Bisimulation inside

From Legal Contracts to Legal Calculi:

the code-driven normativity

Silvia Crafa

Università di Padova

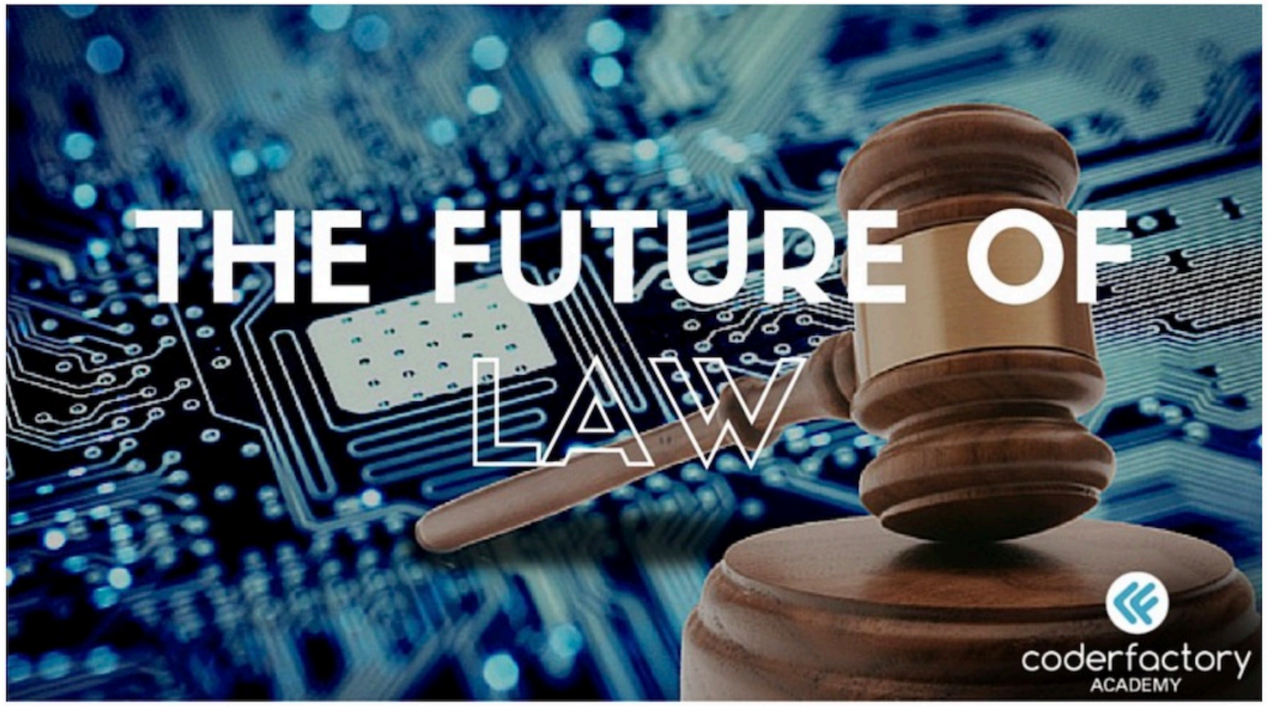
joint work with

Cosimo Laneve **Giovanni Sartor**

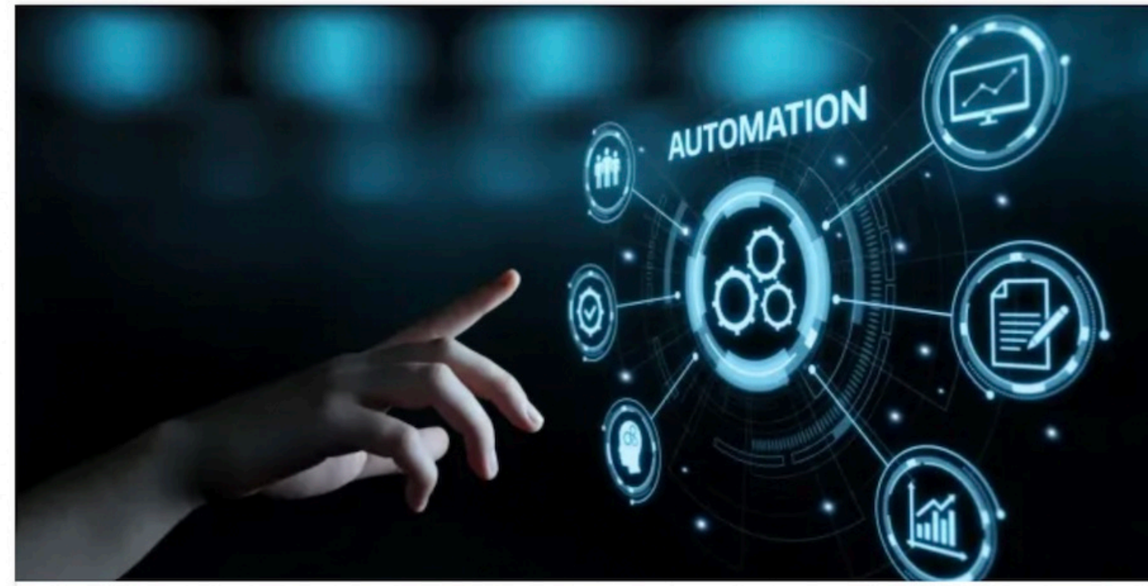
Università di Bologna

Express/SOS 2022

Future of Law: How Coding Will Change the Legal World



LTT LAW TECHNOLOGY TODAY



AUTOMATING LEGAL SERVICES

Law Technology Today | January 7, 2020 | Books | 0 Comments

ANALYSIS

New Tools for Old Rules: How Technology Is Transforming the Lawyer's Tool Kit

Until the time that lawyers and law firms begin to treat information security and data privacy awareness and diligence as key components of their practice management—on an even footing with other critical issues such as conflicts of interest, confidentiality and privilege—our collective blind spot will continue to be a target for rogue actors.

February 07, 2019 at 02:30 PM

Trust

for centuries, people and companies relied on the **principle of trust** between parties (or an authoritative guarantor)

this modality has been so fundamental that there is ***a business of intermediary roles*** (and the Institutions that guarantee justice)

with the blockchain, data and transactions are stored **with no need of intermediaries**

integrity and consistency of data is guaranteed by ***algorithms*** and *economical incentives*

Code is law

SOLIDITY

```
contract NameReg {

  mapping (address => string32) toName;
  mapping (string32 => address) toAddress;

  function register(string32 name) {
    // Don't allow the same name to be overwritten.
    if (toAddress[name] != address(0))
      return;
    // Unregister previous name if there was one.
    if (toName[msg.sender] != "")
      toAddress[toName[msg.sender]] = 0;

    toName[msg.sender] = name;
    toAddress[name] = msg.sender;
    AddressRegistered(msg.sender);
  }

  function unregister() {
    string32 n = toName[msg.sender];
    if (n == "")
      return;
    AddressDeregistered(toAddress[n]);
    toName[msg.sender] = "";
    toAddress[n] = address(0);
  }

  function addressOf(string32 name) constant returns (address addr) {
    return toAddress[name];
  }

  function nameOf(address addr) constant returns (string32 name) {
    return toName[addr];
  }

}
```

- trust is hardwired into intermediary *transparent* algorithms
- software code provides *unambiguous* definition and *automatic execution* of transactions between (mutually untrusted) parties
- when **in disputes**, **the code** of the contract, which is always publicly available, **shall prevail**.

