Towards a verified and computable theory of compositional dynamic systems

Pieter Collins[1], Bas Laarakker[1], Sewon Park[2],
Sacha Sindorf[1] & Holger Thies[2]

[1] Department of Advanced Computing Sciences
Maastricht University

[2] Research Institute for Mathematical Sciences
Kyoto University

Computability and Complexity in Analysis (CCA)
Dubrovnik, Croatia, 9 September3 2023

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 731143.

Outline

- Introduction to Dynamic Systems Theory
- Motivation from Model Checking Hybrid Systems
- Discrete-Time Systems
- Timed-Event Systems
- Further Research

Introduction — Dynamic Systems Theory

Dynamic systems

Dynamic systems are mathematical models of phenomena which change in time.

- Systems have an internal *state*, which describes the system at a given moment in time, and some kind of *law* governing the future evolution.
- An *execution* or *trajectory* of a system is a particular evolution.
- The *behaviour* of a system is the set of *all* possible executions.

We usually cannot see (and are not interested in) the complete working of the system.

- A *trace* of the evolution the restriction of an evolution to the part which is visible to the outside world.
- The *external* or *observable behaviour* comprises the set of all possible traces.

The *composition* of systems is a larger system build up from *component* subsystems.

- The traces of the composed system should be exactly (and definitely not more than) those which project to traces of the component subsystems.
- The order of compositon is irrelevant.
- We can usually derive the law of a composed system from that of its components.

Input-output systems

External variables can be partitioned into:

Inputs, by which the outside world can affect the system, and

Outputs, which are determined by the system law, and can affect the outside world.

The *input-output* behaviour is the mapping from input traces to (sets of possible) output traces.

Input-output behaviours are *causal* or *nonanticipating*, which means that the current output cannot depend on future inputs.

They should be *input-enabling*, meaning that any input has an output.
Combined with causality, this yields *input-accepting*, meaning that for any input, any partial output can be extended to a complete output.

Input-output behaviours with a unique output for every input are *deterministic*.

Composition entails feeding outputs of one component into another, possibly yielding *feedback* loops.

The input-output distinction appears crucial in developing a viable computable *systems theory.*

System classes

In this talk, we will look at two different kinds of system:

- Discrete-time systems, with traces $\ltimes\supset\approx \to W$ for some space W.
- Timed-event systems, with traces $(\mathbb{R}^+ \times E)^*$ for *events* $e : E$, and *increasing* event times t_n.

We will eventually want to extend the results to:

- Continuous-time systems, with traces $\mathbb{R}^+ \to W$, which are common in physics.
- Hybrid-time systems, with traces interleaving discrete events e_n at times t_n, with continuous evolution $[t_n : t_{n+1}] \to W$ inbetween; these are common in engineering.

Mathematical definitions

A *deterministic discrete-time input-output system* has behaviour
$$b : \mathcal{B}\langle U; Y\rangle := (\ltimes\supset\approx \to U) \to (\ltimes\supset\approx \to Y).$$

A *state-space model* is a tuple $s = \langle f : X \times U \to X,\ h : X \times U \to Y,\ e : X\rangle$.

For a given input $u : \ltimes\supset\approx \to U$:

The *trajectory* $x : \ltimes\supset\approx \to X$ is given by $x_0 = e \wedge \forall n,\ x_{n+1} = f(x_n, u_n)$.

The *output* $y : \ltimes\supset\approx \to Y$ is given by $y_n = h(x_n, u_n)$.

The *behaviour operator* gives the system behaviour from the model.

A behaviour b is *causal* if
$$\forall u, u',\ y := b(u),\ y' := b(u'),\ u|_{\leq n} = u'|_{\leq n} \implies y|_{\leq n} = y'|_{\leq n}.$$
The behaviour of a state-space model is causal and (trivially) input-enabling.

A behaviour $b_{12} : \mathcal{B}\langle U; Y_1 \times Y_2\rangle$ is a valid *composed* behaviour of $b_1 : \mathcal{B}\langle U \times Y_2; Y_1\rangle$ and $b_2 : \mathcal{B}\langle U \times Y_1; Y_2\rangle$ if
$$\forall u : \ltimes\supset\approx \to U,\ (y_1, y_2) := b_{12}(u),\ b_1(u, y_2) = y_1 \wedge b_2(u, y_1) = y_2.$$
Here, we pair sequences by $(y_1, y_2)_n := (y_{1,n}, y_{2,n})$.

