Compositional Assume-Guarantee Reasoning of Control Law Diagrams using UTP

Kangfeng Ye Simon Foster
Jim Woodcock
University of York, UK

{kangfeng.ye,simon.foster,jim.woodcock}@york.ac.uk

July 23, 2018

Abstract

This report is a summary of our work for the VeTSS funded project "Mechanised Assume-Guarantee Reasoning for Control Law Diagrams via Circus". Our Assume-Guarantee (AG) reasoning of control law diagrams is based on Hoare and He's Unifying Theories of Programming and their theory of designs. In this report, we present developed theories and laws to map discrete-time Simulink block diagrams to designs in UTP, calculate assumptions and guarantees, and verify properties for modelled systems. A practical application of our AG reasoning to an aircraft cabin pressure control subsystem is also presented. In addition, all mechanised theories in Isabelle/UTP are attached in Appendices. In the end of this report, we summarise current progress for each work package.

Contents

1 Introduction

Control law diagrams such as Simulink [1] and OpenModelica [2] are widely used industrial languages and tool-sets for expressing control laws, including support for simulation and code generation. In particular, Simulink actually is a *de facto* standard in many areas in industry. Its model based design, simulation and code generation make it a very efficient and cost-effective way to develop complex systems. Though empirical analysis through simulation is an important technique to explore and refine models, only formal verification can make specific mathematical guarantees about behaviour, which is crucial to ensure safety of associated implementations. Whilst verification facilities for Simulink exist [3, 4, 5, 6, 7, 8], there is still a need for assertional reasoning techniques that capture the full range of specifiable behaviour, provide non-deterministic specification constructs, and support compositional verification. Such techniques also need to be sufficiently expressive to handle the plethora of additional languages and modelling notations that are used by industry in concert with Simulink, in order to allow formulation of heterogeneous "multi-models" that capture the different paradigms and disciplines used in large scale systems [9]. Applicable tool support for these techniques with a high degree of automation is also of vital importance to enable adoption by industry. Since Simulink diagrams are data rich and usually have an uncountably infinite state space, model checking alone is insufficient and there is a need for theorem proving facilities.

Assume-Guarantee (AG) reasoning is a valuable compositional verification technique for reactive systems [10, 11, 12]. In AG, one demonstrates composite system level properties by decomposing them into a number of contracts for each component subsystem. Each contract specifies the guarantees that the subsystem will make about its behaviour, under certain specified assumptions of the subsystem's environment. Such a decomposition is vital in order to make verification of a complex system tractable, and to allow development of subsystems by separate teams. AG reasoning has previously been applied to verification of discrete time Simulink control law diagrams through mappings into synchronous languages like Lustre [13] and Kahn Process Networks [5]. However such formalisms, whilst theoretically and practically appealing, are limited to expressing processes that are inherently deterministic and non-terminating in nature. Refinement Calculus for Reactive Systems (RCRS) [8] is a methodology that can be applied to reason about non-deterministic and non-input-receptive systems by treating programs as predicate transformers. However, it is not able to reason about multi-rate Simulink diagrams and algebraic loops. Almost all these verification facilities translate Simulink to sequential languages, synchronous languages or reactive languages [7], and then use verification methods for these languages to reason about Simulink diagrams. There is a need to develop a reasoning technique that is based on the semantic understanding of simulation in Simulink as described in Section 2.1. Thus, it is necessary to translate to several additional notations where AG verification can be performed, which hampers both traceability and composition with other languages of different paradigms. What is needed is a rich unified language capable of AG reasoning, and supported by theorem proving, into which Simulink and associated notations can be losslessly translated.

Our proposed approach thus explores development of formal AG-based proof support for discrete-time Simulink diagrams through a semantic embedding of the theory of designs [14] in Unifying Theories of Programming (UTP) [15] in Isabelle/HOL [16] using our developed tool Isabelle/UTP [17]. Initially, we proposed to use *Circus* [18], a formal modelling language for concurrent and reactive systems in the style of CSP, to model Simulink diagrams as shown in [7], and then apply contract-based reasoning to *Circus*. A *Circus* model consists of a network of processes that communicate with one another solely via shared channels that carry typed data. Internal state variables are encapsulated and not directly observable by other parallel processes. *Circus* can capture a variety of languages at the semantic level, and thus supports the formulation of heterogeneous multi-models [9] by acting as a "lingua franca". In addition, a timed version of *Circus* is used to model multi-rate diagrams. However, a *Circus* model has more complex information of blocks in Simulink for AG reasoning. For example, the corresponding *Circus* process for a block uses channels to model connections in diagrams, a non-deterministic internal choice of all input channels to allow an arbitrary input order, and similarly an internal choice of output channels to allow an arbitrary output order.

In order to reason about the *Circus* model, we need to take trace information into account and traces inevitably are more complicated if there are many inputs and outputs for a block. Eventually, using model checking or theorem proving to verify *Circus* models becomes more difficult. According to the semantic understanding of simulation in Simulink in Section 2.1, actually the order of inputs and outputs is irrelevant. Therefore, we have changed our approach to use the theory of designs in UTP to enable AG reasoning for Simulink block diagrams.

A *design* in UTP is a relation between two predicates where the first predicate (precondition) records the assumption and the second one (postcondition) specifies the commitment. *Designs* are intrinsically suitable for modelling and reasoning about state-based programs (such as B machines [19] and Z notations [20]) but not necessary for reactive programs. For simulation of Simulink diagrams, we discretise the simulation time and abstract it into steps (natural numbers), and define inputs and outputs of Simulink blocks as a function from step numbers to a list of inputs or outputs. In this way, the reactive behaviour is encoded in the step numbers in functions. Finally, the theory of designs can be used to reason about reactive behaviour of Simulink diagrams without introduction of detailed implementation information .

Our work presented in this report has multiple contributions. The main contribution is to define a theoretical reasoning framework for control law block diagrams using the theory of designs in UTP. Each block or subsystem is translated to a design and then hierarchical connections of blocks are mapped to a variety of compositions of designs. Additionally, the refinement relation of designs, monotony of composition operators, and closure laws enable compositional reasoning of block diagrams using a contract-based methodology. The second contribution is our mechanisation of theories in the theorem prover Isabelle using our implementation of UTP, Isabelle/UTP. Then the practical contribution is our industrial case study of a subsystem in a safety critical aircraft cabin pressure control system.

In the next section, we describe the relevant preliminary background about Simulink and UTP. Then in Section 3, the assumptions we made are presented and a brief reasoning procedure is described. Section 4 defines our treatment of blocks in UTP and translations of a number of blocks are illustrated. Furthermore, we introduce our composition operators and their corresponding theorems in Section 5. Afterwards, in Section 6 we briefly describe the industrial case study. And we conclude our work in Section 7. Additionally, our mechanised theories, laws and case studies are attached in appendices.

2 Preliminaries

2.1 Control Law Diagrams and Simulink

Simulink is a model-based design modelling, analysis and simulation tool for signal processing systems and control systems. It offers a graphical modelling language which is based on hierarchical block diagrams. Its diagrams are composed of subsystems and blocks as well as connections between these subsystems and blocks. In addition, subsystems also can consists of others subsystems and blocks. And single function blocks have inputs and outputs, and some blocks also have internal states.

There is no formal semantics for Simulink. A consistent understanding [21, 22] of the simulation in Simulink is based on an *idealized* time model. All executions and updates of blocks are performed *instantaneously* (and infinitely fast) at exact simulation steps. Between the simulation steps, the system is *quiescent* and all values held on lines and blocks are constant. The inputs, states and outputs of a block can only be updated when there is a time hit for this block. Otherwise, all values held in the block are constant too though at exact simulation steps. According to this idealized time model, it is inappropriate to assume that blocks are sequentially executed. For example, for a block it is inappropriate to say it takes its inputs, calculates its outputs and states, and then outputs the results from this point of view. Simulation and code generation of Simulink diagrams use sequential semantics for implementation. But it is not always necessary. Simulink needs to have a mathematical and denotational semantics, which UTP provides.

Based on the idealized time model, a single function block can be regarded as a relation between its inputs and outputs. For instance, a unit delay block specifies that its initial output is equal to its initial condition and its subsequent output is equal to previous input. Then connections of blocks establish further relations between blocks. A directed connection from one block to another block specifies that the output of one block is equal to the input of another block. Finally, hierarchical block diagrams establish a relation network between blocks and subsystems.

2.2 Unifying Theories of Programming

UTP is a unified framework to provide a theoretical basis for describing and specifying computer languages across different paradigms such as imperative, functional, declarative, nondeterministic, concurrent, reactive and high-order. A theory in UTP is described using three parts: *alphabet*, a set of variable names for the theory to be studied; *signature*, rules of primitive statements of the theory and how to combine them together to get more complex program; and *healthiness conditions*, a set of mathematically provable laws or equations to characterise the theory.

The alphabetised relational calculus [23] is the most basic theory in UTP. A relation is defined as a predicate with undecorated variables (v) and decorated variables (v') in its alphabet. v denotes the observation made initially and v' denotes the observation made at the intermediate or final state.

The understanding of the simulation in Simulink is very similar to the concept "programs-as-predicates" [24]. This is the similar idea that the Refinement Calculus of Reactive Systems

(RCRS) [8] uses to reason about reactive systems. RCRS is a compositional formal reasoning framework for reactive systems. The language is based on monotonic property transformers which is an extension of monotonic predicate transformers [25]. This semantic understanding makes Unifying Theories of Programming (UTP) [15] intrinsically suitable for reasoning of the semantics of Simulink simulation because UTP uses an alphabetised predicate calculus to model computations.

Refinement calculus is an important concept in UTP. Program correctness is denoted by $S \sqsubseteq P$, which means that the observations of the program P must be a subset of the observations permitted by the specification S. For instance, $(x = 2)$ is a refinement of the predicate $(x > 1)$. A refinement sequence is shown in (1). $S1$ is more general and abstract specification than $S2$ and thus more easier to implement. The predicate *true* is the easiest one and can be implemented by anything. $P2$ is more specific and determinate program than $P1$ and thus $P2$ is more useful in general. *false* is the strongest predicate and it is impossible to implement in practice.

$$\mathbf{true} \sqsubseteq S1 \sqsubseteq S2 \sqsubseteq P1 \sqsubseteq P2 \sqsubseteq \mathbf{false} \tag{1}$$

2.2.1 Designs

Designs are a subset of the alphabetised predicates that use a particular variable *ok* to record information about the start and termination of programs. The behaviour of a design is described from initial observation and final observation by relating its precondition P (assumption) to the postcondition Q (commitment) as $P \vdash Q$ [14, 15] (assuming P holds initially, then Q is established). Therefore, the theory of designs is intrinsically suitable for assume-guarantee reasoning [26].

Definition 2.1 (Design)

$$P \vdash Q \triangleq P \wedge ok \Rightarrow Q \wedge ok'$$

A design is defined in 2.1 where *ok* records the program has started and ok' that it has terminated. It states that if the design has started $(ok = true)$ in a state satisfying its precondition P, then it will terminate $(ok' = true)$ with its postcondition Q established. We introduce some basic designs.

Definition 2.2 (Basic Designs)

$$\begin{aligned} \top_D &\triangleq \textbf{\textsf{\textit{true}}} \vdash \textbf{\textsf{\textit{false}}} = \neg\, ok && \text{[Miracle]} \\ \bot_D &\triangleq \textbf{\textsf{\textit{false}}} \vdash \textbf{\textsf{\textit{false}}} = \textbf{\textsf{true}} && \text{[Abort]} \\ (x := e) &\triangleq \left(\textbf{\textsf{\textit{true}}} \vdash x' = e \wedge y' = y \wedge \cdots\right) && \text{[Assignment]} \\ I\!I_{\boldsymbol{D}} &\triangleq (\textbf{\textsf{\textit{true}}} \vdash I\!I) && \text{[Skip]} \end{aligned}$$

Abort $(\bot_D)$ and miracle $(\top_D)$ are the top and bottom element of a complete lattice formed from designs under the refinement ordering. Abort $(\bot_D)$ is never guaranteed to terminate and miracle establishes the impossible. In addition, abort is refined by any other design and miracle refines any other designs. Assignment has precondition ***true*** provided the expression e is well-defined and establishes that only the variable x is changed to the value of e and other variables have not changed. The skip $I\!I_{\boldsymbol{D}}$ is a design identity that always terminates and leaves all variables unchanged.

Designs can be sequentially composed with the following theorem:

Theorem 2.1 (Sequential Composition)

$$(p_1 \vdash Q_1 \; ; P_2 \vdash Q_2) \quad = \quad ((p_1 \wedge \neg \; (Q_1 \; ; \neg \; P_2)) \vdash Q_1 \; ; \; Q_2) \qquad [p_1\text{-condition}]$$

A sequence of designs terminates when p_1 holds and Q_1 guarantees to establish P_2 provided p_1 is a condition. On termination, sequential composition of their postconditions is established. A condition is a particular predicate that only has input variables in its alphabet. In other words, a design of which its precondition is a condition only makes the assumption about its initial observation (input variables) and without output variables. That is the same case for our treatment of Simulink blocks. Furthermore, sequential composition has two important properties: associativity and monotonicity which are given in the theorem below.

Theorem 2.2 (Associativity, Monotonicity)

$$P_1;\ (P_2;\ P_3) = (P_1;\ P_2);\ P_3 \qquad [\text{Associativity}]$$
$$(P_1;\ Q_1) \sqsubseteq (P_2;\ Q_2) \qquad [\text{Monotonicity}]$$

Refinement of designs is given in the theorem below.

Theorem 2.3 (Refinement)

$$\begin{aligned} (P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2) \quad &= \quad (P_2 \sqsubseteq P_1) \wedge (Q_1 \sqsubseteq P_1 \wedge Q_2) \\ &= \quad [P_1 \Rightarrow P_2] \wedge [P_1 \wedge Q_2 \Rightarrow Q_1] \end{aligned}$$

Refinement of designs is achieved by either weakening the precondition, or strengthening the postcondition in the presence of the precondition.

In addition, we define two notations pre_D and $post_D$ to retrieve the precondition of the design and the postcondition in the presence of the precondition.

Definition 2.3 (pre_D and $post_D$)

$$pre_D\,(P \vdash Q) \triangleq P$$
$$post_D\,(P \vdash Q) \triangleq (P \Rightarrow Q)$$

3 Assumptions and General Procedure of Reasoning

3.1 Assumptions

Causality We assume the discrete-time systems modelled in Simulink diagrams are *causal* where the output at any time only depends on values of present and past inputs. Consequently, if inputs to a casual system are identical up to some time, their corresponding outputs must also be equal up to this time.

Single-rate This mechanised work captures only single sampling rate Simulink models, which means the timestamps of all simulation steps are multiples of a base period T. Eventually, steps are abstracted and measured by step numbers (natural numbers $\mathbb{N}$) and T is removed from its timestamp.

An *algebraic loop* occurs in simulation when there exists a signal loop with only direct feedthrough blocks in the loop, such as instantaneous feedback without delay in the loop. [5, 6, 27] assume there are no algebraic loops in Simulink diagrams and RCRS [8] identifies it as a future work. Our theoretical framework can reason about discrete-time block diagrams with algebraic loops: specifically check if there are solutions and find the solutions.

The signals in Simulink can have many data types, such as signed or unsigned integer, single float, double float, and boolean. The default type for signals are *double* in Simulink. This work uses real numbers in Isabelle/HOL as a universal type for all signals. Real numbers in Isabelle/HOL are modelled precisely using Cauchy sequences, which enables us to reason in the theorem prover. This is a reasonable simplification because all other types could be expressed using real numbers, such as boolean as 0 and 1.

3.2 General Procedure of Applying Assumption-Guarantee Reasoning

Simulink blocks are semantically mapped to designs in UTP where additionally we model assumptions of blocks to avoid unpredictable behaviour (such as a divide by zero error in the Divide block) and ensure healthiness of blocks. The general procedure of applying AG reasoning to Simulink blocks is given below.

- Single blocks and atomic subsystems are translated to single designs with assumptions and guarantees, as well as block parameters. This is shown in Section 4.
- Hierarchical block connections are modelled as compositions of designs (I) by means of sequential composition, parallel composition and feedback.
- Properties or Requirements of block diagrams (S) to be verified are modelled as designs as well.
- The refinement relation ($S \sqsubseteq I$) in UTP is used to verify if a given property is satisfied by a block diagram (or a subsystem) or not. Our approach supports compositional reasoning according to monotonicity of composition operators in terms of the refinement relation. Provided two properties S_1 and S_2 are verified to hold in two blocks or subsystems I_1 and I_2 respectively, then composition of the properties is satisfied by the composition of the blocks or subsystems in terms of the same operator.

$$(S_1 \sqsubseteq I_1 \wedge S_2 \sqsubseteq I_2) \Rightarrow (S_1 \ op \ S_2 \sqsubseteq I_1 \ op \ I_2)$$

4 Semantic Translation of Blocks

In this section, we focus on the methodology to map individual Simulink blocks to designs in UTP semantically. Basically, a block or subsystem is regarded as a relation between inputs and outputs. We use an undashed variable and a dashed variable to denotes input signals and output signals respectively.

4.1 State Space

The state space of our theory for block diagrams is composed of only one variable in addition to *ok*, named *inouts*. Originally, we defined it as a function from real numbers (time t) to a list of inputs or outputs. Each element in the list denotes an input or output and their order in the list is the order of input or output signals.

$$inouts : \mathbb{R}_{\geq 0} \rightarrow \operatorname{seq} \mathbb{R}$$

However, according to our single-rate assumption, the timestamp at time t is equal to multiples of a basic period T: $inouts(t) = inouts(n * T)$. Then T is abstracted away and only the step number n is related. Finally, it is defined below.

$$inouts : \mathbb{N} \rightarrow \operatorname{seq} \mathbb{R}$$

Then a block is a design that establishes the relation between an initial observation *inouts* (a list of input signals) and a final observation *inouts'* (a list of output signals). Additionally, this is subject to the assumption of the design.

4.2 Healthiness Condition: *SimBlock*

This healthiness condition characterises a block with a fixed number of inputs and outputs. Additionally it is feasible. A design is a feasible block if there exists at least a pair of *inouts* and *inouts'* that establishes both the precondition and postcondition of the design.

Definition 4.1 (*SimBlock*) *A design P with m inputs and n outputs is a Simulink block if P is **SimBlock** healthy.*

$$\textbf{\textit{SimBlock}}(m, n, P) \triangleq \begin{pmatrix} (pre_D(P) \wedge post_D(P) \neq \textbf{false}) \wedge \\ ((\forall\, n \bullet \#(inouts\ \ n) = m) \sqsubseteq Dom\ \ (pre_D(P) \wedge post_D(P))) \\ ((\forall\, n \bullet \#(inouts\ \ n) = n) \sqsubseteq Ran\ \ (pre_D(P) \wedge post_D(P))) \end{pmatrix}$$

where Dom and Ran calculate the characteristic predicate for domain and range. Their definitions are shown below.

$$Dom(P) \triangleq (\exists\, inouts' \bullet P)$$
$$Ran(P) \triangleq (\exists\, inouts \bullet P)$$

inps and *outps* are the operators to get the number of input signals and output signals for a block. They are implied from ***SimBlock*** of the block.

Definition 4.2 (*inps* and *outps*)

$$\textbf{\textit{SimBlock}}(m, n, P) \Rightarrow (inps(P) = m \wedge outps(P) = n)$$

Provided that P is a healthy block, inps returns the number of its inputs and outps returns the number of its outputs.

4.3 Blocks

In order to give definitions of the corresponding designs for Simulink blocks, firstly we define a design pattern *FBlock*. Then we illustrate definitions of two typical Simulink blocks and three additional virtual blocks using this pattern. The definitions of all other blocks could be found in Appendix A.

4.3.1 Pattern

We defined a pattern that is used to define all other blocks.

Definition 4.3 (*FBlock*)

$$
\begin{array}{l}
FBlock\,(f_1, m, n, f_2) \\
\triangleq \left(\begin{array}{l}
\forall\, nn \bullet f_1\,(inouts, nn) \\
\vdash \\
\forall\, nn \bullet \left(\begin{array}{l}
\#\,(inouts(nn)) = m \land \\
\#\,(inouts'(nn)) = n \land \\
(inouts'(nn) = f_2\,(inouts'(nn), nn)) \land \\
(\forall\, sigs : \mathbb{N} \rightarrow \mathrm{seq}\ \mathbb{R}, nn : \mathbb{N} \bullet \#\,(sigs\ \ nn) = m \Rightarrow \#\,(f_2(sigs, nn)) = n)
\end{array}\right)
\end{array}\right)
\end{array}
$$

FBlock has four parameters: f_1 is a predicate that specifies the assumption of the block and it is a function on input signals; m and n are the number of inputs and outputs, and f_2 is a function that relates inputs to outputs and is used to establish the postcondition of the block. The precondition of *FBlock* states that f_1 holds for inputs at any step nn. And the postcondition specifies that for any step nn the block always has m inputs and n outputs, the relation between outputs and inputs are given by f_2, and additionally f_2 always produces n outputs provided there are m inputs.

4.3.2 Simulink Blocks

Definition 4.4 (Unit Delay)

$$UnitDelay\,(x_0) \triangleq FBlock\,(true_f, 1, 1, (\lambda\, x, n \bullet \langle x_0 \lhd n = 0 \rhd hd\,(x\ \ (n-1))\rangle))$$

where hd is an operator to get the head of a sequence, and $true_f = (\lambda\, x, n \bullet true)$ *that means no constraints on input signals.*

The definition 4.4 of the Unit Delay block is straightforward: it accepts all inputs, has one input and one output, and produces initial value x_0 in its first step (0) and the previous input otherwise.

Definition 4.5 (Product (Divide))

$$Div2 \triangleq FBlock\,((\lambda\, x, n \bullet hd(tl(x\ \ n)) \neq 0)\,, 2, 1, (\lambda\, x, n \bullet \langle hd(x\ \ n)/hd(tl(x\ \ n))\rangle))$$

where tl is an operator to get the tail of a sequence.

The definition 4.5 of Divide block is slightly different because it assumes the input value of its second input signal is not zero at any step. By this way, the precondition enables modelling of non-input-receptive systems that may reject some inputs at some points.

4.3.3 Virtual Blocks

In addition to Simulink blocks, we have introduced three blocks for the purpose of composition: *Id*, *Split*2, and *Router*. The usage of these blocks is illustrated in Figure 1.

Definition 4.6 (Id)

$$Id \triangleq FBlock\,(true_f, 1, 1, (\lambda\, x, n \bullet \langle hd\,(x\ \ n)\rangle))$$

The identity block *Id* is a block that has one input and one output, and the output value is always equal to the input value. It establishes a fact that a direct signal line in Simulink could be treated as sequential composition of many *Id* blocks. The usage of *Id* is shown in Figure 1a.

Definition 4.7 (Split2)

$$Split2 \triangleq FBlock\left(true_f, 1, 2, \left(\lambda\, x, n \bullet \langle hd\left(x\ \ n\right), hd\left(x\ \ n\right)\rangle\right)\right)$$

*Split*2 corresponds to the signal connection splitter that produces two signals from one and both signals are equal to the input signal. The usage of *Split*2 is shown in Figure 1b.

Definition 4.8 (Router)

$$Router\left(m, table\right) \triangleq FBlock\left(true_f, m, m, \left(\lambda\, x, n \bullet reorder\left(\left(x\ \ n\right), table\right)\right)\right)$$

Router corresponds to the crossing connection of signals and this virtual block changes the order of input and output signals according to the supplied table. The usage of *Router* is shown in Figure 1c.

4.4 Subsystems

The treatment of subsystems (no matter whether hierarchical subsystems or atomic subsystems) in our designs is similar to that of blocks. They could be regarded as a bigger black box that relates inputs to outputs.

5 Block Compositions

In this section, we define three composition operators that are used to compose subsystems or systems from blocks. We also use three virtual blocks to map Simulink's connections in our designs.

For all definitions and laws in this section, if there are no special notes, we assume the following predicates.

$$\begin{array}{l}
\textbf{SimBlock}\left(m_1, n_1, P_1\right)\\
\textbf{SimBlock}\left(m_2, n_2, P_2\right)\\
\textbf{SimBlock}\left(m_3, n_3, P_3\right)\\
\textbf{SimBlock}\left(m_1, n_1, Q_1\right)\\
\textbf{SimBlock}\left(m_2, n_2, Q_2\right)\\
P_1 \sqsubseteq Q_1\\
P_2 \sqsubseteq Q_2
\end{array}$$

5.1 Sequential Composition

The meaning of sequential composition of designs is defined in Theorem 2.1. It corresponds to composition of two blocks in Figure 1d where the outputs of B_1 are equal to the inputs of B_2.

Provided

$$\begin{array}{ll}
P = \left(FBlock\left(true_f, m_1, n_1, f_1\right)\right) & \textbf{SimBlock}\left(m_1, n_1, P\right)\\
Q = \left(FBlock\left(true_f, n_1, n_2, f_2\right)\right) & \textbf{SimBlock}\left(n_1, n_2, Q\right)
\end{array}$$

The expansion law of sequential composition is given below.

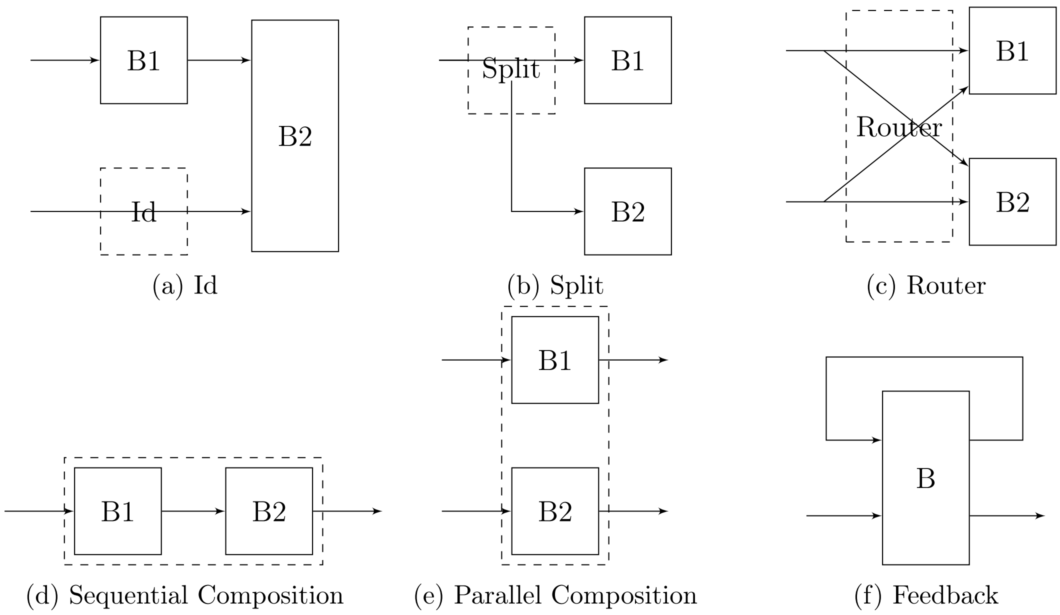

Figure 1: Composition of Blocks

Theorem 5.1 (Expansion)

$$(P;\ Q) = FBlock\,(true_f, m_1, n_2, (f_2 \circ f_1)) \qquad \text{[Expansion]}$$

This theorem establishs that sequential composition of two blocks, where the number of outputs of the first block is equal to the number of inputs of the second block, is simply a new block with the same number of inputs as the first block P and the same number of outputs as the second block Q, and additionally the postcondition of this composed block is function composition. In addition, the composed block is still ***SimBlock*** healthy which is shown in the closure theorem below.

Theorem 5.2 (Closure)

$$\textbf{\textit{SimBlock}}\,(m_1, n_2, (P;\ Q)) \qquad [\textbf{\textit{SimBlock}}\text{ Closure}]$$

5.2 Parallel Composition

Parallel composition of two blocks is a stack of inputs and outputs from both blocks and is illustrated in Figure 1e. It is defined below.

Definition 5.1 (Parallel Composition)

$$P \parallel_B Q \triangleq \left(\begin{array}{l} (takem(inps(P) + inps(Q))\ inps(P);\ P) \\ \parallel_{B_M} \\ (dropm(inps(P) + inps(Q))\ inps(P);\ Q) \end{array} \right)$$

where *takem* and *dropm* are two blocks to split inputs into two parts and their definitions can be found in Appendix A, and B_M is defined below.

Definition 5.2 (B_M)

$$B_M \triangleq \left(ok' = 0.ok \land 1.ok\right) \land \left(inouts' = 0.inouts \frown 1.inouts\right)$$

The definition of parallel composition 5.1 for designs is similar to the parallel-by-merge scheme [15, Sect. 7.2] in UTP. Parallel-by-merge is denoted as $P \parallel_M Q$ where M is a special relation that explains how the output of parallel composition of P and Q should be merged following execution.

However, parallel-by-merge assumes that the initial observations for both predicates should be the same. But that is not the case for our block composition because the inputs to the first block and that to the second block are different. Therefore, in order to use the parallel by merge, firstly we need to partition the inputs to the composition into two parts: one to the first block and another to the second block. This is illustrated in Figure 2 where we assume that P has m inputs and i outputs, and Q has n inputs and j outputs. Finally, it has the same inputs $(m+n)$ and the outputs of P and Q are merged by B_M to get $i+j$ outputs.

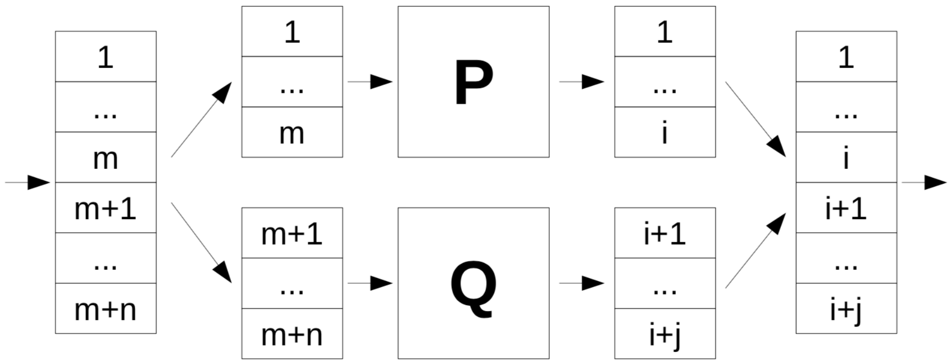

Figure 2: Parallel Composition of Two Blocks

The merge operator B_M states that the parallel composition terminates if both blocks terminate. And on termination, the output of parallel composition is concatenation of the outputs from the first block and the outputs from the second block. *takem* and *dropm* are two blocks that have the same inputs and the number of inputs is equal to addition of the number inputs of P and the number inputs of Q. *takem* only takes the first part of inputs as required by P, and *dropm* takes the second part of inputs as required by Q.

Theorem 5.3 (Associativity, Monotonicity, and *SimBlock* Closure)

$$P_1 \parallel_B (P_2 \parallel_B P_3) = (P_1 \parallel_B P_2) \parallel_B P_3 \qquad \text{[Associativity]}$$
$$(P_1 \parallel_B Q_1) \sqsubseteq (P_2 \parallel_B Q_2) \qquad \text{[Monotonicity]}$$
$$\textbf{\textit{SimBlock}}\,(m1 + m2, n1 + n2, (P_1 \parallel_B P_2)) \qquad [\textbf{\textit{SimBlock}} \text{ Closure}]$$
$$inps\,(P_1 \parallel_B P_2) = m_1 + m_2$$
$$outps\,(P_1 \parallel_B P_2) = n_1 + n_2$$

Parallel composition is associative, monotonic in terms of the refinement relation, and ***SimBlock*** healthy. The inputs and outputs of parallel composition are combination of the inputs and outputs of both blocks.

Theorem 5.4 (Parallel Operator Expansion) *Provided*

$$\begin{array}{ll} P = (FBlock\,(true_f, m_1, n_1, f_1)) & \textbf{\textsf{SimBlock}}\,(m_1, n_1, P) \\ Q = (FBlock\,(true_f, m_2, n_2, f_2)) & \textbf{\textsf{SimBlock}}\,(m_2, n_2, Q) \end{array}$$

then,

$$(P \parallel_B Q) = FBlock \left(\begin{array}{l} true_f, m_1 + m_2, n_1 + n_2, \\ \left(\lambda\, x, n \bullet \left(\begin{array}{l} (f_1 \circ (\lambda\, x, n \bullet take\,(m_1, x\ \ n))) \\ {}^\frown (f_2 \circ (\lambda\, x, n \bullet drop\,(m_1, x\ \ n))) \end{array} \right) \right) \end{array} \right) \qquad \text{[Expansion]}$$

$$\textbf{\textsf{SimBlock}}\,(m_1 + m_2, n_1 + n_2, (P \parallel_B Q)) \qquad [\textbf{\textsf{SimBlock}}\ \text{Closure}]$$

Parallel composition of two *FBlock* defined blocks is expanded to get a new block. Its postcondition is concatenation of the outputs from P and the outputs from Q. The outputs from P (or Q) are function composition of its block definition function f_1 (or f_2) with *take* (or *drop*).

5.3 Feedback

The feedback operator loops an output back to an input, which is illustrated in Figure 1f.

Definition 5.3 (f_D)

$$P\ f_D\ (i, o) \triangleq (\exists\, sig \bullet (PreFD(sig, inps(P), i);\ P;\ PostFD(sig, outps(P), o)))$$

where i and o denotes the index number of the output signal and the input signal, which are looped. *PreFD* denotes a block that adds *sig* into the ith place of the inputs.

Definition 5.4 (*PreFD*)

$$PreFD(sig, m, idx) \triangleq FBlock\,(true_f, m - 1, m, (f\_PreFD(sig, idx)))$$

where $f\_PreFD(sig, idx) = \lambda\, x, n \bullet (take(idx, (x\ \ n)) {}^\frown \langle (sig\ \ n) \rangle {}^\frown drop(idx, (x\ \ n)))$

and *PostFD* denotes a block that removes the oth signal from the outputs of P and this signal shall be equal to *sig*.

