Isabelle/UTP Tutorial

Simon Foster Frank Zeyda

September 5, 2018

Contents

1 Isabelle/UTP Primer

In this section, we will introduce Hoare and He's *Unifying Theories of Programming* [7] through a tutorial about our mechanisation, in Isabelle, called Isabelle/UTP [5, 3, 9]. The UTP is a framework for building and reasoning about heterogeneous semantics of programming and modelling languages. One of the core ideas of the UTP is that any program (or model) can be represented as a logical predicate over the program's state variables. The UTP thus begins from a higher-order logical core, and constructs a semantics for imperative relational programs, which can then be refined and extended with more complex language paradigms and theories. Isabelle/UTP mechanises this language of predicates and relations, and provides proof tactics for solving conjectures. For example, we can prove the following simple conjectures:

lemma $(true \wedge false) = false$
  by *pred-auto*

lemma $(true \Rightarrow P \wedge P) = P$
  by (*pred-auto*)

We discharge these using our predicate calculus tactic, *pred-auto*. It should be noted that *true*, *false*, and the conjunction operator are not simply the HOL operators; rather they act on on our UTP predicate type ($'\alpha$ *upred*).

1.1 State-spaces and Lenses

Predicates in the UTP are alphabetised, meaning they specify beheaviours in terms of a collection of variables, the alphabet, which effectively gives a state-space for a particular program. Thus the type of UTP predicates $'\alpha$ *upred* is parametric in the alphabet $'\alpha$. In Isabelle/UTP we can create a particular state-space with the **alphabet** command:

alphabet *myst* =
  x :: *int*
  y :: *int*
  z :: *int set*

This command creates an alphabet with three variables, x, y, and z, each of which has a defined type. A new Isabelle type is created, *myst*, which can be then used as the parameter for our predicate model, e.g. *myst upred*. In the context of our mechanisation, such variables are represented using *lenses* [5, 4]. A lens, $X : V \Longrightarrow S$, is a pair of functions, $get :: V \to S$ and $put : S \to V \to S$, where S is the source type, and V is the view type. The source type represents a "larger" type that can in some sense be subdivided, and the view type a particular region of the source that can be observed and manipulated independently of the rest of the source.

In Isabelle/UTP, the source type is the state space, and the view type is the variable type. For instance, we here have that x has type $int \Longrightarrow myst$ and z has type $int\ set \Longrightarrow myst$. Thus, performing an assignment to x equates to application of the *put* function, and looking up the present valuation is application of the *get* function.

Since the different variable characterise different regions of the state space we can distinguish them using the independence predicate $x \bowtie z$. Two lenses are independent if they characterise disjoint regions of the source type. In this case we can prove that the two variables are different using the simplifier:

lemma $x \bowtie z$
  by *simp*

However, we cannot prove, for example, that $x \bowtie x$ of course since the same region of the state-space is characterised by both. Lenses thus provide us with a semantic characterisation of variables, rather than a syntactic notion. For more background on this use of lenses please see our recent paper [5].

1.2 Predicate Calculus

We can now use this characterisation of variables to define predicates in Isabelle/UTP, for example $\&y <_u \&x$, which corresponds to all valuations of the state-space in which x is greater than y. Often we have to annotate our variables to help Isabelle understand that we are referring to UTP variables, and not, for example, HOL logical variables. In this case we have to decorate the names with an ampersand. Moreover, we often have to annotate operators with u subscripts to denote that they refer to the UTP version of the operator, and not the HOL version. We can now write down and prove a simple proof goal:

lemma $`(\&x =_u 8 \wedge \&y =_u 5) \Rightarrow \&x >_u \&y`$
  by (*pred-auto*)

The backticks denote that we are writing an tautology. Effectively this goal tells us that $x = 8$ and $y = 5$ are valid valuations for the predicate. Conversely the following goal is not provable.

lemma $`(\&x =_u 5 \wedge \&y =_u 5) \Rightarrow \&x >_u \&y`$
  apply (*pred-simp*) — Results in *False*
oops

We can similarly quantify over UTP variables as the following two examples illustrate.

lemma $(\exists\ x \cdot \&x >_u \&y) = true$
  by (*pred-simp*, *presburger*)

lemma $(\forall\ x \cdot \&x >_u \&y) = false$
  by (*pred-auto*)

The first goal states that for any given valuation of y there is a valuation of x which is greater. Predicate calculus alone is insufficient to prove this and so we can also use Isabelle's *sledgehammer* tool [1] which attempts to solve the goal using an array of automated theorem provers and SMT solvers. In this case it finds that Isabelle's tactic for Presburger arithmetic can solve the goal. In this second case we have a goal which states that every valuation of x is greater than a given valuation of y. Of course, this isn't the case and so we can prove the goal is equivalent to *false*.

1.3 Meta-logical Operators

In addition to predicate calculus operators, we also often need to assert meta-logical properties about a predicate, such as "variable x is not present in predicate P". In Isabelle/UTP we assert this property using the *unrestriction* operator, e.g. $x \sharp true$. Here are some examples of its use, including discharge using our tactic *unrest-tac*.

lemma $x \sharp true$
  by (*unrest-tac*)

lemma $x \sharp (\&y >_u 6)$
  by (*unrest-tac*)

lemma $x \sharp (\forall\ x \cdot \&x =_u \&y)$
  by (*unrest-tac*)

The tactic attempts to prove the unrestriction using a set of built-in unrestriction laws that exist for every operator of the calculus. The final example is interesting, because it shows we are not dealing with a syntactic property but rather a semantic one. Typically, one would describe the (non-)presence of variables syntactically, by checking if the syntax tree of P refers to x. In this case we are actually checking whether the valuation of P depends on x or not. In other words, if we can rewrite P to a form where x is not present, but P is otherwise equivalent, then x is unrestricted – it can take any value. The following example illustrates this:

lemma $x \sharp (\&x <_u 5 \vee \&x =_u 5 \vee \&x >_u 5)$

by (*pred-auto*)

Of course, if x is either less than 5, equal to 5, or greater than 5 then x can take any value and the predicate will still be satisfied. Indeed this predicate is actually equal to *true* and thus x is unrestricted. We will often use unrestriction to encode necessary side conditions on algebraic laws of programming.