Code Driven Law

use software to represent and enact regulation, agreements, law

- identify potential inconsistencies in regulation,
- reduce the complexity and the ambiguity of legal texts
- code-driven enforcement of rules, ex-ante

Code is law

WIRED


GABRIEL NICHOLAS BACKCHANNEL 08.11.17 07:00 AM

ETHEREUM IS CODING'S NEW WILD WEST

- **TheDAO** attack **broke the *code-is-law* dogma**: when large volumes of money are at stake, *a bug is a bug*, not a feature of the signed contract
- blockchain **does not hardwire trust** into algorithms, but rather **reassigns trust to a series of actors** (miners, programmers, companies) who implement, manage and enable the functioning of the platform

Transposing legal rules into technical rules is problematic:

- the **inherent ambiguity** of the legal system is necessary **to ensure a proper application of the law** on a case-by-case basis
  - **regulation by code** is always **more specific** and **less flexible** than the legal provisions it claims to implement. It **moves the problem into another dimension**
  - gives software developers and engineers **the power to embed their own interpretation** of the law into the technical artefacts that they create

Legal contracts

We focus on a specific subset of legal documents, the **legal contracts**:

- *“those agreements that are intended to give rise to a* ***binding legal relationship*** *or to have some other legal effect”*
    - they establish **obligations**, **rights** (such as rights to property), **powers, prohibitions** and **liabilities** between the parties,
    - often subject to specific **conditions** and by taking advantage of **escrows** and **securities**.

- Principle of **freedom of form**, shared by the contractual law of modern legal systems,
    - "*the parties of a legal contract are free to express their agreement using the language and medium they prefer*", including a programming language

Why expressing legal contracts

using a programming language?

Digital Legal contracts (beyond blockchain!)

code provides *unambiguous* and *transparent* definition and *automatic execution* of transactions and enforcement of contractual conditions

Code Driven Law

Yes, But...

- reading the code makes it **understandable**? what is the behaviour and the computational effects of the code execution?

  ➡ the p.l. should be *high-level*, *concise*, *domain specific*, *with a precise semantics*

- legal contracts have an **intrinsic open nature** (off-line/non digital elements):
  - may depend on **external data**,
    - *e.g. a bet on a football match, insurance against a flight delay*
  - may depend on **conditions that can be hardly digitized**,
    - *e.g.* ***diligent*** *storage and* ***care*** *in a rental, using a good only* ***as intended, good faith****, force majeure*

Digital Legal contracts (beyond blockchain!)

code provides *unambiguous* and *transparent* definition and *automatic execution* of transactions and enforcement of contractual conditions

Code Driven Law

Yes, But…

- the **law may deny validity to certain clauses** *(e.g. excessive interests rate)* and/or may **establish additional effects** that were not stated by the parties *(e.g. consumer's power to withdraw from an online sale, warranties, etc.)*

- the contract's institutional effects are **guaranteed by the possibility of**
  - activating ***judicial enforcements:*** each party may start a lawsuit if she believes that the other party has failed to comply with the contract,
  - dynamically **interrupt** or **modify the terms** of the contract in case of *e.g. force majeure, mutual dissent, unilateral modification*

→ **fully automatic execution** and no intermediation is **defective** so **Which** programming language?

Legal Calculi

Legal Calculi

the ***building blocks of legal contracts***
directly map
to primitives and design patterns

A **core language**, that aims at **modelling particular aspects** of its target domain,

- pivoted on few selected, concise and *intelligible* primitives, together with a **precise formalisation** of its **syntax** and **semantics**.
- its **theory** provides *static analysis* and *verification* **tools**
- its design and definition is *implementation agnostic*, but it may be compiled to full-fledged programming languages and platforms

Stipula

a legal contract as an ***interaction protocol***,
that **dynamically** regulates
permissions, prohibitions, obligations, asset exchanges
between **concurrent** parties

concurrency theory

- **Catala**: a language for modelling statutes and regulations clauses,
- **Orlando**: a language for modelling conveyances in property law,
- **Silica**: a language for smart contracts

Bike rental contract

The bike sharing service rests on a legal contract

Bike rental contract

3 days

15

1. Term.
This Agreement shall comme e day the Borrower takes possessio and remain in full force a d effect until Bike is returned to Lender. Borrow er shall return the Bike _______ after the rental date and will pay Euro _________ where half of the amount is of surcharge for late return or loss or damage of the Bike.