Uncomposable systems

The composition of two deterministic causal systems need not be well-defined!

Example: Take U a singleton. Then $b_1 : \mathcal{B}\langle Y_2; Y_1\rangle$ and $b_2 : \mathcal{B}\langle Y_1; Y_2\rangle$.
Take output functions $h_1 : X_1 \times Y_2 \to Y_1$ and $h_2 : X_2 \times Y_1 \to Y_2$ constant in X.
Suppose $Y_1 = Y_2 = \mathbb{R}$, $h_1(x_1, y_2) = y_2$ and $h_2(x_2, y_1) = y_1 + 1$.
Then $y_1 = y_2 = y_1 + 1$, inconsistent!

We can also give examples where there are multiple valid composed outputs.
e.g. If instead $h_2(x_2, y_1) = y_1$, then any pair (y_1, y_2) with $y_1 = y_2$ is a valid output.

The problem is “loops” of inputs which have an immediate effect on outputs!

Awaited and delayed inputs

We overcome this issue using the approach of *reactive modules* [AH99].

The inputs are partitioned into *awaited* inputs U^{A} and *delayed* inputs U^{D}.
The delayed input at any time cannot affect the present output, only output strictly in the future.

For a state-space model, have $f : X \times (U^{\mathrm{A}} \times U^{\mathrm{D}}) \to X$, $h : X \times U^{\mathrm{A}} \to Y$.

For behaviours $b : \mathcal{B}\langle U^{\mathrm{A}} * A^{\mathrm{D}}; Y\rangle$, have the following causality property:

$$\forall u, u',\ y := b(u),\ y' := b(u'),\ u^{\mathrm{a}}_{\leq n} = u'^{\mathrm{a}}_{\leq n} \ \wedge\ u^{\mathrm{d}}_{<n} = u'^{\mathrm{d}}_{<n} \implies y_{\leq n} = y'_{\leq n}.$$

Main Theorem Composing causal behaviours $b_1 : \mathcal{B}\langle U^{\mathrm{A}} * (U^{\mathrm{D}} \times Y_2); Y_1\rangle$ and $b_2 : \mathcal{B}\langle (U^{\mathrm{A}} \times Y_1) * U^{\mathrm{D}}; Y_2\rangle$ results in a well-defined behaviour $b_{12} : \mathcal{B}\langle U^{\mathrm{A}} * U^{\mathrm{D}}, Y_1 \times Y_2\rangle$.

The composed behaviour is causal, and the composition is associative (and commutative in an appropriate sense).

[AH99] Alur & Henzinger, “Reactive Modules”, Formal Methods in System Design 15 (1999).

Stuttering

A behaviour of a timed-event system is a subset $B \subset (\mathbb{R}^+ \times E)^*$.

A straightforward definition of a nondeterministic input-output behaviour is a multivalued map $B_{\mathrm{i/o}} : (\mathbb{R}^+ \times I)^* \rightrightarrows (\mathbb{R}^+ \times O)^*$.

Unfortunately, if an output trace has an event at exactly the same time as an input event, there is no information on which event occured first.

This is important in applications, since often an event is “triggered” by another.

We overcome this issue by introducing a special “stutter” event σ in the output, which occurs at exactly the same times as the input events.

An input-output behaviour is then $B_{\mathrm{i/o}} : (\mathbb{R}^+ \times I)^* \rightrightarrows (\mathbb{R}^+ \times (O \sqcup \{\sigma\})^*$.

Motivation – Model Checking Hybrid Systems

Hybrid systems

A system is *hybrid* if it comprised both discrete and continuous characteristics.

The state evolves by a differential equation $\dot{x} = f_q(x, u)$ until some *guard* condition $g_e(x) \geq 0$ triggers a discrete event e, at which the state is *reset* to a new value $x' = r_e(x)$.

In [BCGSVZ] we give a modelling framework for hybrid systems which is compositional and *computable*.

The framework is so complicated that I am not sure all the subleties are correct...

– We allow *nondeterminism*, distinguish *awaited* and *delayed* variables, introduce *stutters*, don't assume *input enabling*, consider both *finite-* and *infinite-time* trajectories, and need to *combine* discrete- and continuous-time behaviour.
– Even if each issue in isolation seems to be handled reasonably, the combination may have errors.