In addition to presence of variables, we will often want to substitute a variable for an expression. We write this using the familiar syntax $P[\![v/x]\!]$, and also $P[\![v_1,v_2,v_3/x_1,x_2,x_3]\!]$ for an arbitrary number of expressions and variables. We can evaluate substitutions using the tactic *subst-tac* as the following examples show:

lemma $(\&y =_u \&x)[\![2/x]\!] = (\&y =_u 2)$
by (*subst-tac*)

lemma $(\&y =_u \&x \wedge \&y \in_u \&z)[\![2/y]\!] = (2 =_u \&x \wedge 2 \in_u \&z)$
by (*subst-tac*)

lemma $(\exists\ \&x \cdot \&x \in_u \&z)[\![76/\&x]\!] = (\exists\ \&x \cdot \&x \in_u \&z)$
by (*subst-tac*)

lemma $true[\![1,2/\&x,\&y]\!] = true$
by (*subst-tac*)

We can also, of course, combine substitution and predicate calculus to prove conjectures containing substitutions. Below, we prove a theorem in two steps, first applying the substitution, and then secondly utilising predicate calculus. Note the need to add the type coercion: *1* – this is because both 1 and 2 are polymorphic (e.g. they can also have type *real*), and thus explicit types must be added.

lemma $(\&x =_u 1 \wedge \&y =_u \&x)[\![2/x]\!] = false$ (**is** *?lhs* = *?rhs*)
proof –
have $?lhs = (2 =_u (1 :_u int) \wedge \&y =_u 2)$
by (*subst-tac*)
also have ... = *?rhs*
by (*pred-auto*)
finally show *?thesis* **.**
qed

Isabelle/UTP also supports a more general notation for substitutions, where-by variable maplets can be considered as explicit objects that correspond to functions on the state-space. For example, $[\&x \mapsto_s 1, \&y \mapsto_s 2]$ is a substitution that replaces x with 1 and y with 2. Such substitution functions (σ) can be applied to an expression or predicate using the dagger operator: $\sigma \dagger P$. Thus, we can perform the following simplification:

lemma $[\&x \mapsto_s 1, \&y \mapsto_s 2] \dagger (\&x <_u \&y) = ((1 :_u int) <_u 2)$
by (*subst-tac*)

This notation thus allows us to deal with state space much more flexibly. The more standard notation $P[\![v/x]\!]$ is in fact syntactic sugar to application of the singleton substitution function $[x \mapsto_s v]$.

So far, we have considered UTP predicates which contain only UTP variables. However it is possible to have another kind of variable – a logical HOL variable which is sometimes known as a “logical constant” [8]. Such variables are not program or model variables, but they simply exist to assert logical properties of a predicate. The next two examples compare UTP and HOL variables in a quantification.

lemma $(\forall\ x \cdot \&x =_u \&x) = \mathit{true}$
  by (*pred-auto*)

lemma $(\forall\ x \cdot \ll x \gg =_u \ll x \gg) = \mathit{true}$
  by (*pred-auto*)

The first quantification is a quantification of a UTP variable, which we've already encountered. The second is a quantifier over a HOL variable, denoted by the quantifier being bold. In addition we refer to HOL variables, not using the ampersand, but the quotes $\ll k \gg$. These quotes allow us to insert an arbitrary HOL term into a UTP expression, such as a logical variable.

1.4 Relations

Relations, $('\alpha,\ '\beta)$ *urel*, are a class of predicate in which the state space is a product – i.e. $('\alpha,\ '\beta)$ *urel* – and divides the variables into input or "before" variables and output or "after" variables. In Isabelle/UTP we can write down a relational variable using the dollar notation, as illustrated below:

term $(\$x' =_u 1 \wedge \$y' =_u \$y \wedge \$z' =_u \$z) :: \mathit{myst\ hrel}$

Type $'\alpha$ *hrel* is the type of homogeneous relations, which have the same before and after state. This example relation can be intuitively thought of as the relation which sets x to 1 and leaves the other two variables unchanged. We would normally refer to this as an assignment of course, and for convenience we can write such a predicate using a more convenient syntax, $x := 1$, which is equivalent:

lemma $(x := 1) = ((\$x' =_u 1 \wedge \$y' =_u \$y \wedge \$z' =_u \$z) :: \mathit{myst\ hrel})$
  by (*rel-auto*)

Since we are now in the world of relations, we have an additional tactic called *rel-auto* that solves conjectures in relational calculus. We can use relational variables to write to loose specifications for programs, and then prove that a given program is a refinement. Refinement is an order on programs that allows us to assert that a program refines a given specification, for example:

lemma $(\$x' >_u \$y) \sqsubseteq (x,\ y) := ((\&y + 3),\ 5)$
  by (*rel-auto*)

This tells us that the specification that the after value of x must be greater than the initial value of y, is refined by the program which adds 3 to y and assigns this to x, and simultaneously assigns 5 to y. Of course, this is not the only refinement, but an interesting one. A refinement conjecture $P \sqsubseteq Q$ in general asserts that Q is more deterministic than P. In addition to assignments, we can also construct relational specifications and programs using sequential (or relational) composition:

lemma $x := 1 \;;;\; x := (\&x + 1) = x := 2$
  by (*rel-auto*)

Internally, what is happening here is quite subtle, so we can also prove this law in the Isar proof scripting language which allows us to further expose the details of the argument. In this proof we will make use of both the tactic and already proven laws of programming from Isabelle/UTP.

lemma $x := 1 \;;;\; x := (\&x + 1) = x := 2$
proof –
  — We first show that a relational composition of an assignment and some program P corresponds to substitution of the assignment into P, which is proved using the law *assigns-r-comp*.
  have $x := 1 \;;;\; x := (\&x + 1) = (x := (\&x + 1))[\![1/\$x]\!]$

by (*simp add*: *assigns-r-comp alpha usubst*)
— Next we execute the substitution using the relational calculus tactic.
also have ... = $x := (1 + 1)$
by (*rel-auto*)
— Finally by evaluation of the expression, we obtain the desired result of 2.
also have ... = $x := 2$
by (*simp*)
finally show *?thesis* **.**
qed

UTP also gives us an if-then-else conditional construct, written $P \triangleleft b \triangleright Q$, which is a more concise way of writing **if** b **then** P **else** Q. It also allows the expression of while loops, which gives us a simple imperative programming language.

lemma $(x := 1 \;;; (y := 7 \triangleleft \$x >_u 0 \triangleright y := 8)) = (x,y) := (1,7)$
by (*rel-auto*)

Below is an illustration of how we can express a simple while loop in Isabelle/UTP.

term $(x,y) := (3,1);;$ *while* $(\&x >_u 0)$ *do* $x := \&x - 1;;\ y := \&y * 2$ *od*

1.5 Non-determinism and Complete Lattices

So far we have considered only deterministic programming operators. However, one of the key feature of the UTP is that it allows non-deterministic specifications. Determinism is ordered by the refinement order $P \sqsubseteq Q$, which states that P is more deterministic that Q, or alternatively that Q makes fewer commitments than P. The refinement order $P \sqsubseteq Q$ corresponds to a universally closed implication $Q \Rightarrow P$. The most deterministic specification is *false*, which also corresponds to a miraculous program, and the least is *true*, as the following theorems demonstrate.

theorem *false-greatest*: $P \sqsubseteq$ *false*
by (*rel-auto*)

theorem *true-least*: *true* $\sqsubseteq P$
by (*rel-auto*)

In this context *true* corresponds to a programmer error, such as an aborting or non-terminating program (the theory of relations does not distinguish these). We can similarly specify a non-deterministic choice between P and Q with $P \sqcap Q$, or alternatively $\bigsqcap A$ where A is a set of possible behaviours. Predicate $P \sqcap Q$ encapsulates the behaviours of both P and Q, and is thus refined by both. We can also prove a variety of theorems about non-deterministic choice.

