A Mechanisation of FMI in Isabelle/UTP

Frank Zeyda, Ana Cavalcanti, and Simon Foster

June 20, 2017

Contents

1 Theory of Circus

```
theory utp_circus
imports utp_theories_deep "../utp/models/utp_axm"
begin recall_syntax
```

Types are not printed correctly, have a chat with Simon Foster.

```
typ "('σ, 'ε) st_csp"
```

Renaming HOL's relation type available.

```
type_synonym 'a relation = "'a Relation.rel"
```

```
translations (type) "'a relation" ⇌ (type)"'a Relation.rel"
```

hide_type `Relation.rel` — TODO: Let the recall package hide types too!

The below interfere with the corresponding CSP definitions.

```
hide_const utp_cml.Skip
hide_const utp_cml.Stop
```

Move this back to the theory `utp_csp`.

```
definition [lens_defs]: "Σ_CxC = Σ_C ×_L Σ_C"
```

1.1 Channel Event Syntax

The bellow is useful to construct synchronisation sets of events.

```
definition events :: "('a, 'φ) chan ⇒ 'φ set" ("ε'(_')") where
"events c = c ` UNIV"
```

1.2 Process Semantics

An open issue is whether we should contract the alphabet as well i.e. to the type `unit`. Since we use deep (or axiomatic) variables, we can strictly get away without that. But it might be more accurate in terms of the semantics of *Circus* processes. I talked to Simon Foster about this who then added definitions that are necessary to carry out alphabet restrictions of program-state alphabets within various UTP theories.

— **TODO**: Additionally perform an alphabet restriction below.

```
definition Process ::
  "('ε, 'α) action ⇒
   ('ε, 'α) action" where
"Process p = (∃ Σ_CxC · p) (*↾_p Σ_CxC*)"
```

```
definition Action ::
  "('ε, 'α) action ⇒
  (('ε, 'α) action ⇒ ('ε, 'α) action) ⇒
  ('ε, 'α) action" where
"Action = Let"
```

Instead of using the SUPREMUM we should use a mu below.

```
definition RecAction ::
  "(('ε, 'α) action ⇒
    ('ε, 'α) action × ('ε, 'α) action) ⇒
```

```
  ('ε, 'α) action" where
"RecAction act_body =
  Action (SUPREMUM UNIV (λX. fst (act_body X))) (λX. snd (act_body X))"

lemmas circus_syntax =
  Process_def Action_def RecAction_def
```

1.3 Process Syntax

```
nonterminal action and actions

syntax
  "_Action"  :: "[pttrn, logic] ⇒ action"     ("(2_ =/ _)" 10)
  ""         :: "action ⇒ actions"            ("_")
  "_Actions" :: "[action, actions] ⇒ actions" ("_and//_")
  "_Process" :: "[actions,  logic] ⇒ logic"   ("((2begin//(_)//• (_))//end)")
  "_ParamProc" :: "idt ⇒ args ⇒ logic ⇒ bool" ("(process _ _ ≜//_)" [0, 0, 10] 10)
  "_BasicProc" :: "idt ⇒         logic ⇒ bool" ("(process _ ≜//_)" [0, 10] 10)

syntax (output)
  "_Actions_tr'"   :: "[action, actions] ⇒ actions"  ("_//_")

translations
  "_Process (_Actions act acts) e" ⇀ "_Process act (_Process acts e)"
  "_Process (_Actions_tr' act acts) e" ↽ "_Process act (_Process acts e)"

  "_Process (_Action name act) more" ⇌ "(CONST RecAction) (λname. (act, more))"
  "_ParamProc name args body" ⇌ "name = (λargs. (CONST Process) body)"
  "_BasicProc name      body" ⇌ "name = (CONST Process) body"

print_translation {*
  [Syntax_Trans.preserve_binder_abs2_tr'
    @{const_syntax Action} @{syntax_const "_Action"}]
*}
```

Hide non-terminals as this interferes with parsing the action type.

```
hide_type (open)
  utp_circus.action
  utp_circus.actions
```

1.4 Stub Constructs (TODO)

TODO: Define the semantics of the operators below.

Make parallel composition bind weaker than set union, so that the latter can be used to combine channel sets. Operator precedence is still and issue that we need to address within Isabelle/UTP.

```
purge_notation
  ParCSP_NS (infixr "[|_|]" 105) and
  InterleaveCSP (infixr "|||" 105)

purge_syntax
  "_output_prefix" :: "('a, 'σ) uexpr ⇒ prefix_elem'" ("!'(_')")
  "_output_prefix" :: "('a, 'σ) uexpr ⇒ prefix_elem'" (".'(_')")

consts ParCircus ::
```

```
  "('σ, 'φ) action  ⇒ ('φ event set) ⇒ ('σ, 'φ) action ⇒
   ('σ, 'φ) action" (infixl "[|(_)|]" 60)

definition InterleaveCircus ::
  "('σ, 'φ) action  ⇒ ('σ, 'φ) action ⇒
   ('σ, 'φ) action" (infixl "|||" 62) where
"P ||| Q = P [| {} |] Q"

consts HideCircus ::
  "('σ, 'φ) action  ⇒ ('φ event set) ⇒ ('σ, 'φ) action" (infixl "\" 55)

consts InterruptCircus ::
  "('σ, 'φ) action  ⇒ ('σ, 'φ) action ⇒ ('σ, 'φ) action" (infixl "△" 100)
```

1.5 Input Prefix (OLD)

```
definition InputCircus ::
  "('a::{continuum, two}, 'ε) chan ⇒
    ('a, ('σ, 'ε) st_csp) lvar ⇒
    ('a ⇒ ('σ, 'ε) action) ⇒
    (('a ⟹ ('σ, 'ε) st_csp) ⇒ ('σ, 'ε) action) ⇒
    ('σ, 'ε) action" where
"InputCircus c x P A =
  (var_RDES x · R_s(true ⊢ (do_I c x P) ∧ (∃ $x´ · II)) ;; A(x))"
```

A few syntactic adjustments, remove them and adapt `fmi.thy`.

```
no_notation utp_rel_opsem.trel (infix "→_u" 85)

syntax
  "_circus_sync" :: "logic ⇒ logic ⇒ logic" (infixl "→_u" 80)
  "_circus_input" :: "logic ⇒ id ⇒ logic ⇒ logic ⇒ logic"
    ("_?_u'(_ :/ _') → _" [81, 0, 0, 80] 80)
  "_circus_output" :: "logic ⇒ logic ⇒ logic ⇒ logic"
    ("_!_u'(_') → _" [81, 0, 80] 80)

translations
  "c →_u A" ⇌ "(CONST OutputCSP) c ()_u A"
  "c!_u(v) → A" ⇌ "(CONST OutputCSP) c v A"
  "c?_u(x : P) → A" ⇀ "(CONST InputCircus) c
    (*(CONST top_var ...*) (CONST MkDVar IDSTR(x)) (λx. P) (λx. A)"
  "c?_u(x : P) → A" ↽ "(CONST InputCircus) c x (λv. P) (λw. A)"
```

1.6 Mixed Prefixes

In this section, we provide support for mixed prefixes.

1.6.1 Prefix Semantics

We first define a new and simplified version of the `InputCircus` operator. Simplification is due to its action argument being parametrised by the a (HOL) value rather than a variable lens. This makes a subsequent definition of syntax and translations for mixed prefixes much easier. We note that all different kinds of prefixes will be expressed in terms of input communications with appropriate constraints on variables that are introduced by the prefix construct.

```
definition new_do_I :: "
```

```
  ('a, 'ε) chan ⇒ 'a ⇒
  ('a ⇒ ('σ, 'ε) action) ⇒
  ('σ, 'ε) action" where
"new_do_I c x P =
  (($tr´ =_u $tr ∧ {e : «δ_u(c)» | P(e) · (c·«e»)_u}_u ∩_u $ref´ =_u {}_u)
    ◁ $wait´ ▷
  (($tr´ - $tr) ∈_u {e : «δ_u(c)» | P(e) · ⟨(c·«e»)_u⟩}_u ∧ (c·«x»)_u =_u last_u($tr´)))"
```

definition `new_do_I' :: "`
```
  ('a, 'ε) chan ⇒ 'a ⇒
  ('a ⇒ ('σ, 'ε) action) ⇒
  ('σ, 'ε) action" where
"new_do_I' c x P =
  (($tr´ =_u $tr ∧ (c·«x»)_u ∉_u $ref´) ◁ $wait´ ▷ (($tr´ - $tr) =_u ⟨(c·«x»)_u⟩))"
```

definition `NewInputCircus ::`
```
  "('a, 'ε) chan ⇒
    ('a ⇒ ('σ, 'ε) action) ⇒
    ('a ⇒ ('σ, 'ε) action) ⇒
    ('σ, 'ε) action" where
"NewInputCircus c P A = (∃ x · R_s(true ⊢ (new_do_I c x P) ∧ II) ;; A(x))"
```

definition `NewInputCircus' ::`
```
  "('a, 'ε) chan ⇒
    ('a ⇒ ('σ, 'ε) action) ⇒
    ('a ⇒ ('σ, 'ε) action) ⇒
    ('σ, 'ε) action" where
"NewInputCircus' c P A = (∃ x · R_s(true ⊢ (new_do_I' c x P) ∧ II) ;; A(x))"
```

lemma `"NewInputCircus = NewInputCircus'"`
apply `(rule ext)+`
apply `(unfold NewInputCircus_def NewInputCircus'_def)`
apply `(unfold new_do_I_def new_do_I'_def)`
— TODO: Allow simplification theorems to be passed to rel_simp.
apply `(rel_simp)`
apply `(safe; clarsimp?)`
apply `(blast)`
apply `(simp_all add: zero_list_def)`
apply `(blast)`
apply `(metis)`
apply `(metis)`
apply `(blast)`
apply `(blast)`
apply `(metis)+`
done

lemma
`"vwb_lens x ⟹ NewInputCircus c P A = InputCircus c x P B"`
apply `(unfold NewInputCircus_def InputCircus_def)`
apply `(unfold new_do_I_def do_I_def chan_type_def)`
apply `(rel_simp)`
apply `(safe; clarsimp simp add: comp_def)`
oops