2. Payment.
Borrower rents the Bike on __________ and pays Euro _______ in advance. If the rented Bike is damaged or broken, Borrower reserves the right to take any action necessary to get reimbursed.

3.Return of the Bike.
Renter shall return the Bike on the date specified in Article 1 in the agreed return location. If Bike is not returned on said date or the Bike is damaged or loss, Lender reserves the right to take any action necessary to get reimbursed

4. Termination.
This Agreement shall terminate on the date specified in Section 1.

5. Disputes
Every dispute arising from the relationships governed by the above general rental conditions will be managed by the court the Lender company is based, which will decide compensations for Lender and Borrower.

Bike rental contract

similar to a class with fields, a constructor, methods

```
stipula Bike_Rental {
  assets wallet
  fields cost , rentingTime , code

  agreement (Lender,Borrower,Authority){
      Lender , Borrower: rentingTime , cost
  } => @Inactive

  @Inactive Lender : offer(x) { x --> code } => @Payment

  @Payment Borrower : pay[h] (h == cost) {
      h --o wallet
      code --> Borrower
      now + rentingTime >> @Using {"End_Reached" --> Borrower} => @Return
  } => @Using

  @Using Borrower : end { now --> Lender } => @Return

  @Return Lender : rentalOk {
      0.5*wallet --o wallet,Lender
      wallet --o Borrower
  } => @End

  @Using  Lender ,Borrower : dispute(x) { x --> _ } => @Dispute
  @Return Lender ,Borrower : dispute(x) { x --> _ } => @Dispute

  @Dispute Authority : verdict(x,y) (y>=0 && y<=1) { }
      x --> Lender , Borrower
      y*wallet --o wallet , Lender
      wallet --o Borrower
  } => @End
}
```

```
stipula Bike_Rental {

    assets wallet
    fields cost , rentingTime , code

    agreement (Lender,Borrower,Authority){
        Lender , Borrower: rentingTime , cost
    } => @Inactive

    @Inactive Lender : offer(x) { x --> code } => @Payment

    @Payment Borrower
        h --o wallet
        code --> Borr
        now + renting
    } => @Using

    @Using Borrower :

    @Return Lender :
        0.5*wallet --
        wallet --o Bo
    } => @End

    @Using  Lender ,Borrower : dispute(x) { x --> _ } => @Dispute
    @Return Lender ,Borrower : dispute(x) { x --> _ } => @Dispute

    @Dispute Authority : verdict(x,y) (y>=0 && y<=1) { }
        x --> Lender , Borrower
        y*wallet --o wallet , Lender
        wallet --o Borrower
    } => @End
}
```

meeting of the minds

3 parties **express their consent** by

- joining in a *multiparty synchronization* that
- sets the terms of the contract: the initial values of **`rentingTime`** and **`cost`**

and the contract **produces its legal effects** by entering the **initial state `@Inactive`**

```
stipula Bike_Rental {

   assets wallet
   fields cost , rentingTime , code

   agreement (Lender,Borrower,Authority){
      Lender , Borrower: rentingTime , cost
   } => @Inactive

   @Inactive Lender : offer(x) { x --> code } =>

   @Payment  Borrower : pay[h] (h == cost) {
      h --o wallet
      code --> Borrower
      now + rentingTime >> @Using {"End_Reached
   } => @Using

   @Using  Borrower : end { now --> Lender } => @Return

   @Return Lender : rentalOk {
      0.5*wallet --o wallet,Lender
      wallet --o Borrower
   } => @End

   @Using  Lender ,Borrower : dispute(x)  { x --
   @Return Lender ,Borrower : dispute(x)  { x --

   @Dispute Authority : verdict(x,y) (y>=0 && y
      x --> Lender , Borrower
      y*wallet --o wallet , Lender
      wallet --o Borrower
   } => @End
}
```

only this sequence is permitted !