Definition 5.5 (*PostFD*)

$$PostFD(sig, n, idx) \triangleq \left(\begin{array}{l} \textbf{\textsf{true}} \\ \vdash \\ \forall\, nn \bullet \left(\begin{array}{l} \#\,(inouts(nn)) = n \land \\ \#\,(inouts'(nn)) = n - 1 \land \\ (inouts'(nn) = (f\_PostFD(sig, idx, inouts'(nn), nn)) \land \\ sig(nn) = inouts(nn)!idx \end{array} \right) \end{array} \right)$$

where $f\_PostFD(idx) = \lambda\, x, n \bullet (take(idx, (x\ \ n)) {}^\frown drop(idx + 1, (x\ \ n)))$ *and* ! *is an operator to get the element in a list by its index.*

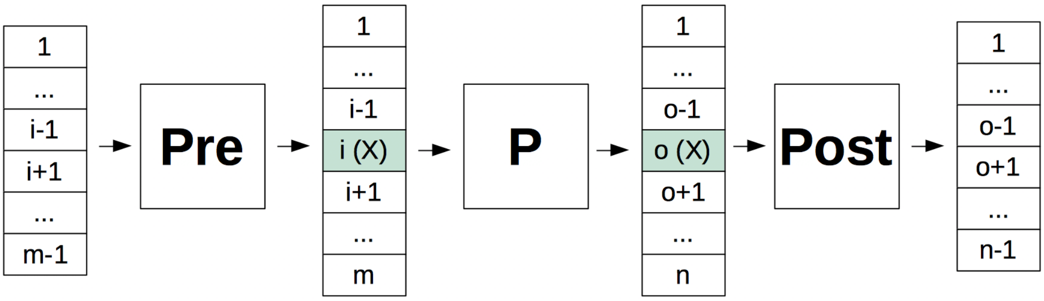

Figure 3: Feedback

The basic idea to construct a feedback operator is to use existential quantification to specify that there exists one signal *sig* that it is the ith input and oth output, and their relation is established by the block P. This is illustrated in Figure 3 where m and n are the number of inputs and outputs of P. *PreFD* adds a signal into the inputs at i and P takes assembled inputs and produces an output in which the oth output is equal to the supplied signal. Finally, the outputs of feedback are the outputs of P without the oth output. Therefore, a block with feedback is translated to a sequential composition of *PreFD*, P, and *PostFD*.

Theorem 5.5 (Monotonicity) *Provided*

$$\begin{array}{ll} \textbf{\textsf{SimBlock}}\,(m_1, n_1, P_1) & \textbf{\textsf{SimBlock}}\,(m_1, n_1, P_2) \\ P_1 \sqsubseteq P_2 & i_1 < m_1 \wedge o_1 < n_1 \end{array}$$

then,

$$(P_1 \; f_D \; (i_1, o_1)) \sqsubseteq (P_2 \; f_D \; (i_1, o_1))$$

The monotonicity law states that if a block is a refinement of another block, then its feedback is also a refinement of the same feedback of another block.

Theorem 5.6 (Expansion) *Provided*

$$\begin{array}{ll} P = \mathit{FBlock}\,(\mathit{true}_f, m, n, f) & \textbf{\textsf{SimBlock}}\,(m, n, P) \\ \mathit{Solvable\_unique}(i, o, m, n, f) & \mathit{is\_Solution}(i, o, m, n, f, \mathit{sig}) \end{array}$$

then,

$$\begin{array}{lr} (P \; f_D \; (i, o)) & \\ = \mathit{FBlock}\,(\mathit{true}_f, m-1, n-1, (\lambda\, x, n \bullet (f\_\mathit{PostFD}(o) \circ f \circ f\_\mathit{PostFD}(\mathit{sig}, x, i)) \;\; x \;\; n)) & \\ & [\text{Expansion}] \\ \textbf{\textsf{SimBlock}}\,(m-1, n-1, (P \; f_D \; (i, o))) & [\textbf{\textsf{SimBlock}}\ \text{Closure}] \end{array}$$

In the expansion theorem, where

Definition 5.6 (*Solvable_unique*)

$$Solvable\_unique\,(i, o, m, n, f) \triangleq \left(\begin{array}{l} (i < m \land o < n) \land \\ \left(\forall\, sigs \bullet \left(\begin{array}{l} (\forall\, nn \bullet \#\,(sigs\ \ nn) = (m-1)) \Rightarrow \\ (\exists_1\, sig \bullet (\forall\, nn \bullet (sig\ \ nn = (f\ (\lambda\, n1 \bullet f\_PreFD\,(sig, i, sigs, n1)\,, nn))!o))) \end{array} \right)\right) \end{array} \right)$$

The *Solvable_unique* predicate characterises a condition that the block with feedback has a unique solution that satisfies the constraint of feedback: the corresponding output and input are equal.

Definition 5.7 (*is_Solution*)

$$is\_Solution\,(i, o, m, n, f, sig) \triangleq \left(\left(\forall\, sigs \bullet \left(\begin{array}{l} (\forall\, nn \bullet \#\,(sigs\ \ nn) = (m-1)) \Rightarrow \\ (\forall\, nn \bullet (sig\ \ nn = (f\ (\lambda\, n1 \bullet f\_PreFD\,(sig, i, sigs, n1)\,, nn))!o)) \end{array} \right)\right) \right)$$

The *is_Solution* predicate evaluates a supplied signal to check if it is a solution for the feedback.

The expansion law of feedback assumes the function f, that is used to define the block P, is solvable in terms of i, o, m and n. In addition, it must have one unique solution sig that resolves the feedback.

Our approach to model feedback in designs enables reasoning about systems with algebraic loops. If a block defined by *FBlock* and $Solvable\_unique\,(i, o, m, n, f)$ is true, then the feedback composition of this block in terms of i and o is feasible no matter whether there are algebraic loops or not.

5.4 Composition Examples

For the compositions in Figure 1, their corresponding maps in our design theory are shown below.

- Figure 1a: $(B_1 \parallel_B Id)\,;\ B_2$
- Figure 1b: $Split2;\ (B_1 \parallel_B B_2)$
- Figure 1c: $(Split2 \parallel_B Split2)\,;\ Router(4, [0, 2, 1, 3]);\ (B_1 \parallel_B B_2)$
- Figure 1d: $B_1;\ B_2$
- Figure 1e: $B_1 \parallel_B B_2$
- Figure 1f: $B\ f_D\ (0, 0)$

6 Case Study

This case study, verification of a post_landing_finalize subsystem, is taken from an aircraft cabin pressure control application. The original Simulink model is from Honeywell through our industrial link with D-RisQ. This case is also studied in [28] and the diagram shown in Figure 4 is from the paper. The purpose of this subsystem is to implement that the output *finalize_event* is triggered after the aircraft door has been open for a minimum specific amount of time following a successful landing.

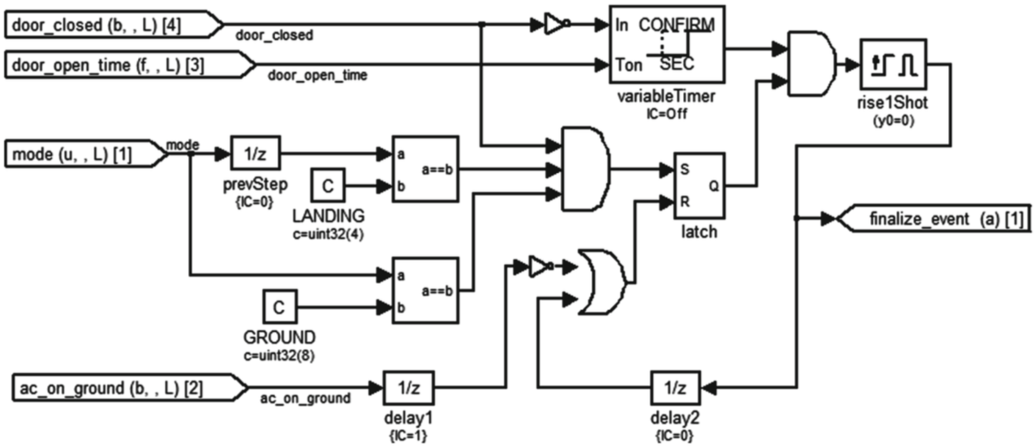

Figure 4: Post Landing Finalize (source: [28])

In order to apply our AG reasoning into this Simulink model, firstly we model the subsystem in our block theories as shown in Section 6.1. Then we verify a number of properties for three small subsystems in this model, which is given in Section 6.2. Finally, in Section 6.3 we present verification of four requirements of this subsystem. To avoid confusion between the subsystem and three small subsystems, in the following sections we use the *system* to denote the post_landing_finalize subsystem to be verified, and the *subsystems* to denote three small subsystems.

6.1 Modelling

We start with translation of three small subsystems (variableTimer, rise1Shot and latch) according to our block theories.

The subsystem latch is modelled as below. It is shown in Appendix **??** as well.

$$(((((UnitDelay\ 0) \parallel_B Id)\,;\ (LopOR\ 2)) \parallel_B (Id;\ LopNOT))\,;\ (LopAND\ 2)\,;\ Split2)\ f_D\ (0,0)$$

The blocks $LopOR$, $LopNOT$ and $LopAND$ correspond to the OR, NOT and AND operators in the logic operator block. Their definitions can be found in Appendix A. Then we apply composition definitions, expansion and SimBlock closure laws to simplify the subsystem. The latch subsystem is finally simplified to a design.

$$latch = FBlock\,(true_f, 2, 1, latch\_simp\_pat\_f)$$

where the definition of $latch\_simp\_pat\_f$ is given in Appendix **??**.

Similarly, variableTimer and rise1Shot are modelled and simplified as shown in Appendix **??** and **??** respectively.

Finally, we can use the similar way to compose the three subsystems with other blocks in this diagram to get the corresponding composition of post_landing_finalise_1, and then apply the similar laws to simplify it further into one block and verify requirements for this system. However, for the outermost feedback it is difficult to use the similar way to simplify it into one block because it is more complicate than feedbacks in other three small subsystems (variableTimer, rise1Shot and latch). In order to use the expansion theorem 5.6 of feedback, we need to find a solution for the block and prove the solution is unique. With increasing complexity of blocks,

Requirement 1	A finalize event will be broadcast after the aircraft door has been open continuously for *door_open_time* seconds while the aircraft is on the ground after a successful landing.
Requirement 2	A finalize event is broadcast only once while the aircraft is on the ground.
Requirement 3	The finalize event will not occur during flight.
Requirement 4	The finalize event will not be enabled while the aircraft door is closed.

Table 1: Requirements for the system (source: [28])

this expansion is becoming harder and harder. Therefore, post_landing_finalise_1 has not been simplified into one block. Instead, it is simplified to a block with a feedback which can be seen in the lemma *post_landing_finalize_1_simp* in Appendix ??.

$$post\_landing\_finalize\_1 = plf\_rise1shot\_simp\ f_D\ (4, 1)$$

6.2 Subsystems Verification

After simplification, we can verify properties of the subsystems using the refinement relation.

We start with verification of a property for variableTimer: *vt_req_00*. This property states that if the door is closed, then the output of this subsystem is always false. The verification of this property is given in Appendix ??. However, this property can not be verified in absence of an assumption made to the second input: *door_open_time*. This is due to a type conversion block `int32` used in the subsystem. If the input to `int32` is larger than 2147483647 (that is, *door_open_time* larger than 2147483647/10), its output is less than zero and finally the output is true. That is not the expected result. Practically, *door_open_time* should be less than 2147483647/10. Therefore, we can make an assumption of the input and eventually verify this property as given in the lemma *vt_req_00*. Additionally, we suggest a substitution of `int32` by `uint32`, or a change of the data type for the input from double to unsigned integer, such as `uint32`.

As for the rise1Shot subsystem, we verified one property: *rise1shot_req_00*. This property specifies that the output is true only when current input is true and previous input is false (see Appendix ??). It means it is triggered only by a rising edge and continuous true inputs will not enable the output.

Furthermore, one property for the latch subsystem (a SR AND-OR latch) is verified (see Appendix ??). The property *latch_req_00* states that as long as the second input R is true, its output is always false. This is consistent with the definition of the SR latch in circuits.

6.3 Requirement Verification

The four requirements to be verified are illustrated in Table 1.

Our approach to cope with the difficulty to simplify this system into one design is to apply compositional reasoning. Generally, application of compositional reasoning to verify requirements is as follows.

- In order to verify the property satisfied by post_landing_finalise_1:

$$C \sqsubseteq post\_landing\_finalise\_1$$

, that is, to verify

$$C \sqsubseteq (plf\_rise1shot\_simp\ f_D\ (4,1))$$

;

- We need to find a decomposed contract C' such that

  $$C \sqsubseteq (C'\ f_D\ (4,1))$$

  and

  $$(C' \sqsubseteq plf\_rise1shot\_simp)$$

  ;

- Then we get

  $$(C'\ f_D\ (4,1)) \sqsubseteq (plf\_rise1shot\_simp\ f_D\ (4,1))$$

  using the monotonicity theorem 5.5 of feedback;

- Finally, according to transitivity of the refinement relation, it establishes that

  $$C \sqsubseteq (plf\_rise1shot\_simp\ f_D\ (4,1))$$

  .

6.3.1 Requirement 3 and 4

Requirement 3 and 4 are verified together as shown in Appendix **??**. *req_04_contract* and *req_04_1_contract* are C and C' described above respectively.

6.3.2 Requirement 1

According to Assumption 3 "*door_open_time* does not change while the aircraft is on the ground" and the fact that this requirement specifies the aircraft is on the ground, therefore *door_open_time* is constant for this scenario. In order to simplify the verification, we assume it is always constant. The contract *req_01_contract* specifies that

- it always has four inputs and one output;
- and the requirement:
  - after a successful landing at step m and $m+1$: the door is closed, the aircraft is on ground, and the mode is switched from LANDING (at step m) to GROUND (at step $m+1$),
  - then the door has been open continuously for *door_open_time* seconds from step $m+2+p$ to $m+2+p+door\_open\_time$, therefore the door is closed at the previous step $m+2+p-1$,
  - while the aircraft is on ground: *ac_on_ground* is true and *mode* is GROUND,
  - additionally, between step m and $m+2+p$, the *finalize_event* is not enabled,
  - then a *finalize_event* will be broadcast at step $m+2+p+door\_open\_time$.

As shown in Appendix **??**, this requirement has been verified.

6.3.3 Requirement 2

The contract *req_02_contract* specifies that

- it always has four inputs and one output;
- and the requirement:
  - if a finalize event has been broadcast at step m,
  - while the aircraft is on ground: *ac_on_ground* is true and *mode* is GROUND,
  - then a finalize event will not be broadcast again.

As shown in Appendix **??**, this requirement has been verified too.

6.4 Summary

In sum, we have translated and mechanised the post_landing_finalize diagram in Isabelle/UTP, simplified its three subsystems (variableTimer, rise1Shot and latch) and the post_landing_finalize into a design with feedback, and finally verified all four requirements of this system. In addition, our work has identified a vulnerable block in variableTimer. This case study demonstrates that our verification framework has rich expressiveness to specify scenarios for requirement verification (as illustrated in the verification of Requirement 1 and 2) and our verification approach is useful in practice.

7 Conclusions

In this report, we present our work for the VeTSS funded project "Mechanised Assume-Guarantee Reasoning for Control Law Diagrams via Circus" from developed theories and laws as well as their mechanisation in Isabelle/UTP. In addition, we present practical application of our theories to reason about a Simulink model in the aircraft cabin pressure control application. Our mechanisation is also attached to this report.

7.1 Progress Summary

The project wss initially proposed to have four work packages. And a summary of progress is shown in Table 2.

WP1 – framework: we reviewed current solutions that use contract-based reasoning and Circus-based program verification for Simulink. Eventually we put forward a new contract-based assume-guarantee reasoning methodology for Simulink diagrams. The theoretical part of this approach is based on the theory of design in UTP that is presented in this report.

WP2 – definition and mechanisation: one advantage of using designs for reasoning is its existing theory and mechanisation in Isabelle/UTP. However, in order to accommodate Simulink diagrams into designs easily, we have defined three additional virtual blocks (Identity, Split and Router) and two extra operators (Parallel Composition and Feedback). They correspond to signal connections and block composition in Simulink. With these new blocks and operators (as well as existing operators for designs), we could translate Simulink diagrams into composition of designs. In addition, we have mechanised (in Isabelle/UTP) the three virtual blocks and 14 Simulink blocks (Constant, Unit Delay, Discrete-Time Integrator, Sum, Product, Gain, Saturation, MinMax, Rounding, Logic Operator, Relational Operator, Switch, Data Type Conversion and Initial Condition) that will be used in our case studies.

Work Package	Description	Progress
WP1	Review current Simulink reasoning solutions and put forward a new contract-based methodology (using UTP design theory) to reason about faulty behaviour through assumptions	100%
WP2	Define assumption-guarantee contracts for the Simulink semantics and mechanise them in Isabelle/UTP, including operators and a limited selection of Simulation discrete blocks that are used in our case studies, and mechanise in Isabelle/UTP	100%
WP3	Mechanise industrial case studies (building case and post landing finalize case) in Isabelle/UTP using mechanised block libraries (produced in WP2), including modelling, contract calculation, and proof	50%
WP4	Investigate the weakest assumption calculus based on the examples, in order to automate reasoning about interferences between blocks and subsystems	25%

Table 2: Project Progress Summary

WP3 – case studies: using definitions and mechanisation of these blocks and operators, we have mechanised one of our case study (the post landing finalize) in Isabelle/UTP.

WP4 - Though time did not permit us to consider the weakest assumption calculus for Simulink in details, in a parallel project we have explored a calculus for weakest reactive rely conditions for reactive contracts based in UTP. The details of this can be found in a draft journal paper under review for Theoretical Computer Science [26]. This initial study provides necessary background for future work with Simulink.

Due to the fact that we started this project two months late since October 2017 because of delays in receiving funding, therefore we have limited time to finish all proposed work. We have not verified all requirements of the post landing finalize case, have not started the second building case study, and have investigaged WP4 partially.

Acknowledgements. This project is funded by the National Cyber Security Centre (NCSC) through UK Research Institute in Verified Trustworthy Software Systems (VeTSS) [29]. We thank Honeywell and D-RisQ for sharing of the industrial case.

A Block Theories

In this section, we define main theories of block diagrams in UTP.

theory *simu-contract-real*
  imports
    ~~/src/HOL/Word/Word
    utp-designs
begin

syntax
  -svid-des :: *svid* ($\mathbf{v}_D$)

translations
  -svid-des => Σ_D

Defined Simulink blocks using designs directly.

named-theorems *sim-blocks*

Functions used to define Simulink blocks via patterns.

named-theorems *f-blocks*

Defined Simulink blocks using functions and patterns.

named-theorems *f-sim-blocks*

SimBlock healthiness.

named-theorems *simblock-healthy*

recall-syntax

A.1 Additional Laws

theorem *ndesign-composition*:
  $((p1 \vdash_n Q1) \;;\; (p2 \vdash_n Q2)) = ((p1 \wedge \neg \lfloor Q1 \;;\; (\neg \lceil p2 \rceil_<) \rfloor_<) \vdash_n (Q1 \;;\; Q2))$
  apply (*ndes-simp*, *simp add*: *wp-upred-def*)
  by (*rel-simp*)

lemma *list-equal-size2*:
  fixes *x*
  assumes $length(x) = 2$
  shows $x = [hd(x)] \bullet [last(x)]$
proof −
  have *1*: $x = [hd(x)] \bullet tl(x)$
    by (*metis append-Cons append-Nil assms hd-Cons-tl length-0-conv zero-not-eq-two*)
  have *2*: $tl(x) = [last(x)]$
    using *assms*
    by (*metis One-nat-def 1 append-butlast-last-id append-eq-append-conv append-is-Nil-conv cancel-ab-semigroup-add-class.add-diff-cancel-left′ length-Cons length-tl list.size*(*3*) *nat-1-add-1 not-Cons-self2*)
  from *1* **and** *2* **show** *?thesis*
    by *auto*
qed

theorem *ndesign-refinement*:
  $(P1 \vdash_n Q1 \sqsubseteq P2 \vdash_n Q2) \longleftrightarrow (`P1 \Rightarrow P2` \wedge `\lceil P1 \rceil_< \wedge Q2 \Rightarrow Q1`)$
  by (*rel-auto*)

theorem *ndesign-refinement′*:
  $(P1 \vdash_n Q1 \sqsubseteq P2 \vdash_n Q2) \longleftrightarrow (P2 \sqsubseteq P1 \wedge Q1 \sqsubseteq (\lceil P1 \rceil_< \wedge Q2))$
  by (*meson ndesign-refinement refBy-order*)

lemma *assume-Ran*: $P \;;\; [Ran(P)]^\top = P$
  apply (*rel-auto*)
done

fun *sum-list1* **where**
sum-list1 [] = *0* |
sum-list1 (*x*#*xs*) = (*sum-list1 xs* + *x*)

A.2 State Space

inouts: input and output signals, abstracted as a function from step numbers to a list of inputs or outputs where we use universal real number as the data type of signals.

alphabet *sim-state* =
  inouts :: *nat* ⇒ *real list*

A.3 Patterns

FBlock is a pattern to define a block with precondition, number of inputs, number of outputs, and postcondition.

definition *FBlock* ::
  ((*nat* ⇒ *real list*) ⇒ *nat* ⇒ *bool*) ⇒
    nat ⇒ *nat* ⇒
   ((*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*)) ⇒
   sim-state hrel-des **where**
[*sim-blocks*]: *FBlock pre m nn f* =
  ((∀ *n*::*nat* · («*pre*» (&*inouts*)$_a$ («*n*»)$_a$)::*sim-state upred*) $\vdash_n$
   ((∀ *n*::*nat* ·
     ((#$_u$($*inouts* («*n*»)$_a$)) $=_u$ «*m*») ∧
     ((#$_u$($*inouts´* («*n*»)$_a$)) $=_u$ «*nn*») ∧
     («*f*» ($*inouts*)$_a$ («*n*»)$_a$ $=_u$ ($*inouts´* («*n*»)$_a$))) ∧
     (∀ *x* · (∀ *n*::*nat* · ((#$_u$(«*x*» («*n*»)$_a$) $=_u$ «*m*») ⇒ (#$_u$(«*f*» («*x*»)$_a$ («*n*»)$_a$) $=_u$ «*nn*»))))
     (∗ *for any inputs*, *f always produces the same size output*. *Useful to prove FBlock-seq-comp* ∗)
   ))

lemma *pre-true* [*simp*]: (∀ *n*::*nat* ·(«λ*x n*. *True*» (&*inouts*)$_a$ («*n*»)$_a$)::*sim-state upred*) = *true*
  by (*rel-simp*)

A.4 Number of Inputs and Outputs

abbreviation *PrePost*(*P*) ≡ $pre_D(P)$ ∧ $post_D(P)$

SimBlock is a condition stating that a design is a Simulink block if it is feasible, and has m inputs and n outputs.

definition *SimBlock* :: *nat* ⇒ *nat* ⇒ *sim-state hrel-des* ⇒ *bool*
  where [*sim-blocks*]:
SimBlock m n P = ((*PrePost*(*P*) ≠ *false*) ∧ (∗ *This is stronger than just excluding abort and miracle*, *and also not the same as H4 feasibility* ∗)
  ((∀ *na* · #$_u$(&*inouts*(«*na*»)$_a$) $=_u$ «*m*») ⊑ *Dom*(*PrePost*(*P*))) ∧
  ((∀ *na* · #$_u$(&*inouts*(«*na*»)$_a$) $=_u$ «*n*») ⊑ *Ran*(*PrePost*(*P*)))(∗ ∧
  (*P is* **N**)∗))

axiomatization
  inps :: *sim-state hrel-des* ⇒ *nat* **and**
  outps :: *sim-state hrel-des* ⇒ *nat*
where
  inps-outps: (*SimBlock m n P*) ⟶ (*inps P* = *m*) ∧ (*outps P* = *n*)

A.5 Operators

A.5.1 Id

definition *f-Id*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**

[*f-blocks*]: *f-Id x n* = [*hd*(*x n*)]

Id block: one input and one output, and the output is always equal to the input

definition *Id* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Id* = *FBlock* ($\lambda x\ n.$ *True*) *1 1* (*f-Id*)

A.5.2 Parallel Composition

definition *mergeB* ::
((*sim-state des*, *sim-state des*, *sim-state des*) *mrg*,
sim-state des) *urel* (B_M) **where**
[*sim-blocks*]: *mergeB* = (($ok´ $=_u$ ($0−ok ∧ $1−ok)) ∧ (
(∀ *n*::*nat* · (($$\mathbf{v}_D$:*inouts*´ («*n*»)$_a$) $=_u$ («*append*» ($0−$\mathbf{v}_D$:*inouts* («*n*»)$_a$)$_a$ ($1−$\mathbf{v}_D$:*inouts* («*n*»)$_a$)$_a$))
(∗∧ ($\#_u$($$\mathbf{v}_D$:$inouts_<$ («*n*»)$_a$) $=_u$ *2*)∗))))

takem: a block that just takes the first *nr2* inputs and ignores the remaining inputs.

definition *takem* :: *nat* ⇒ *nat* ⇒ *sim-state hrel-des* **where**
[*sim-blocks*]: *takem nr1 nr2* = ((«*nr2*» $\leq_u$ «*nr1*») $\vdash_n$
(∀ *n*::*nat* ·
(*uconj* (($\#_u$($*inouts* («*n*»)$_a$)) $=_u$ «*nr1*»)
(*uconj* (($\#_u$($*inouts*´ («*n*»)$_a$)) $=_u$ «*nr2*»)
(*true* ◁ («*nr2*» $=_u$ *0*) ▷ («*take*» («*nr2*»)$_a$ ($*inouts* («*n*»)$_a$)$_a$ $=_u$ ($*inouts*´ («*n*»)$_a$)))
))))

dropm: a block that just drops the first *nr2* inputs and outputs the remaining inputs.

definition *dropm* :: *nat* ⇒ *nat* ⇒ *sim-state hrel-des* **where**
[*sim-blocks*]: *dropm nr1 nr2* = ((«*nr2*» $\leq_u$ «*nr1*») $\vdash_n$
(∀ *n*::*nat* ·
(*uconj* (($\#_u$($*inouts* («*n*»)$_a$)) $=_u$ «*nr1*»)
(*uconj* (($\#_u$($*inouts*´ («*n*»)$_a$)) $=_u$ «*nr2*»)
(*true* ◁ («*nr2*» $=_u$ *0*) ▷ («*drop*» («*nr1*−*nr2*»)$_a$ ($*inouts* («*n*»)$_a$)$_a$ $=_u$ ($*inouts*´ («*n*»)$_a$)))
))))

We use the similar parallel-by-merge in UTP to implement parallel composition.

definition *sim-parallel* ::
sim-state hrel-des ⇒
sim-state hrel-des ⇒
sim-state hrel-des (**infixl** $\|_B$ *60*)
where [*sim-blocks*]: *P* $\|_B$ *Q* =
(((*takem* (*inps P* + *inps Q*) (*inps P*)) ; ; *P*)
$\|_{mergeB}$
((*dropm* (*inps P* + *inps Q*) (*inps Q*)) ; ; *Q*))

A.5.3 Sequential Composition

It is the same as the sequential composition for designs.

A.5.4 Feedback

definition *f-PreFD* :: (*nat* ⇒ *real*) (∗ *input signal*: *introduced by exists* ∗)
⇒ *nat* (∗ *the input index number that is fed back from output*. ∗)
⇒ (*nat* ⇒ *real list*) ⇒ *nat*
⇒ *real list* **where**
[*f-blocks*]: *f-PreFD x idx-fd inouts0 n* =

(*take idx-fd* (*inouts0 n*)) • (*x n*) # (*drop idx-fd* (*inouts0 n*))

definition *f-PostFD* ::
  nat (∗ *the input index number that is fed back from output.* ∗)
  ⇒ (*nat* ⇒ *real list*) ⇒ *nat*
  ⇒ *real list* **where**
[*f-blocks*]: *f-PostFD idx-fd inouts0 n* =
  (*take idx-fd* (*inouts0 n*)) • (*drop* (*idx-fd+1*) (*inouts0 n*))

definition *PreFD* ::
  (*nat* ⇒ *real*) (∗ *input signal*: *introduced by exists* ∗)
  ⇒ *nat* (∗ *m* ∗)
  ⇒ *nat* (∗ *the input index number that is fed back from output.* ∗)
  ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *PreFD x nr-of-inputs idx-fd* = (*true* $\vdash_n$
  (∀ *n*::*nat* · (
    (($\#_u$(\$*inouts* («*n*»)$_a$)) $=_u$ «*nr-of-inputs−1*») ∧
    (($\#_u$(\$*inouts´* («*n*»)$_a$)) $=_u$ «*nr-of-inputs*») ∧
    (\$*inouts´* («*n*»)$_a$ $=_u$ («*f-PreFD x idx-fd*» (\$*inouts*)$_a$ («*n*»)$_a$))
  )))

definition *PostFD* :: (*nat* ⇒ *real*) (∗ *input signal*: *introduced by exists* ∗)
  ⇒ *nat* (∗ *m* ∗)
  ⇒ *nat* (∗ *the input index number that is fed back from output.* ∗)
  ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *PostFD x nr-of-inputs idx-fd* =
  (*true* $\vdash_n$
    (∀ *n*::*nat* · (
      (($\#_u$(\$*inouts* («*n*»)$_a$)) $=_u$ «*nr-of-inputs*») ∧
      (($\#_u$(\$*inouts´* («*n*»)$_a$)) $=_u$ «*nr-of-inputs−1*») ∧
      (\$*inouts´* («*n*»)$_a$ $=_u$ («*f-PostFD idx-fd*» (\$*inouts*)$_a$ («*n*»)$_a$)) ∧
      ((«*nth*» (\$*inouts* («*n*»)$_a$)$_a$ («*idx-fd*»)$_a$ $=_u$ «*x n*»))
  )))

The feedback operator *sim-feedback* is defined via existential quantification.

fun *sim-feedback* :: *sim-state hrel-des*
  ⇒ (*nat* ∗ *nat*)
  ⇒ *sim-state hrel-des* (**infixl** f_D *60*)
where
P f_D (*i1*,*o1*) = (∃ (*x*) · (*PreFD x* (*inps P*) *i1* ; ; *P* ; ; *PostFD x* (*outps P*) *o1*))

Solvable checks if the supplied function for feedback is solvable according to the feedback signal from the output $o1$ to the input $i1$. A function is solvable if its feedback is feasible. Feedback may lead to algebraic loops but this condition states that algebraic loops are solvable.

definition *Solvable*:: *nat* (∗ *the input index for feedback* ∗)
  ⇒ *nat* (∗ *the output index for feedback* ∗)
  ⇒ *nat* (∗ *how many input signals* ∗)
  ⇒ *nat* (∗ *how many output signals* ∗)
  ⇒ ((*nat* ⇒ *real list*) ⇒ *nat* ⇒ *real list*) (∗ *function* ∗)
  ⇒ *bool* **where**
Solvable i1 o1 m nn f = ((*i1* < *m* ∧ *o1* < *nn*) ∧
  (∀ $inouts_0$. (∀ *x*. *length*($inouts_0$ *x*) = (*m−1*)) (∗ *For any* (*m−1*) *inputs* ∗)

⟶
(∃ xx. (∗ there exists a signal xx that is the i1th input and the o1th output ∗)
(∀ n. (xx n = (∗ the o1th output ∗)
(f (λn1. f-PreFD xx i1 $inouts_0$ n1
(∗ ((take i1 ($inouts_0$ n1))•(xx n1)#(drop i1 ($inouts_0$ n1))) ∗)
(∗ assemble of inputs to make xx as i1th ∗)
) n)!o1
)
))))

Solvable-unique: the feedback is solvable and has a unique solution.

definition *Solvable-unique*:: nat (∗ the input index for feedback ∗)
⇒ nat (∗ the output index for feedback ∗)
⇒ nat (∗ how many input signals ∗)
⇒ nat (∗ how many output signals ∗)
⇒ ((nat ⇒ real list) ⇒ nat ⇒ real list) (∗ function ∗)
⇒ bool **where**
Solvable-unique i1 o1 m nn f = ((i1 < m ∧ o1 < nn) ∧
(∀ $inouts_0$. (∀ x. length($inouts_0$ x) = (m−1)) (∗ For any (m−1) inputs ∗)
⟶
(∃! (xx::nat ⇒ real). (∗ there only exists a signal xx that is the i1th input and the o1th output ∗)
(∀ n. (xx n = (∗ the o1th output ∗) (f (λn1. f-PreFD xx i1 $inouts_0$ n1) n)!o1)
)
)
)
)

Solution returns the solution for a feedback block. Here the solution means the signal that could satisfy the feedback constraint (the related input is equal to the output)

definition *Solution*:: nat (∗ the input index for feedback ∗)
⇒ nat (∗ the output index for feedback ∗)
⇒ nat (∗ how many input signals ∗)
⇒ nat (∗ how many output signals ∗)
⇒ ((nat ⇒ real list) ⇒ nat ⇒ real list) (∗ function ∗)
⇒ (nat ⇒ real list)
⇒ (nat ⇒ real) **where**
Solution i1 o1 m nn f $inouts_0$ =
(SOME (xx::nat ⇒ real).
((∗(∀ x. length($inouts_0$ x) = (m−1)) (∗ For any (m−1) inputs ∗)
⟶ ∗)
(∀ n. (xx n =
(f (λn1. f-PreFD xx i1 $inouts_0$ n1
(∗ ((take i1 ($inouts_0$ n1))•[xx n1]•(drop i1 ($inouts_0$ n1)))∗)
) n)!o1
)
)))

is-Solution checks if the supplied solution for a feedback block is a real solution.

definition *is-Solution*:: nat (∗ the input index for feedback ∗)
⇒ nat (∗ the output index for feedback ∗)
⇒ nat (∗ how many input signals ∗)
⇒ nat (∗ how many output signals ∗)
⇒ ((nat ⇒ real list) ⇒ nat ⇒ real list) (∗ function ∗)
⇒ ((nat ⇒ real list) ⇒ (nat ⇒ real))
⇒ bool **where**

$is\text{-}Solution\ i1\ o1\ m\ nn\ f\ xx = ($
  $(\forall\, inouts_0.\ (\forall\, x.\ length(inouts_0\ x) = (m{-}1))$
    $\longrightarrow (\forall\, n.\ (xx\ inouts_0\ n = (f\ (\lambda n1.\ f\text{-}PreFD\ (xx\ inouts_0)\ i1\ inouts_0\ n1)\ n)!o1))))$

A.5.5 Split

definition $f\text{-}Split2$:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[$f\text{-}blocks$]: $f\text{-}Split2\ x\ n = [hd(x\ n),\ hd(x\ n)]$

definition $Split2$ :: $sim\text{-}state\ hrel\text{-}des$ **where**
[$f\text{-}sim\text{-}blocks$]: $Split2 = FBlock\ (\lambda x\ n.\ True)\ 1\ 2\ (f\text{-}Split2)$

A.6 Blocks

A.6.1 Source

A.6.1.1 Constant

Constant Block: no inputs and only one output.

definition $f\text{-}Const$:: $real \Rightarrow (nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[$f\text{-}blocks$]: $f\text{-}Const\ x0\ x\ n = [x0]$

definition $Const$ :: $real \Rightarrow sim\text{-}state\ hrel\text{-}des$ **where**
[$f\text{-}sim\text{-}blocks$]: $Const\ c0 = FBlock\ (\lambda x\ n.\ True)\ 0\ 1\ (f\text{-}Const\ c0)$

A.6.2 Unit Delay

Unit Delay block: one parameter (initial output), one input and one output. And the output is equal to previous input if it is not the initial output; otherwise it is equal to the initial output.

definition $f\text{-}UnitDelay$:: $real \Rightarrow (nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[$f\text{-}blocks$]: $f\text{-}UnitDelay\ x0\ x\ n = [if\ n = 0\ then\ x0\ else\ hd(x\ (n{-}1))]$

definition $UnitDelay$ :: $real \Rightarrow sim\text{-}state\ hrel\text{-}des$ **where**
[$f\text{-}sim\text{-}blocks$]: $UnitDelay\ x0 = FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ (f\text{-}UnitDelay\ x0)$

A.6.3 Discrete-Time Integrator

The Discrete-Time Integrator block: performs discrete-time integration or accumulation of signal. Integration (T=Ts) or Accumulation (T=1) methods: forward Euler, backward Euler, and trapezoidal methods.