1.6.2 Syntax and Translations

We next configure a syntax for mixed prefixes.

```
nonterminal prefix_elem and mixed_prefix

syntax "" :: "prefix_elem ⇒ mixed_prefix" ("_")
```

Input Prefix: ...`?(x)`

```
syntax "_simple_input_prefix" :: "id ⇒ prefix_elem"  ("?'(_')")
```

Input Prefix with Constraint: ...`?(x : P)`

```
syntax "_input_prefix" :: "id ⇒ ('σ, 'ε) action ⇒ prefix_elem" ("?'(_ :/ _')")
```

Output Prefix: ...`![v]e`

A variable name must currently be provided for outputs, too. Fix?!

```
syntax "_output_prefix" :: "id ⇒ ('a, 'σ) uexpr ⇒ prefix_elem" ("![_]_")
syntax "_output_prefix" :: "id ⇒ ('a, 'σ) uexpr ⇒ prefix_elem" (".[_]_")

syntax (output) "_output_prefix_pp" :: "('a, 'σ) uexpr ⇒ prefix_elem" ("!_")
```

Synchronisation Action: `c →𝒞 A`

```
syntax "_sync_action" ::
  "('a, 'ε) chan ⇒ ('σ, 'ε) action ⇒ ('σ, 'ε) action" (infixr "→𝒞" 80)
```

Mixed-Prefix Action: `c`...`(prefix) →𝒞 A`

```
syntax "_mixed_prefix" :: "prefix_elem ⇒ mixed_prefix ⇒ mixed_prefix" ("__")

syntax "_prefix_action" ::
  "('a, 'ε) chan ⇒ mixed_prefix ⇒ ('σ, 'ε) action ⇒ ('σ, 'ε) action"
  ("(__ →𝒞/ _)" [81, 81, 80] 80)
```

Syntax translations

```
translations
  "_simple_input_prefix x" ⇌ "_input_prefix x true"
  "_output_prefix x e" ⇀ "_input_prefix x («x» =ᵤ e)"
  "_output_prefix_pp e" ↽ "_input_prefix v («x» =ᵤ e)"

translations
  "_mixed_prefix (_input_prefix x P) (_input_prefix y Q)" ⇌
  "_input_prefix (_pattern x y) (P ∧ Q)"

translations
  "_sync_action c A" ⇌ "(CONST OutputCSP) c ()ᵤ A"
  "_prefix_action c (_input_prefix x P) A"  ⇀
  "(CONST NewInputCircus) c (λx. P) (λx. A)"
```

The ML print translation for `NewInputCircus` below is a little more robust than using Isabelle in-built **translations** in dealing with unwanted eta-contraction.

```
ML {*
signature CIRCUS_PREFIX =
sig
  val mk_pattern: term list -> term
```

```
  val strip_abs_tr': term list -> term -> term list * term
  val mk_input_prefix: term -> term -> term
  val mk_prefix_action: term -> term -> term -> term -> term
  val InputCircus_tr': term list -> term
end;

structure Circus_Prefix : CIRCUS_PREFIX =
struct
  fun mk_pattern [] = Const (@{syntax_const "_unit"}, dummyT)
  | mk_pattern [x] = x
  | mk_pattern (h :: t) =
    Const (@{syntax_const "_pattern"}, dummyT) $ h $ (mk_pattern t);

  fun strip_abs_tr' vs (Abs abs) =
    let val (v, body) = Syntax_Trans.atomic_abs_tr' abs in
      strip_abs_tr' (vs @ [v]) body
    end
  | strip_abs_tr' vs
      (Const (@{const_syntax case_prod}, _) $ (Abs abs)) =
    let val (v, body) = Syntax_Trans.atomic_abs_tr' abs in
      strip_abs_tr' (vs @ [v]) body
    end
  | strip_abs_tr' vs term = (vs, term);

  fun mk_input_prefix x P =
    Const (@{syntax_const "_input_prefix"}, dummyT) $ x $ P;

  fun mk_prefix_action c x P A =
    Const (@{syntax_const "_prefix_action"}, dummyT) $ c $
      (mk_input_prefix x P) $ A;

  fun InputCircus_tr' [c, P, A] = let
    val (vs, P') = strip_abs_tr' [] P;
    val (vs', A') = strip_abs_tr' [] A in
      if (vs = vs') then
        (mk_prefix_action c (mk_pattern vs) P' A')
      else raise Match
    end
  | InputCircus_tr' _ = raise Match;
end;
*}
```

print_translation {*

```
  [(@{const_syntax NewInputCircus}, K Circus_Prefix.InputCircus_tr')]
*}
```

Testing

All of the below seem to work!

term "c?(x : true) $\rightarrow_{\mathcal{C}}$ A x"
term "c?(x : true)?(y : true) $\rightarrow_{\mathcal{C}}$ A x y"
term "c?(x : true)?(y : true)?(z : true) $\rightarrow_{\mathcal{C}}$ A x y z"
term "c?(x : ≪x≫ $<_u$ ≪10::nat≫) $\rightarrow_{\mathcal{C}}$ A x"
term "c?(x)?(y : true) $\rightarrow_{\mathcal{C}}$ A x y"
term "c?(x : true)![st]≪1::nat≫ $\rightarrow_{\mathcal{C}}$ A x"
term "c![st]≪1::nat≫?(x : true) $\rightarrow_{\mathcal{C}}$ A x"

```
term "c![st]≪1::nat≫ →𝒞 A"
term "c![x]≪1::nat≫![y]≪2::nat≫ →𝒞 A"
term "c →𝒞 A"
```

1.7 Circus Conditional

```
syntax "_if_then_else" ::
  "'a upred ⇒ ('a, 'b) rel ⇒ ('a, 'b) rel ⇒ ('a, 'b) rel"
    ("(if𝒞 (_)/ then𝒞 (_)/ else𝒞 (_))" [0, 0, 10] 10)

translations "if𝒞 b then𝒞 P else𝒞 Q" ⇌ "P ◁ b ▷r Q"
```

1.8 Iterated Constructs

In this section, we define various iterated constructs.

1.8.1 Iterated Sequence

An open question is whether to use a different Skip below. Here, I believe it is not needed; the main issue is to exploit the property of `II` being a right unit (`?P ;; II = ?P`). Alternatively, we may use the singleton list as the base case to circumvent the problem.

```
primrec useq_iter :: "'a list ⇒ ('a ⇒ 'b hrel) ⇒ 'b hrel" where
"useq_iter [] f = II" |
"useq_iter (h # t) f = (f h) ;; (useq_iter t f)"

syntax "_useq_iter" :: "pttrn ⇒ 'a list ⇒ 'σ hrel ⇒ 'σ hrel"
  ("(3;; _ : _ •/ _)" [0, 0, 10] 10)

translations ";; x : l • P" ⇌ "(CONST useq_iter) l (λx. P)"
```

1.8.2 Iterated Interleaving

```
primrec interleave_iter ::
  "'a list ⇒ ('a ⇒ ('σ, 'φ) action) ⇒ ('σ, 'φ) action" where
"interleave_iter [] f = Skip" |
"interleave_iter (h # t) f = (f h) ||| (interleave_iter t f)"

syntax "_interleave_iter_iter" ::
  "pttrn ⇒ 'a list ⇒ ('σ, 'φ) action ⇒ ('σ, 'φ) action"
  ("(3||| _ : _ •/ _)" [0, 0, 10] 10)

translations "||| x : l • P" ⇌ "(CONST interleave_iter) l (λx. P)"
```

1.9 Proof Experiments

```
theorem
"process P ≜ begin A = Act1 and B = (Act2 ;; A) • Main(A, B) end ⟹
 P = Process (Main (Act1, Act2 ;; Act1))"
apply (unfold circus_syntax)
apply (unfold Let_def) — TODO: How to apply the copy rule selectively?!
apply (clarsimp)
done

theorem
```

```
"process P(x::nat) ≜ begin A = Act1 x and B = (Act2 ;; A) • Main(A, B) end ⟹
 P = (λx. Process (Main (Act1 x, Act2 ;; Act1 x)))"
```
apply `(unfold circus_syntax)`
apply `(unfold Let_def)` — **TODO**: How to apply the copy rule selectively?!
apply `(clarsimp)`
done
end

2 FMI Circus Model

```
theory fmi
imports
  "../theories/utp_circus"
  "../utils/Positive_New"
begin recall_syntax

type_synonym 'a relation = "'a Relation.rel"

translations (type) "'a relation" ⇌ (type)"'a Relation.rel"

hide_type Relation.rel — TODO: Let the recall package hide types too!
```

The following adjustment is needed...

```
syntax
  "_csp_sync" :: "logic ⇒ logic ⇒ logic" ("_ →u _" [81, 80] 80)

declare [[typedef_overloaded]]
declare [[quick_and_dirty]]

declare [[syntax_ambiguity_warning=false]]

default_sort type
```