theorem *Choice-equiv*:
fixes $P\ Q$ :: *'α upred*
shows $\bigsqcap \{P, Q\} = P \sqcap Q$
by *simp*

Theorem *Choice-equiv* shows the relationship between the big choice operator and its binary equivalent. The latter is simply a choice over a set with two elements.

theorem *Choice-refine*:
fixes $A\ B$ :: *'α upred set*
assumes $B \subseteq A$
shows $\bigsqcap A \sqsubseteq \bigsqcap B$
by (*simp add*: *Sup-subset-mono assms*)

The intuition of theorem *Choice-refine* is that a specification with more options is refined by one with less options. We can also prove a number of theorems about the binary version of the operator.

theorem *choice-thms*:
fixes $P\ Q :: {'\alpha}\ upred$
shows
$P \sqcap P = P$
$P \sqcap Q = Q \sqcap P$
$(P \sqcap Q) \sqcap R = P \sqcap (Q \sqcap R)$
$P \sqcap true = true$
$P \sqcap false = P$
$P \sqcap Q \sqsubseteq P$
by (*simp-all add*: *lattice-class.inf-sup-aci true-upred-def false-upred-def*)

Non-deterministic choice is idempotent, meaning that a choice between P and P is no choice. It is also commutative and associative. If we make a choice between P and *true* then the erroneous behaviour signified by the latter is always chosen. Thus our operator is a so-called "demonic choice" since the worst possibility is always picked. Similarly, if a choice is made between P and a miracle (*false*) then P is always chosen in order to avoid miracles. Finally, the choice between P and Q can always be refined by removing one of the possibilities.

Since predicates form a complete lattice, then by the Knaster-Tarski theorem the set of fixed points of a monotone function F is also a complete lattice. In particular, this complete lattice has a weakest and strongest element which can be calculated using the notations $\mu\ F$ and $\nu\ F$, respectively. Such fixed point constructions are of particular use for expressing recursive and iterative constructions. Isabelle/HOL provides a number of laws for reasoning about fixed points, a few of which are detailed below.

theorem *mu-id*: $(\mu\ X \cdot X) = true$
by (*simp add*: *mu-id*)

theorem *nu-id*: $(\nu\ X \cdot X) = false$
by (*simp add*: *nu-id*)

theorem *mu-unfold*: $mono\ F \Longrightarrow (\mu\ X \cdot F(X)) = F(\mu\ X \cdot F(X))$
by (*simp add*: *def-gfp-unfold*)

theorem *nu-unfold*: $mono\ F \Longrightarrow (\nu\ X \cdot F(X)) = F(\nu\ X \cdot F(X))$
by (*simp add*: *def-lfp-unfold*)

Perhaps of most interest are the unfold laws, also known as the "copy rule", that allows the function body F of the fixed point equation to be expanded once. These state that, provided that the body of the fixed point is a monotone function, then the body can be copied to the outside. These can be used to prove equivalent laws for operators like the while loop.

1.6 Laws of Programming

Although we have some primitive tactics for proving conjectures in the predicate and relational calculi, in order to build verification tools for programs we need a set of algebraic "laws of programming" [6] that describe important theoretical properties of the operators. Isabelle/UTP contains several hundred examples of such laws, and we here outline a few of them.

theorem *seq-assoc*: $(P\ ;;\ Q)\ ;;\ R = P\ ;;\ (Q\ ;;\ R)$
by (*rel-auto*)

theorem *seq-unit*:
$P\ ;;\ II = P$
$II\ ;;\ P = P$
by (*rel-auto*)+

theorem *seq-zero*:
$P\ ;;\ \mathit{false} = \mathit{false}$
$\mathit{false}\ ;;\ P = \mathit{false}$
by (*rel-auto*)+

Sequential composition is associative, has the operator II as its left and right unit, and *false* as its left and right zeros. The II operator is a form of assignment which simply identifies all the variables between the before and after state, as the following example demonstrates.

lemma $x := \&x = II$
by (*rel-auto*)

In the context of relations, *false* denotes the empty relation, and is usually used to represent a miraculous program. This is intuition of it being a left and right zero: if a miracle occurred then the whole of the program collapses. The conditional $P \triangleleft b \triangleright Q$ also has a number of algebraic laws that we can prove.

theorem *cond-true*: $P \triangleleft \mathit{true} \triangleright Q = P$
by (*rel-auto*)

theorem *cond-false*: $P \triangleleft \mathit{false} \triangleright Q = Q$
by (*rel-auto*)

theorem *cond-commute*: $(P \triangleleft \neg\ b \triangleright Q) = (Q \triangleleft b \triangleright P)$
by (*rel-auto*)

theorem *cond-shadow*: $(P \triangleleft b \triangleright Q) \triangleleft b \triangleright R = P \triangleleft b \triangleright R$
by (*rel-auto*)

A conditional with *true* or *false* as its condition presents no choice. A conditional can also be commuted by negating the condition. Finally, a conditional within a conditional over the same condition, b, presents and unreachable branch. Thus the inner branch can be pruned away. We next prove some useful laws about assignment:

theorem *assign-commute*:
assumes $x \bowtie y\ y \sharp e\ x \sharp f$
shows $x := e;;\ y := f = y := f\ ;;\ x := e$
using *assms* **by** (*rel-auto*)

theorem *assign-twice*:
shows $x := \ll e \gg;;\ x := \ll f \gg = x := \ll f \gg$
by (*rel-auto*)

theorem *assign-null*:
assumes $x \bowtie y$
shows $(x,\ y) := (e,\ \&y) = x := e$
using *assms* **by** (*rel-auto*)

Assignments can commute provided that the two variables are independent, and the expressions being assigned do not depend on the variable of the other assignment. A sequence of assignments to the same variable is equal to the second assignment, provided that the two expressions are both literals, i.e. $\ll e \gg$. Finally, in a multiple assignment, if one of the variables is assigned to itself then this can be hidden, provided the two variables are independent.