1. the Lender sends the bike's usage code to the contract
2. the Borrower pays and receives the bike's code (`cost` **is the double of the fee**, as a safeguard from damages and late returns)
3. the Borrower returns the bike
4. if the bike is not damaged
   - the contract sends the payment (*i.e., half of the content of* `wallet`) to the Lender
   - gives back to the Borrower the escrow
5. the contract terminates

```
stipula Bike_Rental {
   assets wallet
   fields cost , rentingTime , 

   agreement (Lender,Borrower,A
       Lender , Borrower: renti
   } => @Inactive

   @Inactive Lender : offer(x) { x --> code } => @Payment

   @Payment Borrower : pay[h] (h == cost) {
       h --o wallet
       code --> Borrower
       now + rentingTime >> @Using {"End_Reached" --> Borrower} => @Return
   } => @Using

   @Using Borrower : end { now --> Lender } => @Return

   @Return Lender : rentalOk {
       0.5*wallet --o wallet,Lender
       wallet --o Borrower
   } => @End

   @Using  Lender ,Borrower : dispute(x) { x --> _ } => @Dispute
   @Return Lender ,Borrower : dispute(x) { x --> _ } => @Dispute

   @Dispute Authority : verdict(x,y) (y>=0 && y<=1) { }
       x --> Lender , Borrower
       y*wallet --o wallet , Lender
       wallet --o Borrower
   } => @End
}
```

state-based programming style

- widely used to specify interaction protocols
- encodes ***permissions*** and ***prohibitions***

```
stipula Bike_Rental {
    assets wallet
    fields cost , rent

    agreement (Lender,
        Lender , Borro
    } => @Inactive

    @Inactive Lender : offer(x) { x --> code } => @Payment

    @Payment Borrower : pay[h] (h == cost) {
        h --o wallet
        code --> Borrower
        now + rentingTime >> @Using {"End_Reached" --> Borrower} => @Return
    } => @Using

    @Using Borrower : end { now --> Lender } => @Return

    @Return
        0.5*
        wall
    } => @En

    @Using
    @Return Lender ,Borrower : dispute(x) { x --> _ } => @Dispute

    @Dispute Authority : verdict(x,y) (y>=0 && y<=1) { }
        x --> Lender , Borrower
        y*wallet --o wallet , Lender
        wallet --o Borrower
    } => @End
}
```

events encode ***obligations***

- by scheduling a future statement that automatically executes a corresponding action or penalty

This command **issues an event**

- that is executed at the end of the renting time, if the bike is still in use (state **`@Using`**)
- a warning message is sent to the Borrower

```
stipula Bike_Rental {
  assets wallet
  fields code , cost, re

  agreement (Lender,Borro
      Lender , Borrower:
  } => @Inactive
```

Asset-aware programming

values:	*linear assets:*
`1234 --> code`	`10€ --o wallet`
`code --> Borrower`	`wallet --o Lender`

```
  @Inactive Lender : offer(x) { x --> code } => @Payment

  @Payment Borrower : pay[h] (h == cost) {
      h --o wallet
      code --> Borrower
      now + rentingTime >> @Using {"End_Reached"
  } => @Using

  @Using Borrower : end { now --> Lender } => @Re

  @Return Lender : rentalOk {
      0.5*wallet --o wallet,Lender
      wallet --o Borrower
  } => @End

  @Using  Lender ,Borrower : dispute(x) { x --> _
  @Return Lender ,Borrower : dispute(x) { x --> _

  @Dispute Authority : verdict(x,y) (y>=0 && y<=1) { }
      x --> Lender , Borrower
      y*wallet --o wallet , Lender
      wallet --o Borrower
  } => @End
}
```