$DT\text{-}int\text{-}fw$: integration by Forward Euler

fun $sum\text{-}by\text{-}fw\text{-}euler$ :: $nat \Rightarrow real \Rightarrow real \Rightarrow real \Rightarrow (nat \Rightarrow real\ list) \Rightarrow real$ **where**
$sum\text{-}by\text{-}fw\text{-}euler\ 0\ x0\ K\ T\ x = x0\ |$
$sum\text{-}by\text{-}fw\text{-}euler\ (Suc\ m)\ x0\ K\ T\ x =$
  $(sum\text{-}by\text{-}fw\text{-}euler\ m\ x0\ K\ T\ x) + (K * T * (hd(x\ m)))$

definition $f\text{-}DT\text{-}int\text{-}fw$ :: $real \Rightarrow real \Rightarrow real \Rightarrow (nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[$f\text{-}blocks$]: $f\text{-}DT\text{-}int\text{-}fw\ x0\ K\ T\ x\ n = [sum\text{-}by\text{-}fw\text{-}euler\ n\ x0\ K\ T\ x]$

definition $DT\text{-}int\text{-}fw$ :: $real \Rightarrow real \Rightarrow real \Rightarrow$
  $sim\text{-}state\ hrel\text{-}des$ **where**
[$f\text{-}sim\text{-}blocks$]: $DT\text{-}int\text{-}fw\ x0\ K\ T = FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ (f\text{-}DT\text{-}int\text{-}fw\ x0\ K\ T)$

$DT\text{-}int\text{-}bw$: integration by Backward Euler (Initial condition setting is set to State)

fun $sum\text{-}by\text{-}bw\text{-}euler$ :: $nat \Rightarrow real \Rightarrow real \Rightarrow real \Rightarrow (nat \Rightarrow real\ list) \Rightarrow real$ **where**

$sum\text{-}by\text{-}bw\text{-}euler\ 0\ x0\ K\ T\ x = x0 + (K * T * (hd(x\ 0)))$ |
$sum\text{-}by\text{-}bw\text{-}euler\ (Suc\ m)\ x0\ K\ T\ x =$
$(sum\text{-}by\text{-}bw\text{-}euler\ m\ x0\ K\ T\ x) + (K * T * (hd(x\ m)))$

definition *f-DT-int-bw* :: *real* ⇒ *real* ⇒ *real* ⇒ (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-DT-int-bw x0 K T x n* = [*sum-by-bw-euler n x0 K T x*]

definition *DT-int-bw* :: *real* ⇒ *real* ⇒ *real* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *DT-int-bw x0 K T* = *FBlock* (λ*x n. True*) *1 1* (*f-DT-int-bw x0 K T*)

DT-int-trape: integration by Trapezoidal (Initial condition setting is set to State).

fun *sum-by-trape* **where**
$sum\text{-}by\text{-}trape\ 0\ x0\ K\ T\ x = x0 + (K * (T\ div\ 2) * (hd(x\ 0)))$ |
$sum\text{-}by\text{-}trape\ (Suc\ m)\ x0\ K\ T\ x =$
$(sum\text{-}by\text{-}trape\ m\ x0\ K\ T\ x) +$
$(K * (T\ div\ 2) * (hd(x\ m))) +$
$(K * (T\ div\ 2) * (hd(x\ (Suc\ m))))$

definition *f-DT-int-trape* :: *real* ⇒ *real* ⇒ *real* ⇒ (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-DT-int-trape x0 K T x n* = [*sum-by-trape n x0 K T x*]

definition *DT-int-trape* :: *real* ⇒ *real* ⇒ *real* ⇒
sim-state hrel-des **where**
[*f-sim-blocks*]: *DT-int-trape x0 K T* = *FBlock* (λ*x n. True*) *1 1* (*f-DT-int-trape x0 K T*)

A.6.4 Sum

The Sum block performs addition or subtraction on its inputs.

sum-by-sign: Summation or subtraction of a list according to their corresponding signs. It requires the length of inputs are equal to that of signs (true for +)

fun *sum-by-sign* **where**
sum-by-sign [] - = *0* |
sum-by-sign (*x#xs*) (*s#ss*) = (*if s then* (*sum-by-sign xs ss* + *x*) *else* (*sum-by-sign xs ss* − *x*))

definition *f-SumSub*:: *bool list* ⇒ (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-SumSub signs x n* = [*sum-by-sign* (*x n*) *signs*]

SumSub: summation or subtraction according to supplied signs.

definition *SumSub* :: *nat* ⇒ *bool list* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *SumSub nr signs* = *FBlock* (λ*x n. True*) *nr 1* (*f-SumSub signs*)

Sum2 is a special case of *SumSub* and it adds up two inputs

definition *f-Sum2*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-Sum2 x n* = [*hd*(*x n*) + *hd*(*tl*(*x n*))]

definition *Sum2* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Sum2* = *FBlock* (λ*x n. True*) *2 1* (*f-Sum2*)

SumSub2 is a special case of *SumSub* and it is equal to subtract the second input from the first input.

definition *f-SumSub2* :: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-SumSub2 x n* = [*hd*(*x n*) − *hd*(*tl*(*x n*))]

definition *SumSub2* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: $SumSub2 = FBlock\ (\lambda x\ n.\ True)\ 2\ 1\ (f\text{-}SumSub2)$

SubSum2 is a special case of *SumSub* and it is equal to subtract the first input from the second input.

definition *f-SubSum2* :: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: $f\text{-}SubSum2\ x\ n = [-\ hd(x\ n) + hd(tl(x\ n))]$

definition *SubSum2* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: $SubSum2 = FBlock\ (\lambda x\ n.\ True)\ 2\ 1\ (f\text{-}SubSum2)$

A.6.5 Product

The Product block performs multiplication and division.

not-divide-by-zero is a predicate in assumption. For signs, true denotes * and false for /.

fun *not-divide-by-zero* **where**
not-divide-by-zero [] - = *True* |
not-divide-by-zero $(x\#xs)$ $(s\#ss)$ =
  $(HOL.conj\ (not\text{-}divide\text{-}by\text{-}zero\ xs\ ss)\ (if\ s\ then\ True\ else\ (x \neq 0)))$

product-by-sign: multiplies or divides by signs.

fun *product-by-sign* **where**
product-by-sign [] - = *1* |
product-by-sign $(x\#xs)$ $(s\#ss)$ =
  $(if\ s\ then\ (product\text{-}by\text{-}sign\ xs\ ss * x)\ else\ (product\text{-}by\text{-}sign\ xs\ ss\ /\ x))$

definition *f-ProdDiv* :: $bool\ list \Rightarrow (nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: $f\text{-}ProdDiv\ signs\ x\ n = [product\text{-}by\text{-}sign\ (x\ n)\ signs]$

definition *f-no-div-by-zero* :: $bool\ list \Rightarrow (nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow bool$ **where**
[*f-blocks*]: $f\text{-}no\text{-}div\text{-}by\text{-}zero\ signs\ x\ n = not\text{-}divide\text{-}by\text{-}zero\ (x\ n)\ signs$

ProdDiv has additional precondition that assumes all values of the divisor inputs are not equal to zero.

definition *ProdDiv* :: $nat \Rightarrow bool\ list \Rightarrow sim\text{-}state\ hrel\text{-}des$ **where**
[*f-sim-blocks*]: $ProdDiv\ nr\ signs = FBlock\ (\lambda x\ n.\ (f\text{-}no\text{-}div\text{-}by\text{-}zero\ signs\ x\ n))\ nr\ 1\ (f\text{-}ProdDiv\ signs)$

Mul2 is a special case of *ProdDiv* and it multiplies two inputs.

definition *f-Mul2*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: $f\text{-}Mul2\ x\ n = [hd(x\ n) * hd(tl(x\ n))]$

definition *Mul2* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: $Mul2 = FBlock\ (\lambda x\ n.\ True)\ 2\ 1\ (f\text{-}Mul2)$

Div2 is a special case of *ProdDiv* and the first input is divided by the second input.

definition *f-Div2*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: $f\text{-}Div2\ x\ n = [hd(x\ n)\ /\ hd(tl(x\ n))]$

definition *Div2* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: $Div2 = FBlock\ (\lambda x\ n.\ (hd(tl(x\ n)) \neq 0))\ 2\ 1\ (f\text{-}Div2)$

A.6.6 Gain

definition *f-Gain*:: *real* ⇒ (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-Gain k x n* = [*k* ∗ *hd*(*x n*)]

definition *Gain* :: *real* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Gain k* = *FBlock* (λ*x n*. *True*) *1 1* (*f-Gain k*)

A.6.7 Saturation

definition *f-Limit*:: *real* ⇒ *real* ⇒ (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-Limit ymin ymax x n* =
  [*if ymin* > *hd*(*x n*) *then ymin else*
    (*if ymax* < *hd*(*x n*) *then ymax else hd*(*x n*))]

definition *Limit* :: *real* ⇒ *real* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Limit ymin ymax* = *FBlock* (λ*x n*. *True*) *1 1* (*f-Limit ymin ymax*)

A.6.8 MinMax

MinList: return the minimum number from a list of numbers.

fun *MinList* **where**
MinList [] *minx* = *minx* |
MinList (*x*#*xs*) *minx* =
  (*if x* < *minx*
  then MinList xs x
  else MinList xs minx)

The input list must not be empty.

abbreviation *MinLst* ≡ (λ *lst* . *MinList lst* (*hd*(*lst*)))

MaxList: return the maximum number from a list of numbers.

fun *MaxList* **where**
MaxList [] *maxx* = *maxx* |
MaxList (*x*#*xs*) *maxx* =
  (*if x* > *maxx*
  then MaxList xs x
  else MaxList xs maxx)

The input list must not be empty.

abbreviation *MaxLst* ≡ (λ *lst* . *MaxList lst* (*hd*(*lst*)))

MinN returns the minimum value in the inputs.

definition *f-MinN*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-MinN x n* = [*MinLst* (*x n*)]

definition *MinN* :: *nat* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *MinN nr* = *FBlock* (λ*x n*. *True*) *nr 1* (*f-MinN*)

definition *f-Min2*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-Min2 x n* = [*min* (*hd*(*x n*)) (*hd*(*tl*(*x n*)))]

definition *Min2* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Min2* = *FBlock* (λ*x n*. *True*) *2 1* (*f-Min2*)

MaxN returns the maximum value in the inputs.

definition *f-MaxN*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-MaxN x n* = [*MaxLst* (*x n*)]

definition *MaxN* :: *nat* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *MaxN nr* = *FBlock* (λ*x n*. *True*) *nr 1* (*f-MaxN*)

definition *f-Max2*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-Max2 x n* = [*max* (*hd*(*x n*)) (*hd*(*tl*(*x n*)))]

definition *Max2* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Max2* = *FBlock* (λ*x n*. *True*) *2 1* (*f-Max2*)

A.6.9 Rounding

The Rounding Function block applies a rounding function to the input signal to produce the output signal.

RoundFloor rounds inputs using the floor function.

definition *f-RoundFloor*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-RoundFloor x n* = [*real-of-int* ⌊(*hd*(*x n*))⌋]

definition *RoundFloor* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *RoundFloor* = *FBlock* (λ*x n*. *True*) *1 1* (*f-RoundFloor*)

RoundCeil rounds inputs using the ceil function.

definition *f-RoundCeil*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-RoundCeil x n* = [*real-of-int* ⌈(*hd*(*x n*))⌉]

definition *RoundCeil* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *RoundCeil* = *FBlock* (λ*x n*. *True*) *1 1* (*f-RoundCeil*)

A.6.10 Logic Operators

The Logical Operator block performs the specified logical operation on its inputs.

- It supports seven operators: AND, OR, NAND, NOR, XOR, NXOR, NOT;
- An input value is TRUE (1) if it is nonzero and FALSE (0) if it is zero;
- An output value is 1 if TRUE and 0 if FALSE;

A.6.10.1 AND

fun *LAnd* :: *real list* ⇒ *bool* **where**
LAnd [] = *True* |
LAnd (*x*#*xs*) = (*if x* = *0 then False else* (*LAnd xs*))

definition *f-LopAND*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-LopAND x n* = [*if LAnd* (*x n*) *then 1 else 0*]

definition *LopAND* :: *nat* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *LopAND m* = *FBlock* (λ*x n*. *True*) *m 1* (*f-LopAND*)

A.6.10.2 OR **fun** *LOr* :: *real list* ⇒ *bool* **where**
LOr [] = *False* |
LOr (*x*#*xs*) = (*if x* ≠ *0 then True else* (*LOr xs*))

definition *f-LopOR*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-LopOR x n* = [*if LOr* (*x n*) *then 1 else 0*]

definition *LopOR* :: *nat* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *LopOR m* = *FBlock* (λ*x n*. *True*) *m 1* (*f-LopOR*)

A.6.10.3 NAND **fun** *LNand* :: *real list* ⇒ *bool* **where**
LNand [] = *False* |
LNand (*x*#*xs*) = (*if x* = *0 then True else* (*LNand xs*))

definition *f-LopNAND*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-LopNAND x n* = [*if LNand* (*x n*) *then 1 else 0*]

definition *LopNAND* :: *nat* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *LopNAND m* = *FBlock* (λ*x n*. *True*) *m 1* (*f-LopNAND*)

A.6.10.4 NOR **fun** *LNor* :: *real list* ⇒ *bool* **where**
LNor [] = *True* |
LNor (*x*#*xs*) = (*if x* ≠ *0 then False else* (*LNand xs*))

definition *f-LopNOR*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-LopNOR x n* = [*if LNor* (*x n*) *then 1 else 0*]

definition *LopNOR* :: *nat* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *LopNOR m* = *FBlock* (λ*x n*. *True*) *m 1* (*f-LopNOR*)

A.6.10.5 XOR **fun** *LXor* :: *real list* ⇒ *nat* ⇒ *bool* **where**
LXor [] *t* = (*if t mod 2* = *0 then False else True*) |
LXor (*x*#*xs*) *t* = (*if x* ≠ *0 then* (*LXor xs* (*t*+*1*)) *else* (*LXor xs t*))

lemma *LXor* [*0*, *1*, *1*] *0* = *False*
by *auto*

lemma *LXor* [*0*, *1*, *1*, *1*] *0* = *True*
by *auto*

definition *f-LopXOR*:: (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-LopXOR x n* = [*if LXor* (*x n*) *0 then 1 else 0*]

definition *LopXOR* :: *nat* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *LopXOR m* = *FBlock* (λ*x n*. *True*) *m 1* (*f-LopXOR*)

A.6.10.6 NXOR **fun** *LNxor* :: *real list* ⇒ *nat* ⇒ *bool* **where**
LNxor [] *t* = (*if t mod 2* = *0 then True else False*) |
LNxor (*x*#*xs*) *t* = (*if x* ≠ *0 then* (*LNxor xs* (*t*+*1*)) *else* (*LNxor xs t*))

lemma *LNxor* [*0*, *1*, *1*] *0* = *True*
by *auto*

lemma *LNxor* [*0*, *1*, *1*, *1*] *0* = *False*
by *auto*

definition *f-LopNXOR*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-LopNXOR x n* = [*if LNxor* (*x n*) *0 then 1 else 0*]

definition *LopNXOR* :: *nat* $\Rightarrow$ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *LopNXOR m* = *FBlock* ($\lambda x\ n.$ *True*) *m 1* (*f-LopNXOR*)

A.6.10.7 NOT

definition *f-LopNOT*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-LopNOT x n* = [*if hd*(*x n*) = *0 then 1 else 0*]

definition *LopNOT* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *LopNOT* = *FBlock* ($\lambda x\ n.$ *True*) *1 1* (*f-LopNOT*)

A.6.11 Relational Operator

The Relational Operator block performs specified relational operation on inputs.

- It supports six operators for two-input mode: ==, =, <, <=, >, >=;
- An output value is 1 if TRUE and 0 if FALSE;

A.6.11.1 Equal ==

definition *f-RopEQ*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-RopEQ x n* = [*if hd*(*x n*) = *hd*(*tl*(*x n*)) *then 1 else 0*]

definition *RopEQ* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *RopEQ* = *FBlock* ($\lambda x\ n.$ *True*) *2 1* (*f-RopEQ*)

A.6.11.2 Notequal =

definition *f-RopNEQ*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-RopNEQ x n* = [*if hd*(*x n*) = *hd*(*tl*(*x n*)) *then 0 else 1*]

definition *RopNEQ* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *RopNEQ* = *FBlock* ($\lambda x\ n.$ *True*) *2 1* (*f-RopNEQ*)

A.6.11.3 Less Than <

definition *f-RopLT*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-RopLT x n* = [*if hd*(*x n*) $<$ *hd*(*tl*(*x n*)) *then 1 else 0*]

definition *RopLT* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *RopLT* = *FBlock* ($\lambda x\ n.$ *True*) *2 1* (*f-RopLT*)

A.6.11.4 Less Than or Equal to <=

definition *f-RopLE*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-RopLE x n* = [*if hd*(*x n*) $\leq$ *hd*(*tl*(*x n*)) *then 1 else 0*]

definition *RopLE* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *RopLE* = *FBlock* ($\lambda x\ n.$ *True*) *2 1* (*f-RopLE*)

A.6.11.5 Greater Than >

definition *f-RopGT*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-RopGT x n* = [*if hd*(*x n*) $>$ *hd*(*tl*(*x n*)) *then 1 else 0*]

definition *RopGT* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *RopGT* = *FBlock* ($\lambda x\ n.$ *True*) *2 1* (*f-RopGT*)

A.6.11.6 Greater Than or Equal to $>=$

definition *f-RopGE*:: (*nat* $\Rightarrow$ *real list*) $\Rightarrow$ *nat* $\Rightarrow$ (*real list*) **where**
[*f-blocks*]: *f-RopGE x n* = [*if* $hd(x\ n) \geq hd(tl(x\ n))$ *then 1 else 0*]

definition *RopGE* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *RopGE* = *FBlock* ($\lambda x\ n.$ *True*) *2 1* (*f-RopGE*)

A.6.12 Switch

The Switch block switches the output between the first input and the third input based on the value of the second input.

- The first and the third inputs are data inputs;
- The second is the control input.
- Criteria for passing first input: $u2 \geq$ *Threshold*, $u2 >$ *Threshold*, or $u2\ = 0$;

Switch1: criteria is $u2 \geq$ *Threshold*

definition *f-Switch1*:: *real* $\Rightarrow$ (*nat* $\Rightarrow$ *real list*) $\Rightarrow$ *nat* $\Rightarrow$ (*real list*) **where**
[*f-blocks*]: *f-Switch1 th x n* = [*if* $(x\ n)!1 \geq th$ *then* $(x\ n)!0$ *else* $(x\ n)!2$]

definition *Switch1* :: *real* $\Rightarrow$ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Switch1 th* = *FBlock* ($\lambda x\ n.$ *True*) *3 1* (*f-Switch1 th*)

Switch2: criteria is $u2 >$ *Threshold*

definition *f-Switch2*:: *real* $\Rightarrow$ (*nat* $\Rightarrow$ *real list*) $\Rightarrow$ *nat* $\Rightarrow$ (*real list*) **where**
[*f-blocks*]: *f-Switch2 th x n* = [*if* $(x\ n)!1 > th$ *then* $(x\ n)!0$ *else* $(x\ n)!2$]

definition *Switch2* :: *real* $\Rightarrow$ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Switch2 th* = *FBlock* ($\lambda x\ n.$ *True*) *3 1* (*f-Switch2 th*)

Switch3: criteria is $u2\ = 0$

definition *f-Switch3*:: (*nat* $\Rightarrow$ *real list*) $\Rightarrow$ *nat* $\Rightarrow$ (*real list*) **where**
[*f-blocks*]: *f-Switch3 x n* = [*if* $(x\ n)!1 = 0$ *then* $(x\ n)!2$ *else* $(x\ n)!0$]

definition *Switch3* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Switch3* = *FBlock* ($\lambda x\ n.$ *True*) *3 1* (*f-Switch3*)

A.6.13 Data Type Conversion

Data Type Conversion: converts an input signal to the specified data type.

Integer round number towards zero

definition *RoundZero* :: *real* $\Rightarrow$ *int* **where**
RoundZero x = (*if* $x \geq (0::real)$ *then* $\lfloor x \rfloor$ *else* $\lceil x \rceil$)

lemma *RoundZero 1.1 = 1*
apply (*simp add*: *RoundZero-def*)
done

lemma *RoundZero* $(-1.1) = -1$
apply (*simp add*: *RoundZero-def*)
done

int8: convert int to int8.

definition *int8* :: *int* ⇒ *int* **where**
int8 x = ((*x*+*128*) *mod 256*) − *128*

int16: convert int to int16.

definition *int16* :: *int* ⇒ *int* **where**
int16 x = ((*x*+*32768*) *mod 65536*) − *32768*

int32: convert int to int32.

definition *int32* :: *int* ⇒ *int* **where**
int32 x = ((*x*+*2147483648*) *mod 4294967296*) − *2147483648*

lemma *int32-eq*:
  assumes *x* ≥ *0* ∧ *x* < *2147483648*
  shows *int32 x* = *x*
  apply (*simp add*: *int32-def*)
  using *assms* **by** (*smt int-mod-eq*)

lemma *int8* (−*1*) = −*1*
by (*simp add*: *int8-def*)

lemma *int8* (−*128*) = −*128*
by (*simp add*: *int8-def*)

lemma *int8* (−*129*) = *127*
by (*simp add*: *int8-def*)

lemma *int8* (*129*) = −*127*
by (*simp add*: *int8-def*)

lemma *int8* (−*378*) = −*122*
by (*simp add*: *int8-def*)

lemma *int8* (*378*) = *122*
by (*simp add*: *int8-def*)

uint8: convert int to uint8

definition *uint8* :: *int* ⇒ *int* **where**
uint8 x = *x mod 256*

lemma *uint8* (−*1*) = *255*
by (*simp add*: *uint8-def*)

uint16: convert int to uint16

definition *uint16* :: *int* ⇒ *int* **where**
uint16 x = *x mod 65536*

uint32: convert int to uint32

definition *uint32* :: *int* ⇒ *int* **where**
uint32 x = *x mod 4294967296*

lemma (*uint32 4294967296*) = *0*
  by (*simp add*: *uint32-def*)

lemma $(uint32\ 4294967295) = 4294967295$
  by (*simp add*: *uint32-def*)

lemma $(uint32\ (-1)) = 4294967295$
  by (*simp add*: *uint32-def*)

lemma $(uint32\ (-4294967298)) = 4294967294$
  by (*simp add*: *uint32-def*)

DataTypeConvUint32Zero: convert to uint32 and round number towards zero.

definition *f-DTConvUint32Zero*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-DTConvUint32Zero x n* = [*real-of-int* (*uint32* (*RoundZero*(*hd* (*x n*))))]

definition *DataTypeConvUint32Zero* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *DataTypeConvUint32Zero* = *FBlock* ($\lambda x\ n.$ *True*) *1 1* (*f-DTConvUint32Zero*)

DataTypeConvInt32Zero: convert to int32 and round number towards zero.

definition *f-DTConvInt32Zero*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-DTConvInt32Zero x n* = [*real-of-int* (*int32* (*RoundZero*(*hd* (*x n*))))]

definition *DataTypeConvInt32Zero* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *DataTypeConvInt32Zero* = *FBlock* ($\lambda x\ n.$ *True*) *1 1* (*f-DTConvInt32Zero*)

DataTypeConvUint32Floor: convert to uint32 and round number using floor.

definition *f-DTConvUint32Floor*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-DTConvUint32Floor x n* = [*real-of-int* (*uint32* ($\lfloor$(*hd* (*x n*))$\rfloor$))]

definition *DataTypeConvUint32Floor* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *DataTypeConvUint32Floor* = *FBlock* ($\lambda x\ n.$ *True*) *1 1* (*f-DTConvUint32Floor*)

DataTypeConvInt32Floor: convert to int32 and round number using floor.

definition *f-DTConvInt32Floor*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-DTConvInt32Floor x n* = [*real-of-int* (*int32* ($\lfloor$(*hd* (*x n*))$\rfloor$))]

definition *DataTypeConvInt32Floor* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *DataTypeConvInt32Floor* = *FBlock* ($\lambda x\ n.$ *True*) *1 1* (*f-DTConvInt32Floor*)

DataTypeConvUint32Ceil: convert to uint32 and round number using ceil.

definition *f-DTConvUint32Ceil*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-DTConvUint32Ceil x n* = [*real-of-int* (*uint32* ($\lceil$(*hd* (*x n*))$\rceil$))]

definition *DataTypeConvUint32Ceil* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *DataTypeConvUint32Ceil* = *FBlock* ($\lambda x\ n.$ *True*) *1 1* (*f-DTConvUint32Ceil*)

DataTypeConvInt32Ceil: convert to int32 and round number using ceil.

definition *f-DTConvInt32Ceil*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow (real\ list)$ **where**
[*f-blocks*]: *f-DTConvInt32Ceil x n* = [*real-of-int* (*int32* ($\lceil$(*hd* (*x n*))$\rceil$))]

definition *DataTypeConvInt32Ceil* :: *sim-state hrel-des* **where**
[*f-sim-blocks*]: *DataTypeConvInt32Ceil* = *FBlock* ($\lambda x\ n.$ *True*) *1 1* (*f-DTConvInt32Ceil*)

A.6.14 Initial Condition (IC)

The IC block sets the initial condition of the signal at its input port. The block does this by outputting the specified initial condition when you start the simulation, regardless of the actual value of the input signal. Thereafter, the block outputs the actual value of the input signal.

definition *f-IC*:: *real* ⇒ (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-IC x0 x n* = [*if n* = *0 then x0 else hd*(*x n*)]

definition *IC* :: *real* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *IC x0* = *FBlock* (λ*x n*. *True*) *1 1* (*f-IC x0*)

A.6.15 Router Block

A new introduced block to route signals: the same number of inputs and outputs but in different orders.

fun *assembleOutput*:: *real list* ⇒ *nat list* ⇒ *real list* **where**
assembleOutput ins [] = [] |
assembleOutput ins (*x*#*xs*) = (*ins*!*x*)#(*assembleOutput ins* (*xs*))

definition *f-Router*:: *nat list* ⇒ (*nat* ⇒ *real list*) ⇒ *nat* ⇒ (*real list*) **where**
[*f-blocks*]: *f-Router routes x n* = *assembleOutput* (*x n*) *routes*

lemma *f-Router* [*2*,*0*,*1*] (λ*na*. [*11*, *22*, *33*]) *n* = [*33*, *11*, *22*]
  by (*simp add*: *f-blocks*)

definition *Router* :: *nat* ⇒ *nat list* ⇒ *sim-state hrel-des* **where**
[*f-sim-blocks*]: *Router nn routes* = *FBlock* (λ*x n*. *True*) *nn nn* (*f-Router routes*)

end

B Block Laws

In this section, many theorems and laws are proved to facilitate application of our theories in Simulink block diagrams.

theory *simu-contract-real-laws*
  imports
    simu-contract-real
begin

— timeout in seconds
declare [[*smt-timeout* = *600*]]

B.1 Additional Laws

list-len-avail: there always exists some signals that could have a specific size.

lemma *list-len-avail*:
  ∀ *x*≥*0*. (∃ (*xx*::*nat*⇒*real list*). ∀ *n*. *length* (*xx n*) = *x*)
  apply (*rule allI*)
  apply (*auto*)
  apply (*induct-tac x*)
  apply (*rule-tac x* = λ*na*. [] **in** *exI*, *simp*)
  apply (*auto*)
  by (*rule-tac x* = λ*na*. *0*#(*xx na*) **in** *exI*, *simp*)

list-len-avail: there always exists some signals that could have a specific size and the value of each signal is equal to an arbitrary real number.

lemma *list-len-avail'*:
  $\forall r::real.\ \forall x \geq 0.\ (\exists (xx::nat \Rightarrow real\ list).\ (\forall n.\ (length\ (xx\ n) = x) \wedge (\forall y::nat<x.\ ((xx\ n)!y = r))))$
  apply (*rule allI*)
  apply (*auto*)
  apply (*induct-tac x*)
  apply (*rule-tac* $x = \lambda na.$ [] **in** *exI*, *simp*)
  apply (*auto*)
  apply (*rule-tac* $x = \lambda na.\ r\#(xx\ na)$ **in** *exI*, *simp*)
  using *less-Suc-eq-0-disj* **by** *auto*

sum-hd-signal sums up a signal's current value and all past values.

fun *sum-hd-signal*:: $(nat \Rightarrow real\ list) \Rightarrow nat \Rightarrow real$ **where**
sum-hd-signal $x\ 0 = hd(x\ 0)$ |
sum-hd-signal $x\ (Suc\ n) = hd(x\ (Suc\ n))$ + *sum-hd-signal* $x\ (n)$

remove-at removes the ith element from a list.

abbreviation *remove-at* $\equiv (\lambda lst\ i.\ (take\ (i)\ lst) \bullet (drop\ (i+1)\ lst))$

lemma *remove-at* [] 1 = [] **by** *simp*

lemma *remove-at* $[2,3,4]\ 1 = [2,4]$ **by** *simp*

fun-eq: two functions are equal as long as they are equal in all their domains (total functions).

lemma *fun-eq*:
  assumes $\forall x.\ f\ x = g\ x$
  shows $f = g$
  by (*simp add*: *assms ext*)

fun-eq': two functions are equal in all their domains then they are equal functions. (total functions).

lemma *fun-eq'*:
  assumes $f = g$
  shows $\forall x.\ (f\ x = g\ x)$
  by (*simp add*: *assms*)

lemma *fun-neq*:
  assumes $\forall x.\ \neg\ (f\ x = g\ x)$
  shows $\neg\ f = g$
  using *assms* **by** *auto*

ref-eq: two predicates are equal as long as they are refined by each other.

lemma *ref-eq*:
  assumes $P \sqsubseteq Q$
  assumes $Q \sqsubseteq P$
  shows $P = Q$
  by (*simp add*: *antisym assms*(*1*) *assms*(*2*))

lemma *hd-drop-m*:
  $\forall (x::nat \Rightarrow real\ list)\ n::nat.\ length(x\ n) > m \longrightarrow (hd\ (drop\ m\ (x\ n)) = x\ n!m)$
  using *hd-drop-conv-nth* **by** *blast*

lemma *hd-take-m*:
  $m > 0 \longrightarrow (\forall (x::nat \Rightarrow real\ list)\ n::nat.\ (hd\ (take\ m\ (x\ n)) = hd(x\ n)))$
  by (*metis append-take-drop-id hd-append2 less-numeral-extra*(*3*) *take-eq-Nil*)

lemma *hd-tl-take-m*:
  $m > 1 \longrightarrow (\forall (x::nat \Rightarrow real\ list)\ n::nat.\ (hd\ (tl\ (take\ m\ (x\ n))) = hd(tl(x\ n))))$
    by (*metis hd-conv-nth less-numeral-extra*(*3*) *nth-take take-eq-Nil tl-take zero-less-diff*)