2.1 Preliminaries

```
lemma card_gt_two_exists:
"finite S ⟹ 2 ≤ card S ⟹ (∃a∈S. ∃b∈S. a ≠ b)"
apply (atomize (full))
apply (rule impI)
apply (erule finite.induct)
apply (simp_all)
apply (safe; clarsimp)
apply (metis
  One_nat_def Suc_1 card_Suc_eq card_insert_if le_SucE nat.inject singletonI)
apply (auto)
done
```

By default, the product type did not instantiate class `two`.

```
theorem card_ge_two_witness:
"finite S ⟹ 2 ≤ card S = (∃x y. x ∈ S ∧ y ∈ S ∧ x ≠ y)"
apply (rule iffI)
— Subgoal 1
using card_gt_two_exists apply (blast)
— Subgoal 2
apply (case_tac "card S = 0")
apply (clarsimp)
apply (case_tac "card S = 1")
apply (clarsimp)
apply (metis card_Suc_eq singletonD)
apply (clarsimp)
apply (meson Finite_Set.card_0_eq less_2_cases not_le)
done
```

```
lemma instance_twoI:
"(∃(x::'a) (y::'a). x ≠ y) ⟹ ¬ finite (UNIV::'a set) ∨ 2 ≤ card (UNIV::'a set)"
apply (case_tac "finite (UNIV::'a set)")
apply (simp_all)
apply (subst card_ge_two_witness)
apply (simp_all)
done

instance prod :: (two, two) two
apply (intro_classes)
apply (rule instance_twoI)
apply (subgoal_tac "∃(a::'a) (b::'a). a ≠ b")
apply (subgoal_tac "∃(c::'b) (d::'b). c ≠ d")
apply (clarsimp)
apply (rule two_diff)
apply (rule two_diff)
done
```

TODO: Find a better place to carry out the instantiation below.

```
subclass (in infinite) two
apply (intro_classes)
apply (rule disjI1)
apply (auto)
done
```

2.2 Well-defined Values

We declare a type class that introduces a notion of well-definedness of values for some HOL type 'a for which it is instantiated. With this, we can carry out the generic construction of subtypes that include defined values only. We may use this later to obtain types for well-formed events.

```
class wf =
fixes wf :: "'a ⇒ bool"
assumes wf_value_exists: "∃x. wf x"
begin
definition WF_UNIV :: "'a itself ⇒ 'a set" where
"WF_UNIV t = (Collect wf)"
end
```

Generic construction of a subtype comprising of defined values only.

```
typedef (overloaded) 'a::wf wf = "WF_UNIV TYPE('a)"
apply (unfold WF_UNIV_def)
apply (clarsimp)
apply (rule wf_value_exists)
done

setup_lifting type_definition_wf
```

2.3 Lists as Tables

```
type_synonym ('a, 'b) table = "('a × 'b) list"

fun lookup :: "('a × 'b) list ⇒ 'a ⇒ 'b" where
"lookup ((x, y) # t) v = (if v = x then y else (lookup t x))" |
"lookup [] x = undefined"
```

```
syntax "_ulookup" ::
  "('a × 'b, 'σ) uexpr ⇒ ('a, 'σ) uexpr ⇒ ('b, 'σ) uexpr" ("lookupᵤ")

translations "lookupᵤ t x" ⇌ "(CONST bop) (CONST lookup) t x"
```

2.4 Positive Subtype (Laws)

TODO: Move the following to the theory `utp_expr`.

```
syntax "_uRep_pos" :: "('a pos, 'α) uexpr ⇒ ('a, 'α) uexpr" ("§'(_')")

translations "_uRep_pos p" ⇌ "(CONST uop) (CONST Rep_pos) p"
```

TODO: Move the following to the theory `Positive_New`.

```
lemma ge_num_infinite_if_no_top:
"infinite {x::'a::{zero, linorder, no_top}. n ≤ x}"
apply (clarsimp)
— From the assumption that the set is finite.
apply (subgoal_tac "∃y::'a. Max {x. n ≤ x} < y")
apply (clarsimp)
apply (metis Max_ge leD mem_Collect_eq order.strict_implies_order order_refl order_trans)
using gt_ex apply (blast)
done

lemma less_zero_ordLeq_ge_zero:
"|{x::'a::{ordered_ab_group_add}. x < 0}| ≤o |{x::'a. 0 ≤ x}|"
apply (rule_tac f = "uminus" in surj_imp_ordLeq)
apply (simp add: image_def)
apply (clarsimp)
apply (rule_tac x = "- x" in exI)
apply (simp)
done
```

The next theorem is not totally trivial to prove!

```
instance pos :: ("{linordered_ab_group_add, no_top, continuum}") continuum
apply (intro_classes)
apply (case_tac "countable (UNIV :: 'a set)")
— Subgoal 1 (Easy Case)
apply (rule disjI1)
apply (subgoal_tac "∃to_nat::'a ⇒ nat. inj to_nat")
— Subgoal 1.1
apply (clarsimp)
apply (thin_tac "countable UNIV")
apply (rule_tac x = "to_nat o Rep_pos" in exI)
apply (rule inj_comp)
apply (assumption)
apply (meson Positive_New.pos.Rep_pos_inject injI)
— Subgoal 1.2
apply (blast)
— Subgoal 2 (Difficult Case)
apply (rule disjI2)
apply (subst sym [OF equal_card_bij_betw])
apply (rule equal_card_intro)
apply (subgoal_tac "|UNIV::'a pos set| =o |{x::'a. 0 ≤ x}|")
— Subgoal 2.1
```

```
apply (erule ordIso_transitive)
apply (rule ordIso_symmetric)
apply (subgoal_tac "|UNIV::nat set set| =o |UNIV::'a set|")
— Subgoal 2.1.1
apply (erule ordIso_transitive)
apply (subgoal_tac "(UNIV::'a set) = {x.0 ≤ x} ∪ {x. x < 0}")
— Subgoal 2.1.1.1
apply (erule ssubst)
apply (rule card_of_Un_infinite_simps(1))
apply (rule ge_num_infinite_if_no_top)
apply (rule less_zero_ordLeq_ge_zero)
— Subgoal 2.1.1.2
apply (auto)
— Subgoal 2.1.2
apply (rule_tac f = "from_nat_set" in card_of_ordIsoI)
apply (rule_tac bij_betwI'; clarsimp?)
— This is the only place where countable UNIV is needed.
apply (metis bij_betw_imp_surj from_nat_set_def surj_f_inv_f to_nat_set_bij)
apply (rule_tac x = "to_nat_set y" in exI)
apply (clarsimp)
— Subgoal 2.2
apply (rule_tac f = "Rep_pos" in card_of_ordIsoI)
apply (rule_tac bij_betwI'; clarsimp?)
apply (simp add: Positive_New.pos.Rep_pos_inject)
using Positive_New.pos.Rep_pos apply (blast)
apply (rule_tac x = "Abs_pos y" in exI)
apply (simp add: Positive_New.pos.Abs_pos_inverse)
done
```

2.5 Time Model

The rationale in this section is to define an abstract model of time that identifies reasonable assumptions that are sufficient for reasoning about model without having to specify in detail whether we are dealing with, for instance, discrete, continuous or super-dense time.

2.5.1 Abstract Time

We introduce permissible time domains abstractly as a type class. Clearly, the elements of the type have to be linearly ordered, and membership to the class `semidom_divide` entails many key properties of addition, subtraction, multiplication and division. Note that we cannot require time to form a field as there may not be an additive inverse i.e. if we confine ourselves to positive time instants. Lastly, we also assume that time does not stop, meaning that the order must have no top (class `no_top`); it might have a bottom though which, if so, must be the same as `0`.

```
class time = linorder + semidom_divide + no_top

class pos_time = time + zero + order_bot +
assumes zero_is_bot: "0 = bot"
```

I wonder if we can get away with weaker assumptions below. That would mean that we could also instantiate type `int pos` as `time` and `pos_time`. If not, this is not an issue since we would typically use type `nat` here in any case.

```
instance pos :: (linordered_field) time
apply (intro_classes)
```

done

instantiation `pos :: (linordered_field) pos_time`
begin
lift_definition `bot_pos :: "'a pos"`
is `"0"` **..**
instance
apply `(intro_classes)`
apply `(transfer; simp)`
apply `(transfer; simp)`
done
end

2.5.2 Discrete Time

Naturals, integers and rationals are used to model discrete time.

instance `nat :: time`
apply `(intro_classes)`
done

instance `int :: time`
apply `(intro_classes)`
done

instance `rat :: time`
apply `(intro_classes)`
done

instantiation `nat :: pos_time`
begin
instance
apply `(intro_classes)`
apply `(unfold bot_nat_def)`
apply `(rule refl)`
done
end

2.5.3 Continuous Time

Reals and positive reals are used to model continuous time.

type_notation `real ("ℝ")`

type_synonym `pos_real = "real pos" ("ℝ⁺")`

translations `(type) "ℝ⁺"` $\leftharpoondown$ `(type) "real pos"`

instance `real :: time`
apply `(intro_classes)`
done

Membership of $\mathbb{R}^+$ to the sort `time` follows from the earlier-on instantiation of `'a pos` as `timem` provided the type parameter `'a` constitutes a `linordered_field` instance.

