_def always_def
      eventually_def at_fun_def)
   (*expanding definitions*)
   apply (rule_tac x = ⊥ in exI)
   by auto
```

Having proved the above lemma, we can call the RCRS simplification procedure on the serial composition of `Syst8` and `Syst7`:

```
simplify_RCRS "Syst9 = Syst8  o  Syst7" "x" "y"
  use (Syst7_def Syst8_def simp_ltl)
```

Simplification succeeds and produces:

```
Syst9 = ⊥
```

3.3 Contract Inference

Now, suppose that we have a component for which we know something, for instance, that its output y is greater than its input x plus 1:

```
definition [basic_simps]: "A =
          [: x ⇝ y. y ≥ x + 1 :]"
```

Let us see what happens if we connect `A` to `SqrRoot`, and try to simplify:

```
simplify_RCRS "Syst5 = A o SqrRoot" "x" "y"
thm Syst5_simp
```

We get:

```
Syst5 = {.x. ∀y≥ x + 1. 0 ≤ y.}
     o [: x ⇝ y. ∃ z ≥ x + 1. y = sqrt z :]
```

Again, Isabelle has trouble eliminating the quantifiers from the formulas and needs our help. We recognize that the formula of the input condition is equivalent to $x \geq -1$, and state this as a lemma:

```
lemma aux2: "(∀y::real ≥ x + 1. 0 ≤ y) = (-1 ≤ x)"
  by auto
```

We can now use the above lemma to simplify further:

```
simplify_RCRS "Syst6 = A o SqrRoot" "x" "y"
  use (aux2)
```

and we get:

```
Syst6 = {.x. -1 ≤ x.}
     o  [: x ⇝ y. ∃ z ≥ x + 1 . y = sqrt z :]
```

The constraint $-1 \leq x$ can be seen as the weakest constraint on the input (similar to a weakest precondition [11]) which ensures that all internal contracts of the system will be satisfied. Note that the constraint $-1 \leq x$ is as simple as it can be.

Next, consider the postcondition `∃ z ≥ x + 1 . y = sqrt z`. This can be simplified further by eliminating the existential quantifier. This quantifier elimination requires manual intervention in Isabelle, performed as follows.

First we introduce a general lemma for simplifying the relation of an MPT assuming that the input condition is true:

```
lemma prec_rel: "(⋀ x y . p x ⟹ r x y = r' x y)
      ⟹ {.p.} o [:r:] = {.p.} o [:r':]"
  by (auto simp add: fun_eq_iff assert_def
      demonic_def le_fun_def)
```

If for all x and y we have p x ⟹ r x y = r′ x y, then we can replace r by r′ in {.p.} o [:r:].

Next, assuming -1 ≤ x, we show that the existential quantifier can be eliminated in (∃ z. z ≥ x + 1 ∧ y = sqrt z) since z is always greater or equal than 0, i.e., the sqrt function is defined:

```
lemma aux3: "-1 ≤ x ⟹
      (∃ z. z ≥ x + 1 ∧ y = sqrt z)
           = (sqrt (x + 1) ≤ y)"
proof (safe, simp_all)
  fix x y
  assume A[simp]: "sqrt (x + 1) ≤ y"
  assume "-1 ≤ x"
  from this have B[simp]: "0 ≤ x + 1"
    by simp
  have "(sqrt (x + 1))^2 ≤ y^2"
    by (rule power_mono, simp_all)
  from this have [simp]: "x + 1 ≤ y^2"
    by simp
  have [simp]: "0 ≤ y"
    using A B by (metis order_trans
        real_sqrt_ge_0_iff)
  show "∃ z ≥ x + 1. y = sqrt z"
    by (rule_tac x = "y^2" in exI, simp)
qed
```

Finally, we eliminate the existential quantifier in Syst6:

```
lemma "Syst6 = {. x. -1 ≤ x .}
          o [: x ⇝ y. sqrt (x + 1) ≤ y :]"
proof -
  have "Syst6 = {. x . -1 ≤ x .}
          o [:x ⇝ y. ∃ z ≥ x + 1. y = sqrt z:]"
    by (simp add: Syst6_simp)
  also have "... = {. x . -1 ≤ x .}
          o [:x ⇝ y. (sqrt (x + 1) ≤ y):]"
    by (rule prec_rel, simp add: aux3)
  finally show ?thesis
    by simp
qed
```

3.4 Refinement Checking

The above examples illustrated several of the features of RCRS as a reasoning tool, similar to a behavioral type checking and inference engine. Indeed, detecting incompatible connections is akin to catching type errors in programs, and inferring conditions such as the condition on the input in the last example above is akin to type inference. In addition to these capabilities, RCRS can be used to check *refinement* (and its counterpart, *abstraction*) between components. Refinement is a powerful design methodology, useful in many scenarios. For instance, consider a scenario where some component C in the system is to be replaced by C'. In this case, both C and C' are available, and we need to ensure that replacing C by C' does not cause any problems, i.e., that the properties of the original system (containing C) are preserved. Checking that C' refines C is a way to ensure this. Another scenario where refinement is useful is when the system is too detailed/large, and needs to be simplified/abstracted, e.g., in order to make verification easier. In that case, we may choose to abstract some component C with a simpler component C^a. Refinement can be used in this case to ensure that C refines C^a, i.e., that C^a is a proper abstraction of C.

As an example, let us show how to prove that the SqrRoot component refines NonDetSqrt and that the ReceptiveSqrt component in turn refines SqrRoot (components NonDetSqrt and ReceptiveSqrt are defined in Section 2). We have:

```
lemmas [basic_simps] = comp_rel_simps
    basic_block_rel_simps update_def
    refinement_simps
definition [basic_simps] : "NonDetSqrt =
      {. x. 0 ≤ x .} o [: x ⇝ y. 0 ≤ y :]"
lemma "NonDetSqrt ≤ SqrRoot"
  by (auto simp add: basic_simps)
definition [basic_simps] : "ReceptiveSqrt =
     [: x ⇝ y . 0 ≤ x → y = sqrt x :]"
lemma "SqrRoot ≤ ReceptiveSqrt"
  by (simp add: basic_simps)
```

The first line instructs the Analyzer to use simplification rules for relational predicate transformers [30]. Then, we define the new components and state the refinements as lemmas. The proofs are simple and are based on the necessary and sufficient conditions for checking refinement included in the RCRS library:

```
lemma assert_demonic_refinement:
   "({.p.} o [:r:] ≤ {.p'.} o [:r':]) =
    (p ≤ p' ∧ (∀ x . p x → r' x ≤ r x))"
  by (auto simp add: le_fun_def assert_def
      demonic_def)
lemma spec_demonic_refinement:
     "({.p.} o [:r:] ≤ [:r':]) =
      (∀ x . p x → r' x ≤ r x)"
  by (auto simp add: le_fun_def assert_def
      demonic_def)
```

3.5 Discussion

As can be seen by the above examples, RCRS offers powerful reasoning capabilities. Some of these capabilities are fully automatic, despite the fact that they are implemented on top of Isabelle, a general proof assistant which generally requires human intervention. In particular, RCRS theories generated automatically from

Simulink diagrams by our Translator can be processed by our Analyzer fully automatically (see Section 5). Other reasoning capabilities currently require human intervention. We expect that as automated theorem proving capabilities improve, these RCRS capabilities will also become more and more automatic.

4 The Implementation of RCRS in Isabelle

RCRS is fully formalized in the Isabelle theorem prover. The RCRS implementation currently consists of 23 Isabelle *theories* (`.thy` files), totaling 29707 lines of Isabelle code. The structure and dependencies between those theories are shown in Fig. 6.

4.1 Core Theories

Theory `Refinement.thy` (1220 lines) contains a standard implementation of the refinement calculus [4]. Systems are modeled as monotonic predicate transformers [11] with a weakest precondition interpretation. Within this theory we implemented non-deterministic and deterministic update statements, assert statements, parallel composition, refinement and other operations, and proved necessary properties of these. Additionally, this theory implements control statements and proves Hoare total correctness rules required in order to prove the determinacy of our block-diagram translation algorithms [31].

Theory `Temporal.thy` (801 lines) implements a semantic version of QLTL, where temporal operators are interpreted as predicate transformers. For example, the operator $\Box$, when applied to the predicate on infinite traces $(x > 0) : (\texttt{nat} \to \texttt{real}) \to \texttt{bool}$, returns another predicate on infinite traces $\Box(x > 0) : (\texttt{nat} \to \texttt{real}) \to \texttt{bool}$. Temporal operators have been implemented to be polymorphic in the sense that they apply to predicates over an arbitrary number of variables.

Theory `RefinementReactive.thy` (1155 lines) extends `Refinement.thy` and `Temporal.thy` to reactive systems by introducing predicates over infinite traces in addition to predicates over values, and *property* transformers in addition to predicate transformers [33,30].

Theory `TransitionFeedback.thy` (637 lines) defines and implements the *feedback* operator for predicate transformers represented with both deterministic and non-deterministic update statements.

Theory `IterateOperators.thy` (2806 lines) defines the operator that maps predicate transformers on input and current state to output and next state into property transformers on infinite traces of input values to infinite traces of output values.

Theory `ReactiveFeedback.thy` (1268 lines) defines the *feedback* operator for property transformers. For deterministic systems, this theory also establishes the connection between the feedback on predicate transformers and the feedback on property transformers via the iteration operators.

Theory `InstantaneousFeedback.thy` (2405 lines) defines a feedback operator for predicate transformers using unknown values. This operation has been proposed in [34] as an extension of the constructive semantics for synchronous circuits [24] to non-deterministic and non-input receptive systems.

Theory `SimplifyRCRS.thy` (2271 lines) implements several of the Analyzer's procedures. In particular, it contains a simplification procedure which reduces composite RCRS components into atomic ones (see Section 5).

Finally, theory `RCRS_Overview.thy` (1650 lines) summarizes and proves all the results presented in [30].

4.2 Simulink-Related Theories

Theory `Simulink.thy` (886 lines) defines a subset of the basic blocks in the Simulink library as RCRS components (at the time of writing, 48 Simulink block types can be handled).

Theory `SimulinkTypes.thy` (259 lines) defines a polymorphic *simulink* type for wires and functions (e.g., *s_sqrt*) that is used in the formalization of basic blocks. The purpose of this theory is to abstract as much as possible the typing mechanism of Simulink, and to be able to handle as many Simulink diagrams as possible, without modifying them. This theory implements the results presented in [29].

Theory `PythonSimulation.thy` (234 lines) implements rewriting rules for automatically transforming predicate transformers RCRS terms into Python code. This code can be executed to validate the behavior of the system and is of particular interest when Simulink diagrams are modeled with RCRS (see Section 6).

4.3 Theories Regarding the Translation of Block Diagrams

Several theories concern translation methods from hierarchical block diagrams (such as Simulink diagrams) into RCRS composite components, and a proof of correctness (determinacy) of such translations (see Section 6 and [13,31]). In particular:

Theory `ListProp.thy` (671 lines) proves several properties about permutations and substitutions on lists.

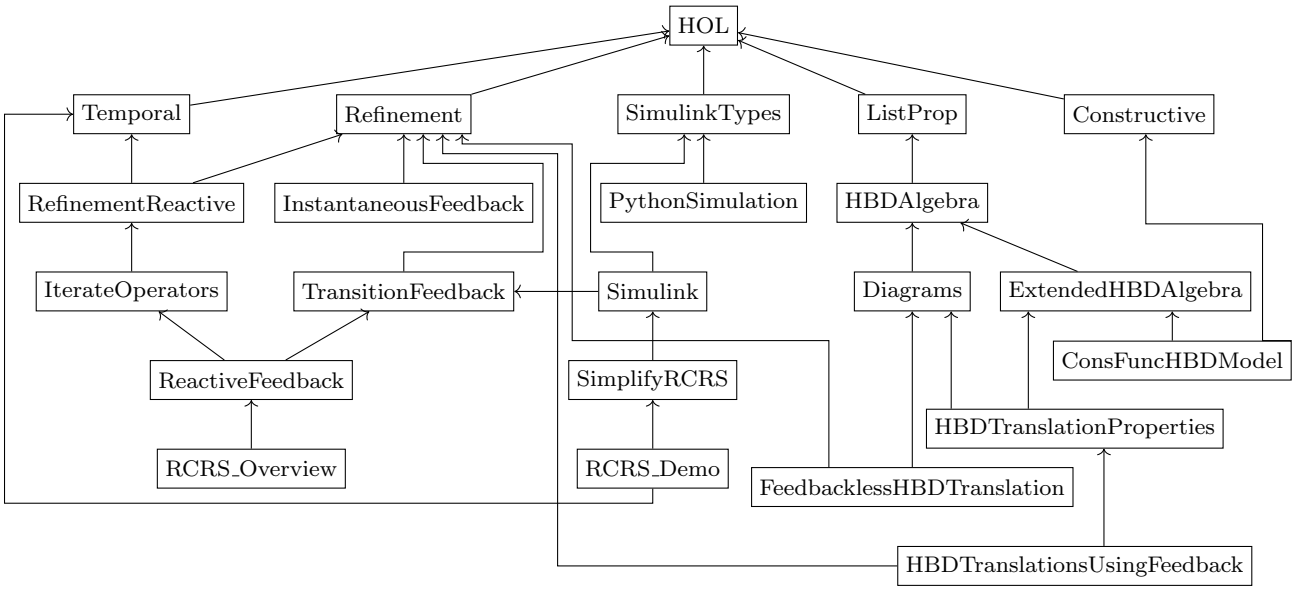


Fig. 6: Structure of the RCRS Isabelle theories. Arrows denote dependencies. Dependencies are from the source to the destination of each arrow, so theory RCRS_Demo depends on theory SimplifyRCRS, the latter depends on theory Simulink, etc.

Theory `HBDAlgebra.thy` (1700 lines) defines the abstract algebra of HBDs presented in [31]. Theory `ExtendedHBDAlgebra.thy` (191 lines) completes it with additional results.

Theory `Constructive.thy` (268 lines) defines the model of constructive functions. In theory `ConsFuncHBDModel.thy` (1479 lines) it is shown that constructive functions are a model of the (extended) HBD algebra.

Theory `Diagrams.thy` (8552 lines) defines the input-output diagrams that can be expressed with RCRS and proves several properties. These results are completed with those from the theory `HBDTranslationProperties` that are used by the main algorithms.

Determinacy is proved in theory `FeedbacklessHBDTranslation.thy` (229 lines) for those diagrams involving only the serial and parallel composition operators. The feedback composition operator is considered in theory `HBDTranslationsUsingFeedback.thy` (484 lines).

4.4 Demo Theory

The results presented in this paper are fully reproducible and can be found in theory `RCRS_Demo.thy` (115 lines).

4.5 Discussion: Shallow vs. Deep Embedding of RCRS Syntax

The syntax of RCRS components is implemented in Isabelle using a *shallow embedding* [6]. This has the advantage of all datatypes and other mechanisms of Isabelle (e.g., renaming) being available for component specification, but also the disadvantage of not being able to express properties and simplifications of the RCRS language within Isabelle, as discussed in [30]. A *deep embedding*, in which the syntax of components is defined as a datatype of Isabelle, is possible, and is left as an open future work direction.

5 The Analyzer

The Analyzer is a set of procedures implemented on top of Isabelle and ML, the programming language of Isabelle. These procedures implement a set of functionalities such as simplification, compatibility checking, refinement checking, etc. Here we describe the main functionalities, implemented by the `simplify_RCRS` construct.

As illustrated in Fig. 4, the general usage of this construct is:

```
simplify_RCRS "Model = C"
  "in" "out"
  use (additional_simps)
```

`C` is a (generally composite) component, `in` and `out` are (nested tuples of) names for its input and output vari-

ables, and `use` indicates what existing proven results should be used by the analyzer. The input and output tuples should match the typing of the component `C`. Conceptually, the tuples `in` and `out` are not required, but we use them to name the input and output variables in the simplified version of `C`, to make the final result more readable.

The result of the above Isabelle declaration is a definition `Model_def` stating the equality `Model = C`, and a theorem `Model_simp`, stating the equality `Model = C'`, where `C'` is the result of the simplification of `C`, and it has the form `{.p.} o [-f-]` for algebraic loop-free deterministic systems or the form `{.p.} o [:r:]` for non-deterministic systems and systems with algebraic loops. The proof of `Model_simp` is automatically constructed by `simplify_RCRS`, i.e. `Model_simp` is proved to follow from some existing RCRS and Isabelle theorems, which in turn are proved to follow from the basic HOL principles.

The internal structure of `simplify_RCRS` is represented by the pseudo-code from Fig. 7.

```
fun simplify_RCRS(Name, Expr, In, Out, Simps)
  def_thm = definition(Name, Expr)
  expd_thm = simplify(Expr, Simps + BasicSimps)
  t = rhs(expd_thm)
  (at_thm, rep) = simplify_comp(t, In, Out)
  at_thm' = simplify(right_hand_side(at_thm))
  simp_thm = trans(def_thm, expd_thm, at_thm, at_thm')
  register def_thm as Name + "_def"
  register simp_thm as Name + "_simp"
```

Fig. 7: Pseudocode for `simplify_RCRS`

Next we explain how this algorithm works on the `Summation` example from Fig. 3[3]:

```
simplify_RCRS "Summation =
    feedback([- f, g, s ⇝ (f, g), s -]
      o (Add ** Id) o UnitDelay o (Split ** Id)
      o [- (f, h), s' ⇝ f, h, s' -])"
    "(g, s)" "(h, s')"
    use (Add_def UnitDelay_def Split_def)
```

The parameters of the algorithm are given by:

```
  Name := Summation
  Expr := feedback([- f, g, s ⇝ (f, g), s -]
       o (Add ** Id) o UnitDelay o (Split ** Id)
       o [- (f, h), s' ⇝ f, h, s' -])
  In := (g, s)
  Out := (h, s')
  Simps := (Add_def UnitDelay_def Split_def)
```

The first step

```
  def_thm = definition(Name, Expr)
```

uses the definition mechanism of Isabelle and creates the theorem `def_thm`:

```
    Summation
=
    feedback([- f, g, s ⇝ (f, g), s -]
      o (Add ** Id) o UnitDelay o (Split ** Id)
      o [- (f, h), s' ⇝ f, h, s' -])
```

Next, the step

```
  expd_thm = simplify(Expr, Simps + BasicSimps)
```

simplifies `Expr` by unfolding the definitions of the basic blocks (`BasicSimps`) and also by using the equalities from `Simps`. In practice for this example the definitions of the basic blocks `Add`, `UnitDelay`, and `Split` are already included in `BasicSimps`, but we included them also in `Simps` for exemplification purposes. The result of this step is the theorem `expd_thm`:

```
    feedback([- f, g, s ⇝ (f, g), s -]
      o (Add ** Id) o UnitDelay o (Split ** Id)
      o [- (f, h), s' ⇝ f, h, s' -])
=
    feedback([- f, g, s ⇝ (f, g), s -]
      o ([- x, y ⇝ x + y -] ** Id)
      o [- x, s ⇝ s, x -]
      o ([- x ⇝ x, x -] ** Id)
      o [- (f, h), s' ⇝ f, h, s' -])
```

The goal of this step is to eliminate all basic blocks and subcomponents and replace them with their *canonical form* `{. _ .} o [- _ -]` or `{. _ .} o [: _ :]`. Some additional small simplifications are also performed at this step (`Id ** Id = Id`, `Id o S = S`). After this step, the simplified result must contain only the atomic RCRS constructs `{. _ .}`, `[- _ -]`, `[: _ :]`, `Id` and the composition operators (feedback, serial, and parallel). We call this form *basic composite property transformer* (BCPT).

Next step:

```
  t = rhs(expd_thm)
```

assigns to variable `t` the *right hand side* of theorem `expd_thm` to be further simplified.

Next step:

```
  (at_thm, rep) = simplify_comp(t, In, Out)
```

simplifies the term `t` into its canonical form by recursion on the structure of `t`. The result of this step is the theorem `at_thm` and the first order representation `rep` of the simplified predicate transformer. For our example the theorem `at_thm` is

```
    feedback([- f, g, s ⇝ (f, g), s -]
      o ([- x, y ⇝ x + y -] ** Id)
      o [- x, s ⇝ s, x -]
      o ([- x ⇝ x, x -] ** Id)
      o [- (f, h), s' ⇝ f, h, s' -])
=
```

[3] Note that we duplicate here the use of atomic components definitions in order to illustrate the algorithm. These are already part of the `basic_simps` simplification rules.

```
{. g, s . True .}  o  [- g, s ⇝ s, s + g -]
```

and the first order representation of the simplified predicate transformer is

```
rep = Func((g,s), True, (s, s + g))
```

This representation is discarded for the topmost call to `simplify_comp`, but is used for the recursive calls to improve the efficiency of the simplification.

Next step:

```
at_thm′ = simplify(rhs(at_thm))
```

eliminates vacuous assertions `{. g, s . True .}` or turns `{. g, s . False .}  o  [- _ -]` into ⊥. For the running example `at_thm'` is

```
  {. g, s . True .}  o  [- g, s ⇝ s, s + g -]
=
  [- g, s ⇝ s, s + g -]
```

The step

```
simp_thm=trans(def_thm, expd_thm, at_thm, at_thm′)
```

applies transitivity of equality for the theorems `def_thm`, `expd_thm`, `at_thm`, `at_thm′` to obtain the final simplification theorem `simp_thm`:

```
Summation = [- g, s ⇝ s, s + g -]
```

The last two steps of the algorithm are registering, in the current Isabelle theory, the definition theorem `def_thm` and the simplification theorem `simp_thm` as `Summation_def` and `Summation_simp`, respectively.

The step performed by `simplify_comp` assumes that the term `t` is in the form of BCPT. If this is not the case, then `simplify_comp` fails. It is the responsibility of the user to ensure that `Simps + BasicSimps` contains all necessary rules such that all non-atomic sub-components of `Expr` are reduced to BCPTs. Our automatic translator from Simulink to RCRS ensures this property. The translator adds all the definitions of the basic Simulink blocks into BasicSimps via the `basic_simps` mechanism:

```
named_theorems basic_simps
lemmas [basic_simps] = comp_func_simps
definition [basic_simps]: "Cst = [- u::unit⇝1 -]"
simplify_RCRS "SubModel = (Cst  **  Id)  o  Add"
    "x" "y"
```

The translator also adds the simplification rules of the sub-components to `Simps` via the `use` mechanism:

```
simplify_RCRS "Model = SubModel o SqrRoot" "x" "z"
    use (SubModel_simp)
```

For the feedback operation `feedback(S)` with `S` deterministic, the function `simplify_comp` tests if `S` is algebraic loop-free, and in this case applies a special simplification rule resulting in a deterministic component, otherwise it applies the feedback definition resulting in a non-deterministic component.

To complete our presentation of the Analyzer, we show how `simplify_comp` is defined for serial composition, the other operations being similar. Assume that `t = t1  o  t2`, and that both `t1` and `t2` are deterministic, then:

```
fun simplify_comp(t1  o  t2, In, Out)
    NewVar = new tuple of output type of t1
    thm1, Func(In1, p1, e1) = simplify_comp(t1, In,
        NewVar)
    thm2, Func(In2, p2, e2) = simplify_comp(t2,
        NewVar, Out)
    p, f = (λIn1.p1), (λIn1.e1)
    f′, p′ = (λIn2.p2), (λIn2.e2)
    p_simp_thm = simplify((p ⊓ (p′  o  f)) In1)
    f_simp_thm = simplify((f′  o  f) In1)
    p_abs_thm = abstract(p_simp_thm)
    f_abs_thm = abstract(f_simp_thm)
    serial_rep = Func(In1, rhs(p_simp_thm),
       rhs(f_simp_thm))
    serial_thm = serial_det OF [thm1, thm2,
       p_abs_thm, f_abs_thm]
    return (serial_thm, serial_rep)
```

The idea of `simplify_comp` is the following. First `t1` and `t2` are simplified using `simplify_comp` recursively. For this we need a new tuple `NewVar` of variable names of the same type as the type of the output of `t1`. After this simplification we obtain the theorems `thm1` and `thm2`:

```
thm1: t1 = {.In1 . p1.} o [-In1 . e1-]
thm2: t2 = {.In2 . p2.} o [-In2 . e2-]
```

and we also obtain the first order representations:

`Func(In1, p1, e1)` and `Func(In2, p2, e2)`

Next we define

`p = (λIn1.p1)` and `f = (λIn1.e1)`
`p′ = (λIn2.p2)` and `f′ = (λIn2.e2)`

With these notations, theorems `thm1` and `thm2` can be stated as:

```
thm1: t1 = {.p.} o [-f-]
thm2: t2 = {.p′.} o [-f′-]
```

We remark that the notation `{.x,y . x + y.}` is just a syntactic sugar for the term `{.(λ(x,y).x+y).}`, and similarly for `[- _ -]` and `[: _ :]`.

Next we compute the simplifications of the terms `(p ⊓ (p′  o  f)) In1` and `(f′  o  f) In1`, where ⊓ is the pointwise extension of the conjunction operation to predicates, as the theorems:

```
p_simp_thm: (p ⊓ (p′  o  f)) In1 = p_exp
f_simp_thm: (f′  o  f) In1 = f_exp
```

We also compute the corresponding abstract versions of these theorems:

```
p_abs_thm: (p ⊓ (p′  o  f)) = (λ In1 . p_exp)
f_abs_thm: f′  o  f) = (λ In1 . f_exp)
```

The Boolean expression `p_exp` is the simplified expression of the assert statement of the serial composition of `t1` and `t2`, and `f_exp` is the simplified expression of the functional outputs for the inputs `In1` of the serial composition of `t1` and `t2`, i.e.

```
serial_thm:
    t1  o  t2 = {.In1.p_exp.}  o  [-In1.f_exp-]
           = {.(λIn1.p_exp).}  o  [-(λIn1.f_exp)-]
```

and

```
    serial_rep = Func(In1, p_exp, f_exp)
```

However, we must construct the theorem `serial_thm` as a consequence of existing proved results. For this purpose we introduce lemma `serial_det`:

```
lemma serial_det: "t1 = {.p.} o [-f-]
    ⟹ t2 = {.p'.} o [-f'-]
    ⟹ p ⊓ (p' o f) = p''
    ⟹ f o f' = f''
    ⟹ t1 o t2 = {.p''.} o [-f''-]"
```

This lemma is the specialization of the definition of the serial composition to deterministic systems, stated in a form that can be applied directly to conclude `serial_thm`. In fact, for `p'' = (λIn1.p_exp)` and `f'' = (λIn1.f_exp)`, and for all other variables as already defined, the conclusion of `serial_det` is exactly `serial_thm`, and the premises of `serial_det` are the theorems `thm1`, `thm2`, `p_abs_thm`, and `f_abs_thm`. Therefore `serial_thm` is a consequence of `serial_det`, `thm1`, `thm2`, `p_abs_thm`, and `f_abs_thm`. This is expressed in the definition of `simplify_comp` as:

```
serial_thm = serial_det OF [thm1, thm2,
    p_abs_thm, f_abs_thm]
```

Note that the execution by the Analyzer of the `.thy` file generated by the Translator is fully automatic, despite the fact that Isabelle generally requires human interaction. This is thanks to the fact that the theory generated by the Translator contains all declarations (equalities, rewriting rules, etc.) necessary for the Analyzer to produce the simplifications and their mechanical proofs, without user interaction.

If the model contains *incompatibilities*, where for instance the input condition of a block like `SqrRoot` cannot be guaranteed by the upstream diagram, the top-level component automatically simplifies to $\bot$. Thus, in this usage scenario, RCRS can be seen as a static analysis and behavioral type checking and inference tool for Simulink.

6 The Translator

So far we have mainly demonstrated the capabilities of the *back-end* of the RCRS Toolset, namely, everything implemented on top of Isabelle (see Fig. 1). This back-end can certainly be used independently of the *front-end*, i.e. the Translator described in this section. Indeed, users of RCRS do not a-priori need the Translator. They can model systems by directly defining RCRS components, including instantiating predefined components from the RCRS component library, and they can reason about such components along the lines of what has been described in the previous sections. However, the Translator is a useful tool to have as it allows users to model systems in the widespread Simulink environment, and generate RCRS formal models automatically. The Translator also enables formal reasoning capabilities for Simulink models.

The Translator is a Python program (about 7100 lines of Python code) called `simulink2isabelle`. The program is based on techniques for translating general *hierarchical block diagrams* (HBDs) into a compositional algebra of components like the one used in RCRS (see [13,32] and Section 6.2 that follows). The current version of the Translator takes as input Simulink diagrams, but could be modified to accept other types of HBD-based languages.

`simulink2isabelle` takes as input a Simulink model (`.slx` file) and a list of options and generates as output an Isabelle theory (`.thy` file). The output file contains, among other things: (1) the definition of all instances of basic blocks in the Simulink diagram (e.g., all Adders, Integrators, Constants, etc.) as atomic RCRS components; (2) the bottom-up definition of all subdiagrams as composite RCRS components; (3) calls to simplification procedures; and (4) theorems stating that the resulting simplified components are equivalent to the original ones. The `.thy` file may also contain additional content depending on user options as explained below.

6.1 Revisiting the Running Examples

The RCRS models shown in Fig. 4 are slightly simplified versions of models automatically generated by the Traslator from the corresponding Simulink diagrams of Fig. 3. Specifically, the two RCRS theories are generated by running the Translator with the following options on the corresponding Simulink files:

```
./simulink2isabelle.py sqrt_syst.slx -ic
./simulink2isabelle.py Summation.slx -ic
```

The Translator will generate the theory header and footer, the definition of basic blocks relying on the library (e.g., `In = Inport`) and the definition of the system. In both cases we call the Translator with the `-ic` option, which specifies the composition strategy to be used (see Section 6.2). This option corresponds to the

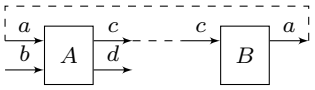


Fig. 8: A simple block diagram.

incremental composition described on the examples in Section 2.3.

6.2 Translation Strategies

Translating HBDs into a formal algebra with composition primitives like the ones used in RCRS is an interesting and non-trivial problem. As shown in [13], there are many possible ways to translate a block diagram into an algebra of components with the three primitive composition operators of RCRS (parallel, serial, feedback). This means that step (2) described at the beginning of this section is not unique. `simulink2isabelle` implements the several translation strategies proposed in [13] as user options.

For example, consider the block diagram shown in Fig. 8. The dashed lines represent connections between the two blocks, A and B. The labels a, b, c, d have been added to facilitate referring to the wires in the discussion that follows.

How could we translate such a diagram into an RCRS component? There are several ways. First, let's assume that `A` and `B` are RCRS components corresponding to the blocks A and B. The latter could be atomic blocks or they could themselves be block diagrams, in which case they would have to be translated first into some composite components `A` and `B` before the diagram of Fig. 8. Independently of whether `A` and `B` are atomic or composite, we assume they are already defined. Our task is to use them to define the diagram of Fig. 8 as an RCRS component.

One first idea is to attempt to connect `A` and `B` in series, thus implementing the connection between the two wires labeled c. This does not directly work, however, because `A` has two outputs whereas `B` only has one input. Therefore, `A o B` is not a syntactically legal RCRS component. We can solve this problem by first composing `B` in parallel with an `Id` component, as explained in Section 2.2. Then we get `A o (B ** Id)`, which is syntactically legal. Moreover, the resulting component has two inputs, the first of which is input wire a, and two outputs, the first of which is output wire a. Therefore we can apply feedback composition to get the final result (also illustrated as a block diagram in Fig. 9a):

```
C1 = feedback(A o (B ** Id))
```

But this is not the only translation possible. Another syntactically legal RCRS term representing the original diagram is

```
C2 = feedback((B  ** Id) o A)
```

illustrated as a block diagram in Fig. 9b. Both `C1` and `C2` correspond to the *incremental translation strategy* described in [13]. The only difference is the order in which the blocks are handled: for `C1`, `A` is picked before `B`, while for `C2`, `B` is picked first and then `A`. RCRS guarantees that no matter which order is picked, the resulting components are semantically equivalent (see [32] and discussion at the end of this subsection).

The subterms of `C1` and `C2` above can also be defined separately and incrementally, as in:

```
C1_1 = A o (B ** Id)
C_1 = feedback(C1_1)
```

This strategy is called *incremental with intermediate outputs* in [13], and can be called with the option `-ici` of the translator.

A third possible translation is:

```
C3 = feedback(feedback(Switch1 o (A ** B) o Switch2))
```

where

```
Switch1 = [- c,a,b ⇝ a,b,c -]
Switch2 = [- c,d,a ⇝ c,a,d -]
```

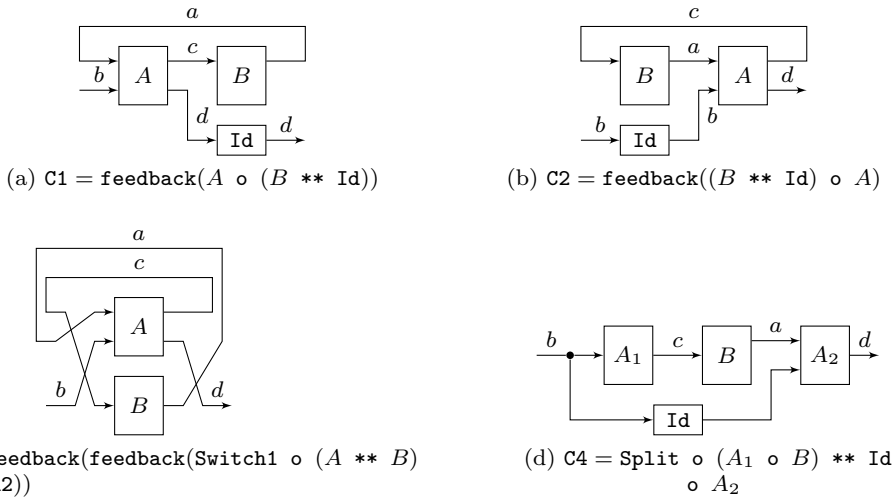
`C3` is illustrated as a block diagram in Fig. 9c. This term corresponds to the so-called *feedback-parallel* translation strategy [13] and can be obtained with the `-fp` option.

A fourth possible RCRS term could be obtained with the *feedbackless* translation strategy (named so because it results in terms which do not use the `feedback` operator at all [13]) which gives