Since alphabetised relations form a complete lattice, we can denote iterative constructions like the while loop which is defined as $while_{\bot}\ b\ do\ P\ od = (\mu\ X \cdot P\ ;;\ X \triangleleft b \triangleright_r II)$. We can then prove some common laws about iteration.

theorem *while-false*: $while_{\bot}\ false\ do\ P\ od = II$
  by (*simp add*: *while-bot-false*)

theorem *while-unfold*: $while_{\bot}\ b\ do\ P\ od = (P\ ;;\ while_{\bot}\ b\ do\ P\ od) \triangleleft b \triangleright_r II$
  using *while-bot-unfold* **by** *blast*

As we have seen, the predicate *true* represents the erroneous program. For loops, we have it that a non-terminating program equates to *true*, as the following example demonstrates.

lemma $while_{\bot}\ true\ do\ x := (\&x + 1)\ od = true$
  by (*simp add*: *assigns-r-feasible while-infinite*)

A program should not be able to recover from non-termination, of course, and therefore it ought to be the case that *true* is a left zero for sequential composition: *true* ;; P = *true*. However this is not the case as the following examples illustrate:

lemma *true* ;; P = *true*
  apply (*rel-simp*)
  nitpick — Counterexample found
oops

lemma $true\ ;;\ (x,y,z) := (\ll c_1 \gg,\ll c_2 \gg,\ll c_3 \gg) = ((x,y,z) := (\ll c_1 \gg,\ll c_2 \gg,\ll c_3 \gg) :: myst\ hrel)$
  by (*rel-auto*)

The latter gives an example of a relation for which *true* is actually a left unit rather than a left zero. The assignment $\langle[\&x \mapsto_s \ll c_1 \gg, \&y \mapsto_s \ll c_2 \gg, \&z \mapsto_s \ll c_3 \gg]\rangle_a$ does not depend on any before variables, and thus it is insensitive to a non-terminating program preceding it. Thus we can see that the theory of relations alone is insufficient to handle non-termination.

1.7 Designs

Though we now have a theory of UTP relations with which can form simple programs, as we have seen this theory experiences some problems. A UTP design, $P \vdash_r Q$, is a relational specification in terms of assumption P and commitment Q. Such a construction states that, if P holds and the program is allowed to execute, then the program will terminate and satisfy its commitment Q. If P is not satisfied then the program will abort yielding the predicate *true*. For example the design $(\$x \neq_u 0) \vdash_r y := \&y\ div\ \&x$ represents a program which, assuming that $x \neq 0$ assigns y divided by x to y.

lemma *dex1*: $(true \vdash_r (x,y) := (2,6))\ ;;\ ((\$x \neq_u 0) \vdash_r (y := (\&y\ div\ \&x))) = true \vdash_r x,y := 2,3$
  by (*rel-auto*, *fastforce+*)

lemma *dex2*: $(true \vdash_r (x,y) := (0,4))\ ;;\ ((\$x \neq_u 0) \vdash_r y := (\&y\ div\ \&x)) = true$
  by (*rel-blast*)

The first example shows the result of pre-composing this design with another design that has a *true* assumption, and assigns 2 and 6 to x and y respectively. Since x satisfies $x \neq 0$, then the design executes and changes y to 3. In the second example 0 is assigned to x, which leads to the design aborting. Unlike with relations, designs do have *true* as a left zero:

theorem *design-left-zero*: $true \;;; (P \vdash_r Q) = true$
by (*simp add*: *H1-left-zero H1-rdesign Healthy-def*)

Thus designs allow us to properly handle programmer error, such as non-termination.

The design turnstile is defined using two observational variables $ok, ok' : \mathbb{B}$, which are used to represent whether a program has been (ok) and whether it has terminated (ok'). Specifically, a design $P \vdash Q$ is defined as $(ok \wedge P) \Rightarrow (ok' \wedge Q)$. This means that if the program was started (ok) and satisfied its assumption (P), then it will terminate (ok') and satisfy its commitment (Q). For more on the theory of designs please see the associated tutorial [2].

1.8 Reactive Designs

A reactive design, $\mathbf{R}_s\ (P \vdash Q)$, is a specialised form of design which is reactive in nature. Whereas designs represents programs that start and terminate, reactive designs also have intermediate "waiting" states. In such a state the reactive design is waiting for something external to occur before it can continue, such as receiving a message or waiting for sufficient time to pass as measured by a clock. When waiting, a reactive design has not terminated, but neither is it an infinite loop or some other error state.

Reactive designs have two additional pair of observational variables:

- $wait, wait' : \mathbb{B}$ – denote whether the predecessor is in a waiting state, and whether the current program is a waiting state;
- $tr, tr' : \mathcal{T}$ – denotes the interaction history using a suitable trace type $\mathcal{T}$.

For more details on reactive designs please see the associated tutorial [2].

2 Example UTP theory: Boyle's laws

In order to exemplify the use of Isabelle/UTP, we mechanise a simple theory representing Boyle's law. Boyle's law states that, for an ideal gas at fixed temperature, pressure p is inversely proportional to volume V, or more formally that for $k = p * V$ is invariant, for constant k. We here encode this as a simple UTP theory. We first create a record to represent the alphabet of the theory consisting of the three variables k, p and V.

alphabet *boyle* =
  k :: *real*
  p :: *real*
  V :: *real*

type-synonym *boyle-rel* = *boyle hrel*

declare *boyle.splits* [*alpha-splits*]

The two locale interpretations below are a technicality to improve automatic proof support via the predicate and relational tactics. This is to enable the (re-)interpretation of state spaces to remove any occurrences of lens types after the proof tactics *pred-simp* and *rel-simp*, or any of their derivatives have been applied. Eventually, it would be desirable to automate both interpretations as part of a custom outer command for defining alphabets.

interpretation *boyle-prd*: — Closed records are sufficient here.
  lens-interp $\lambda r{::}boyle.\ (k_v\ r,\ p_v\ r,\ V_v\ r)$

apply (*unfold-locales*)
apply (*rule injI*)
apply (*clarsimp*)
done

interpretation *boyle-rel*: — Closed records are sufficient here.
lens-interp $\lambda(r::boyle,\ r'::boyle).$
$(k_v\ r,\ k_v\ r',\ p_v\ r,\ p_v\ r',\ V_v\ r,\ V_v\ r')$
apply (*unfold-locales*)
apply (*rule injI*)
apply (*clarsimp*)
done

2.1 Static invariant

We first create a simple UTP theory representing Boyle's laws on a single state, as a static invariant healthiness condition. We state Boyle's law using the function B, which recalculates the value of the constant k based on p and V.

definition $B(\varphi) = ((\exists\ k \cdot \varphi) \wedge (\&k =_u \&p{*}\&V))$

We can then prove that B is both idempotent and monotone simply by application of the predicate tactic. Idempotence means that healthy predicates cannot be made more healthy. Together with idempotence, monotonicity ensures that image of the healthiness functions forms a complete lattice, which is useful to allow the representation of recursive and iterative constructions with the theory.

lemma *B-idempotent*: $B(B(P)) = B(P)$
by *pred-auto*

lemma *B-monotone*: $X \sqsubseteq Y \Longrightarrow B(X) \sqsubseteq B(Y)$
by *pred-blast*

We also create some example observations; the first (φ_1) satisfies Boyle's law and the second doesn't (φ_2).

definition $\varphi_1 = ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 50))$
definition $\varphi_2 = ((\&p =_u 10) \wedge (\&V =_u 5) \wedge (\&k =_u 100))$

We first prove an obvious property: that these two predicates are different observations. We must show that there exists a valuation of one which is not of the other. This is achieved through application of *pred-tac*, followed by *sledgehammer* [1] which yields a *metis* proof.

lemma φ_1-*diff*-φ_2: $\varphi_1 \neq \varphi_2$
by (*pred-simp*, *fastforce*)

