The Refinement Calculus of Reactive Systems Toolset

Iulia Dragomir[1(✉)], Viorel Preoteasa[2(✉)], and Stavros Tripakis[2,3(✉)]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, France
iulia.dragomir@univ-grenoble-alpes.fr
[2] Aalto University, Espoo, Finland
[3] University of California, Berkeley, USA

Abstract. We present the Refinement Calculus of Reactive Systems Toolset, an environment for compositional modeling and reasoning about reactive systems, built on top of Isabelle, Simulink, and Python.

1 Introduction

The *Refinement Calculus of Reactive Systems* (RCRS) is a compositional framework for modeling and reasoning about reactive systems. RCRS has been inspired by component-based frameworks such as interface automata [3] and has its origins in the theory of relational interfaces [14]. The theory of RCRS has been introduced in [13] and is thoroughly described in [11].

RCRS comes with a publicly available toolset, the *RCRS toolset* (Fig. 1), which consists of:

- A full implementation of RCRS in the Isabelle proof assistant [9].
- A set of analysis procedures for RCRS components, 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.

An extended version of this paper contains an additional six-page appendix describing a demo of the RCRS toolset [6]. The extended paper can also be found in a figshare repository [7]. The figshare repository contains all data (code and models) required to reproduce all results of this paper as well as of [6]: see Section "Data Availability Statement" for more details. The RCRS toolset can be downloaded also from the RCRS web page: http://rcrs.cs.aalto.fi/.

This work has been supported by the Academy of Finland and the U.S. National Science Foundation (awards #1329759 and #1139138).

I. Dragomir—Partially supported by the H2020 Programme SRC ESROCOS and ERGO projects.

Grenoble INP—Institute of Engineering Univ. Grenoble Alpes.

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10806, pp. 201–208, 2018.
https://doi.org/10.1007/978-3-319-89963-3_12

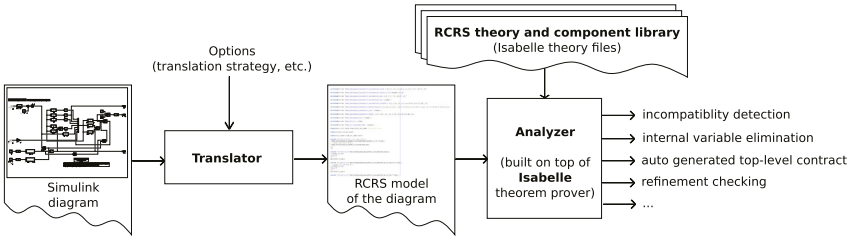


Fig. 1. The RCRS toolset.

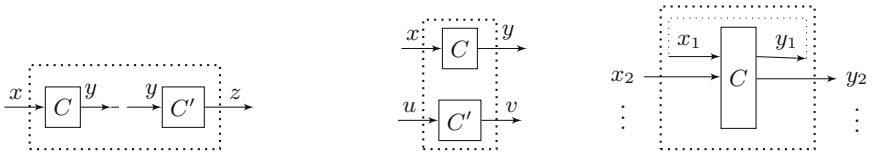
2 Modeling Systems in RCRS

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

```
definition "Id = [: x ⇝ y . y = x :]"
definition "Add = [: (x, y) ⇝ z . z = x + y :]"
definition "Constant c = [: x::unit ⇝ y . y = c :]"
definition "UnitDelay = [: (x,s) ⇝ (y,s') . y = s ∧ s' = x :]"
definition "SqrRoot = {. x . x ≥ 0 .}  o  [- x ⇝ √x -]"
definition "NonDetSqrt = {. x . x ≥ 0 .}  o  [: x ⇝ y . y ≥ 0 :]"
definition "ReceptiveSqrt = [: x ⇝ y . x ≥ 0 ⟶ y = √x :]"
definition "A = {. x . □◊x .}  o  [: x ⇝ y . □◊y :]"
```

`Id` models the identity function: it takes input x and returns y such that $y = x$. `Add` returns the sum of its two inputs. `Constant` is parameterized by `c`, takes no input (equivalent to saying that its input variable is of type `unit`), and returns an output which is always equal to `c`. `UnitDelay` is a *stateful* component: `s` is the current-state variable and `s'` is the next-state variable. `SqrRoot` is a *non-input-receptive* component: its input `x` is required to satisfy `x`$\geq$`0`. (`SqrRoot` may be considered non-atomic as it is defined as the serial composition of two predicate transformers – see Sect. 3.) `NonDetSqrt` is a *non-deterministic* version of `SqrRoot`: it returns an arbitrary (but non-negative) `y`, and not necessarily the square-root of `x`. `ReceptiveSqrt` is an input-receptive version of `SqrRoot`: it accepts negative inputs, but may return an arbitrary output for such inputs. RCRS also allows to describe components using the temporal logic QLTL, an extension of LTL with quantifiers [11]. An example is component `A` above. `A` accepts an infinite input sequence of `x`'s, provided `x` is infinitely often true, and returns a (non-deterministic) output sequence which satisfies the same property.