```
C4 = Split o ((A₁ o B) ** Id) o A₂
```

This strategy can be employed only when the diagram is free from algebraic loops, meaning that it has no instantaneous cyclic dependencies. By *instantaneous* we mean that the values of two signals depend on each other *in the same reaction step*. For example, the output of the `Add` component depends instantaneously on both its inputs. However, the output of the `UnitDelay` only depends on the value of its input at the *previous* step, and therefore, `UnitDelay` can be used in feedback loops without introducing instantaneous (algebraic) dependencies. Several other Simulink blocks like the Integrator, TransferFunction, etc., behave similarly. The feedbackless translation strategy can handle all such models without problems [13].

On the diagram of Fig. 8 this means that c can only depend (instantaneously) on b. In this case, the strategy called with the `-nfb` option produces the result from



(a) `C1` = `feedback`(A `o` (B `**` `Id`))

(b) `C2` = `feedback`((B `**` `Id`) `o` A)

(c) `C3` = `feedback(feedback(Switch1 o` (A `**` B) `o Switch2))`

(d) `C4` = `Split o` (A_1 `o` B) `**` `Id` `o` A_2

Fig. 9: Possible views of the block diagram of Fig. 8.

Fig. 9d. The main idea is to split blocks such that they have only one output and only those inputs on which it depends. Then the connections consist only of serial and parallel. In our example we have $\texttt{A}_1$ with output c and input only b, and $\texttt{A}_2$ with output d and input a and b. Then the feedback is unfolded with serial connections. If c had depended on a (which is equal to c), this strategy could not be applied.

When many translations of a given diagram are possible, which one should be chosen? What are the pros and cons of each possible target term? Are the terms semantically equivalent? How can such terms be generated automatically from a given diagram (what are the translation algorithms)? All these and more interesting questions are beyond the scope of the current paper. We refer the reader to our previous works which provide some (partial) answers. Specifically, [13] presents three translation algorithms, and examines their trade-offs as well as the trade-offs of the respective generated terms. The semantic equivalence of the terms generated by the three translation algorithms proposed in [13] is examined in [32]. In this latter paper we provide a mechanic formalization of the three algorithms (in Isabelle) and prove (also in Isabelle) their semantic equivalence. This formalization and proof correspond to 14797 lines of Isabelle code and is available as part of the RCRS code distribution.

We note that what is formalized in [32] are the translation *strategies* and not the actual Python code of the Translator. The latter is quite detailed in dealing with the actual structure of Simulink files. Our goal was not to formalize this part, but the high-level translation strategies (i.e., algorithms). For the latter, it is far from obvious that the various choices made in the different strategies [13] result in semantically equivalent translations, hence the interest behind the proof of [32].

6.3 Other Functionalities of the Translator

The Translator proposes additional options that allow to explore different expressions of the obtained RCRS model.

The `-flat` option flattens the Simulink model, usually a hierarchical block diagram, before performing the translation. The tradeoffs of using this option on the Analyzer are illustrated in Table 1.

Several options allow to deal with the typing of Simulink diagrams. By default, if the inputs or outputs of a block are typed, the Translator uses them in the blocks definitions generation. Other cases can be handled with the following options as described in [29]. Option `-const` generates additional parameters with types for blocks of type `Constant`, `Relational`, etc. These extra parameters in the definition of a (composed) block ensure the automated execution of the Analyzer in all cases. Option `-generic` generates generic types for all inputs and outputs of blocks. Then the simplified expression obtained is as general as possible. Finally, option `-type` takes a type as parameter expressed in Isabelle, e.g., `real`, and instantiates all untyped inputs and outputs with the given type.

The `-equiv` option generates the RCRS terms with all translation strategies, as well as an extra theory proving the semantical equivalence of the terms.

The `-iter` option additionally generates the property transformer corresponding to the (simplified) system. Additionally the option `-consist` can be used here

to generate the proof that the iteration of the property transformer is consistent as illustrated in Section 7.3.

Finally, the `-sim` option also generates a python template for simulation purposes. The generated theory contains instructions to write the simplified RCRS formula in a format understood by python. This python file contains all the necessary definitions to simulate the simplified RCRS formula of the system and plot it. It also defines additional simulation options such as: `-dt` for the simulation step, `-time` for the duration of simulation and `-plot` to represent in python the simulation results.

7 An Automotive Case Study

We have used the RCRS Toolset on several case studies, the most significant of which is a real-world benchmark provided by Toyota [22]. The benchmark consists of a set of Simulink diagrams modeling a Fuel Control System (FCS).[4] A typical diagram in the above suite contains 3 levels of hierarchy, 104 Simulink blocks in total (out of which 8 subsystems), and 101 wires (out of which 8 result in feedback loops).

In the rest of this section we report on experimental results obtained from running the RCRS Toolset on a Toyota Fuel Control System (FCS) Simulink model, an excerpt of which is shown in Fig. 10. Table 1 presents metrics obtained from running the Translator and the Analyzer on this model, taking into account different translation options. More specifically, we provide

- $\mathcal{T}_{\text{trans}}$ the translation time (in seconds),
- $\mathcal{L}_{\text{cpt}}$ the length of the generated RCRS terms (number of characters),
- $\mathcal{N}_{\text{cpt}}$ the number of generated RCRS terms
- $\mathcal{T}_{\text{simp}}$ the analysis time (in seconds), and
- $\mathcal{L}_{\text{simp}}$ the length of the simplified RCRS term (number of characters).

The results are obtained with all translation strategies: `-fp`, `-ic`, `-ici`, and `-nfb`, and considering `-flat` whenever suitable. For example, using the Translator on the FCS Simulink model with the `-nfb` option results in a `.thy` file of 1671 lines and 56947 characters.

7.1 Running the Translator

We ran the translator as follows, adding different additional options for each different experiment:

```
./simulink2isabelle.py afcs.slx -const -type real
```

We remark that the translation time is negligible in all cases (see Table 1).

7.2 Running the Analyzer

The Analyzer simplifies this model to a top-level atomic RCRS component with no inputs, 7 (external) outputs and 14 state variables (note that all internal wires have been automatically eliminated in this top-level description). For the theory produced with `-nfb` simplification takes 13 seconds and generates a formula which is 10186 characters long. Note that simplification times are significantly longer for theories produced using other translation methods than the feedbackless one. This is because simplifications involving the `feedback` operator are resource consuming.

The Analyzer detects no incompatibilities and computes a simplified top-level atomic component whose description is between 8000 and 10000 characters long depending on the translation strategy used (an excerpt is shown in Fig. 11). This formula simplifies automatically to a non-trivial condition of the form $s \geq 0$, where s is a state variable. This condition is automatically inferred by RCRS and is due to the fact that the stateful block corresponding to s (which is an Integrator) feeds its output into a square root block. This condition indicates that the state of the Integrator should never become negative during execution of the model, and can be seen as an automatically derived invariant which ensures the consistency of the entire FCS model. However, this invariant still needs to be proven. We show how to do this next.

7.3 Proving the Automatically Inferred Invariant

We would like to prove that the automatically inferred invariant $s \geq 0$ above is satisfied during execution of the model. In this case the model does not have any inputs, and all components are deterministic, which implies that the condition $s \geq 0$ must be true at every execution step. In such cases, we have a completely automatic method. We show next how exactly this method works on the FCS model.

Since RCRS discretizes time by performing Euler integration with step dt, this property generally depends on the value of dt: the invariant might hold for some values and not hold for other values. Here, we prove a stronger result, namely, that it holds for any $dt > 0$.

For this we call first the translator with the `-iter` option to obtain the property transformer characterizing the system's behavior over time:

[4] We downloaded the Simulink models from https://cps-vo.org/group/ARCH/benchmarks. One of those models is made available both in the figshare repository [15] and the distribution.

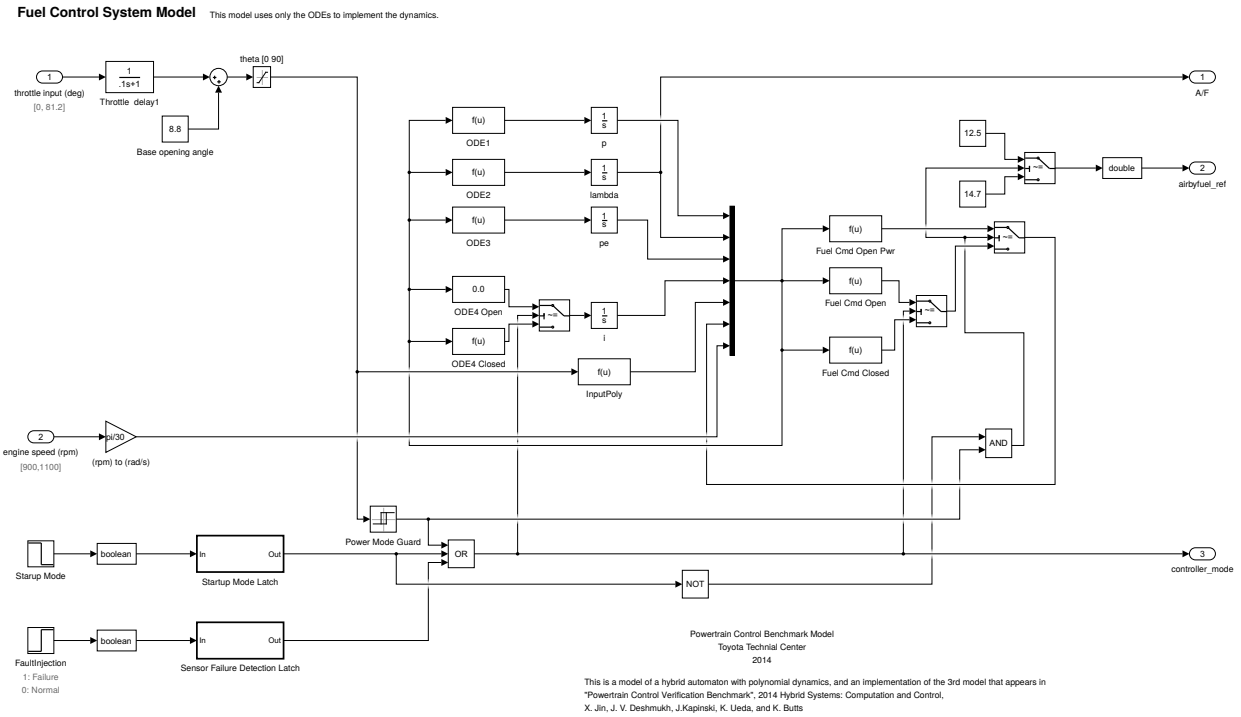


Fig. 10: Excerpt of the Toyota Fuel Control System Simulink model [22].

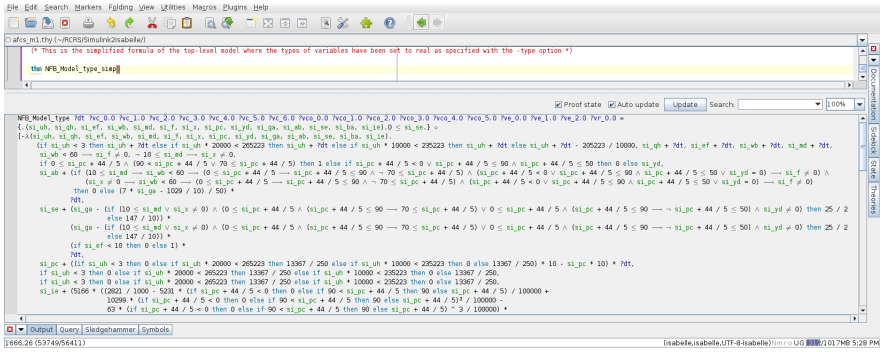


Fig. 11: The simplified predicate transformer of the Fuel Control System.

			-fp		-ic		-ici		-nfb
				-flat		-flat		-flat	
Translator	$\mathcal{T}_{\text{trans}}$	(sec)	0.33	0.30	0.37	0.35	0.41	0.36	0.29
	$\mathcal{L}_{\text{cpt}}$	(#chars)	33126	28999	54624	71362	107362	151537	56947
	$\mathcal{N}_{\text{cpt}}$	(#defs)	148	127	148	127	258	244	269
Analyzer	$\mathcal{T}_{\text{simp}}$	(sec)	284	920	78	294	65	114	13
	$\mathcal{L}_{\text{simp}}$	(#chars)	8343	8324	8324	8324	8344	10167	10186

Table 1: Experimental results for the FCS model.