[BCGSVZ] Bresolin, Collins, Geretti, Segala, Villa & Zivanovic Gonzalez, "A computable and compositional semantics for hybrid automata". In *Proc. Hybrid Systems: Computation and Control (HSCC)*, 2020.

Formal verification in Coq

We therefore aim to develop the framework and verify the semantics in a formal theorem-proving environment (we use Coq).

We start with discrete-time systems and timed-event systems, as these provide simpler cases in which to develop the concepts and techniques.

In particular, for timed-event systems we can study the *stutter* action, which is one of the more subtle issues.

Computable algorithms

The ultimate goal is to also have fully verified and rigorous *algorithms* for systems analysis.

We therefore pay attention to only using *constructive* operations.

In the future, we will combine the systems-theoretic framework with the INCONE library [STT21,KPT23] to obtain computability in the sense of type-two effectivity.

We are also extending a Coq library for verified polynomial function models [CNR11] to support general rigorous numerical algorithms, including interval arithmetic.

[STT21] Steinberg, Thery & Thies. “Computable analysis and notions of continuity in Coq”. *Logical Methods in Computer Science* **17**(2), 2021.
[KPT23] Konecny, Park & Thies. “Formalizing hyperspaces for extracting efficient exact real computation”. *Proc. 48th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2023.
[CNR11] Collins, Niqui & Revol. “A Validated Real Function Calculus”. *Math.Comput.Sci.* **5**, 2011.

Formalisation in Coq for Discrete-Time Systems

Master Thesis work of Sasha Sindorf

Behaviours and systems

A discrete-time input-output behaviour is defined by the type

```
Definition Behaviour {U Y : Type} : Type :=
    (nat -> U) -> (nat -> Y).
```

A system model is given by the inductive type

```
Inductive System {UA UD X Y : Type} : Type :=
    | state_space_model (f:X->(UA*UD)->X) (h:X->UA->Y) (e:X)
    end.
```

Given a system model, we can compute its behaviour:

```
Fixpoint behaviour {UA UD X Y}
    (s : @System UA UD X Y) : @Behaviour (UA*UD) Y.
```

There is no way of going back from a behaviour to a system!

Causality

We have the following definition of “mixed” causality:

```
Definition mixed_causal {UA UD Y} (b : @Behaviour UA*UD Y) :=
    forall (u u' : nat -> (UA*UD)),
        let ua := fun n => fst (u n) in
        let ua' := fun n => fst (u' n) in
        let ud := fun n => snd (u n) in
        let ud' := fun n => snd (u' n) in
    forall (n:nat),
      (forall m, m<=n -> ua m = ua' m) ->
        (forall m, m<n -> ud m = ud' m) ->
          (forall m, m<=n -> b u m = b u' m).
```

The behaviour of a concrete system is causal:

```
Theorem behaviour_mixed_causal {UA UD X Y} :
    forall (s : @System UA UD X Y),
      mixed_causal (behaviour s).
```

Composition

We specify composition by defining when a behaviour is the composition of two others:

We would like to define a general formula:

```
Definition is_composed_behaviour {U Y1 Y2}
    (b1 : @Behaviour (U*Y2) Y1)
    (b2 : @Behaviour (U*Y1) Y2)
    (b12 : @Behaviour U (Y1*Y2)) : Prop :=
  forall u,
    let y12 := b12 u in
    let y1 := fun (n => fst (b12 u n)) in
    let y2 := fun (n => snd (b12 u n)) in
      b1 (fun n => (u n, y2 n)) = y1
        /\  b2 (fun n => (u n, y1 n)) = y2.
```

However, it turns out that we need to take into account the awaited and delayed variables.