We prove that φ_1 satisfies Boyle's law by application of the predicate calculus tactic, *pred-tac*.

lemma *B*-φ_1: φ_1 *is B*
by (*pred-auto*)

We prove that φ_2 does not satisfy Boyle's law by showing that applying B to it results in φ_1. We prove this using Isabelle's natural proof language, Isar.

lemma *B*-φ_2: $B(\varphi_2) = \varphi_1$
proof −
have $B(\varphi_2) = B(\&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u 100)$
by (*simp add*: φ_2-*def*)

also have ... = $((\exists\ k \cdot \&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u 100) \wedge \&k =_u \&p*\&V)$
  by (*simp add*: *B-def*)
  also have ... = $(\&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u \&p*\&V)$
  by *pred-auto*
  also have ... = $(\&p =_u 10 \wedge \&V =_u 5 \wedge \&k =_u 50)$
  by *pred-auto*
  also have ... = φ_1
  by (*simp add*: φ_1-*def*)
  finally show *?thesis* **.**
qed

2.2 Dynamic invariants

Next we build a relational theory that allows the pressure and volume to be changed, whilst still respecting Boyle's law. We create two dynamic invariants for this purpose.

definition $D1(P) = ((\$k =_u \$p*\$V \Rightarrow \$k' =_u \$p'*\$V') \wedge P)$
definition $D2(P) = (\$k' =_u \$k \wedge P)$

D1 states that if Boyle's law satisfied in the previous state, then it should be satisfied in the next state. We define this by conjunction of the formal specification of this property with the predicate. The annotations $\$p$ and $\$p'$ refer to relational variables p and p'. *D2* states that the constant k indeed remains constant throughout the evolution of the system, which is also specified as a conjunctive healthiness condition. As before we demonstrate that *D1* and *D2* are both idempotent and monotone.

lemma *D1-idempotent*: $D1(D1(P)) = D1(P)$ **by** *rel-auto*
lemma *D2-idempotent*: $D2(D2(P)) = D2(P)$ **by** *rel-auto*

lemma *D1-monotone*: $X \sqsubseteq Y \Longrightarrow D1(X) \sqsubseteq D1(Y)$ **by** *rel-auto*
lemma *D2-monotone*: $X \sqsubseteq Y \Longrightarrow D2(X) \sqsubseteq D2(Y)$ **by** *rel-auto*

Since these properties are relational, we discharge them using our relational calculus tactic *rel-tac*. Next we specify three operations that make up the signature of the theory.

definition *InitSys* :: *real* $\Rightarrow$ *real* $\Rightarrow$ *boyle-rel* **where**
  InitSys ip iV
  $= [\ll ip\gg >_u 0 \wedge \ll iV\gg >_u 0]^\top\ ;;\ (p,V,k) := (\ll ip\gg,\ll iV\gg,(\ll ip\gg*\ll iV\gg))$

definition *ChPres* :: *real* $\Rightarrow$ *boyle-rel* **where**
  ChPres dp
  $= ([\&p + \ll dp\gg >_u 0]^\top\ ;;\ p := (\&p + \ll dp\gg)\ ;;\ V := (\&k/\&p))$

definition *ChVol* :: *real* $\Rightarrow$ *boyle-rel* **where**
  ChVol dV
  $= ([\&V + \ll dV\gg >_u 0]^\top\ ;;\ V := (\&V + \ll dV\gg)\ ;;\ p := (\&k/\&V))$

InitSys initialises the system with a given initial pressure (*ip*) and volume (*iV*). It assumes that both are greater than 0 using the assumption construct $?[c]$ which equates to *II* if c is true and *false* (i.e. errant) otherwise. It then creates a state assignment for p and V, uses the B healthiness condition to make it healthy (by calculating k), and finally turns the predicate into a postcondition using the $\lceil P\rceil_>$ function.

ChPres raises or lowers the pressure based on an input *dp*. It assumes that the resulting pressure change would not result in a zero or negative pressure, i.e. $p + dp > 0$. It assigns the updated value to p and recalculates V using the original value of k. *ChVol* is similar but updates the volume.

lemma *D1-InitSystem*: $D1\ (InitSys\ ip\ iV) = InitSys\ ip\ iV$
by *rel-auto*

InitSys is *D1*, since it establishes the invariant for the system. However, it is not *D2* since it sets the global value of k and thus can change its value. We can however show that both *ChPres* and *ChVol* are healthy relations.

lemma *D1*: $D1\ (ChPres\ dp) = ChPres\ dp$ **and** $D1\ (ChVol\ dV) = ChVol\ dV$
by (*rel-auto*, *rel-auto*)

lemma *D2*: $D2\ (ChPres\ dp) = ChPres\ dp$ **and** $D2\ (ChVol\ dV) = ChVol\ dV$
by (*rel-auto*, *rel-auto*)

Finally we show a calculation a simple animation of Boyle's law, where the initial pressure and volume are set to 10 and 4, respectively, and then the pressure is lowered by 2.

lemma *ChPres-example*:
$(InitSys\ 10\ 4\ ;;\ ChPres\ (-2)) = p,V,k := 8,5,40$
proof −
— *InitSys* yields an assignment to the three variables
have $InitSys\ 10\ 4 = p,V,k := 10,4,40$
by (*rel-auto*)
— This assignment becomes a substitution
hence $(InitSys\ 10\ 4\ ;;\ ChPres\ (-2))$
$= (ChPres\ (-2))[\![10,4,40/\$p,\$V,\$k]\!]$
by (*simp add*: *assigns-r-comp alpha usubst*)
— Unfold definition of *ChPres*
also have $... = ([(\&p - 2) >_u 0]^\top [\![10,4,40/\$p,\$V,\$k]\!]$
$;;\ p := (\&p - 2)\ ;;\ V := (\&k\ /\ \&p))$
by (*simp add*: *ChPres-def lit-numeral lit-uminus usubst unrest*)
— Unfold definition of assumption
also have $... = ((p,V,k) := (10,4,40) \triangleleft (8 :_u real) >_u 0 \triangleright false)$
$;;\ p := (\&p - 2)\ ;;\ V := (\&k\ /\ \&p)$
by *rel-auto*
— $(0::'a) < (8::'a)$ is true; simplify conditional
also have $... = (p,V,k) := (10,4,40)\ ;;\ p := (\&p - 2)\ ;;\ V := (\&k\ /\ \&p)$
by *rel-auto*
— Application of both assignments
also have $... = p,V,k := 8,5,40$
by *rel-auto*
finally show *?thesis* **.**
qed

hide-const k
hide-const p
hide-const V
hide-const B

3 Simple Buffer in UTP CSP

theory *utp-csp-buffer*

imports *UTP−Circus.utp-circus*
begin

3.1 Definitions

A stateful CSP (Circus) process is parametrised over two alphabets: one for the state-space, which consists of the state variables, and one for events, which consists of channels. We first define the statespace using the **alphabet** command. The single state variable buf is a list of natural numbers that is currently in the buffer.