Composite components are formed by composing other (atomic or composite) components using three primitive composition operators, as illustrated in Fig. 2: C `o` C' (in series) connects outputs of C to inputs of C'; C `**` C' (in parallel) "stacks" C and C' "on top of each other"; and `feedback`(C) connects the first



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

Fig. 2. The three composition operators of RCRS.

output of C to its first input. These operators are sufficient to express any block diagram, as described in Sect. 4.

3 The Implementation of RCRS in Isabelle

RCRS is fully implemented in the Isabelle theorem prover. The RCRS implementation currently consists of 22 Isabelle *theories* (`.thy` files), totalling 27588 lines of Isabelle code. Some of the main theories are described next.

Theory `Refinement.thy` (1209 lines) contains a standard implementation of refinement calculus [1]. Systems are modeled as monotonic predicate transformers [4] 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.

Theory `RefinementReactive.thy` (1144 lines) extends `Reactive.thy` to reactive systems by introducing predicates over infinite traces in addition to predicates over values, and *property* transformers in addition to predicate transformers [11,13].

Theory `Temporal.thy` (788 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 $(x > 0)$: (`nat` $\rightarrow$ `real`) $\rightarrow$ `bool`, returns another predicate on infinite traces $\Box(x > 0)$: (`nat` $\rightarrow$ `real`) $\rightarrow$ `bool`. Temporal operators have been implemented to be polymorphic in the sense that they apply to predicates over an arbitrary number of variables.

Theory `Simulink.thy` (873 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). In addition to discrete-time, we can handle continuous-time blocks with a fixed-step forward Euler integration scheme. For example, Simulink's integrator block can be defined in two equivalent ways as follows:

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

The syntax `[- x⇝` f`(x) -]` assumes that f is a function, whereas `[: :]` can be used also for relations (i.e., non-deterministic systems). Using the former instead

of the latter to describe deterministic systems aids the Analyzer to perform simplifications – see Sect. 5.

Theory `SimplifyRCRS.thy` (2175 lines) implements several of the Analyzer's procedures. In particular, it contains a simplification procedure which reduces composite RCRS components into atomic ones (see Sect. 5).

In addition to the above, there are several theories containing a proof of correctness of our block-diagram translation strategies (see Sect. 4 and [10]), dealing with Simulink types [12], generating Python simulation code, and many more. A detailed description of all these theories and graphs depicting their dependencies is included in the documentation of the toolset.

The syntax of RCRS components is implemented in Isabelle using a *shallow embedding* [2]. 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 [11]. 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.

4 The Translator

The Translator, called `simulink2isabelle`, translates *hierarchical block diagrams* (HBDs), and in particular Simulink models, into RCRS theories [5]. The Translator (implemented in about 7100 lines of Python code) 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: (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.

As shown in [5], there are many possible ways to translate a block diagram into an algebra of components with the three primitive composition operators of RCRS. This means that step (2) above is not unique. `simulink2isabelle` implements the several translation strategies proposed in [5] as user options.

For example, when run on the Simulink diagram of Fig. 3, the Translator produces a file similar to the one shown in Fig. 4. `IC_Model` and `FP_Model` are composite RCRS components generated automatically w.r.t. two different translation strategies, implemented by user options `-ic` and `-fp`. The `simplify_RCRS` construct is explained in Sect. 5 that follows.

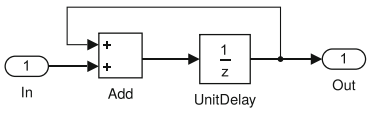


Fig. 3. A Simulink diagram.

Other user options to the Translator include: whether to flatten the input diagram, optional typing information for wires, and whether to generate in addition

to the top-level STS component, a QLTL component representing the temporal behavior of the system. The user can also ask the Translator to generate: (1) components w.r.t. all translation strategies; (2) the corresponding theorems showing that these components are all semantically equivalent; and (3) Python simulation scripts for the top-level component.

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

Fig. 4. Auto-generated Isabelle theory for the Simulink diagram of Fig. 3

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"`, where `C` is a (generally composite) component and `in`, `out` are (tuples of) names for its input and output variables. When such a statement is executed in Isabelle, it performs the following steps: (1) It creates the definition `Model = C`. (2) It *expands* `C`, meaning that it replaces all atomic components and all composition operators in `C` with their definitions. This results in an Isabelle expression `E`. `E` is generally a complicated expression, containing formulas with quantifiers, `case` expressions for tuples, function compositions, and several other operators. (3) `simplify_RCRS` *simplifies* `E`, by eliminating quantifiers, renaming variables, and performing several other simplifications. The simplified expression, `F`, is of the form $\{.p.\}$ `o` `[:`r`:]`, where p is a predicate on input variables and r is a relation on input and

output variables. That is, `F` is an atomic RCRS component. (4) `simplify_RCRS` generates a theorem stating that `Model` is semantically equivalent to `F`, and also the mechanized proof of this theorem (in Isabelle). 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.) neccessary for the Analyzer to produce the simplifications and their mechanical proofs, without user interaction.