Composition

We therefore give the following specification of composition:

```
Definition is_composed_behaviour {UA UD Y1 Y2}
   (b1 : @Behaviour (UA * (UD*Y2)) Y1)}
   (b2 : @Behaviour ((UA*Y1) * UD) Y2)}
   (b12 : @Behaviour (UA*UD) (Y1*Y2)) : Prop :=
 forall u,
   let y12 := b12 u in
   let ua := fun (n => fst (u n)) in
   let ud := fun (n => snd (u n)) in
   let y1 := fun (n => fst (y12 n)) in
   let y2 := fun (n => snd (y12 n)) in
     b1 (fun n => (ua n, (ud n, y2 n))) = y1
       /\  b2 (fun n => ((ua n, y1 n), ud n)) = y2.
```

Composition is unique

When the composed systems are causal, then the composed behaviour (if it exists) is unique:

```
Theorem composed_mixed_causal_behaviour_unique {UA UD Y1 Y2} :
    forall (b1 : @Behaviour UA*(UD*Y2) Y1)
           (b2 : @Behaviour (UA*Y1)*UD Y2)
           (b12 b12' : @Behaviour UA*UD Y1*Y2),
      mixed_causal b1 ->
        mixed_causal b2 ->
          is_composed_behaviour b1 b2 b12 ->
            is_composed_behaviour b1 b2 b12' ->
              forall (u : nat->UA*UD) (n:nat),
                b12 u n = b12' u n.
```

Using functional extensionality, we can simplify the conclusion to `b12 = b12'`

The composed behaviour is itself causal:

```
Theorem behaviour_composition_mixed_causal {UA UD Y1 Y2} :
  forall b1 b2 (b12 : @Behaviour UA*UD Y1*Y2),
    mixed_causal b1 -> mixed_causal b2 ->
      is_composed_behaviour b1 b2 b12 ->
        mixed_causal b12.
```

Composition is a function

To show that a composed behaviour exists, we compute it as a function:

```
Fixpoint compose_behaviours {UA UD Y1 Y2}
    (b1 : @Behaviour (UA * (UD*Y2)) Y1)
    (b2 : @Behaviour ((UA*Y1) * UD) Y2)
    (Y1_default : Y1)
        : @Behaviour (UA*UD) (Y1*Y2)).}
```

Note that we need to know that either Y1 (or Y2) is inhabited.

The composed behaviour of causal systems satisfies the composition property:

```
Theorem mixed_causal_composed {UA UD Y1 Y2} :
  forall (b1 : @Behaviour UA*(UD*Y2) Y1)
         (b2 : @Behaviour (UA*Y1)*UD Y2)
         (y_default : Y1),
         (mixed_causal b1) -> (mixed_causal b2) ->
           is_composed_behaviour b1 b2
             (compose_behaviours b1 b2 y_default).
```

Composition of state space models

We can also define composition of two systems:

```
Definition compose_systems {UA UD X1 X2 Y1 Y2}
    (s1 : @system UA (UD*Y2) X1 Y1)
    (s2 : @system (UA*Y1) UD X2 Y2)
      : (@system UA UD (X1*X2) (Y1*Y2)).
```

The behaviour of the composition is the composition of the behaviours:

```
Theorem composed_system_behaviour {UA UD X1 X2 Y1 Y2} :
   forall (s1 : @system UA (UD*Y2) X1 Y1)
          (s2 : @system (UA*Y1) UD X2 Y2),
    is_composed_behaviour (behaviour s1) (behaviour s2)
      (behaviour (compose_systems s1 s2)).
```

Composition is associative

The composition property is associative:

```
Theorem behaviour_composition_associative {UA UD Y1 Y2 Y3} :
  forall b1 b2 b3 b123,
    mixed_causal b1 -> mixed_causal b2 -> mixed_causal b3 ->
    inhabited (UA * UD) ->
      (exists b12, is_composed_behaviour (preprocess1 b1) b2 b12
          /\ is_composed_behaviour b12 b3 b123)
  <-> (exists b23, is_composed_behaviour b2 (preprocess3 b3) b23
            /\ is_composed_behaviour b1 b23 (unpostprocess b123)).
```

and hence so is the composition function:

```
Theorem compose_behaviours_associative {UA UD Y1 Y2 Y3} :
  forall b1 b2 b3 y1def y2def, forall u n,
  mixed_causal b1 -> mixed_causal b2 -> mixed_causal b3 ->
    (compose_behaviours
      (compose_behaviours (preprocess1 b1) b2 y1def)
        b3 (y1def,y2def)) u n =
    (postprocess (compose_behaviours b1
      (compose_behaviours b2 (preprocess 3 b3) y2def) y1def)) u n.
```

Note the pre/postprocessing of inputs and outputs to reorder variables.

Summary

For the most part, these results work out as expected.