2.5.4 Verifying Instantiations

Instantiation of class time.

```
theorem "OFCLASS(nat, time_class)" ..
theorem "OFCLASS(int, time_class)" ..
theorem "OFCLASS(rat, time_class)" ..
theorem "OFCLASS(real, time_class)" ..
theorem "OFCLASS(rat pos, time_class)" ..
theorem "OFCLASS(real pos, time_class)" ..
```

Instantiation of class pos_time.

```
theorem "OFCLASS(nat, pos_time_class)" ..
theorem "OFCLASS(rat pos, pos_time_class)" ..
theorem "OFCLASS(real pos, pos_time_class)" ..
```

Instantiation of class continuum.

```
theorem "OFCLASS(nat, continuum_class)" ..
theorem "OFCLASS(int, continuum_class)" ..
theorem "OFCLASS(rat, continuum_class)" ..
theorem "OFCLASS(real, continuum_class)" ..
theorem "OFCLASS(int pos, continuum_class)" ..
theorem "OFCLASS(rat pos, continuum_class)" ..
theorem "OFCLASS(real pos, continuum_class)" ..
```

2.6 FMI Types

In this section, we encode the various FMI types in HOL.

2.6.1 TIME and NZTIME

Our aim is to treat time abstractly in the FMI model via some arbitrary type 'τ that is a member of class time or pos_time. We thus introduce some additional syntax here to facilitate obtaining the value universe of a time domain provided through a type 'τ. This is just a syntactic sugar allowing us to write TIME('τ) and NZTIME('τ) while imposing the appropriate sort constraints on the free type 'τ.

```
class ctime = time + linordered_ab_group_add + continuum + two

syntax   "_TIME" :: "type ⇒ type" ("TIME'(_')")
syntax "_NZTIME" :: "type ⇒ type" ("NZTIME'(_')")

translations (type) "TIME('τ)" ⇌ (type) "'τ::ctime"
translations (type) "NZTIME('τ)" ⇌ (type) "'τ::ctime pos"
```

2.6.2 FMI2COMP

The type FMI2COMP of FMI component identifiers is introduced as a given (deferred) type. We can later change this i.e. to give an explicit model that encodes FMI component identifiers as strings or natural numbers, for instance.

```
typedecl FMI2COMP
```

— Syntactic sugar for UNIV.

```
abbreviation FMI2COMP :: "FMI2COMP set" where
"FMI2COMP ≡ UNIV"
```

Instantiation the relevant classes for the axiomatic value model.

```
instantiation FMI2COMP :: typerep
begin
definition typerep_FMI2COMP :: "FMI2COMP itself ⇒ utype" where
[typing]: "typerep_FMI2COMP t = typerep.Typerep (STR ''fmi.FMI2COMP'') []"
instance ..
end

instantiation FMI2COMP :: typedep
begin
definition typedep_FMI2COMP :: "FMI2COMP itself ⇒ utype set" where
[typing]: "typedep_FMI2COMP t = {TYPEREP(FMI2COMP)}"
instance ..
end
```

— The below facilitates evaluation of the transitive closure of the PDG.

```
instantiation FMI2COMP :: equal
begin
definition equal_FMI2COMP ::"FMI2COMP ⇒ FMI2COMP ⇒ bool" where
"equal_FMI2COMP x y = (x = y)"
instance
apply (intro_classes)
apply (unfold equal_FMI2COMP_def)
apply (rule refl)
done
end

inject_type FMI2COMP
```

Instantiation the relevant classes for the deep value model.

Clearly, this is not possible unless we endow the type with a concrete model using a `typedef` instead of a `typedecl`. On the other hand, the axiom introduced by the `sorry` below ought not lead to unsoundness.

— **TODO**: Introduce a concrete model for FMI components in order to be able to prove the instantiations below and remove the `sorry`.

```
instance FMI2COMP :: "{continuum, two}"
sorry
```

2.6.3 FMUSTATE

Likewise, `FMUSTATE` is introduced as a given type for now. We may need to reviewing this in the future; for instance, the universal value model could be used to encode a generic (monomorphic) state type that can encode the state of arbitrary FMUs.

```
typedecl FMUSTATE
```

Instantiation the relevant classes for the axiomatic value model.

```
instantiation FMUSTATE :: typerep
begin
```

```
definition typerep_FMUSTATE :: "FMUSTATE itself ⇒ utype" where
[typing]: "typerep_FMUSTATE t = typerep.Typerep (STR ''fmi.FMUSTATE'') []"
instance ..
end

instantiation FMUSTATE :: typedep
begin
definition typedep_FMUSTATE :: "FMUSTATE itself ⇒ utype set" where
[typing]: "typedep_FMUSTATE t = {TYPEREP(FMUSTATE)}"
instance ..
end
```

```
inject_type FMUSTATE
```

Instantiation the relevant classes for the deep value model.

Clearly, this is not possible unless we endow the type with a concrete model using a `typedef` instead of a `typedecl`. On the other hand, the axiom introduced by the `sorry` below ought not lead to unsoundness.

— **TODO**: Introduce a concrete model for FMI states in order to be able to prove the instantiations below and remove the `sorry`.

```
instance FMUSTATE :: "{continuum, two}"
sorry
```

2.6.4 VAR and VAL

The types of `VAR` and `VAL` are next equated with the unified variable and value types of the axiomatic value model. While we could have alternatively used deep variables here, an approach via axiomatic variables implies that there are no restrictions on modelling FMI types. An issue is that `VAL` is clearly not injectable, at least in the original model. The ranked axiomatic model, however, solves that problem.

```
type_synonym VAR = "uvar.uvar" ("VAR")
type_synonym VAL = "uval.uval" ("VAL")
```

Hack: there are still issues with supporting axiomatic variables.

```
instance uval     ::        "{continuum, two}" sorry
instance uvar_ext :: (type) "{continuum, two}" sorry
```

2.6.5 FMIST and FMISTF

We here declare datatypes for `fmi2Status` of the FMI API.

```
datatype FMI2ST =
  fmi2OK |
  fmi2Error |
  fmi2Fatal
```

Instantiation the relevant classes for the axiomatic value model.

```
inject_type FMI2ST
```

Instantiation the relevant classes for the deep value model.

We note that countability implies membership to `continuum`

```
instance FMI2ST :: countable
apply (countable_datatype)
done

instance FMI2ST :: continuum
apply (intro_classes)
done

instance FMI2ST :: two
apply (intro_classes)
apply (rule instance_twoI)
apply (rule_tac x = "fmi2OK" in exI)
apply (rule_tac x = "fmi2Error" in exI)
apply (clarsimp)
done
```

2.6.6 FMUSTF

```
datatype FMI2STF =
  fmi2Status "FMI2ST" |
  fmi2Discard
```

Instantiation the relevant classes for the axiomatic value model.

```
inject_type FMI2STF
```

Instantiation the relevant classes for the deep value model.

We note that countability implies membership to `continuum`

```
instance FMI2STF :: countable
apply (countable_datatype)
done

instance FMI2STF :: continuum
apply (intro_classes)
done

instance FMI2STF :: two
apply (intro_classes)
apply (rule instance_twoI)
apply (rule_tac x = "fmi2Status _" in exI)
apply (rule_tac x = "fmi2Discard" in exI)
apply (clarsimp)
done
```

2.6.7 ErrorFlags

```
typedef ErrorFlags = "{fmi2Error, fmi2Fatal}"
apply (rule_tac x = "fmi2Error" in exI)
apply (clarsimp)
done
```

TODO: Complete the proof below, should not be too difficult.

```
instance ErrorFlags :: "{continuum, two}" sorry
```

2.7 FMI Events

While the trace type for CSP is fixed to `'a list`, we still have to define the event type `'a`. Generally, we can think of events as sum types. Since events may be parametric, there is once again the issue of how to model events with different (HOL) types as a single unified type. A deep model may encode them as pairs consisting of a name & type. Below, we adopt a shallow model that uses a datatype construction. Similar to the more field of extensible records, we endow the datatype with an extension field. It is still an open issue how we conveniently support compositional declarations of new channels; Simon mentioned *prisms* as an analogue of lenses for sum types. As somewhat *ad hoc* solution is presented below.

2.7.1 FMI API Channels

We note that all constructor functions are of type `chan`. To obtain extensible events types, we introduce a prefixing (scoping) operator for each channel type that lifts the underlying datatype into a `sum` type with an open slot for later extension. Eventually, those prefix operators will be introduced automatically by the tool, namely through a custom `events` command for defining channel events.

```
datatype 'τ::ctime fmi_event =
  fmi2Get "FMI2COMP × VAR × VAL × FMI2ST" |
  fmi2Set "FMI2COMP × VAR × VAL × FMI2STF" |
  fmi2DoStep "FMI2COMP × TIME('τ) × NZTIME('τ) × FMI2STF" |
  fmi2Instantiate "FMI2COMP × bool" |
  fmi2SetUpExperiment "FMI2COMP × TIME('τ) × bool × TIME('τ) × FMI2ST" |
  fmi2EnterInitializationMode "FMI2COMP × FMI2ST" |
  fmi2ExitInitializationMode "FMI2COMP × FMI2ST" |
  fmi2GetBooleanStatusfmi2Terminated "FMI2COMP × bool × FMI2ST" |
  fmi2GetMaxStepSize "FMI2COMP × TIME('τ) × FMI2ST" |
  fmi2Terminate "FMI2COMP × FMI2ST" |
  fmi2FreeInstance "FMI2COMP × FMI2ST" |
  fmi2GetFMUState "FMI2COMP × FMUSTATE × FMI2ST" |
  fmi2SetFMUState "FMI2COMP × FMUSTATE × FMI2ST"

abbreviation fmi_prefix ::
  "('a, 'τ::ctime fmi_event) chan ⇒
   ('a, ('τ::ctime fmi_event) + 'ext) chan" where
"fmi_prefix c ≡ Inl o c"

notation fmi_prefix ("fmi:_" [1000] 1000)

abbreviation "fmi_events ≡
  ε(fmi:fmi2Get) ∪
  ε(fmi:fmi2Set) ∪
  ε(fmi:fmi2DoStep) ∪
  ε(fmi:fmi2Instantiate) ∪
  ε(fmi:fmi2SetUpExperiment) ∪
  ε(fmi:fmi2EnterInitializationMode) ∪
  ε(fmi:fmi2ExitInitializationMode) ∪
  ε(fmi:fmi2GetBooleanStatusfmi2Terminated) ∪
  ε(fmi:fmi2GetMaxStepSize) ∪
  ε(fmi:fmi2Terminate) ∪
  ε(fmi:fmi2FreeInstance) ∪
  ε(fmi:fmi2GetFMUState) ∪
```

```
  ε(fmi:fmi2SetFMUState)"
```