alphabet *st-buffer* =
buff :: *nat list*

Channels are created using the **datatype** command. In this case we can either input a value to go in the buffer, or output one presently in the buffer.

datatype *ch-buffer* =
inp nat | *outp nat*

We create a useful type to describe an action of the buffer as a CSP action parametrised by the state and event alphabet.

type-synonym *act-buffer* = (*st-buffer*, *ch-buffer*) *action*

We define an action that initialises the buffer state by setting it to empty.

abbreviation *Init* :: *act-buffer* **where**
$Init \equiv buff :=_C \langle\rangle$

We define the main body of behaviour for the buffer as an abbreviation. We can either input a value and then place it into the buffer, or else, provided that the buffer is non-empty, we can output a value presently in the buffer.

abbreviation *DoBuff* :: *act-buffer* **where**
$DoBuff \equiv (inp?(v) \rightarrow buff :=_C \&buff \mathbin{\hat{}}_u \langle\ll v\gg\rangle$
$\quad \Box\ (\#_u(\&buff) >_u 0)\ \&_u\ outp!(head_u(\&buff)) \rightarrow buff :=_C tail_u(\&buff))$

The main action of the buffer first initialises the single state variable $buff$, and enters a recursive loop where it does *DoBuff* over and over.

definition *Buffer* :: *act-buffer* **where**
[*rdes-def*]: $Buffer = Init\ ;;\ while_R\ true\ do\ DoBuff\ od$

3.2 Calculations

The $buff :=_C \langle\rangle$ action is represented by a simple contract with a true precondition, false pericondition (i.e. there is no intermediate behaviour), and finally sets the state variable to be empty, whilst leaving the state unchanged. There are no constraints on the initial state.

lemma *Init-contract*:
$Init = \mathbf{R}_s(true_r \vdash false \diamond \Phi(true,[\&buff \mapsto_s \langle\rangle],\langle\rangle))$
by (*rdes-simp*)

lemma *DoBuff-contract*:
$DoBuff = \mathbf{R}_s\ (true_r \vdash$
$\quad \mathcal{E}(true,\langle\rangle, (\bigsqcap\ x \cdot \{(inp\cdot\ll x\gg)_u\}_u) \cup_u (\{(outp\cdot head_u(\&buff))_u\}_u \triangleleft 0 <_u \#_u(\&buff) \triangleright \{\}_u))$
$\diamond$
$\quad ((\bigsqcap\ x \cdot \Phi(true,[\&buff \mapsto_s \&buff \mathbin{\hat{}}_u \langle\ll x\gg\rangle],\langle(inp\cdot\ll x\gg)_u\rangle)) \vee$
$\quad \Phi(0 <_u \#_u(\&buff), [\&buff \mapsto_s tail_u(\&buff)], \langle(outp\cdot head_u(\&buff))_u\rangle)))$

by (*rdes-eq*)

lemma *Buffer-contract*:
Buffer $= \mathbf{R}_s(true_r \vdash \Phi(true,[\&buff \mapsto_s \langle\rangle],\langle\rangle)$;;
$((\bigsqcap\ x \cdot \Phi(true,\ [\&buff \mapsto_s \&buff\ \hat{}_u\ \langle\ll x\gg\rangle],\ \langle(inp\cdot\ll x\gg)_u\rangle)) \vee$
$\Phi(0 <_u \#_u(\&buff),\ [\&buff \mapsto_s tail_u(\&buff)],\ \langle(outp\cdot head_u(\&buff))_u\rangle))^{\star r}$;;
$\mathcal{E}(true,\langle\rangle,\ (\bigsqcap\ x \cdot \{(inp\cdot\ll x\gg)_u\}_u) \cup_u (\{(outp\cdot head_u(\&buff))_u\}_u \triangleleft 0 <_u \#_u(\&buff)$
$\triangleright \{\}_u)) \diamond$
false)
unfolding *Buffer-def DoBuff-contract* **by** *rdes-eq*

3.3 Verifications

We first show that the buffer always outputs the same elements that were input first.

abbreviation *inps* $t \equiv [x.\ inp\ x \leftarrow t]$
abbreviation *outps* $t \equiv [x.\ outp\ x \leftarrow t]$

lemma *P1-lemma*:
$[true \vdash \ll outps(trace)\gg \leq_u \&buff\ \hat{}_u \ll inps(trace)\gg \mid true]_C \sqsubseteq while_R\ true\ do\ DoBuff\ od$
apply (*simp add*: *closure*)
apply (*rdes-refine-split*)
apply (*simp-all add*: *rpred closure usubst*)
apply (*rule conjI*)
apply (*rule rrel-thy.Star-inductl*)
apply (*simp add*: *closure*)
apply (*rule RR-refine-intro*)
apply (*simp-all add*: *closure*)
apply (*rel-auto*)
apply (*smt Prefix-Order.Cons-prefix-Cons Prefix-Order.prefix-Nil append-Cons append-Nil ch-buffer.simps*(*6*) *concat-map-maps hd-Cons-tl maps-simps*(*1*) *not-Cons-self2*)
apply (*rule rrel-thy.Star-inductl*)
apply (*simp add*: *closure*)
apply (*rule RR-refine-intro*)
apply (*simp-all add*: *closure*)
apply (*rel-auto*)
apply (*smt Prefix-Order.Cons-prefix-Cons append.left-neutral append-Cons ch-buffer.simps*(*6*) *concat-map-maps hd-Cons-tl less-eq-list-def maps-simps*(*1*) *prefix-code*(*2*))
done

lemma *P1*: $[true \vdash \ll outps(trace)\gg \leq_u \ll inps(trace)\gg \mid true]_C \sqsubseteq Buffer$
proof −
have $[true \vdash \ll outps(trace)\gg \leq_u \ll inps(trace)\gg \mid true]_C$
$\sqsubseteq$
Init ;; $[true \vdash \ll outps(trace)\gg \leq_u \&buff\ \hat{}_u \ll inps(trace)\gg \mid true]_C$
by (*rdes-refine*)
thus *?thesis*
by (*metis* (*no-types*, *lifting*) *Buffer-def P1-lemma dual-order.trans urel-dioid.mult-isol*)
qed

lemma *Buffer-deadlock-free*: *CDF* $\sqsubseteq$ *Buffer*
unfolding *Buffer-def*
by (*rdes-refine*; *blast*)

end

4 Mini-mondex example

theory *utp-csp-mini-mondex*
  imports *UTP−Circus.utp-circus*
begin

This example is a modified version of the Mini-Mondex card example taken from the 2014 paper "Contracts in CML" by Woodcock et al.

4.1 Types and Statespace

type-synonym *index* = *nat* — Card identifiers
type-synonym *money* = *int* — Monetary amounts.