- The main difficulty in the proofs is bookkeeping the different signals involved.

The trickiest result is the construction of the composed behaviour.

- We first simplified the problem by considering the input-free case.
  The proof works by constructing a fixed-point of a feedback-self-loop.
  The requirement that `Y1` (or `Y2`) is inhabited is required to start the fixed-point iteration.
  (If neither type is inhabited, then `b1` and `b2` have no inputs, but `b12` must have an output, so the construction fails!)

These results can be proved without assuming functional extensionality.

- The causality property allows us to deduce `b u n = b u' n` whenever `forall m, u m = u' m` without requiring `u = u'`.

Formalisation in Coq for Timed-Event Systems

Bachelor Thesis work of Bas Laarakker

Traces

A (finite) *trace* with event set $E \subset U$ is an element $(t_k, e_k)_{k \leq n}$ of $(\mathbb{R}^+ \times U)^*$, and satisfies: $\forall k \leq n,\ e_k \in E$ and $\forall k < n,\ t_k \leq t_{k+1}$.

In Coq, we first define the raw data type

```
Definition raw_tr_list : Type := list (R * U).
```

and the two conditions

```
Fixpoint good_tr_list (E : BoolSet U) (w : raw_tr_list) : bool.
Inductive increasing_time : raw_tr_list -> Prop.
```

Checking all events are in E is decidable, and using `bool` avoids proof irrelevance.

We then package these together in a trace.

```
Record trace_list (E : BoolSet U) : Type := mkTrlist {
  raw_of_tracelist :> raw_tr_list;
  goodprop : good_tr_list E raw_of_tracelist = true; }.
Definition trace (E : BoolSet U) :=
  { tr_list : trace_list E | increasing_time tr_list}.
```

A behaviour is a set of traces:

```
Definition behaviour (E : BoolSet U) := trace E -> Prop.
```

Composition

Given an E_1-trace, we can project to an E_2-trace by removing events not in E_2.

```
Definition proj_trace {E1 : BoolSet U} (E2 : BoolSet U)
    (tr : trace E1) : trace E2.
```

Parallel composition for behaviours is defined using this projection:

```
Definition compose :
  behaviour E1 -> behaviour E2 -> behaviour (E1 :|: E2) :=
    fun B1 B2 tr =>
      B1 (proj_trace E1 tr) /\ B2 (proj_trace E2 tr).
```

The composition is associative:

```
Lemma compose_assoc {E1 E2 E3 : BoolSet U} :
  forall (B1:behaviour E1) (B2:behaviour E2) (B3:behaviour E3),
    ((B1||B2)||B3) = m_eq_type b_union_assoc (B1||(B2||B3)).
```

For this to make sense, we need associativity of union of event sets, given by `b_union_assoc`.

Input-output mappings

To define input-ouput mappings, we first introduce a global stutter event:

```
Variable stut : U.
```

An input-output behavour is a multivalued map $(\mathbb{R}^+ \times I)^* \rightrightarrows (\mathbb{R}^+ \times (O \sqcup \{\sigma\}))^*$.
We require the property that stutters in the output occur at the input event times.
We also require that event-sets I and O are disjoint, and neither contains σ.

```
Record IO_behaviour (I O : BoolSet U) (stut_event : U) :=
  mkIOBehaviour {
    disjoint_IO : Disjoint I O;
    IO_no_stut : In stut_event (I :|: O) = false;
    mapping : forall (I_tr : trace I),
      { O_tr : trace (Add stut_event O)
             | trace_stutters stut_event I_tr O_tr } -> Prop
  }.
```

Conversions between external and an input-output behaviours are constructible:

```
Definition from_IO_behaviour (bio : IO_behaviour I O stut)
    : behaviour (I :|: O).
Definition to_IO_behaviour (b : behaviour (I :|: O))
    : Disjoint I O -> In stut (I :|: O) = false
        -> IO_behaviour I O stut.
```

Composition of input-output mappings

Define composition of input-output behaviours by dropping the input-ouput distinction:

```
Definition compose_IO2 :=
  to_IO_behaviour stut (
    match (u_eqv_comp2) in (_ = t) return behaviour t with
    | eq_refl => (from_IO_behaviour bio1)
                   || (from_IO_behaviour bio2) end
  ) in_out_disjoint2 in_out_no_stut2.
```

A similar operation `compose_IO2` is defined for composition of three behaviours.