2.7.2 Timer Process Channels

```
datatype 'τ::ctime timer_event =
  setT "TIME('τ)" |
  updateSS "NZTIME('τ)" |
  step "TIME('τ) × NZTIME('τ)" |
  endc "unit"

abbreviation timer_prefix ::
  "('a, 'τ::ctime timer_event) chan ⇒
   ('a, ('τ::ctime fmi_event) + ('τ::ctime timer_event) + 'ext) chan" where
"timer_prefix c ≡ Inr o Inl o c"

notation timer_prefix ("tm:_" [1000] 1000)

abbreviation "tm_events ≡
  ε(tm:step) ∪ ε(tm:endc) ∪ ε(tm:setT) ∪ ε(tm:updateSS)"
```

2.7.3 Interaction Process Channels

```
datatype control_event =
  stepToComplete "unit" |
  stepAnalysed "unit" |
  stepComplete "unit" |
  endsimulation "unit" |
  error  "FMI2ST"

abbreviation control_prefix ::
  "('a, control_event) chan ⇒
    ('a, ('τ::ctime fmi_event) + ('τ::ctime timer_event) + (control_event)
      + 'ext) chan" where
"control_prefix c ≡ Inr o Inr o Inl o c"

notation control_prefix ("ctr:_" [1000] 1000)

abbreviation "ctr_events ≡
  ε(ctr:stepToComplete) ∪
  ε(ctr:stepAnalysed) ∪
  ε(ctr:stepComplete) ∪
  ε(ctr:endsimulation)"
```

2.8 FMI Ports

For readability, we introduce a **type_synonym** for ports. A port is encoded by a pair consisting of an FMI component (type `FMI2COMP`) and a variable (type `VAR`). We do not distinguish input and output ports.

```
type_synonym port = "FMI2COMP × VAR"

abbreviation FMU :: "port ⇒ FMI2COMP" where
"FMU port ≡ (fst port)"

abbreviation name :: "port ⇒ VAR" where
"name port ≡ (snd port)"
```

2.9 FMI Configuration

The configuration for a particular FMI system is introduced abstractly via HOL constants. A concrete model can provide overloaded definitions to give concrete meanings to them; this may allow us to potentially prove additional properties. An open question is whether some additional caveats need to be specified already here e.g. that the port dependency graph is acyclic. This could possibly be done through a type definitions. We note that we encode the Z type `seq` by HOL's list type `'a list`.

In line with the example in the deliverable D2.2d, I introduced a function `initialValues` rather than using the `inputs` sequence in order to provide initial values. This also proves to be slightly more convenient in terms of mechanisation. Also, I changed the type of the port-dependency graph to become a function rather than a relation, associating a list of inputs with each outputs. The advantage of this is that it facilitates the definition of the `DistributeInputs` action since currently, iterated sequence of actions is only define by lists but not for (finite) sets.

```
consts FMUs :: "FMI2COMP list"
consts parameters :: "(FMI2COMP × VAR × VAL) list"
consts initialValues :: "(FMI2COMP × VAR × VAL) list"
— Before: consts inputs :: "(FMI2COMP × VAR × VAL) list"
consts inputs :: "port list"
consts outputs :: "port list"
— Before: consts pdg :: "port relation".
consts pdg :: "port ⇒ (port list)"
```

2.10 Simulation Parameters

For now, I added the following as global constants too.

```
consts startTime :: "TIME('τ)"
consts stopTimeDefined :: "bool"
consts stopTime :: "TIME('τ)"
```

We can instantiate them as in the example of the D2.2d deliverable.

```
overloading D2_2d_startTime ≡ "startTime :: TIME('τ)"
begin
  definition D2_2d_startTime :: "TIME('τ)" where
  "D2_2d_startTime = 0"
end

overloading D2_2d_stopTimeDefined ≡ "stopTimeDefined :: bool"
begin
  definition D2_2d_stopTimeDefined :: "bool" where
  "D2_2d_stopTimeDefined = True"
end

overloading D2_2d_stopTime ≡ "stopTime :: TIME('τ)"
begin
  definition D2_2d_stopTime :: "TIME('τ)" where
  "D2_2d_stopTime = 5"
end
```

2.11 FMI Processes

A problem with the *Circus* process encoding below is that, currently, it is not possible to write prefixes that mix inputs and outputs. Hence, I had to adopt a trick which converts everything into a single input prefix. That input imposes constraints so that some parts of the communication effectively behave like outputs. A second issue is that, referring to page 16 of D2.2d, we see that the `AllowGetsAndSets` action is actually parametric. My translation currently does not support parametric actions; hence I adopted a solution that encodes procedure parameters by local variables. Due to the proper treatment of scope by Isabelle/UTP, we generally get away with this. Both issues can thus be overcome; an integration of syntax translations that conceals the manual rewrites and adjustments done below is pending work.

2.11.1 General Timer

TODO: Write the same process as below with axiomatic variables.

definition
```
"process Timer(ct::TIME('τ), hc::NZTIME('τ), tN::TIME('τ)) ≜ begin
  Step =
    (tm:setT?ᵤ(t : «t ≤ tN») → (<currentTime> := &t) ;; Step) □
    (tm:updateSS?ᵤ(ss : true) → (<stepSize> := &ss) ;; Step) □
    (tm:step!ᵤ((&<currentTime>, &<stepsize>)ᵤ) →
      (<currentTime::'τ> :=
        minᵤ(&<currentTime> + §(&<stepSize::'τ pos>), «tN»)) ;; Step) □
    ((&<currentTime> =ᵤ «tN») &ᵤ tm:endc →ᵤ Stop)
  • (<currentTime>, <stepSize> := «ct», «hc») ;; Step
end"
```

definition
```
"process TimerNew(ct::TIME('τ), hc::NZTIME('τ), tN::TIME('τ)) ≜ begin
  Step =
    (tm:setT?(t : «t ≤ tN») →𝒞 (<currentTime> := «t») ;; Step) □
    (tm:updateSS?(ss) →𝒞 (<stepSize> := «ss») ;; Step) □
    (tm:step![out₁]($<currentTime>)![out₂]($<stepsize>) →𝒞
      (<currentTime::'τ> :=
        minᵤ(&<currentTime> + §(&<stepSize::'τ pos>), «tN»)) ;; Step) □
    ((&<currentTime> =ᵤ «tN») &ᵤ tm:endc →𝒞 Stop)
  • (<currentTime>, <stepSize> := «ct», «hc») ;; Step
end"
```

2.11.2 Interaction

Note that I changed the type of `rinps` with respect to the tentative model given in the deliverable D2.2d. That is, instead of using the partial function type `FMI2COMP ⇀ VAR ⇀ VAL` for `rinps`, I decided to use the list `((FMI2COMP × VAR) × VAL) list`. This is (currently) a technicality since there are issues with using function types in prefixes, to do with Simon's embedding of shallow variables. Using lists circumvents the issue for and ought not limit generality since we may reasonably assume that the port-dependency graph is a finite relation.

Process State: `rinps`.