B.2 SimBlock healthiness

lemma *SimBlock-FBlock* [*simblock-healthy*]:
  assumes *s1*: $\exists\, inouts_v\ inouts_v{}'.$
    $\forall\, x.\ length(inouts_v{}'\ x) = n\ \wedge$
      $length(inouts_v\ x) = m\ \wedge$
      $f\ inouts_v\ x = inouts_v{}'\ x$
  assumes *s2*: $\forall\, x\ na.\ length(x\ na) = m \longrightarrow length(f\ x\ na) = n$
  shows $SimBlock\ m\ n\ (FBlock\ (\lambda x\ n.\ True)\ m\ n\ f)$
  apply (*simp add*: *SimBlock-def FBlock-def*)
  apply (*rel-auto*)
  using *s1* **apply** *blast*
  by (*simp add*: *s2*)

lemma *SimBlock-FBlock'* [*simblock-healthy*]:
  assumes *s1*: $\exists\, inouts_v.\ (\forall\, x.\ p1\ inouts_v\ x)\ \wedge$
    $(\exists\ inouts_v{}'.$
    $\forall\, x.\ length(inouts_v{}'\ x) = n\ \wedge$
      $length(inouts_v\ x) = m\ \wedge$
      $f\ inouts_v\ x = inouts_v{}'\ x)$
  assumes *s2*: $\forall\, x\ na.\ length(x\ na) = m \longrightarrow length(f\ x\ na) = n$
  shows $SimBlock\ m\ n\ (FBlock\ (p1)\ m\ n\ f)$
  apply (*simp add*: *SimBlock-def FBlock-def*)
  apply (*rel-auto*)
  using *s1 s2* **by** *blast*

lemma *SimBlock-FBlock-fn* [*simblock-healthy*]:
  assumes *s1*: $SimBlock\ m\ n\ (FBlock\ (\lambda x\ n.\ True)\ m\ n\ f)$
  shows $(\forall\, x\ xa.\ length(x\ xa) = m \longrightarrow length(f\ x\ xa) = n)$
  proof −
    have *1*: $PrePost((FBlock\ (\lambda x\ n.\ True)\ m\ n\ f)) \neq false$
      using *s1 SimBlock-def*
      by *blast*
    then show *?thesis*
      apply (*simp add*: *FBlock-def*)
      apply (*rel-simp*)
    done
  qed

lemma *SimBlock-FBlock-fn'* [*simblock-healthy*]:
  assumes *s1*: $SimBlock\ m\ n\ (FBlock\ (p)\ m\ n\ f)$
  shows $(\forall\, x\ xa.\ length(x\ xa) = m \longrightarrow length(f\ x\ xa) = n)$
  proof −
    have *1*: $PrePost((FBlock\ (p)\ m\ n\ f)) \neq false$
      using *s1 SimBlock-def*
      by *blast*
    then show *?thesis*
      apply (*simp add*: *FBlock-def*)
      apply (*rel-simp*)
    done
  qed

lemma *SimBlock-FBlock-p* [*simblock-healthy*]:
  assumes *s1*: *SimBlock m n* (*FBlock* (*p*) *m n f*)
  shows $\exists\, inouts_v\ .\ \forall\, x.\ p\ inouts_v\ x \wedge length(inouts_v\ x) = m$
  proof −
    have *1*: *PrePost*((*FBlock* (*p*) *m n f*)) ≠ *false*
      using *s1 SimBlock-def*
      by *blast*
    then show *?thesis*
      apply (*simp add*: *FBlock-def*)
      apply (*rel-simp*)
      by *blast*
  qed

lemma *SimBlock-FBlock-p-f* [*simblock-healthy*]:
  assumes *s1*: *SimBlock m n* (*FBlock* (*p*) *m n f*)
  shows $\exists\, inouts_v\ .\ \forall\, x.\ p\ inouts_v\ x \wedge$
   $(\exists\, inouts_v{}'.\ \forall\, x.\ length(inouts_v{}'\ x) = n \wedge length(inouts_v\ x) = m \wedge f\ inouts_v\ x = inouts_v{}'\ x)$
  proof −
    have *1*: *PrePost*((*FBlock* (*p*) *m n f*)) ≠ *false*
      using *s1 SimBlock-def*
      by *blast*
    then show *?thesis*
      apply (*simp add*: *FBlock-def*)
      apply (*rel-simp*)
      by *blast*
  qed

lemma *FBlock-eq*:
  assumes *f1* = *f2*
  shows *FBlock p-f m n f1* = *FBlock p-f m n f2*
  using *assms* **by** *simp*

lemma *FBlock-eq′*:
  assumes $\forall\,(x{::}nat \Rightarrow real\ list)\ n.\ length(x\ n) = m \longrightarrow f1\ x\ n = f2\ x\ n$
  shows *FBlock p-f m n f1* = *FBlock p-f m n f2*
  apply (*simp add*: *FBlock-def*)
  apply (*rule ref-eq*)
  apply (*rel-simp*)
  using *assms* **apply** *simp*
  apply (*rel-simp*)
  using *assms* **by** *metis*

lemma *FBlock-eq′′*:
  assumes *s1*: $\forall\,(x{::}nat \Rightarrow real\ list)\ n.\ (\forall\, na.\ length(x\ na) = m) \longrightarrow f1\ x\ n = f2\ x\ n$
  assumes *s2*: $\forall\,(x{::}nat \Rightarrow real\ list)\ na.\ length(f1\ x\ na) = n$
  assumes *s3*: $\forall\,(x{::}nat \Rightarrow real\ list)\ na.\ length(f2\ x\ na) = n$
  shows *FBlock p-f m n f1* = *FBlock p-f m n f2*
  apply (*simp add*: *FBlock-def*)
  apply (*rule ref-eq*)
  apply (*rel-simp*)
  apply (*rule conjI*)
  apply (*simp add*: *assms*)
  using *assms* **apply** *blast*

apply (*rel-simp*)
using *assms* **by** *metis*

B.3 inps and outps

lemma *inps-P*:
assumes *SimBlock m n P*
shows *inps* $P = m$
using *assms inps-outps* **by** *auto*

lemma *outps-P*:
assumes *SimBlock m n P*
shows *outps* $P = n$
using *assms inps-outps* **by** *auto*

lemma *SimBlock-implies-not-PQ* [*simblock-healthy*]:
assumes *s1*: *SimBlock m n* $(P \vdash_n Q)$
shows $(\lceil P \rceil_< \wedge Q) \neq \mathit{false}$
using *SimBlock-def s1* **by** *auto*

lemma *SimBlock-implies-not-P* [*simblock-healthy*]:
assumes *s1*: *SimBlock m n* $(P \vdash_n Q)$
shows $\lceil P \rceil_< \neq \mathit{false}$
using *SimBlock-def s1*
by (*metis SimBlock-implies-not-PQ aext-false ndesign-def ndesign-refinement′ true-conj-zero*(*1*) *utp-pred-laws.bot.extremum utp-pred-laws.inf.orderE*)

lemma *SimBlock-implies-not-P′* [*simblock-healthy*]:
assumes *s1*: *SimBlock m n* $(P \vdash_n Q)$
shows $P \neq \mathit{false}$
using *SimBlock-def s1*
by (*metis SimBlock-implies-not-PQ aext-false ndesign-def utp-pred-laws.bot.extremum utp-pred-laws.inf.orderE*)

lemma *SimBlock-implies-not-P′′* [*simblock-healthy*]:
assumes *s1*: *SimBlock m n* $(P \vdash_n Q)$
shows $\exists\, inouts_v\ inouts_v{}'.\ [\![\lceil P \rceil_<]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v{}'|\!))$
using *SimBlock-implies-not-P*
by (*metis* (*mono-tags*, *hide-lams*) *bot-bool-def bot-uexpr.rep-eq false-upred-def old.unit.exhaust s1 sim-state.cases-scheme surj-pair udeduct-eqI*)

lemma *SimBlock-implies-not-P-cond* [*simblock-healthy*]:
assumes *s1*: *SimBlock m n* $(P \vdash_r Q)$
assumes *s2*: $out\alpha \sharp P$
shows $\forall\, inouts_v\ inouts_v{}'\ inouts_v{}''.$
$[\![P]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v{}''|\!)) = [\![P]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v{}'|\!))$
using *SimBlock-implies-not-P s1 s2*
by (*rel-simp*)

lemma *SimBlock-implies-not-Q* [*simblock-healthy*]:
assumes *s1*: *SimBlock m n* $(P \vdash_n Q)$
shows $Q \neq \mathit{false}$
using *SimBlock-def s1* **by** *auto*

lemma *SimBlock-implies-not-Q′* [*simblock-healthy*]:

assumes $s1$: $SimBlock\ m\ n\ (P \vdash_n\ Q)$
shows $\exists\, inouts_v\ inouts_v{}'.\ [\![Q]\!]_e\ ((\!|inouts_v = inouts_v|\!),\ (\!|inouts_v = inouts_v{}'|\!))$
using *SimBlock-implies-not-Q*
by (*metis* (*mono-tags*, *hide-lams*) *bot-bool-def bot-uexpr.rep-eq false-upred-def old.unit.exhaust s1 sim-state.cases-scheme surj-pair udeduct-eqI*)

lemma *SimBlock-implies-not-PQ'* [*simblock-healthy*]:
assumes $s1$: $SimBlock\ m\ n\ (P \vdash_n\ Q)$
shows $\exists\, inouts_v\ inouts_v{}'.\ ([\![P]\!]_e\ ((\!|inouts_v = inouts_v|\!)) \wedge$
$[\![Q]\!]_e\ ((\!|inouts_v = inouts_v|\!),\ (\!|inouts_v = inouts_v{}'|\!)))$
using *s1 SimBlock-implies-not-PQ* **apply** (*rel-simp*)
done

lemma *SimBlock-implies-mP* [*simblock-healthy*]:
assumes $s1$: $SimBlock\ m\ n\ (P \vdash_n\ Q)$
shows $\forall\, inouts_v\ inouts_v{}'\ x.$
$[\![P]\!]_e\ ((\!|inouts_v = inouts_v|\!)) \longrightarrow$
$[\![Q]\!]_e\ ((\!|inouts_v = inouts_v|\!),\ (\!|inouts_v = inouts_v{}'|\!)) \longrightarrow$
$length(inouts_v\ x) = m$
proof −
from *s1* **have** $1{:}((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m») \sqsubseteq Dom(PrePost((P \vdash_n\ Q))))$
by (*simp add*: *SimBlock-def*)
then show *?thesis*
by (*rel-auto*)
qed

lemma *SimBlock-implies-Qn* [*simblock-healthy*]:
assumes $s1$: $SimBlock\ m\ n\ (P \vdash_n\ Q)$
shows $\forall\, inouts_v\ inouts_v{}'\ x.$
$[\![P]\!]_e\ ((\!|inouts_v = inouts_v|\!)) \longrightarrow$
$[\![Q]\!]_e\ ((\!|inouts_v = inouts_v|\!),\ (\!|inouts_v = inouts_v{}'|\!)) \longrightarrow$
$length(inouts_v{}'\ x) = n$
proof −
from *s1* **have** $1{:}((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n») \sqsubseteq Ran(PrePost((P \vdash_n\ Q))))$
by (*simp add*: *SimBlock-def*)
then show *?thesis*
by (*rel-auto*)
qed

lemma *sim-refine-implies-inps-outps-eq*:
assumes $s1$: $SimBlock\ m1\ n1\ (P)$
assumes $s2$: $SimBlock\ m2\ n2\ (Q)$
assumes $s3$: $(P) \sqsubseteq (Q)$
assumes $s4$: $(pre_D(P) \wedge post_D(Q)) \neq false$
shows $m1 = m2 \wedge n1 = n2$
proof −
have *ref-des*: $pre_D(Q) \sqsubseteq pre_D(P) \wedge post_D(P) \sqsubseteq (pre_D(P) \wedge post_D(Q))$
using *s3*
by (*simp add*: *design-refine-thms*(*1*) *design-refine-thms*(*2*) *refBy-order*)
have *pred-1*: $PrePost(P) = (pre_D(P) \wedge post_D(P))$
apply (*simp*)
done
have *pred-2*: $PrePost(Q) = (pre_D(Q) \wedge post_D(Q))$
apply (*simp*)
done

have *pred-1-not-false*: $(pre_D(P) \wedge post_D(P)) \neq false$
using *SimBlock-def s1* **by** *force*
have *pred-2-not-false*: $(pre_D(Q) \wedge post_D(Q)) \neq false$
using *SimBlock-def s2* **by** *force*
have *ref-inps-1*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m1») \sqsubseteq Dom((pre_D(P) \wedge post_D(P))))$
using *s1* **apply** (*simp add*: *SimBlock-def*)
done
then have *ref-inps-12*: ... $\sqsubseteq Dom((pre_D(P) \wedge post_D(Q)))$
apply (*simp add*: *ref-des Dom-def*)
by (*smt ref-des arestr.rep-eq conj-upred-def ex.rep-eq inf-bool-def inf-uexpr.rep-eq upred-ref-iff*)
have *ref-inps-2*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m2») \sqsubseteq Dom((pre_D(Q) \wedge post_D(Q))))$
using *s2* **apply** (*simp add*: *SimBlock-def*)
done
have *ref-p2-p1*: $Dom((pre_D(Q) \wedge post_D(Q))) \sqsubseteq Dom((pre_D(P) \wedge post_D(Q)))$
apply (*simp add*: *Dom-def*)
by (*smt ref-des aext-mono arestr-and order-refl utp-pred-laws.ex-mono utp-pred-laws.inf.absorb-iff2 utp-pred-laws.inf-mono*)
from *ref-p2-p1* **and** *ref-inps-2* **have** *ref-inps-2-p1*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m2») \sqsubseteq Dom((pre_D(P) \wedge post_D(Q))))$
by *simp*
from *ref-inps-2-p1* **have** *P1-Q2-implies-m2*: $(\forall\, b.\ [\![Dom((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow [\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m2»)]\!]_e\ b)$
apply (*simp add*: *upred-ref-iff*)
done
from *ref-inps-12* **have** *P1-Q2-implies-m1*: $(\forall\, b.\ [\![Dom((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow [\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m1»)]\!]_e\ b)$
apply (*simp add*: *upred-ref-iff*)
done
from *P1-Q2-implies-m1* **and** *P1-Q2-implies-m2* **have** *P1-Q2-implies-m2-m1*:
$\forall\, b.\ [\![Dom((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow ([\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m2»)]\!]_e\ b \wedge [\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m1»)]\!]_e\ b)$
by *blast*
then have *P1-Q2-implies-m2-m1-1*: $\forall\, b.\ [\![Dom((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow ([\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m2») \wedge (\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m1»)]\!]_e\ b)$
by (*simp add*: *conj-implies2*)
have *forall-comb*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m2») \wedge (\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m1»)) =$
$(\forall\ na \cdot ((\#_u(\&inouts(«na»)_a) =_u «m2») \wedge (\#_u(\&inouts(«na»)_a) =_u «m1»)))$
apply (*rel-auto*)
done
from *P1-Q2-implies-m2-m1-1* **have** *P1-Q2-implies-m2-m1-2*:
$\forall\, b.\ [\![Dom((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow ([\![(\forall\ na \cdot ((\#_u(\&inouts(«na»)_a) =_u «m2») \wedge (\#_u(\&inouts(«na»)_a) =_u «m1»)))]\!]_e\ b)$
by (*simp add*: *forall-comb*)
have *m1-m2-eq*: $m2 = m1$
proof (*rule ccontr*)
assume *ss1*: $m2 \neq m1$
have *conj-false*: $(\forall\ na \cdot ((\#_u(\&inouts(«na»)_a) =_u «m2») \wedge (\#_u(\&inouts(«na»)_a) =_u «m1»))) = false$
using *ss1* **apply** (*rel-auto*)
done
have *imp-false*: $\forall\, b.\ [\![Dom((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow ([\![false]\!]_e\ b)$
using *P1-Q2-implies-m2-m1-2*
apply (*simp add*: *conj-false*)
done

have *dom-false*: $Dom((pre_D(P) \wedge post_D(Q))) = false$
by (*metis imp-false true-conj-zero*(*2*) *udeduct-refineI utp-pred-laws.inf.orderE utp-pred-laws.inf-commute*)
have *P1-Q2-false*: $(pre_D(P) \wedge post_D(Q)) = false$
by (*metis assume-Dom assume-false dom-false seqr-left-zero*)
show *False*
using *s4* **apply** (*simp add*: *P1-Q2-false*)
done
qed

have *ref-inps-1′*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n1») \sqsubseteq Ran((pre_D(P) \wedge post_D(P))))$
using *s1* **apply** (*simp add*: *SimBlock-def*)
done
then have *ref-inps-12′*: $... \sqsubseteq Ran((pre_D(P) \wedge post_D(Q)))$
apply (*simp add*: *ref-des Ran-def*)
by (*smt ref-des arestr.rep-eq conj-upred-def ex.rep-eq inf-bool-def inf-uexpr.rep-eq upred-ref-iff*)
have *ref-inps-2′*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n2») \sqsubseteq Ran((pre_D(Q) \wedge post_D(Q))))$
using *s2* **apply** (*simp add*: *SimBlock-def*)
done
have *ref-p2-p1′*: $Ran((pre_D(Q) \wedge post_D(Q))) \sqsubseteq Ran((pre_D(P) \wedge post_D(Q)))$
apply (*simp add*: *Ran-def*)
by (*smt ref-des aext-mono arestr-and order-refl utp-pred-laws.ex-mono utp-pred-laws.inf.absorb-iff2 utp-pred-laws.inf-mono*)
from *ref-p2-p1′* **and** *ref-inps-2′* **have** *ref-inps-2-p1′*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n2») \sqsubseteq Ran((pre_D(P) \wedge post_D(Q))))$
by *simp*
from *ref-inps-2-p1′* **have** *P1-Q2-implies-n2*: $(\forall b.\ [\![Ran((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow [\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n2»)]\!]_e\ b)$
apply (*simp add*: *upred-ref-iff*)
done
from *ref-inps-12′* **have** *P1-Q2-implies-n1*: $(\forall b.\ [\![Ran((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow [\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n1»)]\!]_e\ b)$
apply (*simp add*: *upred-ref-iff*)
done
from *P1-Q2-implies-n1* **and** *P1-Q2-implies-n2* **have** *P1-Q2-implies-n2-n1*:
$\forall b.\ [\![Ran((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow ([\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n2»)]\!]_e\ b \wedge [\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n1»)]\!]_e\ b)$
by *blast*
then have *P1-Q2-implies-n2-n1-1*:
$\forall b.\ [\![Ran((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow ([\![(\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n2») \wedge (\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n1»)]\!]_e\ b)$
by (*simp add*: *conj-implies2*)
have *forall-comb′*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n2») \wedge (\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n1»)) =$
$(\forall\ na \cdot ((\#_u(\&inouts(«na»)_a) =_u «n2») \wedge (\#_u(\&inouts(«na»)_a) =_u «n1»)))$
apply (*rel-auto*)
done
from *P1-Q2-implies-n2-n1-1* **have** *P1-Q2-implies-n2-n1-2*:
$\forall b.\ [\![Ran((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow ([\![(\forall\ na \cdot ((\#_u(\&inouts(«na»)_a) =_u «n2») \wedge (\#_u(\&inouts(«na»)_a) =_u «n1»)))]\!]_e\ b)$
by (*simp add*: *forall-comb′*)
have *n1-n2-eq*: $n2 = n1$
proof (*rule ccontr*)
assume *ss1*: $n2 \neq n1$
have *conj-false*: $(\forall\ na \cdot ((\#_u(\&inouts(«na»)_a) =_u «n2») \wedge (\#_u(\&inouts(«na»)_a) =_u «n1»))) = false$

using *ss1* **apply** (*rel-auto*)
done
have *imp-false*: $\forall\, b.\ [\![Ran((pre_D(P) \wedge post_D(Q)))]\!]_e\ b \longrightarrow ([\![false]\!]_e\ b)$
using *P1-Q2-implies-n2-n1-2*
apply (*simp add*: *conj-false*)
done
have *dom-false*: $Ran((pre_D(P) \wedge post_D(Q))) = false$
by (*metis imp-false true-conj-zero(2) udeduct-refineI utp-pred-laws.inf.orderE utp-pred-laws.inf-commute*)
have *P1-Q2-false*: $(pre_D(P) \wedge post_D(Q)) = false$
by (*metis assume-Ran assume-false dom-false seqr-right-zero*)
show *False*
using *s4* **apply** (*simp add*: *P1-Q2-false*)
done
qed
show *?thesis*
apply (*simp add*: *n1-n2-eq m1-m2-eq*)
done
qed

B.4 Operators

B.4.1 Id

lemma *SimBlock-Id* [*simblock-healthy*]:
SimBlock 1 1 (*Id*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*simp add*: *f-blocks*)
apply (*metis f-Const-def length-Cons list.size(3)*)
by (*simp add*: *f-blocks*)

lemma *inps-id*: *inps Id* = *1*
using *SimBlock-Id inps-outps* **by** *auto*

lemma *outps-id*: *outps Id* = *1*
using *SimBlock-Id inps-outps* **by** *auto*

B.4.2 Sequential Composition

lemma *refine-seq-mono*:
assumes $P1 \sqsubseteq P2$ **and** $Q1 \sqsubseteq Q2$
shows $P1\ ;;\ Q1 \sqsubseteq P2\ ;;\ Q2$
by (*simp add*: *assms(1) assms(2) seqr-mono*)

lemma *FBlock-seq-comp*:
assumes *s1*: *SimBlock m1 n1* (*FBlock* ($\lambda x\ n.\ True$) *m1 n1 f*)
assumes *s2*: *SimBlock n1 n2* (*FBlock* ($\lambda x\ n.\ True$) *n1 n2 g*)
shows *FBlock* ($\lambda x\ n.\ True$) *m1 n1 f* ;; *FBlock* ($\lambda x\ n.\ True$) *n1 n2 g* = *FBlock* ($\lambda x\ n.\ True$) *m1 n2* ($g \circ f$)
proof −
show *?thesis*
apply (*simp add*: *sim-blocks*)
apply (*rel-simp*)
apply (*rule iffI*)

apply (*clarify*)
apply *presburger*
apply (*rel-auto*)
proof −
fix ok_v $inouts_v$ $ok_v{}'$ $inouts_v{}'$
assume *a0*: $ok_v{}'$
assume *a1*: $(\forall x.\ length(inouts_v\ x) = m1 \land length(inouts_v{}'\ x) = n2 \land$
$g\ (f\ inouts_v)\ x = inouts_v{}'\ x)$
show $\exists\, ok_v{}''\ inouts_v{}''.$
$(ok_v \longrightarrow ok_v{}'' \land (\forall x.\ length(inouts_v{}''\ x) = n1 \land f\ inouts_v\ x = inouts_v{}''\ x)$
$\land (\forall x\ xa.\ length(x\ xa) = m1 \longrightarrow length(f\ x\ xa) = n1)) \land$
$(ok_v{}'' \longrightarrow (\forall x.\ length(inouts_v{}''\ x) = n1 \land g\ inouts_v{}''\ x = inouts_v{}'\ x)$
$\land (\forall x\ xa.\ length(x\ xa) = n1 \longrightarrow length(g\ x\ xa) = n2))$
apply (*rule-tac* $x = ok_v{}'$ **in** *exI*)
apply (*rule-tac* $x = f\ inouts_v$ **in** *exI*, *simp*)
using *SimBlock-FBlock-fn a0 a1 assms*(*2*) *s1* **by** *blast*
qed
qed

lemma *SimBlock-FBlock-seq-comp* [*simblock-healthy*]:
assumes *s1*: *SimBlock m1 n1* (*FBlock* ($\lambda x\ n.$ *True*) *m1 n1 f*)
assumes *s2*: *SimBlock n1 n2* (*FBlock* ($\lambda x\ n.$ *True*) *n1 n2 g*)
shows *SimBlock m1 n2* (*FBlock* ($\lambda x\ n.$ *True*) *m1 n1 f* ;; *FBlock* ($\lambda x\ n.$ *True*) *n1 n2 g*)
apply (*simp add*: *s1 s2 FBlock-seq-comp*)
apply (*rule SimBlock-FBlock*)
proof −
obtain $inouts_v$::*nat* ⇒ *real list* **where** *P*: $\forall na.\ length(inouts_v\ na) = m1$
using *list-len-avail* **by** *auto*
show $\exists\, inouts_v\ inouts_v{}'.\ \forall x.\ length(inouts_v{}'\ x) = n2 \land length(inouts_v\ x) = m1 \land$
$(g \circ f)\ inouts_v\ x = inouts_v{}'\ x$
apply (*rule-tac* $x = inouts_v$ **in** *exI*)
apply (*rule-tac* $x = (g \circ f)\ inouts_v$ **in** *exI*)
using *P SimBlock-FBlock-fn assms*(*2*) *s1* **by** *auto*
next
show $\forall x\ na.\ length(x\ na) = m1 \longrightarrow length((g \circ f)\ x\ na) = n2$
using *SimBlock-FBlock-fn assms*(*2*) *s1* **by** *auto*
qed

lemma *FBlock-seq-comp'*:
assumes *s1*: *SimBlock m1 n1* (*FBlock* (*p1*) *m1 n1 f*)
assumes *s2*: *SimBlock n1 n2* (*FBlock* (*p2*) *n1 n2 g*)
shows *FBlock* ($\lambda x\ n.\ p1\ x\ n \land length(x\ n) = m1$) *m1 n1 f* ;;
FBlock ($\lambda x\ n.\ p2\ x\ n \land length(x\ n) = n1$) *n1 n2 g*
= *FBlock* ($\lambda x\ n.\ p1\ x\ n \land (p2 \circ f)\ x\ n \land length(x\ n) = m1$) *m1 n2* ($g \circ f$)
proof −
from *s1* **have** *1*: $\forall x\ n.\ length(x\ n) = m1 \longrightarrow length(f\ x\ n) = n1$
using *SimBlock-FBlock-fn'* **by** *blast*
from *s2* **have** *2*: $\forall x\ n.\ length(x\ n) = n1 \longrightarrow length(g\ x\ n) = n2$
using *SimBlock-FBlock-fn'* **by** *blast*
show *?thesis*
apply (*simp add*: *sim-blocks*)
apply (*simp add*: *ndesign-composition-wp wp-upred-def*)
apply (*rule ref-eq*)
apply (*rule ndesign-refine-intro*)
apply (*rel-simp*)

using *1* **apply** *fastforce*
apply (*rel-simp*)
apply (*rule-tac* $x = f\ inouts_v$ **in** *exI*)
using *1 2* **apply** *simp*
apply (*rule ndesign-refine-intro*)
apply (*rel-simp*)
apply (*metis ext*)
apply (*rel-simp*)
by *presburger*
qed

lemma *SimBlock-FBlock-seq-comp′* [*simblock-healthy*]:
assumes *s1*: *SimBlock m1 n1* (*FBlock* (*p1*) *m1 n1 f*)
assumes *s2*: *SimBlock n1 n2* (*FBlock* (*p2*) *n1 n2 g*)

assumes *s3*: $\forall\, x\ n.\ (p1\ x\ n) \longrightarrow (p2\ o\ f)\ x\ n$
shows *SimBlock m1 n2* (*FBlock* ($\lambda x\ n.\ p1\ x\ n \wedge length(x\ n) = m1$) *m1 n1 f* ; ;
FBlock ($\lambda x\ n.\ p2\ x\ n \wedge length(x\ n) = n1$) *n1 n2 g*)
apply (*simp add*: *s1 s2 FBlock-seq-comp′*)
apply (*rule SimBlock-FBlock′*)
proof −
obtain $inouts_v$::*nat* ⇒ *real list* **where** *P*: $\forall\, na.\ length(inouts_v\ na) = m1 \wedge p1\ inouts_v\ na$
using *list-len-avail s1 SimBlock-FBlock-p* **by** *metis*
show $\exists\, inouts_v.$
$(\forall\, x.\ p1\ inouts_v\ x \wedge p2\ (f\ inouts_v)\ x \wedge length(inouts_v\ x) = m1) \wedge$
$(\exists\, inouts_v'.\ \forall\, x.\ length(inouts_v'\ x) = n2 \wedge length(inouts_v\ x) = m1 \wedge (g \circ f)\ inouts_v\ x = inouts_v'$
$x)$
apply (*rule-tac* $x = inouts_v$ **in** *exI*)
apply (*rule conjI*)
using *P s3* **apply** *auto*[*1*]
apply (*rule-tac* $x = (g \circ f)\ inouts_v$ **in** *exI*)
using *P assms*(*2*) *SimBlock-FBlock-fn′ s1* **by** *auto*
next
show $\forall\, x\ na.\ length(x\ na) = m1 \longrightarrow length((g \circ f)\ x\ na) = n2$
using *SimBlock-FBlock-fn′ assms*(*2*) *s1* **by** *auto*
qed

B.4.3 Parallel Composition

B.4.3.1 *mergeB* *ThreeWayMerge′*: similar to *ThreeWayMerge*, but it merges 1 and 2 firstly and then merges 0. Instead, *ThreeWayMerge* merges 0 and 1 firstly, then merges 2.

definition *ThreeWayMerge′* :: ′α *merge* ⇒ ((′α, ′α, (′α, ′α, ′α) *mrg*) *mrg*, ′α) *urel* (**M***30′*(-′)) **where**
[*upred-defs*]: *ThreeWayMerge′ M* = $((\$0{-}\mathbf{v}' =_u \$0{-}\mathbf{v} \wedge \$\mathbf{v}_<' =_u \$\mathbf{v}_<) \wedge (\$0{-}\mathbf{v}' =_u \$1{-}0{-}\mathbf{v} \wedge \$1{-}\mathbf{v}' =_u \$1{-}1{-}\mathbf{v} \wedge \$\mathbf{v}_<' =_u \$\mathbf{v}_<)$; ; *M* ; ; *U1*) ; ; *M*

mergeB is associative which means the order of merges applied to 0, 1 and 2 does not matter as long as 0, 1, and 2 are merged in the same order. In other word, M(M(0,1), 2) = M(0, M(1, 2))

lemma *mergeB-assoc*: *ThreeWayMerge* (*mergeB*) = *ThreeWayMerge′* (*mergeB*)
apply (*simp add*: *ThreeWayMerge-def ThreeWayMerge′-def mergeB-def*)
apply (*rel-auto*)
apply (*rename-tac* $inouts_v0\ ok_v0\ inouts_v1\ ok_v1\ inouts_v2\ ok_v2\ inouts_v3\ inouts_v4\ inouts_v5\ inouts_v6$ $inouts_v7$)
apply (*rule-tac* $x = (ok_v1 \wedge ok_v2)$ **in** *exI*)
apply (*rule-tac* $x = \lambda\ na.\ (inouts_v2\ na \bullet inouts_v3\ na)$ **in** *exI*)
apply (*simp*)

apply (*rule-tac* $x = \lambda$ *na*. ($inouts_v 2$ *na* $\bullet$ $inouts_v 3$ *na*) **in** *exI*)
apply (*simp*)
apply (*rename-tac* $inouts_v 0$ $ok_v 0$ $inouts_v 1$ $ok_v 1$ $inouts_v 2$ $ok_v 2$ $inouts_v 3$ $inouts_v 4$ $inouts_v 5$ $inouts_v 6$)
apply (*rule-tac* $x = inouts_v 0$ **in** *exI*)
apply (*rule-tac* $x = (ok_v 0 \wedge ok_v 1)$ **in** *exI*)
apply (*rule-tac* $x = \lambda$ *na*. ($inouts_v 1$ *na* $\bullet$ $inouts_v 2$ *na*) **in** *exI*)
apply (*simp*)
apply (*rule-tac* $x = \lambda$ *na*. ($inouts_v 1$ *na* $\bullet$ $inouts_v 2$ *na*) **in** *exI*)
apply (*simp*)
done

B.4.3.2 *sim-paralell* **lemma** *SimParallel-form*:
assumes *s1*: *SimBlock m1 n1 B1*
assumes *s2*: *SimBlock m2 n2 B2*
shows($B1 \parallel_B B2$) =
$(\exists\ (ok_0, ok_1, inouts_0, inouts_1) \cdot$
$(((takem\ (m1{+}m2)\ (m1))\ ;;\ B1)[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm\ (m1{+}m2)\ (m2))\ ;;\ B2)[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
(**is** *?lhs* = *?rhs*)
proof –
have *s3*: *inps B1* = *m1*
using *s1* **by** (*simp add*: *inps-outps*)
have *s4*: *inps B2* = *m2*
using *s2* **by** (*simp add*: *inps-outps*)
show *?thesis*
apply (*simp add*: *sim-parallel-def*)
apply (*simp add*: *s3 s4 mergeB-def*)
apply (*simp add*: *par-by-merge-alt-def*, *rel-auto*)
apply (*rename-tac* ok_v $inouts_v{}'$ $inouts_v 2$ $inouts_v 3$ $ok_v 3$ $inouts_v 4$ $ok_v 4$ $ok_v 5$ $inouts_v 5$ $inouts_v 6$ $ok_v 6$ $inouts_v 7$)
apply *blast*
by *blast*
qed

lemma *SimBlock-parallel-pre-true* [*simblock-healthy*]:
assumes *s1*: *SimBlock m1 n1* ($true \vdash_n Q1$)
assumes *s2*: *SimBlock m2 n2* ($true \vdash_n Q2$)
shows *SimBlock* (*m1+m2*) (*n1+n2*) (($true \vdash_n Q1$) $\parallel_B$ ($true \vdash_n Q2$))
proof –
— 1. Simplify the parallel operation
have *1*: (($true \vdash_n Q1$) $\parallel_B$ ($true \vdash_n Q2$)) =
$(\exists\ (ok_0, ok_1, inouts_0, inouts_1) \cdot$
$(((takem\ (m1{+}m2)\ (m1))\ ;;\ (true \vdash_n Q1))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm\ (m1{+}m2)\ (m2))\ ;;\ (true \vdash_n Q2))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
using *SimParallel-form s1 s2* **by** *auto*
— 2. Get some basic facts from assumptions
from *s1* **have** $Q1 \neq false$
by (*simp add*: *SimBlock-def*)
then have *Q1-not-false*: $\exists\, inouts_v\ inouts_v{}'.\ [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v{}'|\!))$
by (*rel-simp*)
from *s2* **have** $Q2 \neq false$

by (*simp add*: *SimBlock-def*)
then have *Q2-not-false*: $\exists$ *inouts*$_v$ *inouts*$_v$*'*. $[\![Q2]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v|$), ($|$*inouts*$_v$ = *inouts*$_v'|$))
by (*rel-simp*)
from *s1* **have** (($\forall$ *na* $\cdot$ $\#_u$(&*inouts*(«*na*»)$_a$) $=_u$ «*m1*») $\sqsubseteq$ *Dom*(*PrePost*((*true* $\vdash_n$ *Q1*))))
by (*simp add*: *SimBlock-def*)
then have *ref-m1*: $\forall$ *inouts*$_v$ *inouts*$_v$*'* *x*. $[\![Q1]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v|$), ($|$*inouts*$_v$ = *inouts*$_v'|$)) $\longrightarrow$ *length*(*inouts*$_v$ *x*) = *m1*
by (*rel-simp*)
from *s2* **have** (($\forall$ *na* $\cdot$ $\#_u$(&*inouts*(«*na*»)$_a$) $=_u$ «*m2*») $\sqsubseteq$ *Dom*(*PrePost*((*true* $\vdash_n$ *Q2*))))
by (*simp add*: *SimBlock-def*)
then have *ref-m2*: $\forall$ *inouts*$_v$ *inouts*$_v$*'* *x*. $[\![Q2]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v|$), ($|$*inouts*$_v$ = *inouts*$_v'|$)) $\longrightarrow$ *length*(*inouts*$_v$ *x*) = *m2*
by (*rel-simp*)
have (($\forall$ *na* $\cdot$ $\#_u$(&*inouts*(«*na*»)$_a$) $=_u$ «*n1*») $\sqsubseteq$ *Ran*(*PrePost*((*true* $\vdash_n$ *Q1*))))
using *SimBlock-def s1* **by** *auto*
then have *ref-n1*: $\forall$ *inouts*$_v$ *inouts*$_v$*'* *x*. $[\![Q1]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v'|$), ($|$*inouts*$_v$ = *inouts*$_v|$)) $\longrightarrow$ *length*(*inouts*$_v$ *x*) = *n1*
by (*rel-simp*)
have (($\forall$ *na* $\cdot$ $\#_u$(&*inouts*(«*na*»)$_a$) $=_u$ «*n2*») $\sqsubseteq$ *Ran*(*PrePost*((*true* $\vdash_n$ *Q2*))))
using *SimBlock-def s2* **by** *auto*
then have *ref-n2*: $\forall$ *inouts*$_v$ *inouts*$_v$*'* *x*. $[\![Q2]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v'|$), ($|$*inouts*$_v$ = *inouts*$_v|$)) $\longrightarrow$ *length*(*inouts*$_v$ *x*) = *n2*
by (*rel-simp*)
— Subgoal 1 for *SimBlock-def*
have *c1*: *PrePost*((*true* $\vdash_n$ *Q1*) $\|_B$ (*true* $\vdash_n$ *Q2*)) $\neq$ *false*
apply (*simp add*: *1*)
apply (*simp add*: *sim-blocks*)
apply (*rel-auto*)
proof −
obtain *inouts*$_v$*1* **and** *inouts*$_v$*'1* **and** *inouts*$_v$*2* **and** *inouts*$_v$*'2*
where *P1*: $[\![Q1]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v$*1*$|$), ($|$*inouts*$_v$ = *inouts*$_v$*'1*$|$))
and *P2*: $[\![Q2]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v$*2*$|$), ($|$*inouts*$_v$ = *inouts*$_v$*'2*$|$))
using *Q1-not-false Q2-not-false* **by** *blast*
show $\exists$ *inouts*$_v$ *inouts*$_v$*'*.
($\forall$ *a aa ab*.
($\exists$ *ok*$_v$. *ok*$_v$ $\wedge$
($\exists$ *inouts*$_v$*'*.
($\forall$ *x*. (*m1* = *0* $\longrightarrow$ *length*(*inouts*$_v$ *x*) = *m2* $\wedge$ *inouts*$_v$*'* *x* = []) $\wedge$
(*0* < *m1* $\longrightarrow$
length(*inouts*$_v$ *x*) = *m1* + *m2* $\wedge$
length(*inouts*$_v$*'* *x*) = *m1* $\wedge$ *take m1* (*inouts*$_v$ *x*) = *inouts*$_v$*'* *x*)) $\wedge$
(*ok*$_v$ $\longrightarrow$ *a* $\wedge$ $[\![Q1]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v'|$), ($|$*inouts*$_v$ = *ab*$|$))))) $\longrightarrow$
($\forall$ *b*. ($\exists$ *ok*$_v$. *ok*$_v$ $\wedge$
($\exists$ *inouts*$_v$*'*.
($\forall$ *x*. (*m2* = *0* $\longrightarrow$ *length*(*inouts*$_v$ *x*) = *m1* $\wedge$ *inouts*$_v$*'* *x* = []) $\wedge$
(*0* < *m2* $\longrightarrow$
length(*inouts*$_v$ *x*) = *m1* + *m2* $\wedge$
length(*inouts*$_v$*'* *x*) = *m2* $\wedge$ *drop m1* (*inouts*$_v$ *x*) = *inouts*$_v$*'* *x*)) $\wedge$
(*ok*$_v$ $\longrightarrow$ *aa* $\wedge$ $[\![Q2]\!]_e$ (($|$*inouts*$_v$ = *inouts*$_v'|$), ($|$*inouts*$_v$ = *b*$|$))))) $\longrightarrow$
($\exists$ *x*. $\neg$ *inouts*$_v$*'* *x* = *ab x* $\bullet$ *b x*) $\vee$ *a* $\wedge$ *aa*)) $\wedge$
($\exists$ *a aa*. ($\exists$ *ok*$_v$. *ok*$_v$ $\wedge$
($\exists$ *inouts*$_v$*'*.
($\forall$ *x*. (*m1* = *0* $\longrightarrow$ *length*(*inouts*$_v$ *x*) = *m2* $\wedge$ *inouts*$_v$*'* *x* = []) $\wedge$
(*0* < *m1* $\longrightarrow$
length(*inouts*$_v$ *x*) = *m1* + *m2* $\wedge$