Associativity of composition input-output mappings

The composition is associative;

```
Lemma IO_compose_assoc {I1 O1 I2 O2 I3 O3 stut} :
  forall (bio1 : IO_behaviour I1 O1 stut) (bio2 : IO_behaviour I2 O2 s
    (bio3 : IO_behaviour I3 O3 stut),
  to_IO_behaviour stut (
    match (u_eqv_comp3) in (_ = t) return behaviour t with
    | eq_refl =>
      ((from_IO_behaviour bio1) || (from_IO_behaviour bio2)) || (from_
    end
  ) in_out_disjoint3 (in_out_no_stut3 bio1 bio2 bio3)
  |==
  to_IO_behaviour stut (
    match (u_eqv_comp3') in (_ = t) return behaviour t with
    | eq_refl =>
      (from_IO_behaviour bio1) || ((from_IO_behaviour bio2) || (from_I
    end
  ) in_out_disjoint3 (in_out_no_stut3 bio1 bio2 bio3).
```

The statement requires complicated manipulation of the Coq types.

The proof is actually simple; we just convert to the non input-output case.

Summary

Using a common set of events and taking subsets is technically challenging.

- We needed to implement a new type of sets defined by `bool` predicates.
- We need some complicated type conversions.

We tried using types as event sets, but unions and projections then become even more complicated.

Defining sets of external behaviours, and reducing nondeterministic input-output composition to this yields simple proofs.

Handling increasingness of event-times requires the proof-irrelevance.
Proving properties of set operations requires functional extensionality.

Further Research

Extensions to valuations and external behaviour

The discrete-time framework is defined in terms of spaces/types U, Y.
Similarly to a common set of events, it would be nice to use a common set of *variables*.
A *valuation* takes a subset V_X of variables to the product type $X = \prod_{v \in V_X} \mathrm{type}(v)$.
This should simplify some statements, particularly of associativity.

The discrete-time framework is purely input-output.
We should also formulate composition in terms of external behaviours.
This yields slightly simpler definitions.

Extension to nondeterministic systems

An important extension is to nondeterministic systems.

We have behaviours `(nat -> U) -> Ensemble (nat -> Y)`,
and systems { `f:X->U->Ensemble X; h:X->UA->Y; e:Ensemble X` }.

Note that the output function is not set-valued; this causes issues in composition.

There is a distinction between finite and infinite traces.

Sets of finite traces may be prefix-closed or prefix-free. it is unclear how this interacts with causality and input-acceptingness.

To compute infinite traces from finite ones, we need the axiom of dependent choice.

Extension to monadic systems

We can generalise further to systems on *monads*.

Monads include not only sets (ensembles), but also (nonempty) overt/compact/located sets and probability distributions.

We expect the use of awaited / delayed variables suffices the extend the compositionality results.

Some results may need the assumption that binding a constant function yields the constant.

> This holds for monads of nonempty sets, but not for possibly empty sets.
>
> The assumption is therefore related to input-acceptingness.

Dependent choice seems to be encoded as an inverse limit property:

If $\pi_{j,k} \circ \pi_{i,j} = \pi_{i,k}$ and $\pi_{j,k}[X_j] = X_k$ for $i > j > k \in \ltimes\eth\approx \cup \{\infty\}$,
then $\exists X_\infty,\ \pi_{\infty,i}[X_\infty] = X_i$.

Future research programme

Develop a Coq-verified compositional and computable framework for systems theory.

+ Investigate concepts and approaches for simple systems classes.
– Extend to nondeterministic and monadic systems.
– Extend to continuous-time and hybrid systems.

Formalise the basic theory of computable analysis and complexity theory in Coq.

+ Realisers and simple number, function and set types.
– Extend to piecewise, measurable and set-valued functions.
– Specific computational structures e.g. for countably-based spaces.
– Computational complexity.

Implement a Coq-verified library of rigorous numerical algorithms.

+ Rounded and interval arithmetics and polynomial function models in 1d.
– Extend to higher dimensions, alternative representations and other function spaces
– Algorithms for implicit functions, differential equations and constraint satisfaction.

Develop a Coq-verified library of model-checking algorithms.

– Start with simple algorithms for discrete-time systems.

Put everything together!