```
./simulink2isabelle.py afcs.slx -const -type real
    -iter -consist
```

The `-iter` option will generate the following expression for the iterated simplified system:

```
definition "init_vals = (0.017,0.6353,0.0,
  0.0,14.7,0.5573,0.0,0.0,0.0,0,0.0,0.0,0.0,0)"
```

```
simplify_RCRS "FCS_iter dt =
     DelayFeedbackInit init_vals (FCS dt)"
  "(x)" "(y)"
  use(FCS_simp init_vals_def iter_simps)
  type "real"
```

Here `init_vals` is the tuple of all initial values of all stateful blocks (delays, integrators, transfer functions),

and `DelayFeedbackInit` constructs the systems that takes as input infinite sequences of input values (empty tuples in this case) and produces infinite sequences of output values. `DelayFeedbackInit` iterates the system `FCS` indefinitely, starting from the initial values of the state variables.

Proving the inferred invariant is done automatically in Isabelle using a complete rule, as we show below. Although the rule is complete it relies on Isabelle's simplification mechanism to simplify the first order formulas generated by the rule. The consistency of FCS is obtained by the following lemma and its proof:

```
lemma "dt > 0 ⟹ prec (FCS_iter dt) x"
  apply (unfold FCS_iter_simp prec_assert_update)
  apply (subst iter_prec_induction_unit_iff)
  by auto
```

This lemma states that the invariant of `FCS_iter dt` is true for any input sequence `x`, assuming `dt` > 0. The lemma together with its proof is generated automatically by the translator when using the option `-consist`. First step in the proof replaces `FCS_iter dt` by its simplified version using `FCS_iter_simp`, and then it calculates the input condition using the RCRS library lemma:

```
lemma prec_assert_update: prec ({.p.}  o  [-f-]) =
    p
```

Next step uses the complete induction rule `iter_prec_induction_unit_iff` to reduce the problem to proving first order statements over the logic used for expressing the system (in this case real arithmetic). Finally `auto` is used to prove all these statements.

7.4 Validation by Simulation

The Translator defines a formal semantics for Simulink diagrams in terms of RCRS models. One question that may be raised is how the RCRS semantics compares to Simulink's own semantics, which is essentially "what the Simulink simulator does". Note that there is not one simulator behavior in Simulink, but many, as the tool offers many different types of solvers, with many parameters/options each, etc. Here, we just provide a simple validation by simulation.

First, we use our Translator to generate Python simulation code:

```
./simulink2isabelle.py afcs.slx -const -type real
   -sim
```

Then we execute the generated simulation script with