$length(inouts_v{}' \ x) = m1 \wedge take \ m1 \ (inouts_v \ x) = inouts_v{}' \ x)) \wedge$
$(ok_v \longrightarrow [\![Q1]\!]_e \ ((\!|inouts_v = inouts_v{}'|\!), (\!|inouts_v = aa|\!))))) \wedge$
$(\exists \, b. \ (\exists \, ok_v. \ ok_v \wedge$
$(\exists \, inouts_v{}'.$
$(\forall \, x. \ (m2 = 0 \longrightarrow length(inouts_v \ x) = m1 \wedge inouts_v{}' \ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v \ x) = m1 + m2 \wedge$
$length(inouts_v{}' \ x) = m2 \wedge drop \ m1 \ (inouts_v \ x) = inouts_v{}' \ x)) \wedge$
$(ok_v \longrightarrow a \wedge [\![Q2]\!]_e \ ((\!|inouts_v = inouts_v{}'|\!), (\!|inouts_v = b|\!))))) \wedge$
$(\forall \, x. \ inouts_v{}' \ x = aa \ x \bullet b \ x) \wedge a))$
apply (*rule-tac* $x = \lambda na. \ inouts_v 1 \ na \bullet inouts_v 2 \ na$ **in** *exI*)
apply (*rule-tac* $x = \lambda na. \ inouts_v{}'1 \ na \bullet inouts_v{}'2 \ na$ **in** *exI*)
apply (*rule conjI*)
apply *blast*
apply (*rule-tac* $x = True$ **in** *exI*)
apply (*rule-tac* $x = \lambda na. \ inouts_v{}'1 \ na$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = True$ **in** *exI*)
apply (*simp*)
apply (*rule-tac* $x = \lambda na. \ inouts_v 1 \ na$ **in** *exI*)
using *P1 P2 ref-m1 ref-m2* **apply** *fastforce*
apply (*rule-tac* $x = \lambda na. \ inouts_v{}'2 \ na$ **in** *exI*)
apply (*simp*)
apply (*rule-tac* $x = True$ **in** *exI*)
apply (*simp*)
apply (*rule-tac* $x = \lambda na. \ inouts_v 2 \ na$ **in** *exI*)
using *P1 P2 ref-m1 ref-m2* **by** *force*
qed
— Subgoal 2 for *SimBlock-def*
have *c2*: $((\forall \ na \cdot \#_u(\&inouts(«na»)_a) =_u «m1+m2») \sqsubseteq Dom(PrePost((true \vdash_n Q1) \parallel_B (true \vdash_n Q2))))$
apply (*simp add*: *1*)
apply (*simp add*: *sim-blocks*)
apply (*rel-simp*)
using *assms*
by (*metis add.right-neutral not-gr-zero*)
— Subgoal 3 for *SimBlock-def*
have *c3*: $((\forall \ na \cdot \#_u(\&inouts(«na»)_a) =_u «n1+n2») \sqsubseteq Ran(PrePost((true \vdash_n Q1) \parallel_B (true \vdash_n Q2))))$
apply (*simp add*: *1*)
apply (*simp add*: *sim-blocks*)
apply (*rel-simp*)
by (*simp add*: *ref-n1 ref-n2*)

from *c1 c2 c3* **show** *?thesis*
apply (*simp add*: *SimBlock-def*)
done
qed

Parallel composition of two SimBlocks (provided that the preconditions of both are condition) are still SimBlock.

lemma *SimBlock-parallel* [*simblock-healthy*]:
assumes *s1*: *SimBlock m1 n1* $(P1 \vdash_n Q1)$
assumes *s2*: *SimBlock m2 n2* $(P2 \vdash_n Q2)$
shows *SimBlock* $(m1+m2)$ $(n1+n2)$ $((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2))$

proof −
  have *pform*: $((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2)) =$
    $(\exists\ (ok_0,\ ok_1,\ inouts_0,\ inouts_1) \cdot$
      $(((takem\ (m1+m2)\ (m1))\ ;;\ (P1 \vdash_n Q1))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
      $((dropm\ (m1+m2)\ (m2))\ ;;\ (P2 \vdash_n Q2))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
      $(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
      $(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
   using *SimParallel-form s1 s2* **by** *auto*
  — Subgoal 1 for *SimBlock-def*
  have *c1*: $PrePost((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2)) \neq false$
   apply (*simp add*: *pform*)
   apply (*simp add*: *sim-blocks*)
   apply (*rel-auto*)
   proof −
    obtain $inouts_v1{::}nat \Rightarrow real\ list$ **and** $inouts_v'1{::}nat \Rightarrow real\ list$ **and**
      $inouts_v2{::}nat \Rightarrow real\ list$ **and** $inouts_v'2{::}nat \Rightarrow real\ list$ **where**
     P1: $[\![P1]\!]_e\ ((\!|inouts_v = inouts_v1|\!))$ **and**
     Q1: $[\![Q1]\!]_e\ ((\!|inouts_v = inouts_v1|\!), (\!|inouts_v = inouts_v'1|\!))$ **and**
     P2: $[\![P2]\!]_e\ ((\!|inouts_v = inouts_v2|\!))$ **and**
     Q2: $[\![Q2]\!]_e\ ((\!|inouts_v = inouts_v2|\!), (\!|inouts_v = inouts_v'2|\!))$
      using *s1 s2 SimBlock-implies-not-PQ'*
      by *blast*
    have *inps1*: $length(inouts_v1\ na) = m1$
      using *P1 Q1 SimBlock-implies-mP s1* **by** *blast*
    have *inps2*: $length(inouts_v2\ na) = m2$
      using *P2 Q2 SimBlock-implies-mP s2* **by** *blast*
    have *outps1*: $length(inouts_v'1\ na) = n1$
      using *P1 Q1 SimBlock-implies-Qn s1* **by** *blast*
    have *outps2*: $length(inouts_v'2\ na) = n2$
      using *P2 Q2 SimBlock-implies-Qn s2* **by** *blast*
    show $\exists\, inouts_v\ inouts_v'.$
     $(\forall\, a\ aa\ ab.$
       $(\exists\, ok_v.\ ok_v \wedge$
         $(\exists\, inouts_v'.$
           $(\forall\, x.\ (m1 = 0 \longrightarrow length(inouts_v\ x) = m2 \wedge inouts_v'\ x = []) \wedge$
             $(0 < m1 \longrightarrow$
              $length(inouts_v\ x) = m1 + m2 \wedge$
              $length(inouts_v'\ x) = m1 \wedge take\ m1\ (inouts_v\ x) = inouts_v'\ x)) \wedge$
           $(ok_v \wedge [\![P1]\!]_e\ (\!|inouts_v = inouts_v'|\!) \longrightarrow$
            $a \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = ab|\!))))) \longrightarrow$
       $(\forall\, b.\ (\exists\, ok_v.\ ok_v \wedge$
           $(\exists\, inouts_v'.$
             $(\forall\, x.\ (m2 = 0 \longrightarrow length(inouts_v\ x) = m1 \wedge inouts_v'\ x = []) \wedge$
               $(0 < m2 \longrightarrow$
                $length(inouts_v\ x) = m1 + m2 \wedge$
                $length(inouts_v'\ x) = m2 \wedge drop\ m1\ (inouts_v\ x) = inouts_v'\ x)) \wedge$
             $(ok_v \wedge [\![P2]\!]_e\ (\!|inouts_v = inouts_v'|\!) \longrightarrow$
              $aa \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = b|\!))))) \longrightarrow$
         $(\exists\, x.\ \neg\ inouts_v'\ x = ab\ x \bullet b\ x) \vee a \wedge aa)) \wedge$
     $(\exists\, a\ aa.\ (\exists\, ok_v.\ ok_v \wedge$
           $(\exists\, inouts_v'.$
             $(\forall\, x.\ (m1 = 0 \longrightarrow length(inouts_v\ x) = m2 \wedge inouts_v'\ x = []) \wedge$
               $(0 < m1 \longrightarrow$
                $length(inouts_v\ x) = m1 + m2 \wedge$
                $length(inouts_v'\ x) = m1 \wedge take\ m1\ (inouts_v\ x) = inouts_v'\ x)) \wedge$

$(ok_v \wedge [\![P1]\!]_e\ (\!|inouts_v = inouts_v'|\!) \longrightarrow$
$[\![Q1]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = aa|\!))))) \wedge$
$(\exists\, b.\ (\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v'.$
$(\forall\, x.\ (m2 = 0 \longrightarrow length(inouts_v\ x) = m1 \wedge inouts_v'\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v\ x) = m1 + m2 \wedge$
$length(inouts_v'\ x) = m2 \wedge drop\ m1\ (inouts_v\ x) = inouts_v'\ x)) \wedge$
$(ok_v \wedge [\![P2]\!]_e\ (\!|inouts_v = inouts_v'|\!) \longrightarrow$
$a \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = b|\!))))) \wedge$
$(\forall\, x.\ inouts_v'\ x = aa\ x \bullet b\ x) \wedge a))$
apply (*rule-tac* $x = \lambda na\ .\ (inouts_v 1\ na\ \bullet inouts_v 2\ na)$ **in** *exI*)
apply (*rule-tac* $x = \lambda na\ .\ (inouts_v'1\ na\ \bullet inouts_v'2\ na)$ **in** *exI*)
apply (*rule conjI*)
apply (*rule allI*)+
apply (*simp*)
apply (*rule impI*)
apply (*rule allI*)+
apply (*rule impI*)
proof −
fix $ok_v 1$ **and** $ok_v 2$ **and** $inouts_v 1'::nat \Rightarrow real\ list$ **and** $inouts_v 2'::nat \Rightarrow real\ list$
assume *a1*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v'.$
$(\forall\, x.\ (m1 = 0 \longrightarrow length(inouts_v 1\ x) + length(inouts_v 2\ x) = m2 \wedge inouts_v'\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v'\ x) = m1 \wedge$
$take\ m1\ (inouts_v 1\ x) \bullet take\ (m1 - length(inouts_v 1\ x))\ (inouts_v 2\ x) =$
$inouts_v'\ x)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v'|\!)) \longrightarrow$
$ok_v 1 \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = inouts_v 1'|\!))))$
assume *a2*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v'.$
$(\forall\, x.\ (m2 = 0 \longrightarrow length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 \wedge inouts_v'\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v'\ x) = m2 \wedge$
$drop\ m1\ (inouts_v 1\ x) \bullet drop\ (m1 - length(inouts_v 1\ x))\ (inouts_v 2\ x) =$
$inouts_v'\ x)) \wedge$
$(ok_v \wedge [\![P2]\!]_e\ ((\!|inouts_v = inouts_v'|\!)) \longrightarrow$
$ok_v 2 \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = inouts_v 2'|\!))))$
from *a1* **have** *1*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v'.$
$(\forall\, x.\ (m1 = 0 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m2 \wedge$
$inouts_v 1\ x = [] \wedge$
$inouts_v'\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v'\ x) = m1 \wedge$
$inouts_v 1\ x = inouts_v'\ x)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v'|\!)) \longrightarrow$
$ok_v 1 \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = inouts_v 1'|\!))))$
using *inps1 P1 Q1 SimBlock-implies-mP s1*
by (*smt append-take-drop-id cancel-comm-monoid-add-class.diff-cancel length-0-conv*

length-drop take-eq-Nil)
then have *2*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v{}'.$
$(\forall\, x.\ inouts_v 1\ x = inouts_v{}'\ x \wedge$
$(m1 = 0 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m2 \wedge$
$inouts_v 1\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 1\ x) = m1)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v{}'|\!)) \longrightarrow$
$ok_v 1 \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v{}'|\!),\ (\!|inouts_v = inouts_v 1'|\!))))$
by (*metis (full-types) inps1 length-0-conv length-greater-0-conv*)
then have *3*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v{}'.$
$(\forall\, x.\ inouts_v 1\ x = inouts_v{}'\ x) \wedge$
$(\forall\, x.\ (m1 = 0 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m2 \wedge$
$inouts_v 1\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 1\ x) = m1)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v{}'|\!)) \longrightarrow$
$ok_v 1 \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v{}'|\!),\ (\!|inouts_v = inouts_v 1'|\!))))$
by *smt*
then have *4*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v{}'.$
$(\forall\, x.\ (m1 = 0 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m2 \wedge$
$inouts_v 1\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 1\ x) = m1)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!)) \longrightarrow$
$ok_v 1 \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!),\ (\!|inouts_v = inouts_v 1'|\!))))$
by (*metis 2 3 append-Nil ext length-append less-not-refl neq0-conv*)
then have *5*: $\exists\, ok_v.\ ok_v \wedge$
$(\forall\, x.\ (m1 = 0 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m2 \wedge$
$inouts_v 1\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 1\ x) = m1)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!)) \longrightarrow$
$ok_v 1 \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!),\ (\!|inouts_v = inouts_v 1'|\!)))$
by (*simp*)
then have *6*:
$(\forall\, x.\ (m1 = 0 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m2 \wedge$
$inouts_v 1\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 1\ x) = m1)) \wedge$
$([\![P1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!)) \longrightarrow$
$ok_v 1 \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!),\ (\!|inouts_v = inouts_v 1'|\!)))$

by *blast*
then have *7*: $(\llbracket P1 \rrbracket_e\ (\lparr inouts_v = inouts_v 1 \rparr) \longrightarrow$
$ok_v 1 \wedge \llbracket Q1 \rrbracket_e\ (\lparr inouts_v = inouts_v 1 \rparr, \lparr inouts_v = inouts_v 1' \rparr))$
by *simp*
from *a2* **have** *11*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v'.$
$(\forall\, x.\ (m2 = 0 \longrightarrow length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 \wedge$
$inouts_v'\ x = [] \wedge inouts_v 2\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v'\ x) = m2 \wedge$
$(inouts_v 2\ x) = inouts_v'\ x)) \wedge$
$(ok_v \wedge \llbracket P2 \rrbracket_e\ (\lparr inouts_v = inouts_v' \rparr) \longrightarrow$
$ok_v 2 \wedge \llbracket Q2 \rrbracket_e\ (\lparr inouts_v = inouts_v' \rparr, \lparr inouts_v = inouts_v 2' \rparr))))$
using *inps1 P2 Q2 SimBlock-implies-mP s2*
by (*smt P1 Q1 append-self-conv2 cancel-comm-monoid-add-class.diff-cancel drop-0 drop-eq-Nil order-refl s1*)
then have *12*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v'.$
$(\forall\, x.\ inouts_v 2\ x = inouts_v'\ x \wedge$
$(m2 = 0 \longrightarrow length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 \wedge$
$inouts_v 2\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 2\ x) = m2)) \wedge$
$(ok_v \wedge \llbracket P2 \rrbracket_e\ (\lparr inouts_v = inouts_v' \rparr) \longrightarrow$
$ok_v 2 \wedge \llbracket Q2 \rrbracket_e\ (\lparr inouts_v = inouts_v' \rparr, \lparr inouts_v = inouts_v 2' \rparr))))$
by (*metis* (*full-types*) *inps2 length-0-conv length-greater-0-conv*)
then have *13*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v'.$
$(\forall\, x.\ inouts_v 2\ x = inouts_v'\ x) \wedge$
$(\forall\, x.\ (m2 = 0 \longrightarrow length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 \wedge$
$inouts_v 2\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 2\ x) = m2)) \wedge$
$(ok_v \wedge \llbracket P2 \rrbracket_e\ (\lparr inouts_v = inouts_v' \rparr) \longrightarrow$
$ok_v 2 \wedge \llbracket Q2 \rrbracket_e\ (\lparr inouts_v = inouts_v' \rparr, \lparr inouts_v = inouts_v 2' \rparr))))$
by *smt*
then have *14*: $\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v'.$
$(\forall\, x.\ (m2 = 0 \longrightarrow length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 \wedge$
$inouts_v 2\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 2\ x) = m2)) \wedge$
$(ok_v \wedge \llbracket P2 \rrbracket_e\ (\lparr inouts_v = inouts_v 2 \rparr) \longrightarrow$
$ok_v 2 \wedge \llbracket Q2 \rrbracket_e\ (\lparr inouts_v = inouts_v 2 \rparr, \lparr inouts_v = inouts_v 2' \rparr))))$
by (*metis 12 13 append-Nil ext length-append less-not-refl neq0-conv*)
then have *15*: $\exists\, ok_v.\ ok_v \wedge$
$(\forall\, x.\ (m2 = 0 \longrightarrow length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 \wedge$
$inouts_v 2\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 2\ x) = m2)) \wedge$

$(ok_v \wedge [\![P2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!)) \longrightarrow$
$ok_v 2 \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!), (\!|inouts_v = inouts_v 2'|\!)))$
by (*simp*)
then have *16*:
$(\forall x.\ (m2 = 0 \longrightarrow length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 \wedge$
$inouts_v 2\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v 1\ x) + length(inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v 2\ x) = m2)) \wedge$
$(\ [\![P2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!)) \longrightarrow$
$ok_v 2 \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!), (\!|inouts_v = inouts_v 2'|\!)))$
by *blast*
then have *17*: $(\ [\![P2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!)) \longrightarrow$
$ok_v 2 \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!), (\!|inouts_v = inouts_v 2'|\!)))$
by *simp*
show $(\exists x.\ \neg\ inouts_v'1\ x \bullet inouts_v'2\ x = inouts_v 1'\ x \bullet inouts_v 2'\ x) \vee ok_v 1 \wedge ok_v 2$
proof (*rule ccontr*)
assume *aa*: $\neg\ ((\exists x.\ \neg\ inouts_v'1\ x \bullet inouts_v'2\ x = inouts_v 1'\ x \bullet inouts_v 2'\ x) \vee ok_v 1 \wedge$
$ok_v 2)$
from *aa* **have** *b1*: $(\forall x.\ inouts_v'1\ x \bullet inouts_v'2\ x = inouts_v 1'\ x \bullet inouts_v 2'\ x) \wedge (\neg\ ok_v 1$
$\vee \neg\ ok_v 2)$
by (*simp*)
from *b1* **have** *b2*: $(\forall x.\ inouts_v'1\ x \bullet inouts_v'2\ x = inouts_v 1'\ x \bullet inouts_v 2'\ x)$
by (*simp*)
from *b1* **have** *b3*: $(\neg\ ok_v 1 \vee \neg\ ok_v 2)$
by (*simp*)
from *b3 7 17* **have** *b4*:
$\neg\ [\![P2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!)) \vee$
$\neg\ [\![P1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!))$
by *blast*
from *s1* **have** *b5*: $[\![P1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!))$
using *P1 SimBlock-implies-not-P-cond*
by *blast*
from *s2* **have** *b6*: $[\![P2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!))$
using *P2 SimBlock-implies-not-P-cond* **by** *blast*
show *False*
using *b4 b5 b6* **by** (*auto*)
qed
next
show $\exists a\ aa.\ (\exists ok_v.\ ok_v \wedge$
$(\exists inouts_v'.$
$(\forall x.\ (m1 = 0 \longrightarrow length(inouts_v 1\ x \bullet inouts_v 2\ x) = m2 \wedge inouts_v'\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v 1\ x \bullet inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v'\ x) = m1 \wedge take\ m1\ (inouts_v 1\ x \bullet inouts_v 2\ x) = inouts_v'\ x)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v'|\!)) \longrightarrow$
$[\![Q1]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = aa|\!))))) \wedge$
$(\exists b.\ (\exists ok_v.\ ok_v \wedge$
$(\exists inouts_v'.$
$(\forall x.\ (m2 = 0 \longrightarrow length(inouts_v 1\ x \bullet inouts_v 2\ x) = m1 \wedge inouts_v'\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v 1\ x \bullet inouts_v 2\ x) = m1 + m2 \wedge$
$length(inouts_v'\ x) = m2 \wedge drop\ m1\ (inouts_v 1\ x \bullet inouts_v 2\ x) = inouts_v'\ x)) \wedge$
$(ok_v \wedge [\![P2]\!]_e\ ((\!|inouts_v = inouts_v'|\!)) \longrightarrow$
$a \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v'|\!), (\!|inouts_v = b|\!))))) \wedge$

$(\forall x.\ inouts_v{}'1\ x \bullet inouts_v{}'2\ x = aa\ x \bullet b\ x) \wedge a)$
apply (*rule-tac* $x = True$ **in** *exI*)
apply (*rule-tac* $x = inouts_v{}'1$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = True$ **in** *exI*, *simp*)
apply (*rule-tac* $x = inouts_v 1$ **in** *exI*)
using *P1 P2 Q1 Q2 SimBlock-implies-mP s1 s2*
apply (*smt add-eq-self-zero append.right-neutral cancel-ab-semigroup-add-class.add-diff-cancel-left′ order-refl sum-eq-sum-conv take-all take-eq-Nil*)
apply (*rule-tac* $x = inouts_v{}'2$ **in** *exI*, *simp*)
apply (*rule-tac* $x = True$ **in** *exI*, *simp*)
apply (*rule-tac* $x = inouts_v 2$ **in** *exI*)
using *P1 P2 Q1 Q2 SimBlock-implies-mP s1 s2*
by (*smt add-eq-self-zero append-eq-append-conv-if cancel-ab-semigroup-add-class.add-diff-cancel-left′ drop-0 list-exhaust-size-eq0 sum-eq-sum-conv*)
qed
qed
— Subgoal 2 for *SimBlock-def*
have *c2*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «m1+m2») \sqsubseteq Dom(PrePost((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2))))$
apply (*simp add*: *pform*)
apply (*simp add*: *sim-blocks*)
apply (*rel-simp*)
using *assms*
by (*metis add.right-neutral not-gr-zero*)
— Subgoal 3 for *SimBlock-def*
have *c3*: $((\forall\ na \cdot \#_u(\&inouts(«na»)_a) =_u «n1+n2») \sqsubseteq Ran(PrePost((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2))))$
apply (*simp add*: *pform*)
apply (*simp add*: *sim-blocks*)
apply (*rel-simp*)
apply (*rename-tac* $inouts_v{}'\ inouts_v\ n\ ok_v q1\ ok_v q2\ inouts_v 1'\ ok_v\ inouts_v 2'\ inouts_v 1\ ok_v{}'\ inouts_v 2$)
proof −
fix $inouts_v{}'\ inouts_v\ n\ ok_v q1\ ok_v q2\ inouts_v 1'\ ok_v\ inouts_v 2'\ inouts_v 1\ ok_v{}'\ inouts_v 2$
assume *a1*: $[\![P1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!)) \longrightarrow [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!), (\!|inouts_v = inouts_v 1'|\!))$
assume *a2*: $[\![P2]\!]_e\ (\!|inouts_v = inouts_v 2|\!) \longrightarrow [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!), (\!|inouts_v = inouts_v 2'|\!))$
assume *a3*: $\forall\, a\ aa\ ab.$
$(\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v.$
$(\forall x.\ (m1 = 0 \longrightarrow inouts_v\ x = []) \wedge$
$(0 < m1 \longrightarrow length(inouts_v\ x) = m1 \wedge inouts_v 1\ x = inouts_v\ x)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ (\!|inouts_v = inouts_v|\!) \longrightarrow$
$a \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = ab|\!))))) \longrightarrow$
$(\forall\, b.\ (\exists\, ok_v.\ ok_v \wedge$
$(\exists\, inouts_v.$
$(\forall x.\ (m2 = 0 \longrightarrow inouts_v\ x = []) \wedge$
$(0 < m2 \longrightarrow length(inouts_v\ x) = m2 \wedge inouts_v 2\ x = inouts_v\ x)) \wedge$
$(ok_v \wedge [\![P2]\!]_e\ (\!|inouts_v = inouts_v|\!) \longrightarrow$
$aa \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = b|\!))))) \longrightarrow$
$(\exists\, x.\ \neg\ inouts_v 1'\ x \bullet inouts_v 2'\ x = ab\ x \bullet b\ x) \vee a \wedge aa)$
assume *a4*: $\forall\, x.\ inouts_v{}'\ x = inouts_v 1'\ x \bullet inouts_v 2'\ x$
assume *a5*: $\forall\, x.\ (m1 = 0 \longrightarrow length(inouts_v\ x) = m2 \wedge inouts_v 1\ x = []) \wedge$

$(0 < m1 \longrightarrow length(inouts_v\ x) = m1 + m2 \wedge length(inouts_v 1\ x) = m1 \wedge$
$take\ m1\ (inouts_v\ x) = inouts_v 1\ x)$
assume $a6$: $\forall x.\ (m2 = 0 \longrightarrow length(inouts_v\ x) = m1 \wedge inouts_v 2\ x = []) \wedge$
$(0 < m2 \longrightarrow length(inouts_v\ x) = m1 + m2 \wedge length(inouts_v 2\ x) = m2 \wedge$
$drop\ m1\ (inouts_v\ x) = inouts_v 2\ x)$
from $a5$ **have** 1: $length(inouts_v 1\ na) = m1$
by *blast*
from $a6$ **have** 2: $length(inouts_v 2\ na) = m2$
by *blast*
from $a3$ **have** $(\forall a\ aa\ ab.$
$(\exists ok_v.\ ok_v \wedge$
$(\exists inouts_v.$
$(\forall x.\ (m1 = 0 \longrightarrow inouts_v\ x = []) \wedge$
$(0 < m1 \longrightarrow length(inouts_v\ x) = m1 \wedge inouts_v 1\ x = inouts_v\ x)) \wedge$
$(ok_v \wedge [\![P1]\!]_e\ (\!|inouts_v = inouts_v|\!) \longrightarrow$
$a \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = ab|\!))))) \longrightarrow$
$(\forall b.\ (\exists ok_v.\ ok_v \wedge$
$(\exists inouts_v.$
$(\forall x.\ (m2 = 0 \longrightarrow inouts_v\ x = []) \wedge$
$(0 < m2 \longrightarrow length(inouts_v\ x) = m2 \wedge inouts_v 2\ x = inouts_v\ x)) \wedge$
$(ok_v \wedge [\![P2]\!]_e\ (\!|inouts_v = inouts_v|\!) \longrightarrow$
$aa \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = b|\!))))) \longrightarrow$
$(\exists x.\ \neg\ inouts_v 1'\ x \bullet inouts_v 2'\ x = ab\ x \bullet b\ x) \vee a \wedge aa))$
$\longrightarrow (\forall a\ aa\ ab.$
$([\![P1]\!]_e\ (\!|inouts_v = inouts_v 1|\!) \longrightarrow$
$a \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!), (\!|inouts_v = ab|\!))) \longrightarrow$
$(\forall b.\ ([\![P2]\!]_e\ (\!|inouts_v = inouts_v 2|\!) \longrightarrow$
$aa \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!), (\!|inouts_v = b|\!))) \longrightarrow$
$(\exists x.\ \neg\ inouts_v 1'\ x \bullet inouts_v 2'\ x = ab\ x \bullet b\ x) \vee a \wedge aa))$
apply (*simp*)
apply (*rule allI*)+
apply (*rename-tac* $ok_v q\ inouts_v 1'q\ inouts_v 2'q$)
apply (*rule impI*)
apply (*rule allI*)
apply (*rule impI*)
by (*smt a5 a6 neq0-conv*)
then have $a3'$: $(\forall a\ aa\ ab.$
$([\![P1]\!]_e\ (\!|inouts_v = inouts_v 1|\!) \longrightarrow$
$a \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!), (\!|inouts_v = ab|\!))) \longrightarrow$
$(\forall b.\ ([\![P2]\!]_e\ (\!|inouts_v = inouts_v 2|\!) \longrightarrow$
$aa \wedge [\![Q2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!), (\!|inouts_v = b|\!))) \longrightarrow$
$(\exists x.\ \neg\ inouts_v 1'\ x \bullet inouts_v 2'\ x = ab\ x \bullet b\ x) \vee a \wedge aa))$
using $a3$ **by** *smt*
have $P1$: $[\![P1]\!]_e\ (\!|inouts_v = inouts_v 1|\!)$
using $a3'$ **using** $a2$ **by** *blast*
then have $Q1$: $[\![Q1]\!]_e\ ((\!|inouts_v = inouts_v 1|\!), (\!|inouts_v = inouts_v 1'|\!))$
using $a1$ **by** *auto*
then have $N1$: $length(inouts_v 1'\ n) = n1$
using $P1$ *SimBlock-implies-Qn* $s1$ **by** *blast*
have $P2$: $[\![P2]\!]_e\ (\!|inouts_v = inouts_v 2|\!)$
using $a3'$ **using** $a1$ **by** *blast*
then have $Q2$: $[\![Q2]\!]_e\ ((\!|inouts_v = inouts_v 2|\!), (\!|inouts_v = inouts_v 2'|\!))$
using $a2$ **by** *auto*
then have $N2$: $length(inouts_v 2'\ n) = n2$
using $P2$ *SimBlock-implies-Qn* $s2$ **by** *blast*

show $length(inouts_v 1' \ n) + length(inouts_v 2' \ n) = n1 + n2$
using *N1 N2* **by** *auto*
qed
from *c1 c2 c3* **show** *?thesis*
apply (*simp add*: *SimBlock-def*)
done
qed

lemma *inps-parallel*:
assumes *s1*: *SimBlock m1 n1* $(P1 \vdash_n Q1)$
assumes *s2*: *SimBlock m2 n2* $(P2 \vdash_n Q2)$
shows *inps* $((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2)) = m1 + m2$
using *SimBlock-parallel inps-outps s1 s2* **by** *blast*

lemma *outps-parallel*:
assumes *s1*: *SimBlock m1 n1* $(P1 \vdash_n Q1)$
assumes *s2*: *SimBlock m2 n2* $(P2 \vdash_n Q2)$
shows *outps* $((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2)) = n1 + n2$
using *SimBlock-parallel inps-outps*
using *s1 s2* **by** *blast*

Associativity of parallel composition.