definition
```
"process Interaction ≜ begin
  Instantiation = (;; i : FMUs •
    fmi:fmi2Instantiate?ᵤ(i_sc : π₁(«i_sc») =ᵤ «i») → Skip) and

  InstantiationMode =
    (if𝒞 «parameters = []» then𝒞
      (;; i : FMUs •
```

```
      fmi:fmi2SetUpExperiment?_u(i_startTime_stopTimeDefined_stopTime_st :
        π_1(«i_startTime_stopTimeDefined_stopTime_st») =_u «i» ∧
        π_1(π_2(«i_startTime_stopTimeDefined_stopTime_st»)) =_u «startTime» ∧
        π_1(π_2(π_2(«i_startTime_stopTimeDefined_stopTime_st»))) =_u «stopTimeDefined» ∧
        π_1(π_2(π_2(π_2(«i_startTime_stopTimeDefined_stopTime_st»)))) =_u «stopTime»)
          → Skip) ;;
    (;; i : FMUs ·
      fmi:fmi2EnterInitializationMode?_u(i_st : π_1(«i_st») =_u «i») → Skip)
  else_C
    (;; i_x_v : parameters ·
      fmi:fmi2Set?_u(i_x_v_st :
        π_1(«i_x_v_st») =_u π_1(«i_x_v») ∧
        π_1(π_2(«i_x_v_st»)) =_u π_1(π_2(«i_x_v»)) ∧
        π_1(π_2(π_2(«i_x_v_st»))) =_u π_2(π_2(«i_x_v»))) → Skip) ;;
    (;; i : FMUs ·
      fmi:fmi2SetUpExperiment?_u(i_startTime_stopTimeDefined_stopTime_st :
        π_1(«i_startTime_stopTimeDefined_stopTime_st») =_u «i» ∧
        π_1(π_2(«i_startTime_stopTimeDefined_stopTime_st»)) =_u «startTime» ∧
        π_1(π_2(π_2(«i_startTime_stopTimeDefined_stopTime_st»))) =_u «stopTimeDefined» ∧
        π_1(π_2(π_2(π_2(«i_startTime_stopTimeDefined_stopTime_st»)))) =_u «stopTime»)
          → Skip) ;;
    (;; i : FMUs ·
      fmi:fmi2EnterInitializationMode?_u(i_st : π_1(«i_st») =_u «i») → Skip)) and

InitializationMode =
  (if_C «initialValues = []» then_C
    (;; i : FMUs ·
      fmi:fmi2ExitInitializationMode?_u(i_st : π_1(«i_st») =_u «i») → Skip)
  else_C
    (;; i_x_v : initialValues ·
      fmi:fmi2Set?_u(i_x_v_st :
        π_1(«i_x_v_st») =_u π_1(«i_x_v») ∧
        π_1(π_2(«i_x_v_st»)) =_u π_1(π_2(«i_x_v»)) ∧
        π_1(π_2(π_2(«i_x_v_st»))) =_u π_2(π_2(«i_x_v»))) → Skip) ;;
    (;; i : FMUs ·
      fmi:fmi2ExitInitializationMode?_u(i_st : π_1(«i_st») =_u «i») → Skip)) and

Terminated = (;; i : FMUs ·
  fmi:fmi2Terminate?_u(i_st : π_1(«i_st») =_u «i») →
  fmi:fmi2FreeInstance?_u(i_st : π_1(«i_st») =_u «i») → Skip) ;;
  ctr:endsimulation →_u Skip and

TakeOutputs = <rinp::(port × VAL) list> := ⟨⟩ ;;
  (;; out : outputs · fmi:fmi2Get?_u(i_x_v_st :
    π_1(«i_x_v_st») =_u «FMU out» ∧
    π_1(π_2(«i_x_v_st»)) =_u «name out») →
      (;; inp : pdg out ·
        <rinp> := &<rinp> ^_u ⟨(«inp», π_1(π_2(π_2(&i_x_v_st))))_u⟩)) and

DistributeInputs = (;; inp : inputs ·
  fmi:fmi2Set?_u(i_x_v_st :
    π_1(«i_x_v_st») =_u «FMU inp» ∧
    π_1(π_2(«i_x_v_st»)) =_u «name inp» ∧
    π_1(π_2(π_2(«i_x_v_st»))) =_u (lookup_u $<rinp> «inp»)) → Skip) and
```

```
  Step = (;; i : [0..(length FMUs)] ·
    (if (i::int) = 0 then
      ctr:stepToComplete →u
        (fmi:fmi2DoStep?u(i_t_hc_st :
          π1(«i_t_hc_st») =u «nth FMUs 1» ∧
          π1(π2(«i_t_hc_st»)) =u $<t> ∧
          π1(π2(π2(«i_t_hc_st»))) =u $<hc>) → Skip)
    else if (i::int) < (length FMUs) then
      (⨅X.
        (fmi:fmi2GetBooleanStatusfmi2Terminated?u(i_b_st :
          π1(«i_b_st») =u «nth FMUs (nat i)») → X) □
        (fmi:fmi2GetMaxStepSize?u(i_t_st :
          π1(«i_t_st») =u «nth FMUs (nat i)») → X)) □
        (fmi:fmi2DoStep?u(i_t_hc_st :
          π1(«i_t_hc_st») =u «nth FMUs (nat (i+1))» ∧
          π1(π2(«i_t_hc_st»)) =u $<t> ∧
          π1(π2(π2(«i_t_hc_st»))) =u $<hc>) → Skip)
    else
      (⨅X.
        (fmi:fmi2GetBooleanStatusfmi2Terminated?u(i_b_st :
          π1(«i_b_st») =u «nth FMUs (nat i)») → X) □
        (fmi:fmi2GetMaxStepSize?u(i_t_st :
          π1(«i_t_st») =u «nth FMUs (nat i)») → X)) □
        (ctr:stepAnalysed →u Skip))) and

  slaveInitialized =
    (tm:endc →u Terminated) □
    (tm:step?u(t_hc : true) →
      (* Used local variables to pass action parameters! *)
      (<t>, <hc> := π1(&t_hc), π2(&t_hc)) ;;
      TakeOutputs ;; DistributeInputs ;; Step) and

  NextStep =
    (tm:updateSS?u(d : true) → NextStep) □
    (tm:setT?u(t : true) → NextStep) □
    (slaveInitialized ;; NextStep) □
    (Terminated)

  · Instantiation ;; InstantiationMode ;; InitializationMode ;; slaveInitialized
end"
```

theorem "P Interaction"
apply (unfold Interaction_def)
apply (simp add: circus_syntax Let_def)
oops

A simplified definition of the same (?) process is given below.

definition

```
"process InteractionSimplified ≜ begin
  Instantiation = (;; i : FMUs ·
    fmi:fmi2Instantiate?u(i_sc : π1(«i_sc») =u «i») → Skip) and

  InstantiationMode =
    (;; i_x_v : parameters ·
      fmi:fmi2Set?u(i_x_v_st :
```

```
      π_1(≪i_x_v_st≫) =_u π_1(≪i_x_v≫) ∧
      π_1(π_2(≪i_x_v_st≫)) =_u π_1(π_2(≪i_x_v≫)) ∧
      π_1(π_2(π_2(≪i_x_v_st≫))) =_u π_2(π_2(≪i_x_v≫))) → Skip) ;;
  (;; i : FMUs ·
    fmi:fmi2SetUpExperiment?_u(i_startTime_stopTimeDefined_stopTime_st :
      π_1(≪i_startTime_stopTimeDefined_stopTime_st≫) =_u ≪i≫ ∧
      π_1(π_2(≪i_startTime_stopTimeDefined_stopTime_st≫)) =_u ≪startTime≫ ∧
      π_1(π_2(π_2(≪i_startTime_stopTimeDefined_stopTime_st≫))) =_u ≪stopTimeDefined≫ ∧
      π_1(π_2(π_2(π_2(≪i_startTime_stopTimeDefined_stopTime_st≫)))) =_u ≪stopTime≫)
        → Skip) ;;
  (;; i : FMUs ·
    fmi:fmi2EnterInitializationMode?_u(i_st : π_1(≪i_st≫) =_u ≪i≫) → Skip) and

InitializationMode =
  (;; i_x_v : initialValues ·
    fmi:fmi2Set?_u(i_x_v_st :
      π_1(≪i_x_v_st≫) =_u π_1(≪i_x_v≫) ∧
      π_1(π_2(≪i_x_v_st≫)) =_u π_1(π_2(≪i_x_v≫)) ∧
      π_1(π_2(π_2(≪i_x_v_st≫))) =_u π_2(π_2(≪i_x_v≫))) → Skip) ;;
  (;; i : FMUs ·
    fmi:fmi2ExitInitializationMode?_u(i_st : π_1(≪i_st≫) =_u ≪i≫) → Skip) and

Terminated = (;; i : FMUs ·
  fmi:fmi2Terminate?_u(i_st : π_1(≪i_st≫) =_u ≪i≫) →
  fmi:fmi2FreeInstance?_u(i_st : π_1(≪i_st≫) =_u ≪i≫) → Skip) ;;
  ctr:endsimulation →_u Skip and

TakeOutputs = <rinp::(port × VAL) list> := ⟨⟩ ;;
  (;; out : outputs · fmi:fmi2Get?_u(i_x_v_st :
    π_1(≪i_x_v_st≫) =_u ≪FMU out≫ ∧
    π_1(π_2(≪i_x_v_st≫)) =_u ≪name out≫) →
      (;; inp : pdg out ·
        <rinp> := &<rinp> ^_u ⟨(≪inp≫, π_1(π_2(π_2(&i_x_v_st))))_u⟩)) and

DistributeInputs = (;; inp : inputs ·
  fmi:fmi2Set?_u(i_x_v_st :
    π_1(≪i_x_v_st≫) =_u ≪FMU inp≫ ∧
    π_1(π_2(≪i_x_v_st≫)) =_u ≪name inp≫ ∧
    π_1(π_2(π_2(≪i_x_v_st≫))) =_u (lookup_u $<rinp> ≪inp≫)) → Skip) and

Step = (;; i : [0..(length FMUs)] ·
  (if (i::int) = 0 then
    ctr:stepToComplete →_u
      (fmi:fmi2DoStep?_u(i_t_hc_st :
        π_1(≪i_t_hc_st≫) =_u ≪nth FMUs 1≫ ∧
        π_1(π_2(≪i_t_hc_st≫)) =_u $<t> ∧
        π_1(π_2(π_2(≪i_t_hc_st≫))) =_u $<hc>) → Skip)
  else if (i::int) < (length FMUs) then
    (⨅X.
      (fmi:fmi2GetBooleanStatusfmi2Terminated?_u(i_b_st :
        π_1(≪i_b_st≫) =_u ≪nth FMUs (nat i)≫) → X) □
      (fmi:fmi2GetMaxStepSize?_u(i_t_st :
        π_1(≪i_t_st≫) =_u ≪nth FMUs (nat i)≫) → X)) □
      (fmi:fmi2DoStep?_u(i_t_hc_st :
        π_1(≪i_t_hc_st≫) =_u ≪nth FMUs (nat (i+1))≫ ∧
```

```
        π1(π2(«i_t_hc_st»)) =u $<t> ∧
        π1(π2(π2(«i_t_hc_st»))) =u $<hc>) → Skip)
  else
    (⨅X.
      (fmi:fmi2GetBooleanStatusfmi2Terminated?u(i_b_st :
        π1(«i_b_st») =u «nth FMUs (nat i)») → X) □
      (fmi:fmi2GetMaxStepSize?u(i_t_st :
        π1(«i_t_st») =u «nth FMUs (nat i)») → X)) □
      (ctr:stepAnalysed →u Skip))) and

 slaveInitialized =
   (tm:endc →u Terminated) □
   (tm:step?u(t_hc : true) →
     (* Used local variables to pass action parameters! *)
     (<t>, <hc> := π1(&t_hc), π2(&t_hc)) ;;
     TakeOutputs ;; DistributeInputs ;; Step) and

 NextStep =
   (tm:updateSS?u(d : true) → NextStep) □
   (tm:setT?u(t : true) → NextStep) □
   (slaveInitialized ;; NextStep) □
   (Terminated)

 · Instantiation ;; InstantiationMode ;; InitializationMode ;; slaveInitialized
end"
```