In the paper money is represented as a nat, here we use an int so that we have the option of modelling negative balances. This also eases proof as integers form an algebraic ring.

alphabet *st-mdx* =
  accts :: (*index*, *money*) *pfun* — Index record of each card's balance

datatype *ch-mdx* =
  pay index × *index* × *money* | — Request a payment between two cards
  transfer index × *index* × *money* | — Effect the transfer
  accept index | — Accept the payment
  reject index — Reject it

type-synonym *action-mdx* = (*st-mdx*, *ch-mdx*) *action*

4.2 Actions

The Pay action describes the protocol when a payment of n is requested between two cards, i and j. It is slightly modified from the paper, as we firstly do not use operations but effect the transfer using indexed assignments directly, and secondly because before the transfer can proceed we need to check the balance is both sufficient, and that the transfer amount is greater than 0. It should also be noted that the indexed assignments give rise to preconditions that the list is defined at the given index. In other words, the given card records must be present.

definition *Pay* :: *index* ⇒ *index* ⇒ *money* ⇒ *action-mdx* **where**
$Pay\ i\ j\ n =$
  $pay.((\ll i\gg,\ll j\gg,\ll n\gg)_u) \rightarrow$
    $((reject.(\ll i\gg) \rightarrow Skip)$
      $\triangleleft (\ll i\gg =_u \ll j\gg \vee \ll i\gg \notin_u dom_u(\&accts) \vee \ll n\gg \leq_u 0 \vee \ll n\gg >_u \&accts(\ll i\gg)_a) \triangleright_R$
    $(accts[\ll i\gg] :=_C (\&accts(\ll i\gg)_a - \ll n\gg)\ ;;$
     $accts[\ll j\gg] :=_C (\&accts(\ll j\gg)_a + \ll n\gg)\ ;;$
     $accept.(\ll i\gg) \rightarrow Skip))$

definition *PaySet* :: *index* ⇒ (*index* × *index* × *money*) *set* **where**
[*upred-defs*]: $PaySet\ cardNum = \{(i,j,k).\ i < cardNum \wedge j < cardNum \wedge i \neq j\}$

definition *AllPay* :: *index* ⇒ *action-mdx* **where**
$AllPay\ cardNum = (\bigsqcap\ (i,\ j,\ n) \in PaySet\ cardNum \cdot Pay\ i\ j\ n)$

The Cycle action just repea the payments over and over for any extant and different card indices. In order to be well-formed we require that $cardNum \geq 2$.

definition *Cycle* :: *index* ⇒ *action-mdx* **where**

$Cycle\ cardNum = (\mu_C\ X \cdot AllPay(cardNum)\ ;;\ X)$

The Mondex action is a sample setup. It requires creates *cardNum* cards each with 100 units present.

definition $Mondex :: index \Rightarrow action\text{-}mdx$ **where**
$Mondex(cardNum) = (accts :=_C entr_u(\{0..10\}_u,\ \lambda\ x.\ 100)\ ;;\ Cycle(cardNum))$

4.3 Pre/peri/post calculations

The behaviour of a reactive program is described in three parts: (1) the precondition, that describes how the state and environment must behave to ensure valid behaviour; (2) the pericondition that describes the commitments the program makes whilst in an intermediate state in terms of events only; and (3) the postcondition that describes the commitments after the process terminates. The pericondition refers only to the trace, as the state is invisible in intermediate states – it can only be observed through events. The pre- and postcondition can refer to both the state and the trace; although the form can only refer to a prefix of the trace and before state variables – only the postcondition refers to after state.

lemma *Pay-CSP* [*closure*]: *Pay i j n is CSP*
 by (*simp add*: *Pay-def closure*)

The precondition of pay requires that, under the assumption that a payment was requested by the environment (pay is present at the trace head), and that the given amount can be honoured by the sending card, then the two cards must exist. This arises directly from the indexed assignment preconditions.

The pericondition has three cases: (1) nothing has happened and we are not refusing the payment request, (2) the payment request happened, but there isn't enough (or non-positive) money and reject is being offered, or (3) there was enough money and accept is being offered.

The postcondition has two options. Firstly, the amount was wrong, and so the trace was extended by both pay and reject, with the state remaining unchanged. Secondly, the payment was fine and so the trace was extended by pay and accept, and the states of the two cards was updated appropriately.

lemma *Pay-contract* [*rdes-def*]:
 assumes $i \neq j$
 shows
 $Pay\ i\ j\ n =$
 $\mathbf{R}_s\ (\ (\mathcal{I}(true,\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u\rangle) \Rightarrow_r$
 $[(\ll i\gg \notin_u dom_u(\&accts) \vee \ll n\gg \leq_u 0 \vee \&accts(\ll i\gg)_a <_u \ll n\gg) \vee \ll i\gg \in_u dom_u(\&accts) \wedge$
$\ll j\gg \in_u dom_u(\&accts)]_{S<})$

 $\vdash (\mathcal{E}(true,\langle\rangle,\ \{(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u\}_u) \vee$
 $\mathcal{E}(true,\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u\rangle,\ \{(reject\cdot\ll i\gg)_u\}_u)$
 $\triangleleft\ (\ll i\gg \notin_u dom_u(\&accts) \vee \ll n\gg \leq_u 0 \vee \&accts(\ll i\gg)_a <_u \ll n\gg)\ \triangleright_R$
 $\mathcal{E}(true,\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u\rangle,\ \{(accept\cdot\ll i\gg)_u\}_u))$

 $\diamond\ (\Phi(true,id,\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u,\ (reject\cdot\ll i\gg)_u\rangle)$
 $\triangleleft\ (\ll i\gg \notin_u dom_u(\&accts) \vee \ll n\gg \leq_u 0 \vee \&accts(\ll i\gg)_a <_u \ll n\gg)\ \triangleright_R$
 $\Phi(true,[\&accts \mapsto_s \&accts(\ll i\gg \mapsto \&accts(\ll i\gg)_a - \ll n\gg,\ \ll j\gg \mapsto \&accts(\ll j\gg)_a + \ll n\gg)_u]$
 $,\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u,\ (accept\cdot\ll i\gg)_u\rangle)))$
 using *assms* **by** (*simp add*: *Pay-def closure*, *rdes-simp*, *simp add*: *pr-var-def*, *rel-auto*)

lemma *Pay-wf* [*closure*]:
 Pay i j n is NCSP

by (*simp add*: *Pay-def closure*)

lemma *Pay-Productive* [*closure*]: *Pay i j n is Productive*
by (*simp add*: *Pay-def closure*)

lemma *PaySet-cardNum-nempty* [*closure*]:
$cardNum \geq 2 \Longrightarrow \neg\ PaySet\ cardNum = \{\}$
by (*rel-simp*, *presburger*)

lemma *AllPay-wf* [*closure*]:
$cardNum \geq 2 \Longrightarrow AllPay\ cardNum$ *is NCSP*
by (*simp add*: *AllPay-def closure*)

lemma *AllPay-Productive* [*closure*]:
$cardNum \geq 2 \Longrightarrow AllPay\ cardNum$ *is Productive*
by (*simp add*: *AllPay-def closure*)

4.4 Verification

We perform verification by writing contracts that specify desired behaviours of our system. A contract $[P \vdash Q \mid R]_C$ consists of three predicates that correspond to the pre-, peri-, and postconditions, respectively. The precondition talks about initial state variables and the *trace* contribution via a special variable. The pericondition likewise talks about initial states and traces. The postcondition also talks about final states.