- assets are **linear resources** like *(crypto-) **currency***, or ***tokens*** *(a smart lock, a NFT)*
- useful for **payments, escrows** and **securities**
- asset cannot be *forged*, nor *double spent,* nor be *locked* into the contract

```
stipula Bike_Rental {

   assets wallet
   fields cost , rentingTime , code

   agreement (Lender,Borrower,Authority){
       Lender , Borrower: rentingTime , cost
   } => @Inactive

   @Inactive Lender : offer(x) { x --> code } => @Payment

   @Payment Borrower : pay[h]  (h == cost) {
       h --o wallet
       code --> Borrower
       now + rentingTime >> @Usi
   } => @Using

   @Using Borrower : end { now --

   @Return Lender : rentalOk {
       0.5*wallet --o wallet,Lenc
       wallet --o Borrower
   } => @End

   @Using  Lender ,Borrower : dispute(x) { x --> _ } => @Dispute
   @Return Lender ,Borrower : dispute(x) { x --> _ } => @Dispute

   @Dispute Authority : verdict(x,y) (y>=0 && y<=1) { }
       x --> Lender   x --> Borrower
       y*wallet --o wallet , Lender   // a fraction of wallet goes to Lender
       wallet --o Borrower            // the rest goes t
   } => @End
}
```

judicial enforcement pattern

- the agreement specifies the **Authority** that manages the **litigations**

- Anyone can invoke the authority at any time by moving to the state **@Dispute**
- the Authority communicate the decision by sending a string **x** and splitting the escrow money between the litigants according to the fraction **y**

```
stipula Bike_Rental {

   assets wallet
   fields cost , rentingTime , code

   agreement (Lender,Borrower,Authority){
       Lender , Borrower: rentingTime , cost
   } => @Inactive

   @Inactive Lender : offer(x) { x --> code } => @Payment
```

judicial enforcement pattern

- the agreement specifies the **Authority** that manages the **litigations**

a controlled amount of intermediation:

fully automatic execution is defective

can invoke the authority at any time by
to the state **@Dispute**

hority communicate the decision by
a string **x** and splitting the escrow
money between the litigants according to the fraction **y**

```
   @Return Lender : rentalOk {
       0.5*wallet --o wallet,Lend
       wallet --o Borrower
   } => @End

   @Using  Lender ,Borrower : dispute(x) { x --> _ } => @Dispute
   @Return Lender ,Borrower : dispute(x) { x --> _ } => @Dispute

   @Dispute Authority : verdict(x,y) (y>=0 && y<=1) { }
       x --> Lender , Borrower
       y*wallet --o wallet , Lender    // a fraction of wallet goes to Lender
       wallet --o Borrower             // the rest goes t
   } => @End
}
```

```
stipula Bet {
   assets wallet1, wallet2
   fields alea, val1, val2, data_source, fee, amount, t_before, t_after

   agreement(Better1,Better2,DataProvider){
     DataProvider , Better1 , Better2 : fee, data_source, alea, t_after
     Better1 , Better2 : amount , t_before
   } ⇒ @Init

   @Init Better1 : place_bet(x)[h]  (h == amount){
     h --o wallet1
     x --> val1
     t_before >> @First { wallet1 --o Better1 } ⇒ @Fail
   } ⇒ @First

   @First Better2 : place_bet(x)[h]  (h == amount){
     h --o wallet2
     x --> val2
     t_before >> @Run { wallet1 --o Better1  wallet2 --o Better2 } ⇒ @Fail
   } ⇒ @Run

   @Run DataProvider : data(x,y,z)[] (x == data_source && y==alea){
     if (z==val1 && z != val2){  // Better1 wins
        fee --o wallet2 ,DataProvider
        wallet2 --o Better1
        wallet1 --o Better1 }
     else
        ...
   }⇒ @End
```

```
stipula Bet {
   assets wallet1, wallet2
   fields alea, val1, val2, data_source, fee, amount, t_before, t_after

   agreement(Better1,Better2,DataProvider){
     DataProvider , Better1 , Better2 : fee, data_source, alea, t_after
     Better1 , Better2 : amount , t_before
   } ⇒ @Init

   @Init Better1 : place_bet(x)[h
     h --o wallet1
     x --> val1
     t_before >> @First { wallet1
   } ⇒ @First

   @First Better2 : place_bet(x)[
     h --o wallet2
     x --> val2
     t_before >> @Run { wallet1 --o Better1  wallet2 --o Better2 } ⇒ @Fail
   } ⇒ @Run

   @Run DataProvider : data(x,y,z)[] (x == data_source && y==alea){
     if (z==val1 && z != val2){  // Better1 wins
        fee --o wallet2 ,DataProvider
        wallet2 --o Better1
        wallet1 --o Better1 }
     else
        ...
   }⇒ @End
```