definition

```
"process InteractionNew ≜ begin
 Instantiation = (;; i : FMUs ·
   fmi:fmi2Instantiate.[out1](«i»)?(sc) →C Skip) and

 InstantiationMode =
   (;; (i, x, v) : parameters ·
     fmi:fmi2Set![out1](«i»)![out2](«x»)![out3](«v»)?(st) →C Skip) ;;
   (;; i : FMUs ·
     fmi:fmi2SetUpExperiment![out1](«i»)![out2](«startTime»)
       ![out3](«stopTimeDefined»)![out4](«stopTime»)?(st) →C Skip) ;;
   (;; i : FMUs ·
     fmi:fmi2EnterInitializationMode.[out1](«i»)?(sc) →C Skip) and

 InitializationMode =
   (;; (i, x, v) : initialValues ·
     fmi:fmi2Set![out1](«i»)![out2](«x»)![out3](«v»)?(st) →C Skip) ;;
   (;; i : FMUs ·
     fmi:fmi2ExitInitializationMode![out1](«i»)?(sc) →C Skip) and

 Terminated =
   (;; i : FMUs ·
     fmi:fmi2Terminate.[out1](«i»)?(sc) →C
     fmi:fmi2FreeInstance.[out1](«i»)?(sc) →C Skip) ;;
   ctr:endsimulation →C Skip and

 TakeOutputs = <rinp::(port × VAL) list> := ⟨⟩ ;;
   (;; out : outputs ·
     fmi:fmi2Get.[out1](«FMU out»).[out2](«name out»)?(v)?(st) →C
```

```
      (;; inp : pdg out · <rinp> := &<rinp> ^_u ⟨(≪inp≫, ≪v≫)_u⟩)) and

  DistributeInputs = (;; inp : inputs ·
    fmi:fmi2Set.[out_1](≪FMU inp≫).[out_2](≪name inp≫)
      ![out_3](lookup_u $<rinp> ≪inp≫)?(st) →_C Skip) and

  Step = (;; i : [0..(length FMUs)] ·
    (if (i::int) = 0 then
      ctr:stepToComplete →_C
        (fmi:fmi2DoStep.[out_1](≪FMUs!0≫).[out_2]($<t>).[out_3]($<hc>)?(st) →_C Skip)
    else if (i::int) < (length FMUs) then
      (⊓X.
        (fmi:fmi2GetBooleanStatusfmi2Terminated.[out_1](≪FMUs!(nat (i-1))≫)?(b)?(st) →_C
X) □
        (fmi:fmi2GetMaxStepSize.[out_1](≪FMUs!(nat (i-1))≫)?(t)?(st) →_C X)) □
        (fmi:fmi2DoStep.[out_1](≪FMUs!(nat i)≫).[out_2]($<t>).[out_3]($<hc>)?(st) →_C Skip)
    else
      (⊓X.
        (fmi:fmi2GetBooleanStatusfmi2Terminated.[out_1](≪FMUs!(nat (i-1))≫)?(b)?(st) →_C
X) □
        (fmi:fmi2GetMaxStepSize.[out_1](≪FMUs!(nat (i-1))≫)?(t)?(st) →_C X)) □
        (ctr:stepAnalysed →_C Skip))) and

  slaveInitialized =
    (tm:endc →_C Terminated) □
    (tm:step?(t)?(hc) →_C
      (* We use local variables to pass action parameters! *)
      (<t>, <hc> := ≪t≫, ≪hc≫) ;;
      TakeOutputs ;; DistributeInputs ;; Step) and

  NextStep =
    (tm:updateSS?(d) →_C NextStep) □
    (tm:setT?(t) →_C NextStep) □
    (slaveInitialized ;; NextStep) □
    (Terminated)

  · Instantiation ;; InstantiationMode ;; InitializationMode ;; slaveInitialized
end"
```

print_theorems

2.11.3 End Simulation

definition
"endSimulation = ctr:endsimulation →_u Skip"

2.11.4 States Managers

TODO: Write the same process as below with axiomatic variables.

definition

```
"process FMUStatesManager(i::FMI2COMP) ≜ begin
  AllowsGetsAndSets =
    (fmi:fmi2GetFMUState?_u(i_s_st : π_1(≪i_s_st≫) =_u ≪i≫) →
      (<s> := π_1(π_2(&i_s_st)) ;; AllowsGetsAndSets)) □
    (fmi:fmi2SetFMUState?_u(i_s_st : π_1(≪i_s_st≫) =_u ≪i≫ ∧ π_1(π_2(≪i_s_st≫)) =_u $<s>) →
```

```
    (<s> := π₁(π₂(&i_s_st)) ;; AllowsGetsAndSets)) and
  AllowAGet =
    (fmi:fmi2GetFMUState?ᵤ(i_s_st : π₁(≪i_s_st≫) =ᵤ ≪i≫) →
      (<s> := π₁(π₂(&i_s_st)) ;; AllowsGetsAndSets))
  • fmi:fmi2Instantiate?ᵤ(i_b : π₁(≪i_b≫) =ᵤ ≪i≫) → AllowAGet
end"
```

definition
```
"process NoStatesManager ≜
  ( ||| i : FMUs • fmi:fmi2Instantiate?ᵤ(i_b : π₁(≪i_b≫) =ᵤ ≪i≫) → Stop) △
    endSimulation"
```

theorem `"FMUStatesManager i = ABC"`
apply `(unfold FMUStatesManager_def)`
apply `(simp add: circus_syntax)`
oops

2.11.5 Error Handling

definition
```
"process ErrorMonitor(mst::FMI2ST) ≜
begin
  StopError =
    ((&<st> =ᵤ ≪mst≫) &ᵤ ctr:error!ᵤ(≪mst≫) → (*Monitor*) Skip) □
    ((&<st> ≠ᵤ ≪mst≫) &ᵤ ctr:error!ᵤ(≪mst≫) → (*Monitor*) Skip) and

  Monitor =
    (fmi:fmi2Get?ᵤ(i_n_v_st : true) →
      (<st> := π₂(π₂(π₂(&i_n_v_st))) ;; StopError))

  • Monitor △ (ctr:endsimulation →ᵤ Skip)
end"
```

2.11.6 Master Algorithm

definition
```
"process FMUStatesManagers ≜
  ||| i : FMUs • FMUStatesManager(i) △ endSimulation"
```

definition
```
"process TimedInteraction(t0, tN) ≜
  ((Timer(t0, Abs_pos 2, tN) △ endSimulation)
    [| tm_events ∪ ε(ctr:endsimulation) |]
  Interaction) \ (ε(ctr:stepAnalysed) ∪ ε(ctr:stepComplete)) \ tm_events"
```

definition
```
"process MAlgorithm(t0, tN) ≜
  (TimedInteraction(t0, tN)
    [| ε(ctr:endsimulation) ∪ ε(fmi:fmi2Instantiate) |]
  FMUStatesManagers)"
```

2.11.7 General Bejaviour of an FMU

definition `RUN :: "'ε set ⇒ ('σ, 'ε) action"` **where**
`"RUN evts = undefined"`

2.12 Proof Experiments

```
term "setT!u(«0») → SKIP"
term "InputCSP setT x"
term "⌈''x''⌉d"
term "(dvar_lens ⌈''x''⌉d)"
term "⌈''x''⌉d↑"
term "InputCircus setT (⌈''x''⌉d↑)"
end
```

3 Railways Mechanisation

```
theory railways
imports fmi String
begin
```

3.1 FM2 Types

This should be moved to the Isabelle theory `fmi`.

```
type_synonym fmi2Real = "real"
type_synonym fmi2Integer = "int"
type_synonym fmi2String = "string"
type_synonym fmi2Boolean = "bool"
```

3.2 Railways Constants

Track Segments: CDV1-CDV11

```
definition "CDV1 = (1::fmi2Integer)"
definition "CDV2 = (2::fmi2Integer)"
definition "CDV3 = (3::fmi2Integer)"
definition "CDV4 = (4::fmi2Integer)"
definition "CDV5 = (5::fmi2Integer)"
definition "CDV6 = (6::fmi2Integer)"
definition "CDV7 = (7::fmi2Integer)"
definition "CDV8 = (8::fmi2Integer)"
definition "CDV9 = (9::fmi2Integer)"
definition "CDV10 = (10::fmi2Integer)"
definition "CDV11 = (11::fmi2Integer)"
```