```
python afcs.py -dt 0.001 -time 50 -plot
```

We compare the simulation results obtained from Simulink to those obtained from the RCRS-generated simulation code. Since this model is closed (i.e., has no external inputs) and deterministic, it only has a single behavior (assuming all Simulink solver parameters are fixed). Therefore, we only generate one simulation plot for each method. Graphically the plots look identical and if we compute their maximum difference between the values computed by Simulink and our simulation we find that it is on the order of magnitude e-05. Better results can be obtained by reducing the time step dt. For instance, a step of 5e-05 gives a maximum error of 2.0354e-06.

8 Related Work

The theory of RCRS is related to FOCUS [7], input-output automata [23], reactive modules [3], interface automata [9], and Dill's trace theory [12]. Although RCRS shares with these theories several compositionality principles, such as refinement, it differs from them in important ways. Specifically, FOCUS, IO-automata, and reactive modules, are limited to input-receptive systems, while RCRS can handle non-input-receptive specifications (for an extensive discussion of the benefits of the latter see [41]). Interface automata and Dill's trace theory are asynchronous and low-level theories whereas RCRS is synchronous and can model and reason about systems at a higher, symbolic level. Additional discussion of work related to the RCRS theory can be found in [30,34]. In the rest of this section we limit our discussion to work related to the RCRS Toolset.

The RCRS Toolset is related to verification tools for hybrid systems, such as Hytech [21], Charon [2], SpaceEx [19], KeYmaera [35], and C2E2 [17]. However, the focus of the above tools is on verification (using reachability analysis or theorem proving techniques) whereas the focus of RCRS is compositional reasoning and static analysis. In particular, RCRS does not perform reachability analysis (except of the kind described in Section 7.3). As far as we know, none of the above tools allow modeling of both non-deterministic and non-input-receptive systems.

SimCheck [38] is a contract-based tool for Simulink which allows the user to annotate ports and wires with types and also units (e.g., *cm*). A translation to Yices [16] supports the automated static and behavioral type checking. In contrast to SimCheck, RCRS automatically infers the types and dimensions of signals from the Simulink diagrams, although it does not infer or check for physical units. RCRS contracts can be very expressive and can relate input, output, and state variables, whereas SimCheck contracts appear much more restrictive (separate constraints on inputs and outputs

but not on both). SimCheck also does not appear to have a notion of refinement, neither a compositional algebra of block diagrams.

Several tools translate Simulink diagrams to various types of models, including to Hybrid Automata [1], BIP [39], NuSMV [25], Lustre [42], Boogie [36], Timed Interval Calculus [8], Function Blocks [45], I/O Extended Finite Automata [46], Hybrid CSP [47], and SpaceEx [27]. These target languages are all different from RCRS, and in particular do not allow, to our knowledge, to describe both non-deterministic and non-input-receptive systems. In addition, it is unclear whether these approaches provide any formal guarantees on the determinism of the translation. For example, the order in which blocks in the Simulink diagram are processed might a-priori influence the result. Some works fix this order, e.g., [36] computes the control flow graph and translates the model according to this computed order. In contrast, the RCRS Translator has been formally and mechanically proven to be deterministic, in the sense that the different results of the translation are semantically equivalent independently of the composition order [32,13].

RCRS can be seen as a type system and tool for Simulink. The RCRS Analyzer can be seen as performing type checking and type inference (synthesizing contracts of composite components from the contracts of their children components). From that point of view, RCRS is related to type systems for programming languages, such as Standard ML [26], Refinement Types for ML [18], Dependent Types [44], or Liquid Types [37]. The main difference between these languages and RCRS is that RCRS focuses on reactive systems and dynamic behavior, whereas the above are classic programming languages focusing on input-output behavior. Another difference is that the above type systems use techniques based on subtypes and dependent-types that allow checking invariants about the program at compile time. These techniques need to be in general automated, and therefore apply only to certain classes of decidable problems. The compatibility checks that RCRS performs for system compositions are more general, and not necessarily decidable, as the logic on which RCRS operates is very general. In the RCRS case checking compatibility of system compositions is reduced to checking satisfiability of formulas. If the underlying logic of these formulas happens to be decidable, then we can have an automatic compatibility test.

9 Conclusions

The RCRS Toolset is a state-of-the-art compositional formal modeling and reasoning tool for embedded and cyber-physical systems. It is, to our knowledge, the only tool which is able to capture both non-deterministic and non-input-receptive reactive systems. It is able to perform various types of formal reasoning (built on top of Isabelle), including compatibility checks, contract synthesis, and refinement checking. It also comes with a Translator of Simulink diagrams. The Translator implements a set of translation strategies which have been formally verified to yield deterministic results. The RCRS Toolset has been evaluated on a set of examples, including real-world benchmarks provided by industrial partners (Toyota).

Future work includes extending the Toolset along several dimensions. First, the library of Simulink blocks can be extended to include more of Simulink's basic blocks. More challenging would be an extension to handle Stateflow state machines. Such machines can have an arbitrary number of states and structure. Therefore, automatically extracting a contract for such a machine involves non-trivial static analysis of the structure of the machine. Another non-trivial extension involves being able to handle continuous-time dynamics without discretizing time. An alternative is to be able to handle other types of integration than the basic Euler scheme which RCRS currently uses. It would be interesting also to be able to handle acausal models, similar to those captured in languages such as Modelica [20]. Several improvements to the usability, effectiveness, and efficiency of the tools in the Toolset can also be envisaged. Among those are techniques to localize incompatibilities when debugging a model.

Acknowledgements This work has been supported by the Academy of Finland and the U.S. National Science Foundation (awards 1329759 and 1801546). Dragomir was partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement #730080 (ESROCOS). Preoteasa was partially supported by the ECSEL JU MegaM@Rt2 project under grant agreement #737494.

References

1. A. Agrawal, G. Simon, and G. Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109:43 – 56, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).
2. R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In

4th International Workshop on Hybrid Systems: Computation and Control, HSCC '01, pages 33–48. Springer, 2001.
3. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
4. R.-J. Back and J. von Wright. *Refinement Calculus*. Springer, 1998.
5. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.
6. R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In *IFIP TC10/WG 10.2 Intl. Conf. on Theorem Provers in Circuit Design*, pages 129–156. North-Holland Publishing Co., 1992.
7. M. Broy and K. Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer, 2001.
8. C. Chen, J. S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*, 21(5):451–483, 2009.
9. L. de Alfaro and T. Henzinger. Interface automata. In *Foundations of Software Engineering (FSE)*. ACM Press, 2001.
10. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
11. E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.
12. D. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. MIT Press, Cambridge, MA, USA, 1987.
13. I. Dragomir, V. Preoteasa, and S. Tripakis. Compositional Semantics and Analysis of Hierarchical Block Diagrams. In *SPIN*, pages 38–56. Springer, 2016.
14. I. Dragomir, V. Preoteasa, and S. Tripakis. The Refinement Calculus of Reactive Systems Toolset. In *TACAS*, 2018.
15. I. Dragomir, V. Preoteasa, and S. Tripakis. The Refinement Calculus of Reactive Systems Toolset - Feb 2018. figshare. https://doi.org/10.6084/m9.figshare.5900911, Feb. 2018.
16. B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.
17. C. Fan, B. Qi, S. Mitra, M. Viswanathan, and P. S. Duggirala. Automatic reachability analysis for nonlinear hybrid models with C2E2. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 531–538. Springer, 2016.
18. T. Freeman and F. Pfenning. Refinement Types for ML. *SIGPLAN Not.*, 26(6):268–277, May 1991.
19. G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.
20. P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley, 2 edition, 2014.
21. T. Henzinger, P.-H. Ho, and H. Wong Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1, 1997.
22. X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain Control Verification Benchmark. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC'14, pages 253–262. ACM, 2014.
23. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
24. S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(7):950–956, 1994.
25. B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In *Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 606–620. Springer, 2006.
26. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.
27. S. Minopoli and G. Frehse. SL2SX Translator: From Simulink to SpaceEx Verification Tool. In *19th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2016.
28. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
29. V. Preoteasa, I. Dragomir, and S. Tripakis. Type Inference of Simulink Hierarchical Block Diagrams in Isabelle. In *FORTE*, 2017.
30. V. Preoteasa, I. Dragomir, and S. Tripakis. The Refinement Calculus of Reactive Systems. *CoRR*, abs/1710.03979, 2018.
31. V. Preoteasa, I. Dragomir, and S. Tripakis. Mechanically proving determinacy of hierarchical block diagram translations. In C. Enea and R. Piskac, editors, *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, volume 11388 of *Lecture Notes in Computer Science*, pages 577–600. Springer, 2019.
32. V. Preoteasa, I. Dragomir, and S. Tripakis. Mechanically Proving Determinacy of Hierarchical Block Diagram Translations. In *VMCAI 2019 - 20th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2019. Extended version available as arXiv report 1611.01337.
33. V. Preoteasa and S. Tripakis. Refinement calculus of reactive systems. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10, Oct 2014.
34. 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.
35. J. Quesel, S. Mitsch, S. M. Loos, N. Arechiga, and A. Platzer. How to model and prove hybrid systems with keymaera: a tutorial on safety. *STTT*, 18(1):67–91, 2016.

36. R. Reicherdt and S. Glesner. Formal Verification of Discrete-Time MATLAB/Simulink Models Using Boogie. In D. Giannakopoulou and G. Salaün, editors, *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 190–204, Cham, 2014. Springer International Publishing.
37. P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.
38. P. Roy and N. Shankar. SimCheck: a contract type system for Simulink. *Innovations in Systems and Software Engineering*, 7(2):73–83, 2011.
39. V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis. Compositional translation of Simulink models into synchronous BIP. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 217–220, July 2010.
40. A. P. Sistla, M. Y. Vardi, and P. Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.
41. 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.
42. S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating Discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, Nov. 2005.
43. 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.
44. H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *POPL*, pages 214–227. ACM, 1999.
45. C. Yang and V. Vyatkin. Transformation of Simulink models to IEC 61499 Function Blocks for verification of distributed control systems. *Control Engineering Practice*, 20(12):1259–1269, 2012.
46. C. Zhou and R. Kumar. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.
47. L. Zou, N. Zhany, S. Wang, M. Franzle, and S. Qin. Verifying Simulink diagrams via a Hybrid Hoare Logic Prover. In *Embedded Software (EMSOFT)*, Sept 2013.