For example, when the theory in Fig. 4 is executed, the following theorem is generated and proved automatically:

$$\texttt{Model = [- (g, s)} \rightsquigarrow \texttt{(s, s+g) -]}$$

where `Model` is either `IC_Model` or `FP_Model`. The rightmost expression is the automatically generated simplification of the top-level system to an atomic RCRS component.

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 $\bot$ (i.e., false). Thus, in this usage scenario, RCRS can be seen as a static analysis and behavioral type checking and inference tool for Simulink.

6 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 [8]. The benchmark consists of a set of Simulink diagrams modeling a Fuel Control System.[1] 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 are feedbacks, the most complex composition operator in RCRS). Using the Translator on this diagram results in a `.thy` file of 1671 lines and 57037 characters. Translation time is negligible. The Analyzer simplifies this model to a top-level atomic STS 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). Simplification takes approximately 15 seconds and generates a formula which is 8337 characters long. The formula is consistent (not false), which proves statically that the original Simulink diagram has no incompatibilities. More details about the case study can be found in [5,6].

[1] We downloaded the Simulink models from https://cps-vo.org/group/ARCH/benchmarks. One of those models is made available in the figshare repository [7] – see also Section "Data Availability Statement".

7 Data Availability Statement

All results mentioned in this paper as well as in the extended version of this paper [6] are fully reproducible using the code, data, and instructions available in the figshare repository: https://doi.org/10.6084/m9.figshare.5900911.v1.

The figshare repository contains the full implementation of the RCRS toolset, including the formalization of RCRS in Isabelle, the Analyzer, the RCRS Simulink library, and the Translator. The figshare repository also contains sample Simulink models, including the Toyota model discussed in Sect. 6, a demo file named `RCRS_Demo.thy`, and detailed step-by-step instructions on how to conduct a demonstration and how to reproduce the results of this paper. Documentation on RCRS is also provided.

The figshare repository provides a snapshot of RCRS as of February 2018. Further developments of RCRS will be reflected on the RCRS web page: http://rcrs.cs.aalto.fi/.

References

1. Back, R.-J., von Wright, J.: Refinement Calculus. Springer, Heidelberg (1998)
2. Boulton, R.J., Gordon, A., Gordon, M.J.C., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design, pp. 129–156. North-Holland Publishing Co., Amsterdam (1992)
3. de Alfaro, L., Henzinger, T.: Interface automata. In: Foundations of Software Engineering (FSE). ACM Press, New York (2001)
4. Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. Comm. ACM **18**(8), 453–457 (1975)
5. Dragomir, I., Preoteasa, V., Tripakis, S.: Compositional semantics and analysis of hierarchical block diagrams. In: Bošnački, D., Wijs, A. (eds.) SPIN 2016. LNCS, vol. 9641, pp. 38–56. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32582-8_3
6. Dragomir, I., Preoteasa, V., Tripakis, S.: The Refinement Calculus of Reactive Systems Toolset. CoRR, abs/1710.08195:1–12 (2017)
7. Dragomir, I., Preoteasa, V., Tripakis, S.: The Refinement Calculus of Reactive Systems Toolset, February 2018. figshare. https://doi.org/10.6084/m9.figshare.5900911.v1
8. Jin, X., Deshmukh, J.V., Kapinski, J., Ueda, K., Butts, K.: Powertrain control verification benchmark. In: Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control, HSCC 2014, pp. 253–262. ACM (2014)
9. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
10. Preoteasa, V., Dragomir, I., Tripakis, S.: A nondeterministic and abstract algorithm for translating hierarchical block diagrams. CoRR, abs/1611.01337 (2016)
11. Preoteasa, V., Dragomir, I., Tripakis, S.: The Refinement Calculus of Reactive Systems. CoRR, abs/1710.03979 (2017)
12. Preoteasa, V., Dragomir, I., Tripakis, S.: Type inference of Simulink hierarchical block diagrams in Isabelle. In: Bouajjani, A., Silva, A. (eds.) FORTE 2017. LNCS, vol. 10321, pp. 194–209. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60225-7_14

13. Preoteasa, V., Tripakis, S.: Refinement calculus of reactive systems. In: 2014 International Conference on Embedded Software (EMSOFT), pp. 1–10, October 2014
14. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM TOPLAS **33**(4), 14:1–14:41 (2011)