lemma *parallel-ass*:
assumes *s1*: *SimBlock m0 n0* $(P0 \vdash_n Q0)$
assumes *s2*: *SimBlock m1 n1* $(P1 \vdash_n Q1)$
assumes *s3*: *SimBlock m2 n2* $(P2 \vdash_n Q2)$
shows $((P0 \vdash_n Q0) \parallel_B ((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2))) = (((P0 \vdash_n Q0) \parallel_B (P1 \vdash_n Q1)) \parallel_B (P2 \vdash_n Q2))$
(**is** *?lhs* = *?rhs*)
proof −
let $?P12 = \exists \ (ok_1, ok_2, inouts_1, inouts_2) \cdot$
$(((takem \ (m1{+}m2) \ (m1)) \ ;; \ (P1 \vdash_n Q1))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm \ (m1{+}m2) \ (m2)) \ ;; \ (P2 \vdash_n Q2))[\![«ok_2»,«inouts_2»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall \ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´ \ («n»)_a =_u («append» \ («inouts_1 \ n»)_a \ («inouts_2 \ n»)_a))) \wedge$
$(\$ok´ =_u («ok_1» \wedge «ok_2»)))$
have *lhs-12*: $((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2)) = ?P12$
using *SimParallel-form s2 s3* **by** *blast*
have *lhs-12-sim*: *SimBlock* $(m1{+}m2)$ $(n1{+}n2)$ $((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2))$
by (*simp add*: *SimBlock-parallel s2 s3*)
then have *lhs-sim*: *?lhs* =
$(\exists \ (ok_0, ok_{12}, inouts_0, inouts_{12}) \cdot$
$(((takem \ (m0{+}(m1{+}m2)) \ (m0)) \ ;; \ (P0 \vdash_n Q0))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm \ (m0{+}(m1{+}m2)) \ (m1{+}m2)) \ ;; \ ?P12)[\![«ok_{12}»,«inouts_{12}»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall \ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´ \ («n»)_a =_u («append» \ («inouts_0 \ n»)_a \ («inouts_{12} \ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_{12}»))))$
using *lhs-12-sim lhs-12 SimParallel-form s1 s2 s3* **by** *auto*

let $?P01 = \exists \ (ok_0, ok_1, inouts_0, inouts_1) \cdot$
$(((takem \ (m0{+}m1) \ (m0)) \ ;; \ (P0 \vdash_n Q0))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm \ (m0{+}m1) \ (m1)) \ ;; \ (P1 \vdash_n Q1))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall \ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´ \ («n»)_a =_u («append» \ («inouts_0 \ n»)_a \ («inouts_1 \ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»)))$
have *rhs-01*: $((P0 \vdash_n Q0) \parallel_B (P1 \vdash_n Q1)) = ?P01$
using *SimParallel-form s1 s2* **by** *blast*
have *rhs-01-sim*: *SimBlock* $(m0{+}m1)$ $(n0{+}n1)$ $((P0 \vdash_n Q0) \parallel_B (P1 \vdash_n Q1))$

by (*simp add*: *SimBlock-parallel s1 s2*)
then have *rhs-sim*: *?rhs* =
$(\exists\ (ok_{01},\ ok_2,\ inouts_{01},\ inouts_2) \cdot$
$(((takem\ ((m0{+}m1){+}m2)\ (m0{+}m1))\ ;;\ \ ?P01)[\![«ok_{01}»,«inouts_{01}»/\$ok\acute{},\$\mathbf{v}_D{:}inouts\acute{}]\!] \wedge$
$((dropm\ ((m0{+}m1){+}m2)\ (m2))\ ;;\ \ (P2 \vdash_n Q2))[\![«ok_2»,«inouts_2»/\$ok\acute{},\$\mathbf{v}_D{:}inouts\acute{}]\!] \wedge$
$(\forall\ \ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts\acute{}\ («n»)_a =_u («append»\ («inouts_{01}\ n»)_a\ («inouts_2\ n»)_a))) \wedge$
$(\$ok\acute{} =_u («ok_{01}» \wedge «ok_2»))))$
using *rhs-01-sim rhs-01 SimParallel-form s1 s2 s3* **by** *auto*
show *?thesis*
apply (*simp add*: *lhs-sim rhs-sim*)
apply (*simp add*: *sim-blocks*)
apply (*rel-simp*)
apply (*rule iffI*)
— Subgoal 1: lhs –> rhs
apply (*clarify*)
apply (*rename-tac* ok_v $inouts_v$ ok_v' $inouts_v'$ $ok_v'q0$ *aa* $inouts_v'q0$ ok_vp0 $inouts_v'12$ $inouts_vp0$ ok_v12
$inouts_v12$ $ok_v'q1$ $ok_v'q2$ $inouts_v'q1$ ok_vp1 $inouts_v'q2$ $inouts_vp1$ ok_vp2 $inouts_vp2$)
apply (*rule-tac* $x = ok_v'q0 \wedge ok_v'q1$ **in** *exI*)
apply (*rule-tac* $x = ok_v'q2$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ (inouts_v'q0\ na \bullet inouts_v'q1\ na)$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_v$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ (inouts_vp0\ na \bullet inouts_vp1\ na)$ **in** *exI*)
apply (*rule conjI*)
apply (*clarify*)
apply (*smt ab-semigroup-add-class.add-ac*(*1*) *drop-0 gr0I length-append list.size*(*3*)
self-append-conv take-add)
apply (*rule-tac* $x = ok_v'q0$ **in** *exI*)
apply (*rule-tac* $x = ok_v'q1$ **in** *exI*)
apply (*rule-tac* $x = inouts_v'q0$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_vp0$ **in** *exI*)
apply (*rule-tac* $x = inouts_vp0$ **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*metis gr0I length-0-conv*)
apply *blast*
apply (*rule-tac* $x = inouts_v'q1$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_vp1$ **in** *exI*)
apply (*rule-tac* $x = inouts_vp1$ **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*metis append-eq-conv-conj drop-append list.size*(*3*) *neq0-conv*)
apply *blast*
apply *blast*
apply (*rule-tac* $x = inouts_v'q2$ **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*rule-tac* $x = ok_vp2$ **in** *exI*)
apply (*rule-tac* $x = inouts_vp2$ **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*metis add-cancel-left-right drop-drop gr0I semiring-normalization-rules*(*24*))
apply *blast*
apply *auto*[*1*]
— Subgoal 2: rhs –> lhs
apply (*clarify*)

apply (*rename-tac* ok_v $inouts_v$ ok_v' $inouts_v'$ *a* $ok_v'q2$ $inouts_v'01$ ok_v01 $inouts_v'q2$ $inouts_v01$ ok_vp2 $inouts_vp2$
$ok_v'q0$ $ok_v'q1$ $inouts_v'q0$ ok_vp0 $inouts_v'q1$ $inouts_vp0$ ok_vp1 $inouts_vp1$)
apply (*rule-tac* $x = ok_v'q0$ **in** *exI*)
apply (*rule-tac* $x = ok_v'q1 \wedge ok_v'q2$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ (inouts_v'q0\ na)$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_v$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ (inouts_vp0\ na)$ **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*rule impI*)
apply (*rule allI*)
apply (*rule conjI*)
apply (*metis add-cancel-left-left zero-less-iff-neq-zero*)
apply (*metis append.right-neutral append-take-drop-id diff-is-0-eq le-add1 take-0 take-append*)
apply *blast*
apply (*rule-tac* $x = \lambda na.\ (inouts_v'q1\ na \bullet inouts_v'q2\ na)$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_v$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ (inouts_vp1\ na \bullet inouts_vp2\ na)$ **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*rule impI*)
apply (*rule allI*)
apply (*rule conjI*)
apply (*smt add.commute append-take-drop-id drop-drop length-append length-greater-0-conv less-add-same-cancel2 neq0-conv take-drop*)
apply (*rule impI*)
apply (*rule conjI*)
apply (*metis gr-zeroI list.size*(*3*))
apply (*metis* (*no-types*, *hide-lams*) *add.left-neutral append-take-drop-id diff-add-zero drop-0 drop-append neq0-conv plus-list-def zero-list-def*)
apply (*rule-tac* $x = ok_v'q1$ **in** *exI*)
apply (*rule-tac* $x = ok_v'q2$ **in** *exI*)
apply (*rule-tac* $x = inouts_v'q1$ **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*metis gr0I length-0-conv*)
apply (*rule-tac* $x = inouts_v'q2$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_vp2$ **in** *exI*)
apply (*rule-tac* $x = inouts_vp2$ **in** *exI*)
apply (*rule conjI*, *simp*)
apply (*metis append-eq-conv-conj drop-append list.size*(*3*) *neq0-conv*)
apply *blast*
apply *blast*
apply (*rule conjI*, *simp*)
by *blast*
qed

lemma *refinement-implies-r*:
assumes *s1*: $(P1 \vdash_r Q1) \sqsubseteq (P1r \vdash_r Q1r)$
shows $\forall\, ok_v\ inouts_v\ ok_v'\ inouts_v'.$
$(ok_v \wedge [\![P1r]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v'|\!)) \longrightarrow$
$ok_v' \wedge [\![Q1r]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v'|\!))) \longrightarrow$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v'|\!)) \longrightarrow$

$ok_v{}' \wedge [\![Q1]\!]_e$ $((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v{}'|\!)))$
using *s1* **apply** (*rel-simp*)
by *blast*

lemma *refinement-implies*:
assumes *s1*: $(P1 \vdash_n Q1) \sqsubseteq (P1r \vdash_n Q1r)$
shows $\forall\, ok_v\ inouts_v\ ok_v{}'\ inouts_v{}'.$
$(ok_v \wedge [\![P1r]\!]_e\ ((\!|inouts_v = inouts_v|\!)) \longrightarrow$
$ok_v{}' \wedge [\![Q1r]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v{}'|\!))) \longrightarrow$
$(ok_v \wedge [\![P1]\!]_e\ ((\!|inouts_v = inouts_v|\!)) \longrightarrow$
$ok_v{}' \wedge [\![Q1]\!]_e\ ((\!|inouts_v = inouts_v|\!), (\!|inouts_v = inouts_v{}'|\!)))$
using *s1* **apply** (*rel-simp*)
by *blast*

lemma *parallel-mono-r*:
assumes *s1*: *SimBlock m1 n1* $(P1 \vdash_r Q1)$
assumes *s2*: *SimBlock m2 n2* $(P2 \vdash_r Q2)$
assumes *s3*: *SimBlock m1 n1* $(P1r \vdash_r Q1r)$
assumes *s4*: *SimBlock m2 n2* $(P2r \vdash_r Q2r)$
assumes *s5*: $(P1 \vdash_r Q1) \sqsubseteq (P1r \vdash_r Q1r)$
assumes *s6*: $(P2 \vdash_r Q2) \sqsubseteq (P2r \vdash_r Q2r)$
shows $((P1 \vdash_r Q1) \parallel_B (P2 \vdash_r Q2)) \sqsubseteq ((P1r \vdash_r Q1r) \parallel_B (P2r \vdash_r Q2r))$
proof −
have *pform*: $((P1 \vdash_r Q1) \parallel_B (P2 \vdash_r Q2)) =$
$(\exists\ (ok_0, ok_1, inouts_0, inouts_1) \cdot$
$(((takem\ (m1{+}m2)\ (m1))\ ;;\ (P1 \vdash_r Q1))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm\ (m1{+}m2)\ (m2))\ ;;\ (P2 \vdash_r Q2))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
using *SimParallel-form s1 s2* **by** *auto*
have *pform'*: $((P1r \vdash_r Q1r) \parallel_B (P2r \vdash_r Q2r)) =$
$(\exists\ (ok_0, ok_1, inouts_0, inouts_1) \cdot$
$(((takem\ (m1{+}m2)\ (m1))\ ;;\ (P1r \vdash_r Q1r))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm\ (m1{+}m2)\ (m2))\ ;;\ (P2r \vdash_r Q2r))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
using *SimParallel-form s3 s4* **by** *auto*
show *?thesis*
apply (*simp add*: *pform pform'*)
apply (*simp add*: *sim-blocks*)
apply (*rel-simp*)
apply (*rename-tac* ok_v $inouts_v$ $inouts_v{}'$ ok_vq1r ok_vq2r $inouts_v1r'$ ok_vp1r $inouts_v2r'$ $inouts_v1r$ ok_vp2r $inouts_v2r$)
apply (*rule-tac* $x = ok_vq1r$ **in** *exI*)
apply (*rule-tac* $x = ok_vq2r$ **in** *exI*)
apply (*rule-tac* $x = inouts_v1r'$ **in** *exI*)
apply (*simp*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_vp1r$ **in** *exI*, *simp*)
apply (*rule-tac* $x = inouts_v1r$ **in** *exI*)
apply (*rule conjI*)
apply *simp*
using *s5 s1 refinement-implies-r* **apply** (*metis*)
apply (*rule-tac* $x = inouts_v2r'$ **in** *exI*, *simp*)
apply (*rule-tac* $x = ok_vp2r$ **in** *exI*)

apply *simp*
apply (*rule-tac x* = $inouts_v 2r$ **in** *exI*, *simp*)
using *s6 s2 refinement-implies-r* **apply** (*metis*)
done
qed

lemma *parallel-mono*:
assumes *s1*: *SimBlock m1 n1* ($P1 \vdash_n Q1$)
assumes *s2*: *SimBlock m2 n2* ($P2 \vdash_n Q2$)
assumes *s3*: *SimBlock m1 n1* ($P1r \vdash_n Q1r$)
assumes *s4*: *SimBlock m2 n2* ($P2r \vdash_n Q2r$)
assumes *s5*: $(P1 \vdash_n Q1) \sqsubseteq (P1r \vdash_n Q1r)$
assumes *s6*: $(P2 \vdash_n Q2) \sqsubseteq (P2r \vdash_n Q2r)$
shows $((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2)) \sqsubseteq ((P1r \vdash_n Q1r) \parallel_B (P2r \vdash_n Q2r))$
proof −
have *pform*: $((P1 \vdash_n Q1) \parallel_B (P2 \vdash_n Q2)) =$
$(\exists\ (ok_0,\ ok_1,\ inouts_0,\ inouts_1) \cdot$
$(((takem\ (m1+m2)\ (m1))\ ;;\ (P1 \vdash_n Q1))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm\ (m1+m2)\ (m2))\ ;;\ (P2 \vdash_n Q2))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
using *SimParallel-form s1 s2* **by** *auto*
have *pform'*: $((P1r \vdash_n Q1r) \parallel_B (P2r \vdash_n Q2r)) =$
$(\exists\ (ok_0,\ ok_1,\ inouts_0,\ inouts_1) \cdot$
$(((takem\ (m1+m2)\ (m1))\ ;;\ (P1r \vdash_n Q1r))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$((dropm\ (m1+m2)\ (m2))\ ;;\ (P2r \vdash_n Q2r))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!] \wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
using *SimParallel-form s3 s4* **by** *auto*
show *?thesis*
apply (*simp add*: *pform pform'*)
apply (*simp add*: *sim-blocks*)
apply (*rel-simp*)
apply (*rename-tac* ok_v $inouts_v$ $inouts_v'$ $ok_v q1r$ $ok_v q2r$ $inouts_v 1r'$ $ok_v p1r$ $inouts_v 2r'$ $inouts_v 1r$ $ok_v p2r$ $inouts_v 2r$)
apply (*rule-tac x* = $ok_v q1r$ **in** *exI*)
apply (*rule-tac x* = $ok_v q2r$ **in** *exI*)
apply (*rule-tac x* = $inouts_v 1r'$ **in** *exI*)
apply (*simp*)
apply (*rule conjI*)
apply (*rule-tac x* = $ok_v p1r$ **in** *exI*, *simp*)
apply (*rule-tac x* = $inouts_v 1r$ **in** *exI*)
apply (*rule conjI*)
apply *simp*
using *s5 s1 refinement-implies* **apply** (*metis*)
apply (*rule-tac x* = $inouts_v 2r'$ **in** *exI*, *simp*)
apply (*rule-tac x* = $ok_v p2r$ **in** *exI*)
apply *simp*
apply (*rule-tac x* = $inouts_v 2r$ **in** *exI*, *simp*)
using *s6 s2 refinement-implies* **apply** (*metis*)
done
qed

lemma *FBlock-parallel-comp-id*:
assumes *s1*: *SimBlock 1 1* (*FBlock* ($\lambda x\ n.\ True$) *1 1 f-Id*)

shows $(FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ f\text{-}Id) \parallel_B (FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ f\text{-}Id)$
$= FBlock\ (\lambda x\ n.\ True)\ 2\ 2\ (\lambda x\ n.\ (((f\text{-}Id \circ (\lambda xx\ nn.\ take\ 1\ (xx\ nn)))\ x\ n)$
$\bullet\ ((f\text{-}Id \circ (\lambda xx\ nn.\ drop\ 1\ (xx\ nn))))\ x\ n))$
proof −
have *inps-1*: $inps\ (FBlock\ (\lambda x\ n.\ True)\ (Suc\ 0)\ (Suc\ 0)\ f\text{-}Id) = 1$
using *s1* **by** (*simp add*: *inps-P*)
have *form*: $((FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ f\text{-}Id) \parallel_B (FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ f\text{-}Id)) =$
$(\exists\ (ok_0,\ ok_1,\ inouts_0,\ inouts_1) \cdot$
$(((takem\ (1{+}1)\ (1))\ ;;\ (FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ f\text{-}Id))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!]$
$\wedge$
$((dropm\ (1{+}1)\ (1))\ ;;\ (FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ f\text{-}Id))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!]$
$\wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
using *s1* **by** (*simp add*: *SimParallel-form*)
have *2*: $(\exists\ (ok_0,\ ok_1,\ inouts_0,\ inouts_1) \cdot$
$(((takem\ (1{+}1)\ (1))\ ;;\ (FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ f\text{-}Id))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!]$
$\wedge$
$((dropm\ (1{+}1)\ (1))\ ;;\ (FBlock\ (\lambda x\ n.\ True)\ 1\ 1\ f\text{-}Id))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!]$
$\wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
$= FBlock\ (\lambda x\ n.\ True)\ 2\ 2\ (\lambda x\ n.\ (((f\text{-}Id \circ (\lambda xx\ nn.\ take\ 1\ (xx\ nn)))\ x\ n)$
$\bullet\ ((f\text{-}Id \circ (\lambda xx\ nn.\ drop\ 1\ (xx\ nn))))\ x\ n))$
apply (*simp add*: *FBlock-def f-Id-def takem-def dropm-def*)
apply (*rel-auto*)
apply (*simp add*: *f-Id-def*)
apply (*rule-tac* $x = ok_v'$ **in** *exI*)
apply (*rule-tac* $x = ok_v'$ **in** *exI*)
apply (*rule-tac* $x = inouts_v'$ **in** *exI*)
apply (*rule conjI*)
apply *blast*
apply (*rule-tac* $x = \lambda na.\ []$ **in** *exI*)
apply *blast*
apply (*rule-tac* $x = ok_v'$ **in** *exI*)
apply (*rule-tac* $x = ok_v'$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ take\ (Suc\ 0)\ (inouts_v\ na)$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_v'$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ take\ (Suc\ 0)\ (inouts_v\ na)$ **in** *exI*)
apply (*metis* (*no-types*, *lifting*) *Nitpick.size-list-simp*(*2*) *f-Id-def less-numeral-extra*(*3*) *list.sel*(*1*) *pos2 take-Suc take-eq-Nil take-tl*)
apply (*rule-tac* $x = \lambda na.\ drop\ (Suc\ 0)\ (inouts_v\ na)$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_v'$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ drop\ (Suc\ 0)\ (inouts_v\ na)$ **in** *exI*)
apply (*metis* (*no-types*, *lifting*) *Cons-nth-drop-Suc One-nat-def Suc-le-mono diff-Suc-1 drop-eq-Nil f-Id-def hd-drop-conv-nth le-numeral-extra*(*4*) *length-drop lessI numeral-2-eq-2*)
by (*metis Cons-nth-drop-Suc Suc-1 Suc-eq-plus1 add.left-neutral append-take-drop-id drop-0 drop-eq-Nil lessI list.sel*(*1*) *order-refl take-Suc zero-less-Suc*)
show *?thesis*
using *form 2*
by *simp*
qed

lemma *FBlock-parallel-comp*:
assumes *s1*: $\mathit{SimBlock}\ m1\ n1\ (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m1\ n1\ f)$
assumes *s2*: $\mathit{SimBlock}\ m2\ n2\ (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m2\ n2\ g)$
shows $(\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m1\ n1\ f) \parallel_B (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m2\ n2\ g)$
$= \mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ (m1{+}m2)\ (n1{+}n2)$
$(\lambda x\ n.\ (((f \circ (\lambda xx\ nn.\ \mathit{take}\ m1\ (xx\ nn)))\ x\ n) \bullet ((g \circ (\lambda xx\ nn.\ \mathit{drop}\ m1\ (xx\ nn))))\ x\ n))$
proof $-$
have *inps-1*: $\mathit{inps}\ (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m1\ n1\ f) = m1$
using *s1* **by** (*simp add*: *inps-P*)
have *inps-2*: $\mathit{inps}\ (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m2\ n2\ g) = m2$
using *s2* **by** (*simp add*: *inps-P*)
have *form*: $((\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m1\ n1\ f) \parallel_B (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m2\ n2\ g)) =$
$(\exists\ (ok_0,\ ok_1,\ inouts_0,\ inouts_1) \cdot$
$(((\mathit{takem}\ (m1{+}m2)\ (m1))\ ;;\ (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m1\ n1\ f))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!]$
$\wedge$
$((\mathit{dropm}\ (m1{+}m2)\ (m2))\ ;;\ (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m2\ n2\ g))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!]$
$\wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
using *s1 s2* **by** (*simp add*: *SimParallel-form*)
have *2*: $(\exists\ (ok_0,\ ok_1,\ inouts_0,\ inouts_1) \cdot$
$(((\mathit{takem}\ (m1{+}m2)\ (m1))\ ;;\ (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m1\ n1\ f))[\![«ok_0»,«inouts_0»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!]$
$\wedge$
$((\mathit{dropm}\ (m1{+}m2)\ (m2))\ ;;\ (\mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ m2\ n2\ g))[\![«ok_1»,«inouts_1»/\$ok´,\$\mathbf{v}_D{:}inouts´]\!]$
$\wedge$
$(\forall\ n{::}nat \cdot (\$\mathbf{v}_D{:}inouts´\ («n»)_a =_u («append»\ («inouts_0\ n»)_a\ («inouts_1\ n»)_a))) \wedge$
$(\$ok´ =_u («ok_0» \wedge «ok_1»))))$
$= \mathit{FBlock}\ (\lambda x\ n.\ \mathit{True})\ (m1{+}m2)\ (n1{+}n2)$
$(\lambda x\ n.\ (((f \circ (\lambda xx\ nn.\ \mathit{take}\ m1\ (xx\ nn)))\ x\ n) \bullet ((g \circ (\lambda xx\ nn.\ \mathit{drop}\ m1\ (xx\ nn))))\ x\ n))$
apply (*simp add*: *FBlock-def f-Id-def takem-def dropm-def*)
apply (*rel-simp*)
apply (*rule iffI*)
apply (*clarify*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
proof $-$
fix $ok_v\ inouts_v\ inouts_v{}'\ a\ aa\ ab\ ok_v{}''\ b\ inouts_v{}''{::}nat \Rightarrow real\ list$ **and** $ok_v{}'''$ **and**
$inouts_v{}'''{::}nat \Rightarrow real\ list$
assume *a1*: $\forall x.\ (m1 = 0 \longrightarrow length(inouts_v\ x) = m2 \wedge inouts_v{}''\ x = []) \wedge$
$(0 < m1 \longrightarrow length(inouts_v\ x) = m1 + m2 \wedge take\ m1\ (inouts_v\ x) = inouts_v{}''\ x)$
assume *a2*: $\forall x.\ (m2 = 0 \longrightarrow length(inouts_v\ x) = m1 \wedge inouts_v{}'''\ x = []) \wedge$
$(0 < m2 \longrightarrow length(inouts_v\ x) = m1 + m2 \wedge drop\ m1\ (inouts_v\ x) = inouts_v{}'''\ x)$
assume *a3*: $\forall x.\ length(inouts_v{}''\ x) = m1 \wedge length(ab\ x) = n1 \wedge f\ inouts_v{}''\ x = ab\ x$
assume *a4*: $\forall x.\ length(inouts_v{}'''\ x) = m2 \wedge length(b\ x) = n2 \wedge g\ inouts_v{}'''\ x = b\ x$
from *a1* **have** *1*: $\forall x.\ take\ m1\ (inouts_v\ x) = inouts_v{}''\ x$
by *fastforce*
then have *11*: $inouts_v{}'' = (\lambda x.\ take\ m1\ (inouts_v\ x))$
using *a1* **by** *force*
from *a3* **have** *2*: $\forall x.\ f\ inouts_v{}''\ x = ab\ x$
by *blast*
from *11* **and** *2* **have** *3*: $\forall x.\ f\ (\lambda x.\ take\ m1\ (inouts_v\ x))\ x = ab\ x$
by *blast*
from *a2* **have** *g1*: $\forall x.\ (drop\ m1\ (inouts_v\ x) = inouts_v{}'''\ x)$
by *fastforce*
then have *g11*: $inouts_v{}''' = (\lambda x.\ drop\ m1\ (inouts_v\ x))$

by *force*
from *a4* **have** $g2$: $\forall x.\ g\ inouts_v{'''}\ x = b\ x$
by *blast*
from *g11* **and** *g2* **have** $g3$: $\forall x.\ g\ (\lambda x.\ drop\ m1\ (inouts_v\ x))\ x = b\ x$
by *blast*
show $\forall x.\ length(inouts_v\ x) = m1 + m2 \wedge$
$f\ (\lambda nn.\ take\ m1\ (inouts_v\ nn))\ x \bullet g\ (\lambda nn.\ drop\ m1\ (inouts_v\ nn))\ x = ab\ x \bullet b\ x$
apply (*rule allI*)
apply (*rule conjI*)
using *a2* **apply** *auto*[*1*]
by (*simp add*: *3 g3*)
next
assume $a1$: $\forall x\ xa.\ length(x\ xa) = m1 \longrightarrow length(f\ x\ xa) = n1$
assume $a2$: $\forall x\ xa.\ length(x\ xa) = m2 \longrightarrow length(g\ x\ xa) = n2$
show $\forall x\ xa.\ length(x\ xa) = m1 + m2 \longrightarrow$
$length(f\ (\lambda nn.\ take\ m1\ (x\ nn))\ xa) + length(g\ (\lambda nn.\ drop\ m1\ (x\ nn))\ xa) = n1 + n2$
using *a1 a2* **by** *simp*
next
fix $ok_v\ inouts_v\ ok_v{'}\ inouts_v{'}$
assume $a1$: $ok_v \longrightarrow$
$ok_v{'} \wedge$
$(\forall x.\ length(inouts_v\ x) = m1 + m2 \wedge$
$length(inouts_v{'}\ x) = n1 + n2 \wedge$
$f\ (\lambda nn.\ take\ m1\ (inouts_v\ nn))\ x \bullet g\ (\lambda nn.\ drop\ m1\ (inouts_v\ nn))\ x = inouts_v{'}\ x) \wedge$
$(\forall x\ xa.\ length(x\ xa) = m1 + m2 \longrightarrow$
$length(f\ (\lambda nn.\ take\ m1\ (x\ nn))\ xa) + length(g\ (\lambda nn.\ drop\ m1\ (x\ nn))\ xa) = n1 + n2)$
from *a1* **show** $\exists a\ aa\ ab.$
$(\exists ok_v{'}\ inouts_v{'}.$
$(ok_v \longrightarrow$
$ok_v{'} \wedge$
$(\forall x.\ (m1 = 0 \longrightarrow length(inouts_v\ x) = m2 \wedge inouts_v{'}\ x = []) \wedge$
$(0 < m1 \longrightarrow$
$length(inouts_v\ x) = m1 + m2 \wedge length(inouts_v{'}\ x) = m1 \wedge take\ m1\ (inouts_v\ x) =$
$inouts_v{'}\ x))) \wedge$
$(ok_v{'} \longrightarrow$
$a \wedge (\forall x.\ length(inouts_v{'}\ x) = m1 \wedge length(ab\ x) = n1 \wedge f\ inouts_v{'}\ x = ab\ x) \wedge$
$(\forall x\ xa.\ length(x\ xa) = m1 \longrightarrow length(f\ x\ xa) = n1))) \wedge$
$(\exists b.\ (\exists ok_v{'}\ inouts_v{'}.$
$(ok_v \longrightarrow$
$ok_v{'} \wedge$
$(\forall x.\ (m2 = 0 \longrightarrow length(inouts_v\ x) = m1 \wedge inouts_v{'}\ x = []) \wedge$
$(0 < m2 \longrightarrow$
$length(inouts_v\ x) = m1 + m2 \wedge$
$length(inouts_v{'}\ x) = m2 \wedge drop\ m1\ (inouts_v\ x) = inouts_v{'}\ x))) \wedge$
$(ok_v{'} \longrightarrow$
$aa \wedge (\forall x.\ length(inouts_v{'}\ x) = m2 \wedge length(b\ x) = n2 \wedge g\ inouts_v{'}\ x = b\ x) \wedge$
$(\forall x\ xa.\ length(x\ xa) = m2 \longrightarrow length(g\ x\ xa) = n2))) \wedge$
$(\forall x.\ inouts_v{'}\ x = ab\ x \bullet b\ x) \wedge ok_v{'} = (a \wedge aa))$
apply (*rel-auto*)
apply (*rule-tac* $x = ok_v{'}$ **in** *exI*)
apply (*rule-tac* $x = ok_v{'}$ **in** *exI*)
apply (*rule-tac* $x = inouts_v{'}$ **in** *exI*)
apply (*rule conjI*)
apply *blast*
using *take-0* **apply** *blast*

apply (*rule-tac* $x = ok_v{}'$ **in** *exI*)
apply (*rule-tac* $x = ok_v{}'$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ f\ (\lambda nx.\ take\ m1\ (inouts_v\ nx))\ na$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_v{}'$ **in** *exI*)
apply (*rule-tac* $x = \lambda nx.\ take\ m1\ (inouts_v\ nx)$ **in** *exI*)
using *SimBlock-FBlock-fn s1* **apply** *auto*[*1*]
apply (*rule-tac* $x = \lambda na.\ g\ (\lambda nx.\ drop\ m1\ (inouts_v\ nx))\ na$ **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = ok_v{}'$ **in** *exI*)
apply (*rule-tac* $x = \lambda nx.\ drop\ m1\ (inouts_v\ nx)$ **in** *exI*)
using *SimBlock-FBlock-fn s2* **apply** *auto*[*1*]
by *simp*
qed
show *?thesis*
using *2 form* **by** *simp*
qed

lemma *SimBlock-FBlock-parallel-comp* [*simblock-healthy*]:
assumes *s1*: *SimBlock m1 n1* (*FBlock* ($\lambda x\ n.\ True$) *m1 n1 f*)
assumes *s2*: *SimBlock m2 n2* (*FBlock* ($\lambda x\ n.\ True$) *m2 n2 g*)
shows *SimBlock* (*m1*+*m2*) (*n1*+*n2*) ((*FBlock* ($\lambda x\ n.\ True$) *m1 n1 f*) $\|_B$ (*FBlock* ($\lambda x\ n.\ True$) *m2 n2 g*))
apply (*simp add*: *s1 s2 FBlock-parallel-comp*)
apply (*rule SimBlock-FBlock*)
proof −
obtain $inouts_v$::*nat* ⇒ *real list* **where** *P*: $\forall na.\ length(inouts_v\ na) = m1 + m2$
using *list-len-avail* **by** *auto*
show $\exists inouts_v\ inouts_v{}'.$
$\forall x.\ length(inouts_v{}'\ x) = n1 + n2\ \wedge$
$length(inouts_v\ x) = m1 + m2\ \wedge$
$f\ (\lambda nn.\ take\ m1\ (inouts_v\ nn))\ x \bullet g\ (\lambda nn.\ drop\ m1\ (inouts_v\ nn))\ x = inouts_v{}'\ x$
apply (*rule-tac* $x = inouts_v$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ (f\ (\lambda nn.\ take\ m1\ (inouts_v\ nn))\ na \bullet g\ (\lambda nn.\ drop\ m1\ (inouts_v\ nn))\ na)$ **in** *exI*)
using *P SimBlock-FBlock-fn s1 s2* **by** *auto*
next
show $\forall x\ na.\ length(x\ na) = m1 + m2 \longrightarrow$
$length(f\ (\lambda nn.\ take\ m1\ (x\ nn))\ na \bullet g\ (\lambda nn.\ drop\ m1\ (x\ nn))\ na) = n1 + n2$
using *SimBlock-FBlock-fn s1 s2* **by** *auto*
qed

B.4.4 Feedback

B.4.4.1 feedback

lemma *feedback-mono*:
fixes *m1* :: *nat* **and** *n1* :: *nat* **and** *i1* :: *nat* **and** *o1* :: *nat*
assumes *s1*: *SimBlock m1 n1 P1*
assumes *s2*: *SimBlock m1 n1 P2*
assumes *s3*: $P1 \sqsubseteq P2$
assumes *s4*: $i1 < m1$
assumes *s5*: $o1 < n1$
shows $(P1\ f_D\ (i1,o1)) \sqsubseteq (P2\ f_D\ (i1,o1))$
apply (*simp add*: *f-sim-blocks*)
using *s1 s2* **apply** (*simp add*: *inps-P outps-P*)
apply (*rel-simp*)
apply (*auto*)

apply (*metis s3 upred-ref-iff*)
apply (*rule-tac x = x* **in** *exI*)
apply (*rule-tac x =* ok_v'' **in** *exI*)
apply (*rule-tac x =* $inouts_v''$ **in** *exI*)
apply (*rule-tac x =* ok_v''' **in** *exI*)
apply (*rule-tac x =* $inouts_v'''$ **in** *exI*)
apply (*metis s3 upred-ref-iff*)
apply (*rule-tac x = x* **in** *exI*)
apply (*rule-tac x = True* **in** *exI*)
apply (*rule-tac x =* $inouts_v''$ **in** *exI*)
apply (*rule conjI*)
apply *blast*
apply (*rule-tac x = False* **in** *exI*)
apply (*rule-tac x =* $inouts_v'''$ **in** *exI*)
apply (*meson s3 upred-ref-iff*)
apply (*rule-tac x = x* **in** *exI*)
apply (*rule-tac x = True* **in** *exI*)
apply (*rule-tac x =* $inouts_v''$ **in** *exI*)
apply (*rule conjI*)
apply *blast*
apply (*rule-tac x =* ok_v''' **in** *exI*)
apply (*rule-tac x =* $inouts_v'''$ **in** *exI*)
by (*metis s3 upred-ref-iff*)

lemma *sol-f-id*: *Solvable 0 0 1 1 f-Id*
by (*simp add*: *Solvable-def f-Id-def f-PreFD-def*)

lemma *sol-f-ud*: *Solvable 0 0 1 1* (*f-UnitDelay x0*)
apply (*simp add*: *Solvable-def f-UnitDelay-def f-PreFD-def*)
by (*auto*)