Intermediary pattern

- who takes the role of **`DataProvider`** **takes the legal responsibility** of providing the correct data form the expected **`data_source`**
- an **`Authority`** can be added to deal with litigations

Stipula

tested over a set of **archetypal legal contracts**
(*free rent*, *license to access a digital service*,
bet contract on an aleatory event, *remote purchase*)

- common **legal patterns** correspond to Stipula **design pattern**

meeting of the minds	agreement primitive
permissions, prohibitions	state-based programming
obligations	event primitive
transfer of currency or other assets	asset-aware (linear) programming
openness to external conditions or data	Intermediary pattern
judicial enforcement and exceptional behaviours	Authority pattern

Unleashing formal methods

✓ Clear semantics

- Stipula syntax and operational semantics are formally defined.
- the execution prevents unsafe assets operations, *e.g.* attempts to drain too much value from an asset or to forge new assets.

✓ Observational equivalence

- using a *bisimulation* technique we developed an equational theory that identifies contracts with different hidden elements but the same observable behavior

✓ Type inference

- the syntax is untyped for simplicity but we developed an algorithm for deriving types of assets, fields and functions, so to statically prevent basic programming errors.

✓ Liquidity analyser

- we developed a verification technique to statically check liquid contracts, that do not freeze any asset forever, *i.e.* that are not redeemable by any party

[AGREE]

$$\frac{\texttt{assets}\ \overline{\mathtt{h}}\ \in\ \mathtt{C} \qquad \texttt{agreement}(\overline{\mathtt{A}})\{\ \overline{\mathtt{A}_1} : \overline{\mathtt{x}_1}\ \cdots\ \overline{\mathtt{A}_n} : \overline{\mathtt{x}_n}\ \} \Rightarrow \texttt{@Q}\ \in\ \mathtt{C}}{\mathtt{C}(\_\,,\ \varnothing\,,\ \_\,,\ \_)\ ,\ \mathbb{t} \xrightarrow{(\overline{A},\ \overline{A_i : \overline{v_i}}^{\,i\in 1..n})} \mathtt{C}(\mathtt{Q}\,,\ [\overline{\mathtt{A}} \mapsto \overline{A}, \overline{\mathtt{x}_i}\ \mapsto\ \overline{v_i}^{\,i\in 1..n}, \overline{\mathtt{h}}\ \mapsto\ \overline{0}]\,,\ \_\,,\ \_)\ ,\ \mathbb{t}}$$

[FUNCTION]

$$\frac{\begin{array}{c}\texttt{@Q}\ \mathtt{A} : \mathtt{f}\,(\overline{\mathtt{y}})\,[\overline{\mathtt{k}}]\ (E)\,\{\,S\,W\,\} \Rightarrow \texttt{@Q}' \in \mathtt{C} \\ \Psi\ ,\ \mathbb{t} \not\rightarrow \\ \ell(\mathtt{A}) = A \quad \ell' = \ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}] \quad [\![E]\!]_{\ell'} = \mathbf{true}\end{array}}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\,,\ \Psi)\ ,\ \mathbb{t} \xrightarrow{A:\mathtt{f}(\overline{u})[\overline{v}]} \mathtt{C}(\mathtt{Q}\,,\ \ell'\,,\ S\,W \Rightarrow \texttt{@Q}'\,,\ \Psi)\ ,\ \mathbb{t}}$$