We first show that any payment leaves the total value shared between the cards unchanged. This is under the assumption that at least two cards exist. The contract has as its precondition that initially the number of cards is *cardNum*. The pericondition is *true* as we don't care about intermediate behaviour here. The postcondition has that the summation of the sequence of card values remains the same, though of course individual records will change.

lemma *uminus-inter-insert* [*simp*]:
$(-\ A) \cap (-\ insert\ x\ B) = (-\ insert\ x\ A) \cap (-\ B)$
by (*auto*)

theorem *money-constant*:
assumes $finite\ cards\ i \in cards\ j \in cards\ i \neq j$
shows $[dom_u(\&accts) =_u \ll cards \gg \vdash true \mid sum_u(\$accts) =_u sum_u(\$accts')]_C \sqsubseteq Pay\ i\ j\ n$
— We first calculate the reactive design contract and apply refinement introduction
proof (*simp add*: *assms Pay-contract*, *rule CRD-refine-rdes*)

— Three proof obligations result for the pre/peri/postconditions. The first requires us to show that the contract's precondition is weakened by the implementation precondition. It is because the implementation's precondition is under the assumption of receiving an input and the money amount constraints. We discharge by first calculating the precondition, as done above, and then using the relational calculus tactic.

from *assms*
show ‘$[dom_u(\&accts) =_u \ll cards \gg]_{S<} \Rightarrow$
$\mathcal{I}(true, \langle (pay \cdot (\ll i \gg, \ll j \gg, \ll n \gg)_u)_u \rangle) \Rightarrow_r$
$[(\ll i \gg \notin_u dom_u(\&accts) \vee \ll n \gg \leq_u 0 \vee \&accts(\ll i \gg)_a <_u \ll n \gg) \vee$
$\ll i \gg \in_u dom_u(\&accts) \wedge \ll j \gg \in_u dom_u(\&accts)]_{S<}$‘
by (*rel-auto*)

— The second is trivial as we don't care about intermediate states.

show $[true]_{S<}\llbracket x \rightarrow \&tt \rrbracket \llbracket r \rightarrow \$ref' \rrbracket \sqsubseteq$
$([dom_u(\&accts) =_u \ll cards \gg]_{S<} \wedge$
$(\mathcal{E}(true,\langle\rangle,\ \{(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u\}_u) \vee$
$\mathcal{E}(true,\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u\rangle,\ \{(reject\cdot\ll i\gg)_u\}_u) \triangleleft (\ll i\gg \notin_u dom_u(\&accts) \vee \ll n\gg \leq_u 0 \vee \&accts(\ll i\gg)_a <_u \ll n\gg) \triangleright_R$
$\mathcal{E}(true,\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u\rangle,\ \{(accept\cdot\ll i\gg)_u\}_u)))$
by (*simp add*: *rpred usubst*, *rel-auto*)

— The third requires that we show that the postcondition implies that the total amount remains unaltered. We calculate the postcondition, and then use relational calculus. In this case, this is not enough and an additional property of lists is required ($?i < \#_u(?xs) \Longrightarrow foldr\ (+)\ (?xs[?i := ?v])\ 0 = foldr\ (+)\ ?xs\ 0 - ?xs(?i)_a + ?v$) that can be retrieved by sledgehammer. However, we actually had to prove that property first and add it to our library.

from *assms*
show $[sum_u(\$accts) =_u sum_u(\$accts')]_{S'}\llbracket x \rightarrow \&tt \rrbracket \sqsubseteq$
$([dom_u(\&accts) =_u \ll cards \gg]_{S<} \wedge$
$\Phi(true,id,\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u,\ (reject\cdot\ll i\gg)_u\rangle) \triangleleft (\ll i\gg \notin_u dom_u(\&accts) \vee \ll n\gg \leq_u 0 \vee \&accts(\ll i\gg)_a <_u \ll n\gg) \triangleright_R$
$\Phi(true,[\&accts \mapsto_s \&accts(\ll i\gg \mapsto \&accts(\ll i\gg)_a - \ll n\gg,\ \ll j\gg \mapsto \&accts(\ll j\gg)_a + \ll n\gg)_u],\langle(pay\cdot(\ll i\gg,\ \ll j\gg,\ \ll n\gg)_u)_u,\ (accept\cdot\ll i\gg)_u\rangle))$
by (*rel-auto*, *simp-all add*: *pfun-sums-upd-2*)
qed

The next property is that no card value can go below 0, assuming it was non-zero to start with.

theorem *no-overdrafts*:
assumes *finite cards* $i \in cards$ $j \in cards$ $i \neq j$
shows $[dom_u(\&accts) =_u \ll cards\gg \vdash true \mid (\forall\ k \cdot \ll k\gg \in_u \ll cards\gg \wedge \$accts(\ll k\gg)_a \geq_u 0 \Rightarrow \$accts'(\ll k\gg)_a \geq_u 0)]_C \sqsubseteq Pay\ i\ j\ n$
using *assms*
apply (*simp add*: *Pay-contract*)
apply (*rule CRD-refine-rdes*)
apply (*rel-auto*)
apply (*rel-auto*)
apply (*rel-simp*)
apply (*metis diff-ge-0-iff-ge dual-order.trans le-add-same-cancel2 less-le not-le pfun-app-upd-1 pfun-app-upd-2*)
done

The next property shows liveness of transfers. If a payment is accepted, and we have enough money, then the acceptance of the transfer cannot be refused. Unlike the previous two examples, this is specified using the pericondition as we are talking about intermediate states and refusals.

theorem *transfer-live*:
assumes *finite cards* $i \in cards$ $j \in cards$ $i \neq j$ $n > 0$
shows $[dom_u(\&accts) =_u \ll cards\gg$
$\vdash \ll trace\gg \neq_u \langle\rangle \wedge last_u(\ll trace\gg) =_u (pay\cdot(\ll(i,j,k)\gg))_u \wedge \ll n\gg \leq_u \&accts(\ll i\gg)_a \Rightarrow (accept\cdot(\ll(i)\gg))_u \notin_u \ll refs\gg$
$\mid true]_C \sqsubseteq Pay\ i\ j\ n$
using *assms*
apply (*simp add*: *Pay-contract*)
apply (*rule CRD-refine-rdes*)
apply (*rel-auto*)+
done

end

References

[1] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.

[2] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.

[3] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.

[4] J. Foster. *Bidirectional programming languages.* PhD thesis, University of Pennsylvania, 2009.

[5] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.

[6] T. Hoare, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sørensen, J. Spivey, and B. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.

[7] T. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[8] C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[9] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming*, volume 10134 of *LNCS*, pages 155–175. Springer, 2016.