— The function which output is equal to its input plus 1 is not solvable
lemma ¬ *Solvable 0 0 1 1* (λx *n*. [*hd*(*x n*) + *1*])
apply (*simp add*: *Solvable-def f-PreFD-def*)
by (*auto*)

lemma *sol-f-id-ud*: *Solvable 0 0 1 1* ((*f-UnitDelay x0*) ∘ (*f-Id*))
apply (*simp add*: *Solvable-def f-UnitDelay-def f-Id-def f-PreFD-def*)
by (*auto*)

lemma *sol-f-integrator*:
Solvable 1 1 2 2 (λx *n*. [*if n = 0 then x0 else* (*x* (*n*−*1*)!*0*) + (*x* (*n*−*1*)!*1*),
if n = 0 then x0 else (*x* (*n*−*1*)!*0*) + (*x* (*n*−*1*)!*1*)])
apply (*simp add*: *Solvable-def f-PreFD-def*)
apply (*clarify*)
apply (*rule-tac x =* λna. (*if na = 0 then x0 else* (*x0*+*sum-hd-signal* $inouts_0$ (*na*−*1*))) **in** *exI*)
apply (*simp*, *clarify*)
apply (*rule conjI*)
apply (*clarify*)

apply (*metis Nil-is-append-conv One-nat-def add.commute hd-append2 hd-conv-nth list.size*(*3*) *nth-append-length zero-neq-one*)
apply (*clarify*)
proof −
fix $inouts_0$::*nat* ⇒ *real list* **and** *n*::*nat*
assume *a1*: ∀ *x*. *length*($inouts_0$ *x*) = *Suc 0*
assume *a2*: ¬ *n* ≤ *Suc 0*
have *1*: ($inouts_0$ (*n* − *Suc 0*) • [*x0* + *sum-hd-signal* $inouts_0$ (*n* − *Suc* (*Suc 0*))])!(*0*)
= *hd*($inouts_0$ (*n* − *Suc 0*))
using *a1 a2*
by (*metis One-nat-def hd-conv-nth le-numeral-extra*(*4*) *less-numeral-extra*(*1*) *list.size*(*3*) *not-one-le-zero nth-append*)
have *2*: ($inouts_0$ (*n* − *Suc 0*) • [*x0* + *sum-hd-signal* $inouts_0$ (*n* − *Suc* (*Suc 0*))])!(*Suc 0*)
= *x0* + *sum-hd-signal* $inouts_0$ (*n* − *Suc* (*Suc 0*))
using *a1 a2*
by (*metis nth-append-length*)
have *3*: (*n* − (*Suc 0*)) = *Suc* (*n* − (*Suc* (*Suc 0*)))
using *a2* **by** *linarith*
show *x0* + *sum-hd-signal* $inouts_0$ (*n* − *Suc 0*) =
($inouts_0$ (*n* − *Suc 0*) • [*x0* + *sum-hd-signal* $inouts_0$ (*n* − *Suc* (*Suc 0*))])!(*0*) +
($inouts_0$ (*n* − *Suc 0*) • [*x0* + *sum-hd-signal* $inouts_0$ (*n* − *Suc* (*Suc 0*))])!(*Suc 0*)
apply (*simp add*: *1 2*)
using *a1 a2 3*
by *simp*
qed

lemma *Solvable-unique-is-solvable*:
assumes *Solvable-unique i1 o1 m n* (*f*)
shows *Solvable i1 o1 m n* (*f*)
using *assms* **apply** (*simp add*: *Solvable-unique-def Solvable-def*)
apply (*clarify*)
by *blast*

unique-solution-integrator: the integrator diagram has a unique solution.

lemma *unique-solution-integrator*:
fixes $inouts_0$::*nat* ⇒ *real list*
assumes *s1*: ∀ *n*. *length*($inouts_0$ *n*) = *1*
shows ∃!*xx*. (∀ *n*. (*n* = *0* ⟶ *xx 0* = *x0*) ∧
(*0* < *n* ⟶ *xx n* = *hd*(($inouts_0$ (*n* − *Suc 0*))) + *xx* (*n* − *Suc 0*)))
apply (*rule ex-ex1I*)
apply (*rule-tac x* = *λna*. (*if na* = *0 then x0 else* (*x0*+($\sum$ *i* ∈ {*0*..(*na*−*1*)}. *hd*(($inouts_0$ *i*))))) **in** *exI*)
apply (*simp*)
apply (*rule allI*)
proof −
fix *n*::*nat*
show ¬ *n* ≤ *Suc 0* ⟶
($\sum$ *i* = *0*..*n* − *Suc 0*. *hd* ($inouts_0$ *i*)) =
hd ($inouts_0$ (*n* − *Suc 0*)) + ($\sum$ *i* = *0*..*n* − *Suc* (*Suc 0*). *hd* ($inouts_0$ *i*))
proof (*induct n*)
case *0*
thus *?case* **by** *auto*
next
case (*Suc n*) **note** *IH* = *this*
{ **assume** *Suc n* = *1*

```
          hence ?case by auto
        }
        also {
          assume Suc n > 1
          {
            assume Suc n = 2
            hence ?case by auto
          }
          also {
            assume Suc n > 2
            have ?case
              by (smt One-nat-def Suc-diff-Suc ‹1 < Suc n› sum.atLeast0-atMost-Suc)
          }
        }

        ultimately show ?case
          by (smt One-nat-def Suc-1 Suc-lessI cancel-comm-monoid-add-class.diff-cancel
              diff-Suc-1 not-less sum.atLeast0-atMost-Suc)
      qed
  next
    fix xx:: nat ⇒ real and y:: nat ⇒ real
    assume a1: ∀n. (n = 0 ⟶ xx 0 = x0) ∧ (0 < n ⟶ xx n = hd (inouts₀ (n − Suc 0)) + xx (n
− Suc 0))
    assume a2: ∀n. (n = 0 ⟶ y 0 = x0) ∧ (0 < n ⟶ y n = hd (inouts₀ (n − Suc 0)) + y (n −
Suc 0))
    have 1: ∀n. xx n = y n
      apply (rule allI)
      proof −
        fix n::nat
        show xx n = y n
          proof (induct n)
            case 0
            then show ?case
              using a1 a2 by simp
          next
            case (Suc n) note IH = this
            then show ?case
              using a1 a2 by simp
          qed
      qed
    show xx = y
      using 1 fun-eq by (blast)
  qed

lemma FBlock-feedback:
  assumes s1: SimBlock m n (FBlock (λx n. True) m n f)
  assumes s2: Solvable-unique i1 o1 m n (f)
  shows (FBlock (λx n. True) m n f) f_D (i1, o1)
      = (FBlock (λx n. True) (m−1) (n−1)
          (λx na. ((f-PostFD o1) o f o (f-PreFD (Solution i1 o1 m n f x) i1)) x na))
  proof −
    have inps-1: inps (FBlock (λx n. True) m n f) = m
      using s1 by (simp add: inps-P)
    have outps-1: outps (FBlock (λx n. True) m n f) = n
      using s1 by (simp add: outps-P)
```

have *i1-lt-m*: $i1 < m$
  using *s2* **by** (*simp add*: *Solvable-unique-def*)
have *o1-lt-n*: $o1 < n$
  using *s2* **by** (*simp add*: *Solvable-unique-def*)
have *1*: $(FBlock\ (\lambda x\ n.\ True)\ m\ n\ f)\ f_D\ (i1,\ o1) = (true \vdash_n (\exists\ x \cdot$
$(\forall\ n \cdot \#_u(\$inouts(«n»)_a) =_u «m - Suc\ 0» \wedge$
$\#_u(\$inouts´(«n»)_a) =_u «m» \wedge \$inouts´(«n»)_a =_u «f\text{-}PreFD\ x\ i1»(\$inouts)_a(«n»)_a)$
; ;
$((\forall\ na \cdot \#_u(\$inouts(«na»)_a) =_u «m» \wedge$
$\#_u(\$inouts´(«na»)_a) =_u «n» \wedge «f»(\$inouts)_a(«na»)_a =_u \$inouts´(«na»)_a) \wedge$
$(\forall\ x \cdot \forall\ na \cdot \#_u(«x\ na») =_u «m» \Rightarrow \#_u(«f\ x\ na») =_u «n»))$; ;
$(\forall\ na \cdot \#_u(\$inouts(«na»)_a) =_u «n» \wedge$
$\#_u(\$inouts´(«na»)_a) =_u «n - Suc\ 0» \wedge$
$\$inouts´(«na»)_a =_u «f\text{-}PostFD\ o1»(\$inouts)_a(«na»)_a \wedge$
$«uapply»(\$inouts(«na»)_a)_a(«o1»)_a =_u «x\ na»)))$
  apply (*simp add*: *inps-1 outps-1*)
  apply (*simp add*: *PreFD-def PostFD-def FBlock-def Solution-def*)
  apply (*simp add*: *ndesign-composition-wp wp-upred-def*)
  by (*rel-simp*)
have *2*: $(true \vdash_n (\exists\ x \cdot$
$(\forall\ n \cdot \#_u(\$inouts(«n»)_a) =_u «m - Suc\ 0» \wedge$
$\#_u(\$inouts´(«n»)_a) =_u «m» \wedge \$inouts´(«n»)_a =_u «f\text{-}PreFD\ x\ i1»(\$inouts)_a(«n»)_a)$
; ;
$((\forall\ na \cdot \#_u(\$inouts(«na»)_a) =_u «m» \wedge$
$\#_u(\$inouts´(«na»)_a) =_u «n» \wedge «f»(\$inouts)_a(«na»)_a =_u \$inouts´(«na»)_a) \wedge$
$(\forall\ x \cdot \forall\ na \cdot \#_u(«x\ na») =_u «m» \Rightarrow \#_u(«f\ x\ na») =_u «n»))$; ;
$(\forall\ na \cdot \#_u(\$inouts(«na»)_a) =_u «n» \wedge$
$\#_u(\$inouts´(«na»)_a) =_u «n - Suc\ 0» \wedge$
$\$inouts´(«na»)_a =_u «f\text{-}PostFD\ o1»(\$inouts)_a(«na»)_a \wedge$
$«uapply»(\$inouts(«na»)_a)_a(«o1»)_a =_u «x\ na»)))$
  $= (FBlock\ (\lambda x\ n.\ True)\ (m-1)\ (n-1)$
$(\lambda x\ na.\ ((f\text{-}PostFD\ o1)\ o\ f\ o\ (f\text{-}PreFD\ (Solution\ i1\ o1\ m\ n\ f\ x)\ i1))\ x\ na))$
  apply (*simp add*: *FBlock-def Solution-def*)
  apply (*rule ref-eq*)
  apply (*rule ndesign-refine-intro*, *simp+*)
  apply (*rel-simp*)
  apply (*rule-tac* $x = (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))$ **in** *exI*)
  apply (*rule-tac* $x = \lambda na.\ f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))$
$i1\ inouts_v\ na$ **in** *exI*, *simp*)
  apply (*rule conjI*)
  apply (*simp add*: *f-PreFD-def*)
  using *i1-lt-m* **apply** *linarith*
  apply (*rule-tac* $x = \lambda na.\ (f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))$
$i1\ inouts_v)\ na)$ **in** *exI*, *simp*)
  apply (*rule conjI*)
  apply (*simp add*: *f-PreFD-def*)
  apply (*rule conjI*)
  using *i1-lt-m* **apply** *linarith*

  defer
  apply (*rule conjI*)
  using *SimBlock-FBlock-fn s1* **apply** *blast*
  apply (*rule allI*, *rule conjI*)

  defer

defer
apply (*rule ndesign-refine-intro*, *simp*+)
apply (*rel-simp*)
apply (*rule conjI*)
defer
apply (*simp add*: *f-PreFD-def f-PostFD-def*)
using *o1-lt-n* **apply** *linarith*
prefer *3*
proof −
fix $inouts_v$::*nat* ⇒ *real list* **and** $inouts_v$′::*nat* ⇒ *real list* **and** *x*::*nat*
assume *a1*: ∀ *x*. *length*($inouts_v$ *x*) = *m* − *Suc 0* ∧
length($inouts_v$′ *x*) = *n* − *Suc 0* ∧
f-PostFD o1 (*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1* $inouts_v$) *n*!(*o1*)) *i1* $inouts_v$))
x = $inouts_v$′ *x*
let *?P*= λ*xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1* $inouts_v$) *n*!(*o1*)
have *1*: (*?P* (*SOME xx*. *?P xx*))
apply (*rule someI-ex*[*of ?P*])
using *s2* **apply** (*simp add*: *Solvable-unique-def*)
using *a1* **by** *blast*
show *f* (*f-PreFD* (*SOME xx*. *?P xx*) *i1* $inouts_v$) *x*!(*o1*) = (*SOME xx*. *?P xx*) *x*
by (*simp add*: *1*)
next
fix $inouts_v$ $inouts_v$′
assume *a1*: ∀ *x*. *length*($inouts_v$ *x*) = *m* − *Suc 0* ∧
length($inouts_v$′ *x*) = *n* − *Suc 0* ∧
f-PostFD o1 (*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1* $inouts_v$) *n*!(*o1*)) *i1* $inouts_v$))
x =
$inouts_v$′ *x*
assume *a2*: ∀ *x xa*. *length*(*x xa*) = *m* − *Suc 0* ⟶
length(*f-PostFD o1* (*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1 x*) *n*!(*o1*)) *i1 x*))
xa) =
n − *Suc 0*
from *a1* **have** *a1*′: ∀ *x*. *length*($inouts_v$ *x*) = *m* − *Suc 0*
by (*simp*)
have ∀ *na*. *length*((*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1* $inouts_v$) *n*!(*o1*)) *i1* $inouts_v$)
na) = *m*
using *a1*′ *f-PreFD-def* **apply** (*simp*)
using *i1-lt-m* **by** *linarith*
then show ∀ *x*. *length*(*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1* $inouts_v$) *n*!(*o1*)) *i1*
$inouts_v$) *x*) = *n*
using *SimBlock-FBlock-fn s1* **by** *blast*
next
fix $inouts_v$ $inouts_v$′ *x*
assume *a1*: ∀ *x*. *length*($inouts_v$ *x*) = *m* − *Suc 0* ∧
length($inouts_v$′ *x*) = *n* − *Suc 0* ∧
f-PostFD o1 (*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1* $inouts_v$) *n*!(*o1*)) *i1* $inouts_v$))
x =
$inouts_v$′ *x*
assume *a2*: ∀ *x xa*. *length*(*x xa*) = *m* − *Suc 0* ⟶
length(*f-PostFD o1* (*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1 x*) *n*!(*o1*)) *i1 x*))
xa) =
n − *Suc 0*
from *a1* **have** *a1*′: ∀ *x*. *length*($inouts_v$ *x*) = *m* − *Suc 0*
by (*simp*)
have ∀ *na*. *length*((*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1* $inouts_v$) *n*!(*o1*)) *i1* $inouts_v$)

$na) = m$
using *a1' f-PreFD-def* **apply** (*simp*)
using *i1-lt-m* **by** *linarith*
then show $length(f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))\ i1\ inouts_v)\ x) = n$
using *SimBlock-FBlock-fn s1* **by** *blast*
next
fix $inouts_v$::*nat* ⇒ *real list* **and** $inouts_v'$::*nat* ⇒ *real list* **and** *x*::*nat* ⇒ *real* **and** $inouts_v''$::*nat* ⇒ *real list* **and** $inouts_v'''$::*nat* ⇒ *real list*
assume *a1*: $\forall xa.\ length(inouts_v\ xa) = m - Suc\ 0 \wedge inouts_v''\ xa = f\text{-}PreFD\ x\ i1\ inouts_v\ xa$
assume *a2*: $\forall xa.\ length(f\text{-}PreFD\ x\ i1\ inouts_v\ xa) = m \wedge f\ inouts_v''\ xa = inouts_v'''\ xa$
assume *a3*: $\forall xa.\ length(inouts_v'''\ xa) = n \wedge length(inouts_v'\ xa) = n - Suc\ 0 \wedge inouts_v'\ xa = f\text{-}PostFD\ o1\ inouts_v'''\ xa \wedge inouts_v'''\ xa!(o1) = x\ xa$
have *unique-sol*:
$(\exists!\ (xx::nat \Rightarrow real).$
$(\forall n.\ (xx\ n = (f\ (\lambda n1.\ f\text{-}PreFD\ xx\ i1\ inouts_v\ n1)\ n)!o1)))$
using *s2 a1* **by** (*simp add*: *Solvable-unique-def*)
from *a1 a2* **have** $\forall xa.\ inouts_v'''\ xa = f\ inouts_v''\ xa$
by *simp*
then have $\forall xa.\ inouts_v'''\ xa = f\ (f\text{-}PreFD\ x\ i1\ inouts_v)\ xa$
using *a1* **by** *presburger*
then have *0*: $inouts_v''' = f\ (f\text{-}PreFD\ x\ i1\ inouts_v)$
by (*rule fun-eq*)
have *1*: $(SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1)) = x$
apply (*rule some-equality*)
using *0 a3 unique-sol* **by** *auto*
then have *2*: $\forall n.\ f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))\ i1\ inouts_v))\ n$
$= f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ x\ i1\ inouts_v))\ n$
by *blast*
then have *3*: $\forall n.\ f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))\ i1\ inouts_v))\ n$
$= f\text{-}PostFD\ o1\ inouts_v'''\ n$
using *0* **by** *blast*
show $\forall x.\ length(f\text{-}PostFD\ o1\ inouts_v'''\ x) = n - Suc\ 0 \wedge$
$f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))\ i1\ inouts_v))\ x$
$= f\text{-}PostFD\ o1\ inouts_v'''\ x$
apply (*rule allI*, *rule conjI*)
apply (*simp add*: *f-PostFD-def*)
using *a3 o1-lt-n* **apply** *auto*[*1*]
using *3* **by** *blast*
qed
show *?thesis*
using *1* **by** (*simp add*: *2*)
qed

lemma *unique-solution*:
assumes *s1*: *Solvable-unique i1 o1 m n* (*f*)
assumes *s2*: *is-Solution i1 o1 m n* (*f*) (*xx*)
assumes *s3*: $\forall n.\ length(ins\ n) = m{-}1$
shows *xx ins* = (*Solution i1 o1 m n f ins*)
using *s1 s2* **apply** (*simp add*: *Solution-def Solvable-unique-def is-Solution-def*)
apply (*clarify*)
proof −

assume *a1*: $\forall$ *inouts*$_0$. ($\forall$ *x*. *length*(*inouts*$_0$ *x*) = *m* − *Suc 0*) $\longrightarrow$
($\forall$ *n*. *xx inouts*$_0$ *n* = *f* (*f-PreFD* (*xx inouts*$_0$) *i1 inouts*$_0$) *n*!(*o1*))
assume *a2*: $\forall$ *inouts*$_0$. ($\forall$ *x*. *length*(*inouts*$_0$ *x*) = *m* − *Suc 0*) $\longrightarrow$
($\exists$!*xx*. $\forall$ *n*. *xx n* = *f* (*f-PreFD xx i1 inouts*$_0$) *n*!(*o1*))
have (*SOME xx*. $\forall$ *n*. *xx n* = *f* (*f-PreFD xx i1 ins*) *n*!(*o1*)) = *xx ins*
apply (*rule some-equality*)
using *a1 s3* **apply** *simp*
using *a2* **apply** (*simp add*: *Ex1-def*)
proof −
fix *xxa*
assume *a3*: $\forall$ *n*. *xxa n* = *f* (*f-PreFD xxa i1 ins*) *n*!(*o1*)
assume *a4*: $\forall$ *inouts*$_0$.
($\forall$ *x*. *length*(*inouts*$_0$ *x*) = *m* − *Suc 0*) $\longrightarrow$
($\exists$ *x*. ($\forall$ *n*. *x n* = *f* (*f-PreFD x i1 inouts*$_0$) *n*!(*o1*)) $\wedge$
($\forall$ *y*. ($\forall$ *n*. *y n* = *f* (*f-PreFD y i1 inouts*$_0$) *n*!(*o1*)) $\longrightarrow$ *y* = *x*))
from *a4 s3* **have** *1*: ($\exists$ *x*. ($\forall$ *n*. *x n* = *f* (*f-PreFD x i1 ins*) *n*!(*o1*)) $\wedge$
($\forall$ *y*. ($\forall$ *n*. *y n* = *f* (*f-PreFD y i1 ins*) *n*!(*o1*)) $\longrightarrow$ *y* = *x*))
by *simp*
from *s2* **have** *2*: $\forall$ *n*. (*xx ins*) *n* = *f* (*f-PreFD* (*xx ins*) *i1 ins*) *n*!(*o1*)
using *a1 s3* **by** *simp*
show *xxa* = *xx ins*
using *a3 a4 s3 1 2* **by** *blast*
qed
then show *xx ins* = (*SOME xx*. $\forall$ *n*. *xx n* = *f* (*f-PreFD xx i1 ins*) *n*!(*o1*))
by *simp*
qed

lemma *FBlock-feedback'*:
assumes *s1*: *SimBlock m n* (*FBlock* (λ*x n*. *True*) *m n f*)
assumes *s2*: *Solvable-unique i1 o1 m n* (*f*)
assumes *s3*: *is-Solution i1 o1 m n* (*f*) (*xx*)
shows (*FBlock* (λ*x n*. *True*) *m n f*) f_D (*i1*, *o1*)
= (*FBlock* (λ*x n*. *True*) (*m*−*1*) (*n*−*1*)
(λ*x na*. ((*f-PostFD o1*) *o f o* (*f-PreFD* (*xx x*) *i1*)) *x na*))
using *s1 s2 FBlock-feedback* **apply** (*simp*)
proof −
have *i1-lt-m*: *i1* < *m*
using *s2* **by** (*simp add*: *Solvable-unique-def*)
have *o1-lt-n*: *o1* < *n*
using *s2* **by** (*simp add*: *Solvable-unique-def*)
show *FBlock* (λ*x n*. *True*) (*m* − *Suc 0*) (*n* − *Suc 0*)
(λ*x*. *f-PostFD o1* (*f* (*f-PreFD* (*Solution i1 o1 m n f x*) *i1 x*))) =
FBlock (λ*x n*. *True*) (*m* − *Suc 0*) (*n* − *Suc 0*) (λ*x*. *f-PostFD o1* (*f* (*f-PreFD* (*xx x*) *i1 x*)))
apply (*simp* (*no-asm*) *add*: *FBlock-def*)
apply (*rel-simp*)
apply (*rule iffI*, *clarify*)
defer
apply (*clarify*)
defer
proof −
fix *ok*$_v$ *inouts*$_v$ *ok*$_v$' *inouts*$_v$'
assume *a1*: $\forall$ *x*. *length*(*inouts*$_v$ *x*) = *m* − *Suc 0* $\wedge$
length(*inouts*$_v$' *x*) = *n* − *Suc 0* $\wedge$
f-PostFD o1 (*f* (*f-PreFD* (*Solution i1 o1 m n f inouts*$_v$) *i1 inouts*$_v$)) *x* = *inouts*$_v$' *x*
assume *a2*: $\forall$ *x xa*. *length*(*x xa*) = *m* − *Suc 0* $\longrightarrow$

$length(f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (Solution\ i1\ o1\ m\ n\ f\ x)\ i1\ x))\ xa) = n - Suc\ 0$

have 1: $\forall x.\ length(inouts_v\ x) = m - Suc\ 0$
using $a1$ **by** $simp$
have 2: $xx\ inouts_v = (Solution\ i1\ o1\ m\ n\ f\ inouts_v)$
apply $(rule\ unique\text{-}solution)$
using $s2$ **apply** $(simp)$
using $s3$ **apply** $(simp)$
using 1 **by** $(simp)$
show $(\forall x.\ length(inouts_v\ x) = m - Suc\ 0 \wedge length(inouts_v{}'\ x) = n - Suc\ 0 \wedge f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (xx\ inouts_v)\ i1\ inouts_v))\ x = inouts_v{}'\ x) \wedge (\forall x\ xa.\ length(x\ xa) = m - Suc\ 0 \longrightarrow length(f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (xx\ x)\ i1\ x))\ xa) = n - Suc\ 0)$
apply $(rule\ conjI)$
using $2\ a1$ **apply** $simp$
apply $(rule\ allI)$
apply $(clarify)$
proof $-$
fix $x::nat \Rightarrow real\ list$ **and** $xa::nat$
assume $a11$: $length\ (x\ xa) = m - Suc\ 0$
have 1: $length((f\text{-}PreFD\ (xx\ x)\ i1\ x)\ xa) = m$
using $a11$ **apply** $(simp\ add{:}\ f\text{-}PreFD\text{-}def)$
using $i1\text{-}lt\text{-}m$ **by** $linarith$
have 2: $length((f\ (f\text{-}PreFD\ (xx\ x)\ i1\ x))\ xa) = n$
using $1\ SimBlock\text{-}FBlock\text{-}fn\ s1$ **by** $blast$
show $length(f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (xx\ x)\ i1\ x))\ xa) = n - Suc\ 0$
apply $(simp\ add{:}\ f\text{-}PostFD\text{-}def\ f\text{-}PreFD\text{-}def)$
using $1\ 2\ o1\text{-}lt\text{-}n$ **by** $linarith$
qed
next
fix $ok_v\ inouts_v\ ok_v{}'\ inouts_v{}'$
assume $a1$: $\forall x.\ length(inouts_v\ x) = m - Suc\ 0 \wedge length(inouts_v{}'\ x) = n - Suc\ 0 \wedge f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (xx\ inouts_v)\ i1\ inouts_v))\ x = inouts_v{}'\ x$
assume $a2$: $\forall x\ xa.\ length(x\ xa) = m - Suc\ 0 \longrightarrow length(f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (xx\ x)\ i1\ x))\ xa) = n - Suc\ 0$
have 1: $\forall x.\ length(inouts_v\ x) = m - Suc\ 0$
using $a1$ **by** $simp$
have 2: $xx\ inouts_v = (Solution\ i1\ o1\ m\ n\ f\ inouts_v)$
apply $(rule\ unique\text{-}solution)$
using $s2$ **apply** $(simp)$
using $s3$ **apply** $(simp)$
using 1 **by** $(simp)$
show $(\forall x.\ length(inouts_v\ x) = m - Suc\ 0 \wedge length(inouts_v{}'\ x) = n - Suc\ 0 \wedge f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (Solution\ i1\ o1\ m\ n\ f\ inouts_v)\ i1\ inouts_v))\ x = inouts_v{}'\ x) \wedge (\forall x\ xa.\ length(x\ xa) = m - Suc\ 0 \longrightarrow length(f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (Solution\ i1\ o1\ m\ n\ f\ x)\ i1\ x))\ xa) = n - Suc\ 0)$
apply $(rule\ conjI)$
using $2\ a1$ **apply** $auto[1]$
apply $(rule\ allI)$
apply $(clarify)$
proof $-$
fix $x::nat \Rightarrow real\ list$ **and** $xa::nat$
assume $a11$: $length\ (x\ xa) = m - Suc\ 0$
have 1: $length((f\text{-}PreFD\ (Solution\ i1\ o1\ m\ n\ f\ x)\ i1\ x)\ xa) = m$
using $a11$ **apply** $(simp\ add{:}\ f\text{-}PreFD\text{-}def)$
using $i1\text{-}lt\text{-}m$ **by** $linarith$

have *2*: *length*((*f* (*f-PreFD* (*Solution i1 o1 m n f x*) *i1 x*)) *xa*) = *n*
using *1 SimBlock-FBlock-fn s1* **by** *blast*
show *length*(*f-PostFD o1* (*f* (*f-PreFD* (*Solution i1 o1 m n f x*) *i1 x*)) *xa*) = *n* − *Suc 0*
apply (*simp add*: *f-PostFD-def f-PreFD-def*)
using *1 2 o1-lt-n* **by** *linarith*
qed
qed
qed

lemma *FBlock-feedback-ref*:
assumes *s1*: *SimBlock m n* (*FBlock* (*λx n. True*) *m n f*)
assumes *s2*: *Solvable i1 o1 m n* (*f*)
shows (*FBlock* (*λx n. True*) *m n f*) f_D (*i1*, *o1*)
⊑ (*FBlock* (*λx n. True*) (*m*−*1*) (*n*−*1*)
(*λx na.* ((*f-PostFD o1*) *o f o* (*f-PreFD* (*Solution i1 o1 m n f x*) *i1*)) *x na*))
proof −
have *inps-1*: *inps* (*FBlock* (*λx n. True*) *m n f*) = *m*
using *s1* **by** (*simp add*: *inps-P*)
have *outps-1*: *outps* (*FBlock* (*λx n. True*) *m n f*) = *n*
using *s1* **by** (*simp add*: *outps-P*)
have *i1-lt-m*: *i1* < *m*
using *s2* **by** (*simp add*: *Solvable-def*)
have *o1-lt-n*: *o1* < *n*
using *s2* **by** (*simp add*: *Solvable-def*)
have *1*: (*FBlock* (*λx n. True*) *m n f*) f_D (*i1*, *o1*) = (*true* $\vdash_n$ (∃ *x* ·
(∀ *n* · $\#_u$(\$*inouts*(«*n*»)$_a$) $=_u$ «*m* − *Suc 0*» ∧
$\#_u$(\$*inouts*´(«*n*»)$_a$) $=_u$ «*m*» ∧ \$*inouts*´(«*n*»)$_a$ $=_u$ «*f-PreFD x i1*»(\$*inouts*)$_a$(«*n*»)$_a$)
; ;
((∀ *na* · $\#_u$(\$*inouts*(«*na*»)$_a$) $=_u$ «*m*» ∧
$\#_u$(\$*inouts*´(«*na*»)$_a$) $=_u$ «*n*» ∧ «*f*»(\$*inouts*)$_a$(«*na*»)$_a$ $=_u$ \$*inouts*´(«*na*»)$_a$) ∧
(∀ *x* · ∀ *na* · $\#_u$(«*x na*») $=_u$ «*m*» ⇒ $\#_u$(«*f x na*») $=_u$ «*n*»)) ; ;
(∀ *na* · $\#_u$(\$*inouts*(«*na*»)$_a$) $=_u$ «*n*» ∧
$\#_u$(\$*inouts*´(«*na*»)$_a$) $=_u$ «*n* − *Suc 0*» ∧
\$*inouts*´(«*na*»)$_a$ $=_u$ «*f-PostFD o1*»(\$*inouts*)$_a$(«*na*»)$_a$ ∧
«*uapply*»(\$*inouts*(«*na*»)$_a$)$_a$(«*o1*»)$_a$ $=_u$ «*x na*»)))
apply (*simp add*: *inps-1 outps-1*)
apply (*simp add*: *PreFD-def PostFD-def FBlock-def Solution-def*)
apply (*simp add*: *ndesign-composition-wp wp-upred-def*)
by (*rel-simp*)
have *2*: (*true* $\vdash_n$ (∃ *x* ·
(∀ *n* · $\#_u$(\$*inouts*(«*n*»)$_a$) $=_u$ «*m* − *Suc 0*» ∧
$\#_u$(\$*inouts*´(«*n*»)$_a$) $=_u$ «*m*» ∧ \$*inouts*´(«*n*»)$_a$ $=_u$ «*f-PreFD x i1*»(\$*inouts*)$_a$(«*n*»)$_a$)
; ;
((∀ *na* · $\#_u$(\$*inouts*(«*na*»)$_a$) $=_u$ «*m*» ∧
$\#_u$(\$*inouts*´(«*na*»)$_a$) $=_u$ «*n*» ∧ «*f*»(\$*inouts*)$_a$(«*na*»)$_a$ $=_u$ \$*inouts*´(«*na*»)$_a$) ∧
(∀ *x* · ∀ *na* · $\#_u$(«*x na*») $=_u$ «*m*» ⇒ $\#_u$(«*f x na*») $=_u$ «*n*»)) ; ;
(∀ *na* · $\#_u$(\$*inouts*(«*na*»)$_a$) $=_u$ «*n*» ∧
$\#_u$(\$*inouts*´(«*na*»)$_a$) $=_u$ «*n* − *Suc 0*» ∧
\$*inouts*´(«*na*»)$_a$ $=_u$ «*f-PostFD o1*»(\$*inouts*)$_a$(«*na*»)$_a$ ∧
«*uapply*»(\$*inouts*(«*na*»)$_a$)$_a$(«*o1*»)$_a$ $=_u$ «*x na*»)))
⊑ (*FBlock* (*λx n. True*) (*m*−*1*) (*n*−*1*)
(*λx na.* ((*f-PostFD o1*) *o f o* (*f-PreFD* (*Solution i1 o1 m n f x*) *i1*)) *x na*))
apply (*simp add*: *FBlock-def Solution-def*)
apply (*rule ndesign-refine-intro*, *simp*+)
apply (*rel-simp*)

apply (*rule-tac* $x = (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))$
$i1\ inouts_v\ na$ **in** *exI*, *simp*)
apply (*rule conjI*)
apply (*simp add*: *f-PreFD-def*)
using *i1-lt-m* **apply** *linarith*
apply (*rule-tac* $x = \lambda na.\ (f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))$
$i1\ inouts_v)\ na)$ **in** *exI*, *simp*)
apply (*rule conjI*)
apply (*simp add*: *f-PreFD-def*)
apply (*rule conjI*)
using *i1-lt-m* **apply** *linarith*

defer
apply (*rule conjI*)
using *SimBlock-FBlock-fn s1* **apply** *blast*
apply (*rule allI*, *rule conjI*)

defer
proof −
fix $inouts_v$::*nat* ⇒ *real list* **and** $inouts_v{}'$::*nat* ⇒ *real list* **and** *x*::*nat*
assume *a1*: $\forall x.\ length(inouts_v\ x) = m - Suc\ 0\ \wedge$
$length(inouts_v{}'\ x) = n - Suc\ 0\ \wedge$
$f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))\ i1\ inouts_v))$
$x = inouts_v{}'\ x$
let $?P = \lambda xx.\ \ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1)$
have *1*: $(?P\ (SOME\ xx.\ ?P\ xx))$
apply (*rule someI-ex*[*of ?P*])
using *s2* **apply** (*simp add*: *Solvable-def*)
using *a1* **by** *blast*
show $f\ (f\text{-}PreFD\ (SOME\ xx.\ ?P\ xx)\ i1\ inouts_v)\ x!(o1) = (SOME\ xx.\ ?P\ xx)\ x$
by (*simp add*: *1*)
next
fix $inouts_v\ inouts_v{}'$
assume *a1*: $\forall x.\ length(inouts_v\ x) = m - Suc\ 0\ \wedge$
$length(inouts_v{}'\ x) = n - Suc\ 0\ \wedge$
$f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))\ i1\ inouts_v))$
$x =$
$inouts_v{}'\ x$
assume *a2*: $\forall x\ xa.\ length(x\ xa) = m - Suc\ 0 \longrightarrow$
$length(f\text{-}PostFD\ o1\ (f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ x)\ n!(o1))\ i1\ x))$
$xa) =$
$n - Suc\ 0$
from *a1* **have** *a1'*: $\forall x.\ length(inouts_v\ x) = m - Suc\ 0$
by (*simp*)
have $\forall na.\ length((f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))\ i1\ inouts_v)$
$na) = m$
using *a1' f-PreFD-def* **apply** (*simp*)
using *i1-lt-m* **by** *linarith*
then show $\forall x.\ length(f\ (f\text{-}PreFD\ (SOME\ xx.\ \forall n.\ xx\ n = f\ (f\text{-}PreFD\ xx\ i1\ inouts_v)\ n!(o1))\ i1$
$inouts_v)\ x) = n$
using *SimBlock-FBlock-fn s1* **by** *blast*
next
fix $inouts_v\ inouts_v{}'\ x$
assume *a1*: $\forall x.\ length(inouts_v\ x) = m - Suc\ 0\ \wedge$