Available Routes: V1Q1/V1Q2/Q2V2/V1Q3/Q3V2

```
definition "V1Q1 = (1::fmi2Integer)"
definition "V1Q2 = (2::fmi2Integer)"
definition "Q2V2 = (3::fmi2Integer)"
definition "V1Q3 = (4::fmi2Integer)"
definition "Q3V2 = (5::fmi2Integer)"
```

Signal Encoding

TODO: Use "Isabelle Theories for Machine Words" by Jeremy Dawson.

```
definition "RED == False"
definition "GREEN == True"

fun signals :: "(bool × bool × bool) ⇒ fmi2Integer" where
"signals (s1, s2, s3) =
  (if s1 then 1 else 0) +
  (if s2 then 2 else 0) +
  (if s3 then 4 else 0)"
```

Track Switch Encoding

TODO: Use "Isabelle Theories for Machine Words" by Jeremy Dawson.

```
definition "STRAIGHT == False"
definition "DIVERGING == True"
```

```
fun switches :: "(bool × bool × bool × bool × bool) ⇒ fmi2Integer" where
"switches (sw1, sw2, sw3, sw4, sw5) =
  (if sw1 then 1 else 0) +
  (if sw2 then 2 else 0) +
  (if sw3 then 4 else 0) +
  (if sw4 then 8 else 0) +
  (if sw5 then 16 else 0)"
```

Railways FMUs

```
axiomatization
  train1 :: "FMI2COMP" and train2 :: "FMI2COMP" and
  merger :: "FMI2COMP" and interlocking :: "FMI2COMP" where
  fmus_distinct : "distinct [train1, train2, merger, interlocking]" and
  FMI2COMP_def : "FMI2COMP = {train1, train2, merger, interlocking}"
```

Proof Support

```
code_datatype "train1" "train2" "merger" "interlocking"

lemma fmus_simps [simp]:
"train1 ≠ train2"
"train1 ≠ merger"
"train1 ≠ interlocking"
"train2 ≠ train1"
"train2 ≠ merger"
"train2 ≠ interlocking"
"merger ≠ train1"
"merger ≠ train2"
"merger ≠ interlocking"
"interlocking ≠ train1"
"interlocking ≠ train2"
"interlocking ≠ merger"
using railways.fmus_distinct apply (auto)
done

lemma fmus_eq_simps [code]:
"equal_class.equal train1 train1 ≡ True"
"equal_class.equal train1 train2 ≡ False"
"equal_class.equal train1 merger ≡ False"
"equal_class.equal train1 interlocking ≡ False"
"equal_class.equal train2 train1 ≡ False"
"equal_class.equal train2 train2 ≡ True"
"equal_class.equal train2 merger ≡ False"
"equal_class.equal train2 interlocking ≡ False"
"equal_class.equal merger train1 ≡ False"
"equal_class.equal merger train2 ≡ False"
"equal_class.equal merger merger ≡ True"
"equal_class.equal merger interlocking ≡ False"
"equal_class.equal interlocking train1 ≡ False"
"equal_class.equal interlocking train2 ≡ False"
"equal_class.equal interlocking merger ≡ False"
"equal_class.equal interlocking interlocking ≡ True"
apply (unfold equal_FMI2COMP_def)
apply (simp_all only: fmus_simps refl)
done
```

3.3 Parameters

```
overloading
  railways_parameters ≡ "parameters :: (FMI2COMP × VAR × VAL) list"
begin
  definition railways_parameters :: "(FMI2COMP × VAR × VAL) list" where
  "railways_parameters = [
    (train1, $max_speed:{fmi2Real}_u, InjU (4.16::real)),
    (train2, $max_speed:{fmi2Real}_u, InjU (4.16::real)),
    (train1, $fixed_route:{fmi2Integer}_u, InjU V1Q2),
    (train2, $fixed_route:{fmi2Integer}_u, InjU Q3V2)]"
end
```

3.4 Inputs

```
overloading
  railways_inputs ≡ "inputs :: (FMI2COMP × VAR) list"
begin
  definition railways_inputs :: "(FMI2COMP × VAR) list" where
  "railways_inputs = [
    (train1, $signals:{fmi2Integer}_u),
    (train1, $switches:{fmi2Integer}_u),
    (train2, $signals:{fmi2Integer}_u),
    (train2, $switches:{fmi2Integer}_u),
    (merger, $track_segment1:{fmi2Integer}_u),
    (merger, $track_segment2:{fmi2Integer}_u),
    (merger, $telecommand1:{fmi2Integer}_u),
    (merger, $telecommand2:{fmi2Integer}_u),
    (interlocking, $CDV:{fmi2Integer}_u),
    (interlocking, $TC:{fmi2Integer}_u)]"
end
```

3.5 Outputs

```
overloading
  railways_outputs ≡ "outputs :: (FMI2COMP × VAR) list"
begin
  definition railways_outputs :: "(FMI2COMP × VAR) list" where
  "railways_outputs = [
    (train1, $track_segment:{fmi2Integer}_u),
    (train1, $telecommand:{fmi2Integer}_u),
    (train2, $track_segment:{fmi2Integer}_u),
    (train2, $telecommand:{fmi2Integer}_u),
    (merger, $CDV:{fmi2Integer}_u),
    (merger, $TC:{fmi2Integer}_u),
    (merger, $collision:{fmi2Boolean}_u),
    (merger, $derailment:{fmi2Boolean}_u),
    (interlocking, $signals:{fmi2Integer}_u),
    (interlocking, $switches:{fmi2Integer}_u)]"
end
```

3.6 Initial Values

The following constants have to be defined as appropriate.

```
definition "initialSignals = InjU (signals (RED, RED, RED))"
definition "initialSwitches =
```

```
  InjU (switches (STRAIGHT, STRAIGHT, STRAIGHT, STRAIGHT, STRAIGHT))"
definition "initialTrack1 = InjU CDV3"
definition "initialTrack2 = InjU CDV2"
```

What about the initial values for telecommand, CDV and TC?

```
overloading
  railways_initialValues ≡ "initialValues :: (FMI2COMP × VAR × VAL) list"
begin
  definition railways_initialValues :: "(FMI2COMP × VAR × VAL) list" where
  "railways_initialValues = [
    (train1, $signals:{fmi2Integer}u, initialSignals),
    (train1, $switches:{fmi2Integer}u, initialSwitches),
    (train2, $signals:{fmi2Integer}u, initialSignals),
    (train2, $switches:{fmi2Integer}u, initialSwitches),
    (merger, $track_segment1:{fmi2Integer}u, initialTrack1),
    (merger, $track_segment2:{fmi2Integer}u, initialTrack2),
    (merger, $telecommand1:{fmi2Integer}u, undefined),
    (merger, $telecommand2:{fmi2Integer}u, undefined),
    (interlocking, $CDV:{fmi2Integer}u, undefined),
    (interlocking, $TC:{fmi2Integer}u, undefined)]"
end
```

3.7 Port Dependency Graph (PDG)

```
definition pdg :: "port relation" where
"pdg = {
  (* External Dependencies (Connections) *)
  ((train1, $track_segment:{fmi2Integer}u), (merger, $track_segment1:{fmi2Integer}u)),
  ((train2, $track_segment:{fmi2Integer}u), (merger, $track_segment2:{fmi2Integer}u)),
  ((train1, $telecommand:{fmi2Integer}u), (merger, $telecommand:{fmi2Integer}u)),
  ((train2, $telecommand:{fmi2Integer}u), (merger, $telecommand:{fmi2Integer}u)),
  ((merger, $CDV:{fmi2Integer}u), (interlocking, $CDV:{fmi2Integer}u)),
  ((merger, $TC:{fmi2Integer}u), (interlocking, $TC:{fmi2Integer}u)),
  ((interlocking, $signals:{fmi2Integer}u), (train1, $signals:{fmi2Integer}u)),
  ((interlocking, $signals:{fmi2Integer}u), (train2, $signals:{fmi2Integer}u)),
  ((interlocking, $switches:{fmi2Integer}u), (train1, $switches:{fmi2Integer}u)),
  ((interlocking, $switches:{fmi2Integer}u), (train2, $switches:{fmi2Integer}u)),
  (* Internal Dependencies (Direct) *)
  (* The next are not direct dependencies due to integrators in the CTL. *)
  (* ((train1, $switches:{fmi2Integer}u), (train1, $track_segment:{fmi2Integer}u)), *)
  (* ((train2, $switches:{fmi2Integer}u), (train1, $track_segment:{fmi2Integer}u)), *)
  ((merger, $track_segment1:{fmi2Integer}u), (merger, $CDV:{fmi2Integer}u)),
  ((merger, $track_segment2:{fmi2Integer}u), (merger, $CDV:{fmi2Integer}u)),
  ((merger, $telecommand:{fmi2Integer}u), (merger, $TC:{fmi2Integer}u)),
  ((merger, $telecommand:{fmi2Integer}u), (merger, $TC:{fmi2Integer}u)),
  ((interlocking, $CDV:{fmi2Integer}u), (interlocking, $signals:{fmi2Integer}u)),
  ((interlocking, $CDV:{fmi2Integer}u), (interlocking, $switches:{fmi2Integer}u)),
  ((interlocking, $TC:{fmi2Integer}u), (interlocking, $signals:{fmi2Integer}u)),
  ((interlocking, $TC:{fmi2Integer}u), (interlocking, $switches:{fmi2Integer}u))
}"
```

Needed to enable evaluation of `STR s1 = STR s2` terms.

```
declare equal_literal.rep_eq [code del]
```

We next prove via code evaluation that the PDG is acyclic indeed.

lemma "acyclic pdg"
apply (eval)
done
end