length(*inouts*$_v$′ *x*) = *n* − *Suc 0* ∧
f-PostFD o1 (*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1 inouts*$_v$) *n*!(*o1*)) *i1 inouts*$_v$))
x =
inouts$_v$′ *x*
assume *a2*: ∀ *x xa*. *length*(*x xa*) = *m* − *Suc 0* ⟶
length(*f-PostFD o1* (*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1 x*) *n*!(*o1*)) *i1 x*))
xa) =
n − *Suc 0*
from *a1* **have** *a1*′: ∀ *x*. *length*(*inouts*$_v$ *x*) = *m* − *Suc 0*
by (*simp*)
have ∀ *na*. *length*((*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1 inouts*$_v$) *n*!(*o1*)) *i1 inouts*$_v$)
na) = *m*
using *a1*′ *f-PreFD-def* **apply** (*simp*)
using *i1-lt-m* **by** *linarith*
then show *length*(*f* (*f-PreFD* (*SOME xx*. ∀ *n*. *xx n* = *f* (*f-PreFD xx i1 inouts*$_v$) *n*!(*o1*)) *i1*
inouts$_v$) *x*) = *n*
using *SimBlock-FBlock-fn s1* **by** *blast*
qed
show *?thesis*
by (*metis 1 2*)
qed

lemma *SimBlock-FBlock-feedback* [*simblock-healthy*]:
assumes *s1*: *SimBlock m n* (*FBlock* (λ*x n*. *True*) *m n f*)
assumes *s2*: *Solvable i1 o1 m n* (*f*)
shows *SimBlock* (*m*−*1*) (*n*−*1*) ((*FBlock* (λ*x n*. *True*) *m n f*) *f*$_D$ (*i1*, *o1*))
proof −
have *m1-ge-0*: (*m* − (*Suc 0*)) ≥ *0*
using *s2* **by** (*simp add*: *Solvable-def*)
have *m1-gt-0*: *m* > *0*
using *s2* **by** (*simp add*: *Solvable-def*)
have *inps-1*: *inps* (*FBlock* (λ*x n*. *True*) *m n f*) = *m*
using *inps-outps s1* **by** *blast*
have *outps-1*: *outps* (*FBlock* (λ*x n*. *True*) *m n f*) = *n*
using *inps-outps s1* **by** *blast*
have *i1-le-m*: *i1* ≤ *m* − *Suc 0*
using *s2* **apply** (*simp add*: *Solvable-def*)
by *linarith*
have *o1-le-n*: *o1* ≤ *n* − *Suc 0*
using *s2* **apply** (*simp add*: *Solvable-def*)
by *linarith*
obtain *inouts*$_0$::*nat* ⇒ *real list* **where** *P0*: ∀ *x*. *length*(*inouts*$_0$ *x*) = (*m* − *1*)
using *m1-gt-0 list-len-avail*
by *blast*
have (∀ *inouts*$_0$. (∀ *x*. *length*(*inouts*$_0$ *x*) = (*m*−*1*))
⟶ (∃ *xx*.
(∀ *n*. (*xx n* =
(*f* (λ*n1*.
((*take i1* (*inouts*$_0$ *n1*))•(*xx n1*)#(*drop i1* (*inouts*$_0$ *n1*)))
) *n*)!*o1*
)
)))
using *s2* **by** (*simp add*: *Solvable-def f-PreFD-def*)
then have *1*: ∃ *xx*. (∀ *n*. (*xx n* = (*f* (λ*n1*. ((*take i1* (*inouts*$_0$ *n1*))•(*xx n1*)#(*drop i1* (*inouts*$_0$ *n1*))))
n)!*o1*))

apply (*simp*)
using *P0* **by** *simp*
obtain *xx*::*nat* ⇒ *real*
where *P1*: ($\forall n.$ $(xx\ n = (f\ (\lambda n1.\ ((take\ i1\ (inouts_0\ n1))\bullet(xx\ n1)\#(drop\ i1\ (inouts_0\ n1))))\ n)!o1$
))
using *1 P0* **by** *blast*
have *2*: $Suc\ (m - Suc\ 0) = m$
using *m1-gt-0* **by** *simp*
show *?thesis*
apply (*simp add*: *SimBlock-def inps-1 outps-1 PreFD-def PostFD-def*)
apply (*simp add*: *FBlock-def*)
apply (*rel-auto*)
apply (*simp add*: *f-blocks*)

apply (*rule-tac* $x = inouts_0$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.$
$(remove\text{-}at\ (f\ (\lambda n1.\ ((take\ i1\ (inouts_0\ n1))\bullet[xx\ n1]\bullet(drop\ i1\ (inouts_0\ n1))))\ na)\ o1)$ **in** *exI*)
apply (*rule-tac* $x = xx$ **in** *exI*)
apply (*rule-tac* $x = True$ **in** *exI*, *simp*)
apply (*rule-tac* $x = \lambda na.$ (
$(\lambda n1.\ ((take\ i1\ (inouts_0\ n1))\bullet[xx\ n1]\bullet(drop\ i1\ (inouts_0\ n1))))\ na)$ **in** *exI*)
apply (*simp*)
apply (*rule conjI*)
apply (*rule allI*)
apply (*rule conjI*)
using *P0* **apply** (*simp*)
apply (*simp add*: *2 P0*)
apply (*rule-tac* $x = True$ **in** *exI*, *simp*)
apply (*rule-tac* $x = \lambda na.$
$((f\ (\lambda n1.\ ((take\ i1\ (inouts_0\ n1))\bullet[xx\ n1]\bullet(drop\ i1\ (inouts_0\ n1))))\ na))$ **in** *exI*)
apply (*simp*)
apply (*rule conjI*)
using *2 P0 SimBlock-FBlock-fn s1*
apply (*smt One-nat-def add-Suc-right append-take-drop-id length-Cons length-append*)
apply (*rule conjI*)
using *SimBlock-FBlock-fn s1* **apply** *blast*
apply (*rule allI*)
apply (*rule conjI*)
using *SimBlock-FBlock-fn s1*
apply (*smt 2 One-nat-def P0 add-Suc-right append-take-drop-id length-Cons length-append*)
apply (*rule conjI*)
defer
using *P1* **apply** *metis*
proof −
fix *x*
have *1*: $length(f\ (\lambda n1.\ take\ i1\ (inouts_0\ n1)\bullet xx\ n1\ \#\ drop\ i1\ (inouts_0\ n1))\ x) = n$
using *2 P0 SimBlock-FBlock-fn s1*
by (*smt One-nat-def add-Suc-right append-take-drop-id length-Cons length-append*)
show $min\ (length(f\ (\lambda n1.\ take\ i1\ (inouts_0\ n1)\bullet xx\ n1\ \#\ drop\ i1\ (inouts_0\ n1))\ x))\ o1 +$
$(length(f\ (\lambda n1.\ take\ i1\ (inouts_0\ n1)\bullet xx\ n1\ \#\ drop\ i1\ (inouts_0\ n1))\ x) - Suc\ o1) =$
$n - Suc\ 0$
apply (*simp add*: *1*)
using *o1-le-n* **by** *linarith*
qed
qed

B.4.5 Split

lemma *SimBlock-Split2* [*simblock-healthy*]:
  SimBlock 1 2 (*Split2*)
  apply (*simp add*: *f-sim-blocks*)
  apply (*rule SimBlock-FBlock*)
  apply (*simp add*: *f-blocks*)
  apply (*rule-tac x* = *λna*. [*1*] **in** *exI*)
  apply *force*
  by (*simp add*: *f-blocks*)

B.5 Blocks

B.5.1 Source

B.5.1.1 Const

lemma *SimBlock-Const* [*simblock-healthy*]:
  SimBlock 0 1 (*Const c0*)
  apply (*simp add*: *f-sim-blocks*)
  apply (*rule SimBlock-FBlock*)
  apply (*simp add*: *f-blocks*)
  apply (*rule-tac x* = *λna*. [] **in** *exI*)
  apply *force*
  by (*simp add*: *f-blocks*)

B.5.1.2 Pulse Generator

B.5.2 Unit Delay

lemma *SimBlock-UnitDelay* [*simblock-healthy*]:
  SimBlock 1 1 (*UnitDelay x0*)
  apply (*simp add*: *f-sim-blocks*)
  apply (*rule SimBlock-FBlock*)
  apply (*simp add*: *f-blocks*)
  apply (*rule-tac x* = *λna*. [*1*] **in** *exI*)
  apply (*rule-tac x* = *λna*. [*if na* = *0 then x0 else 1*] **in** *exI*)
  apply (*simp*)
  by (*simp add*: *f-blocks*)

B.5.3 Discrete-Time Integrator

B.5.4 Sum

lemma *SimBlock-Sum2* [*simblock-healthy*]:
  SimBlock 2 1 (*Sum2*)
  apply (*simp add*: *f-sim-blocks*)
  apply (*rule SimBlock-FBlock*)
  apply (*simp add*: *f-blocks*)
  apply (*rule-tac x* = *λna*. [*1*,*1*] **in** *exI*)
  apply (*rule-tac x* = *λna*. [*2*] **in** *exI*)
  apply (*simp*)
  by (*simp add*: *f-blocks*)

B.5.5 Product

lemma *SimBlock-Mul2* [*simblock-healthy*]:
  SimBlock 2 1 (*Mul2*)
  apply (*simp add*: *f-sim-blocks*)

apply (*rule SimBlock-FBlock*)
apply (*simp add*: *f-blocks*)
apply (*rule-tac* $x = \lambda na.$ [*1*,*1*] **in** *exI*)
apply (*rule-tac* $x = \lambda na.$ [*1*] **in** *exI*)
apply (*simp*)
by (*simp add*: *f-blocks*)

lemma *SimBlock-Div2* [*simblock-healthy*]:
SimBlock 2 1 (*Div2*)
apply (*simp add*: *f-sim-blocks*)
apply (*simp add*: *SimBlock-def FBlock-def*)
apply (*rel-auto*)
apply (*rule-tac* $x = \lambda na.$ [*1*,*1*] **in** *exI*)
apply (*simp*)
apply (*rule conjI*)
apply (*rule-tac* $x = \lambda na.$ [*1*] **in** *exI*)
apply (*simp add*: *f-blocks*)
by (*simp add*: *f-blocks*)

B.5.6 Gain

lemma *SimBlock-Gain* [*simblock-healthy*]:
SimBlock 1 1 (*Gain k*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*simp add*: *f-blocks*)
apply (*rule-tac* $x = \lambda na.$ [*1*] **in** *exI*)
apply (*rule-tac* $x = \lambda na.$ [*k*] **in** *exI*)
apply (*simp*)
by (*simp add*: *f-blocks*)

B.5.7 Saturation

lemma *SimBlock-Limit* [*simblock-healthy*]:
assumes $ymin \leq ymax$
shows *SimBlock 1 1* (*Limit ymin ymax*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*simp add*: *f-blocks*)
apply (*rule-tac* $x = \lambda na.$ [*ymin*] **in** *exI*)
apply (*rule-tac* $x = \lambda na.$ [*ymin*] **in** *exI*)
using *assms* **apply** (*simp*)
by (*simp add*: *f-blocks*)

B.5.8 MinMax

lemma *SimBlock-Min2* [*simblock-healthy*]:
shows *SimBlock 2 1* (*Min2*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*simp add*: *f-blocks*)
apply (*rule-tac* $x = \lambda na.$ [*1*,*2*] **in** *exI*)
apply (*rule-tac* $x = \lambda na.$ [*1*] **in** *exI*)
apply (*simp*)
by (*simp add*: *f-blocks*)

lemma *SimBlock-Max2* [*simblock-healthy*]:
  shows *SimBlock 2 1* (*Max2*)
  apply (*simp add*: *f-sim-blocks*)
  apply (*rule SimBlock-FBlock*)
  apply (*simp add*: *f-blocks*)
  apply (*rule-tac x* = λ*na*. [*1*,*2*] **in** *exI*)
  apply (*rule-tac x* = λ*na*. [*2*] **in** *exI*)
  apply (*simp*)
  by (*simp add*: *f-blocks*)

B.5.9 Rounding

lemma *SimBlock-RoundFloor* [*simblock-healthy*]:
  shows *SimBlock 1 1* (*RoundFloor*)
  apply (*simp add*: *f-sim-blocks*)
  apply (*rule SimBlock-FBlock*)
  apply (*simp add*: *f-blocks*)
  apply (*rule-tac x* = λ*na*. [*1*] **in** *exI*)
  apply (*rule-tac x* = λ*na*. [*1*] **in** *exI*)
  apply *auto*[*1*]
  by (*simp add*: *f-blocks*)

lemma *SimBlock-RoundCeil* [*simblock-healthy*]:
  shows *SimBlock 1 1* (*RoundCeil*)
  apply (*simp add*: *f-sim-blocks*)
  apply (*rule SimBlock-FBlock*)
  apply (*simp add*: *f-blocks*)
  apply (*rule-tac x* = λ*na*. [*1*] **in** *exI*)
  apply (*rule-tac x* = λ*na*. [*1*] **in** *exI*)
  apply *auto*[*1*]
  by (*simp add*: *f-blocks*)

B.5.10 Combinatorial Logic

B.5.11 Logic Operators

B.5.11.1 AND

lemma *LAnd* [*1*,*1*] = *True*
  by *auto*

lemma *LAnd* [*1*,*1*,*0*] = *False*
  by *auto*

lemma *LAnd-and-not*: *LAnd* [*a*,*b*] = ($a \neq 0 \wedge b \neq 0$)
  by (*simp*)

lemma *LAnd-not-or*: *LAnd* [*a*,*b*] = ($\neg$ ($a = 0 \vee b = 0$))
  by (*simp*)

lemma *SimBlock-LopAND* [*simblock-healthy*]:
  assumes *s1*: $m > 0$
  shows *SimBlock m 1* (*LopAND m*)
  apply (*simp add*: *f-sim-blocks*)
  apply (*rule SimBlock-FBlock*)
  proof −
    obtain $inouts_v$::*nat* $\Rightarrow$ *real list*
    where *P*: $\forall$ *na*. *length*($inouts_v$ *na*) = *m* $\wedge$ ($\forall x<m$. (($inouts_v$ *na*)!*x* = *0*))

using *list-len-avail'* **by** *fastforce*
have *1*: $(\forall x<m.\ ((inouts_v\ na)!x = 0))$
using *P* **by** *blast*
have *2*: $length(inouts_v\ na) = m$
using *P* **by** *blast*
from *1 2* **have** *3*: $(LAnd\ (inouts_v\ x) = False)$
using *P s1* **by** (*metis LAnd.simps(2) hd-Cons-tl length-0-conv neq0-conv nth-Cons-0*)
show $\exists\, inouts_v\ inouts_v'.$
$\forall x.\ length(inouts_v'\ x) = Suc\ 0 \wedge length(inouts_v\ x) = m \wedge f\text{-}LopAND\ inouts_v\ x = inouts_v'\ x$
apply (*rule-tac* $x = inouts_v$ **in** *exI*)
apply (*simp add*: *f-blocks*)
apply (*rule-tac* $x = \lambda na.\ [0]$ **in** *exI*)
using *P 3*
by (*metis* (*full-types*) *LAnd.simps(2) hd-Cons-tl length-0-conv length-Cons nth-Cons-0 s1*)
next
show $\forall x\ na.\ length(x\ na) = m \longrightarrow length(f\text{-}LopAND\ x\ na) = Suc\ 0$
by (*simp add*: *f-blocks*)
qed

B.5.11.2 OR

lemma *LOr* $[0,0] = False$
by *auto*

lemma *LOr* $[0,1,0] = True$
by *auto*

lemma *SimBlock-LopOR* [*simblock-healthy*]:
assumes *s1*: $m > 0$
shows *SimBlock m 1* (*LopOR m*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
proof −
obtain $inouts_v::nat \Rightarrow real\ list$
where *P*: $\forall na.\ length(inouts_v\ na) = m \wedge (\forall x<m.\ ((inouts_v\ na)!x = 1))$
using *list-len-avail'* **by** *fastforce*
have *1*: $(\forall x<m.\ ((inouts_v\ na)!x = 1))$
using *P* **by** *blast*
have *2*: $length(inouts_v\ na) = m$
using *P* **by** *blast*
from *1 2* **have** *3*: $(LOr\ (inouts_v\ x) = True)$
using *P s1*
by (*metis LOr.elims(3) length-0-conv neq0-conv nth-Cons-0 zero-neq-one*)
show $\exists\, inouts_v\ inouts_v'.$
$\forall x.\ length(inouts_v'\ x) = Suc\ 0 \wedge length(inouts_v\ x) = m \wedge f\text{-}LopOR\ inouts_v\ x = inouts_v'\ x$
apply (*rule-tac* $x = inouts_v$ **in** *exI*)
apply (*simp add*: *f-blocks*)
apply (*rule-tac* $x = \lambda na.\ [1]$ **in** *exI*)
using *P 3*
by (*metis* (*full-types*) *LOr.simps(2) hd-Cons-tl length-0-conv length-Cons nth-Cons-0 s1*)
next
show $\forall x\ na.\ length(x\ na) = m \longrightarrow length(f\text{-}LopOR\ x\ na) = Suc\ 0$
by (*simp add*: *f-blocks*)
qed

B.5.11.3 NAND

lemma *LNand* $[1,1] = False$
by *auto*

lemma *LNand* [*1*,*1*,*0*] = *True*
by *auto*

lemma *SimBlock-LopNAND* [*simblock-healthy*]:
assumes *s1*: $m > 0$
shows *SimBlock m 1* (*LopNAND m*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
proof −
obtain $inouts_v$::*nat* ⇒ *real list*
where *P*: $\forall na.\ length(inouts_v\ na) = m \land (\forall x<m.\ ((inouts_v\ na)!x = 0))$
using *list-len-avail'* **by** *fastforce*
have *1*: $(\forall x<m.\ ((inouts_v\ na)!x = 0))$
using *P* **by** *blast*
have *2*: $length(inouts_v\ na) = m$
using *P* **by** *blast*
from *1 2* **have** *3*: $(LNand\ (inouts_v\ x) = True)$
using *P s1*
by (*metis LNand.elims*(*3*) *length-0-conv neq0-conv nth-Cons-0*)
show $\exists\, inouts_v\ inouts_v{}'.$
$\forall x.\ length(inouts_v{}'\ x) = Suc\ 0 \land length(inouts_v\ x) = m \land f\text{-}LopNAND\ inouts_v\ x = inouts_v{}'\ x$
apply (*rule-tac* $x = inouts_v$ **in** *exI*)
apply (*simp add*: *f-blocks*)
apply (*rule-tac* $x = \lambda na.\ [1]$ **in** *exI*)
using *P 3*
by (*metis* (*full-types*) *LNand.simps*(*2*) *hd-Cons-tl length-0-conv length-Cons nth-Cons-0 s1*)
next
show $\forall x\ na.\ length(x\ na) = m \longrightarrow length(f\text{-}LopNAND\ x\ na) = Suc\ 0$
by (*simp add*: *f-blocks*)
qed

B.5.11.4 NOR

lemma *LNor* [*1*,*0*] = *False*
by *auto*

lemma *LNor* [*0*,*0*,*0*] = *True*
by *auto*

B.5.11.5 XOR

lemma *LXor* [*1*,*0*] *0* = *True*
by *auto*

lemma *LXor* [*1*,*0*,*1*] *0* = *False*
by *auto*

B.5.11.6 NXOR

lemma *LNxor* [*1*,*0*] *0* = *False*
by *auto*

lemma *LNxor* [*1*,*0*,*1*] *0* = *True*
by *auto*

B.5.11.7 NOT

lemma *SimBlock-LopNOT* [*simblock-healthy*]:
shows *SimBlock 1 1* (*LopNOT*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac* $x = \lambda na.\ [0]$ **in** *exI*)

apply (*rule-tac* $x = \lambda na.\ [1]$ **in** *exI*)
apply (*simp add*: *f-LopNOT-def*)
by (*simp add*: *f-blocks*)

B.5.12 Relational Operator

B.5.12.1 Equal == **lemma** *SimBlock-RopEQ* [*simblock-healthy*]:
shows *SimBlock 2 1* (*RopEQ*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac* $x = \lambda na.\ [0,0]$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ [1]$ **in** *exI*)
apply (*simp add*: *f-RopEQ-def*)
by (*simp add*: *f-blocks*)

B.5.12.2 Notequal = **lemma** *SimBlock-RopNEQ* [*simblock-healthy*]:
shows *SimBlock 2 1* (*RopNEQ*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac* $x = \lambda na.\ [0,0]$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ [0]$ **in** *exI*)
apply (*simp add*: *f-RopNEQ-def*)
by (*simp add*: *f-blocks*)

B.5.12.3 Less Than < **lemma** *SimBlock-RopLT* [*simblock-healthy*]:
shows *SimBlock 2 1* (*RopLT*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac* $x = \lambda na.\ [0,0]$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ [0]$ **in** *exI*)
apply (*simp add*: *f-RopLT-def*)
by (*simp add*: *f-blocks*)

B.5.12.4 Less Than or Equal to <= **lemma** *SimBlock-RopLE* [*simblock-healthy*]:
shows *SimBlock 2 1* (*RopLE*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac* $x = \lambda na.\ [0,0]$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ [1]$ **in** *exI*)
apply (*simp add*: *f-blocks*)
by (*simp add*: *f-blocks*)

B.5.12.5 Greater Than > **lemma** *SimBlock-RopGT* [*simblock-healthy*]:
shows *SimBlock 2 1* (*RopGT*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac* $x = \lambda na.\ [0,0]$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ [0]$ **in** *exI*)
apply (*simp add*: *f-blocks*)
by (*simp add*: *f-blocks*)

B.5.12.6 Greater Than or Equal to >= **lemma** *SimBlock-RopGE* [*simblock-healthy*]:
shows *SimBlock 2 1* (*RopGE*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)

apply (*rule-tac x* = λ*na*. [*0*,*0*] **in** *exI*)
apply (*rule-tac x* = λ*na*. [*1*] **in** *exI*)
apply (*simp add*: *f-blocks*)
by (*simp add*: *f-blocks*)

B.5.13 Switch

lemma *SimBlock-Switch1* [*simblock-healthy*]:
shows *SimBlock 3 1* (*Switch1 th*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac x* = λ*na*. [*0*,*th*,*1*] **in** *exI*)
apply (*rule-tac x* = λ*na*. [*0*] **in** *exI*)
apply (*simp add*: *f-blocks*)
by (*simp add*: *f-blocks*)

lemma *SimBlock-Switch2* [*simblock-healthy*]:
shows *SimBlock 3 1* (*Switch2 th*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac x* = λ*na*. [*0*,*th*+*1*,*1*] **in** *exI*)
apply (*rule-tac x* = λ*na*. [*0*] **in** *exI*)
apply (*simp add*: *f-blocks*)
by (*simp add*: *f-blocks*)

lemma *SimBlock-Switch3* [*simblock-healthy*]:
shows *SimBlock 3 1* (*Switch3*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac x* = λ*na*. [*0*,*1*,*1*] **in** *exI*)
apply (*rule-tac x* = λ*na*. [*0*] **in** *exI*)
apply (*simp add*: *f-blocks*)
by (*simp add*: *f-blocks*)

B.5.14 Merge

B.5.15 Subsystem

B.5.16 Enabled Subsystem

B.5.17 Triggered Subsystem

B.5.18 Enabled and Triggered Subsystem

B.5.19 Data Type Conversion

lemma *SimBlock-DataTypeConvUint32Zero* [*simblock-healthy*]:
shows *SimBlock 1 1* (*DataTypeConvUint32Zero*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac x* = λ*na*. [*3294967295.5*] **in** *exI*)
apply (*rule-tac x* = λ*na*. [*3294967295*] **in** *exI*)
apply (*simp add*: *f-blocks RoundZero-def uint32-def*)
by (*simp add*: *f-blocks*)

lemma *SimBlock-DataTypeConvInt32Zero* [*simblock-healthy*]:
shows *SimBlock 1 1* (*DataTypeConvInt32Zero*)
apply (*simp add*: *f-sim-blocks*)

apply (*rule SimBlock-FBlock*)
apply (*rule-tac* $x = \lambda na.\ [-4.5]$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ [-4]$ **in** *exI*)
apply (*simp add*: *f-blocks RoundZero-def int32-def*)
by (*simp add*: *f-blocks*)

B.5.20 Initial Condition (IC)

lemma *SimBlock-IC* [*simblock-healthy*]:
shows *SimBlock 1 1* (*IC x0*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
apply (*rule-tac* $x = \lambda na.\ [x0]$ **in** *exI*)
apply (*rule-tac* $x = \lambda na.\ [x0]$ **in** *exI*)
apply (*simp add*: *f-blocks*)
by (*simp add*: *f-blocks*)

B.5.21 Router Block

lemma *assembleOutput-len*:
$\forall x\ na.\ length(assembleOutput\ (x\ na)\ routes) = length(routes)$
apply (*auto*)
proof (*induction routes*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a routes*)
then show *?case*
by (*simp*)
qed

lemma *SimBlock-Router* [*simblock-healthy*]:
assumes *s1*: $length(routes) = m$
shows *SimBlock m m* (*Router m routes*)
apply (*simp add*: *f-sim-blocks*)
apply (*rule SimBlock-FBlock*)
proof −
obtain $inouts_v::nat \Rightarrow real\ list$
where *P*: $\forall na.\ length(inouts_v\ na) = m \wedge (\forall x{<}m.\ ((inouts_v\ na)!x = 0))$
using *list-len-avail'* **by** *fastforce*
have *1*: $(\forall x{<}m.\ ((inouts_v\ na)!x = 0))$
using *P* **by** *blast*
have *2*: $length(inouts_v\ na) = m$
using *P* **by** *blast*
have *3*: $\forall x.\ length(assembleOutput\ (inouts_v\ x)\ routes) = length(routes)$
by (*simp add*: *assembleOutput-len*)
then have *4*: $\forall x.\ length(assembleOutput\ (inouts_v\ x)\ routes) = m$
using *s1* **by** *simp*
show $\exists\, inouts_v\ inouts_v'.$
$\forall x.\ length(inouts_v'\ x) = m \wedge length(inouts_v\ x) = m \wedge f\text{-}Router\ routes\ inouts_v\ x = inouts_v'\ x$
apply (*rule-tac* $x = inouts_v$ **in** *exI*)
apply (*rule-tac* $x = f\text{-}Router\ routes\ inouts_v$ **in** *exI*)
apply (*simp add*: *f-blocks*)
using *4 s1*
by (*simp add*: *P*)

next
  show $\forall\, x\ na.\ length(x\ na) = m \longrightarrow length(f\text{-}Router\ routes\ x\ na) = m$
    apply (*simp add*: *f-blocks*)
    using *s1* **by** (*simp add*: *assembleOutput-len*)
qed

B.6 Frequently Used Composition of Blocks

lemma *UnitDelay-Id-parallel-comp*:
$(UnitDelay\ 0 \parallel_B Id) = (FBlock\ (\lambda x\ n.\ True)\ (2)\ (2)$
    $(\lambda x\ n.\ [if\ n = 0\ then\ 0\ else\ hd(x\ (n-1)),\ hd(tl(x\ n))]))$
proof −
  have *f1*: $(UnitDelay\ 0 \parallel_B Id) = (FBlock\ (\lambda x\ n.\ True)\ (2)\ (2)$
    $(\lambda x\ n.\ ((((f\text{-}UnitDelay\ 0) \circ (\lambda xx\ nn.\ take\ 1\ (xx\ nn)))\ x\ n)$
      $\bullet\ ((f\text{-}Id \circ (\lambda xx\ nn.\ drop\ 1\ (xx\ nn))))\ x\ n)))$
    using *SimBlock-UnitDelay SimBlock-Id* **apply** (*simp add*: *FBlock-parallel-comp f-sim-blocks*)
    by (*simp add*: *numeral-2-eq-2*)
  then have *f1-0*: ... $= (FBlock\ (\lambda x\ n.\ True)\ (2)\ (2)$
    $(\lambda x\ n.\ [if\ n = 0\ then\ 0\ else\ hd(x\ (n-1)),\ hd(tl(x\ n))]))$
    proof −
      have $\forall\, (f{::}nat \Rightarrow real\ list)\ (n{::}nat).$
        $((\lambda x\ n.\ ((((f\text{-}UnitDelay\ 0) \circ (\lambda xx\ nn.\ take\ 1\ (xx\ nn)))\ x\ n)$
          $\bullet\ ((f\text{-}Id \circ (\lambda xx\ nn.\ drop\ 1\ (xx\ nn))))\ x\ n))\ f\ n =$
         $((\lambda x\ n.\ [if\ n = 0\ then\ 0\ else\ hd(x\ (n-1)),\ hd(tl(x\ n))])\ f\ n))$
        using *f-Id-def f-UnitDelay-def* **apply** (*simp*)
        by (*metis drop-0 drop-Suc list.sel(1) take-Nil take-Suc*)
      then show *?thesis*
        by *auto*
    qed
  then show *?thesis*
    by (*simp add*: *f1 f1-0*)
qed

end

References

[1] MathWorks, “Simulink.” [Online]. Available: https://uk.mathworks.com/products/simulink.html

[2] OSMC, “Openmodelica.” [Online]. Available: https://openmodelica.org/

[3] R. D. Arthan, P. Caseley, C. O’Halloran, and A. Smith, “Clawz: Control laws in Z,” in *3rd IEEE International Conference on Formal Engineering Methods, ICFEM 2000, York, England, UK, September 4-7, 2000, Proceedings.* IEEE Computer Society, 2000, pp. 169–176.

[4] P. Roy and N. Shankar, “Simcheck: a contract type system for simulink,” *Innovations in Systems and Software Engineering*, vol. 7, no. 2, p. 73, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1007/s11334-011-0145-4

[5] P. Boström and J. Wiik, “Contract-based verification of discrete-time multi-rate simulink models,” *Software and System Modeling*, vol. 15, no. 4, pp. 1141–1161, 2016.

[6] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis, “Translating discrete-time simulink to lustre,” in *Embedded Software, Third International Conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15, 2003, Proceedings*, ser. Lecture Notes in Computer Science, R. Alur and I. Lee, Eds., vol. 2855. Springer, 2003, pp. 84–99.

[7] A. Cavalcanti, P. Clayton, and C. O’Halloran, “Control law diagrams in *Circus*,” in *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, ser. Lecture Notes in Computer Science, J. S. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds., vol. 3582. Springer, 2005, pp. 253–268.

[8] V. Preoteasa, I. Dragomir, and S. Tripakis, “The refinement calculus of reactive systems,” *CoRR*, vol. abs/1710.03979, 2017. [Online]. Available: http://arxiv.org/abs/1710.03979

[9] F. Zeyda, J. Ouy, S. Foster, and A. Cavalcanti, “Formalising cosimulation models,” *Software Engineering and Formal Methods*, Jan. 2018. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-74781-1_31

[10] B. Meyer, “Applying "design by contract",” *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[11] C. B. Jones, *Wanted: a compositional approach to concurrency.* New York, NY: Springer New York, 2003, pp. 5–15. [Online]. Available: https://doi.org/10.1007/978-0-387-21798-7_1

[12] S. S. Bauer, A. David, R. Hennicker, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, “Moving from specifications to contracts in component-based design,” in *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, ser. Lecture Notes in Computer Science, J. de Lara and A. Zisman, Eds., vol. 7212. Springer, 2012, pp. 43–58.

[13] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, “Translating discrete-time simulink to lustre,” *ACM Trans. Embedded Comput. Syst.*, vol. 4, no. 4, pp. 779–818, 2005.

[14] J. Woodcock and A. Cavalcanti, “A tutorial introduction to designs in unifying theories of programming,” in *Integrated Formal Methods*, E. A. Boiten, J. Derrick, and G. Smith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 40–66.

[15] C. Hoare and J. He, *Unifying theories of programming.* Prentice Hall, 1998, vol. 14.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[17] S. Foster, F. Zeyda, and J. Woodcock, “Isabelle/utp: A mechanised theory engineering framework,” in *Unifying Theories of Programming - 5th International Symposium, UTP 2014, Singapore, May 13, 2014, Revised Selected Papers*, ser. Lecture Notes in Computer Science, D. Naumann, Ed., vol. 8963. Springer, 2014, pp. 21–41.

[18] M. Oliveira, A. Cavalcanti, and J. Woodcock, “A UTP Semantics for Circus,” *Formal Asp. Comput.*, vol. 21, no. 1-2, pp. 3–32, 2009.

[19] J.-R. Abrial, *The B-book: assigning programs to meanings.* Cambridge University Press, 2005.

[20] J. M. Spivey, *The Z Notation: A Reference Manual*, ser. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[21] N. Marian and Y. Ma, *Translation of Simulink Models to Component-based Software Models.* Forlag uden navn, 2007, pp. 274–280.

[22] A. Cavalcanti, A. Mota, and J. Woodcock, “Simulink timed models for program verification,” in *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, ser. Lecture Notes in Computer Science, Z. Liu, J. Woodcock, and H. Zhu, Eds., vol. 8051. Springer, 2013, pp. 82–99.

[23] A. Cavalcanti and J. Woodcock, “A tutorial introduction to CSP in *Unifying Theories of Programming*,” in *Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004, Revised Lectures*, ser. Lecture Notes in Computer Science, A. Cavalcanti, A. Sampaio, and J. Woodcock, Eds., vol. 3167. Springer, 2004, pp. 220–268.

[24] C. A. R. Hoare and A. W. Roscoe, “Programs as Executable Predicates,” in *FGCS*, 1984, pp. 220–228.

[25] V. Preoteasa and S. Tripakis, “Refinement calculus of reactive systems,” *CoRR*, vol. abs/1406.6035, 2014. [Online]. Available: http://arxiv.org/abs/1406.6035

[26] S. Foster, A. Cavalcanti, S. Canham, J. Woodcock, and F. Zeyda, “Unifying theories of reactive design contracts,” *In preparation for Theoretical Computer Science*, vol. abs/1712.10233, 2017.

[27] I. Dragomir, V. Preoteasa, and S. Tripakis, “Compositional semantics and analysis of hierarchical block diagrams,” in *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, D. Bosnacki and A. Wijs, Eds., vol. 9641. Springer, 2016, pp. 38–56.

[28] D. Bhatt, A. Chattopadhyay, W. Li, D. Oglesby, S. Owre, and N. Shankar, "Contract-based verification of complex time-dependent behaviors in avionic systems," in *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, ser. Lecture Notes in Computer Science, S. Rayadurgam and O. Tkachuk, Eds., vol. 9690. Springer, 2016, pp. 34–40.

[29] VeTSS, "Uk research institute in verified trustworthy software systems." [Online]. Available: https://vetss.org.uk/