[STATE-CHANGE]

$$\frac{[\![W\{^{\mathbb{t}}/_{\mathtt{now}}\}]\!]_{\ell} = \Psi'}{\mathtt{C}(\mathtt{Q}\,,\ \ell,\ \_\,W \Rightarrow \texttt{@Q}'\,,\ \Psi)\ ,\ \mathbb{t} \longrightarrow \mathtt{C}(\mathtt{Q}'\,,\ \ell,\ \_\,,\ \Psi' \mid \Psi)\ ,\ \mathbb{t}}$$

[EVENT-MATCH]

$$\frac{\Psi = \mathbb{t} \gg \texttt{@Q}\ \{\ S\ \} \Rightarrow \texttt{@Q}' \mid \Psi'}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\,,\ \Psi)\ ,\ \mathbb{t} \longrightarrow \mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ S \Rightarrow \texttt{@Q}'\,,\ \Psi')\ ,\ \mathbb{t}}$$

[TICK]

$$\frac{\Psi\ ,\ \mathbb{t} \not\rightarrow}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\,,\ \Psi)\ ,\ \mathbb{t} \longrightarrow \mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\,,\ \Psi)\ ,\ \mathbb{t}+1}$$

[VALUE-SEND]

$$\frac{[\![E]\!]_{\ell} = v \quad \ell(\mathtt{A}) = A}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ E \rightarrow \mathtt{A}\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t} \xrightarrow{v \rightarrow A} \mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t}}$$

[ASSET-SEND]

$$\frac{[\![E]\!]^{\mathsf{a}}_{\ell} = a \quad \ell(\mathtt{A}) = A \quad [\![\mathbf{h} - a]\!]^{\mathsf{a}}_{\ell} = a'}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ E \multimap \mathbf{h}, \mathtt{A}\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t} \xrightarrow{a \multimap A} \mathtt{C}(\mathtt{Q}\,,\ \ell[\mathbf{h} \mapsto a']\,,\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t}}$$

[FIELD-UPDATE]

$$\frac{[\![E]\!]_{\ell} = v}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ E \rightarrow \mathtt{x}\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t} \longrightarrow \mathtt{C}(\mathtt{Q}\,,\ \ell[\mathtt{x} \mapsto v]\,,\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t}}$$

[ASSET-UPDATE]

$$\frac{\begin{array}{c}[\![E]\!]^{\mathsf{a}}_{\ell} = a \quad [\![\mathbf{h} - a]\!]^{\mathsf{a}}_{\ell} = a' \quad [\![\mathbf{h}' + a]\!]^{\mathsf{a}}_{\ell} = a'' \\ \ell' = \ell[\mathbf{h} \mapsto a', \mathbf{h}' \mapsto a'']\end{array}}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ E \multimap \mathbf{h}, \mathbf{h}'\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t} \longrightarrow \mathtt{C}(\mathtt{Q}\,,\ \ell'\,,\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t}}$$

[COND-TRUE]

$$\frac{[\![E]\!]_{\ell} = \mathbf{true}}{\begin{array}{l}\mathtt{C}(\mathtt{Q}\,,\ \ell,\ (\mathbf{if}\ (E)\,\{\,S\,\}\,\mathbf{else}\,\{\,S'\,\}\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t} \\ \qquad\qquad \longrightarrow \mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ S\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t}\end{array}}$$

[COND-FALSE]

$$\frac{[\![E]\!]_{\ell} = \mathbf{false}}{\begin{array}{l}\mathtt{C}(\mathtt{Q}\,,\ \ell,\ \mathbf{if}\ (E)\,\{\,S\,\}\,\mathbf{else}\,\{\,S'\,\}\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t} \\ \qquad\qquad \longrightarrow \mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ S'\ \Sigma\,,\ \Psi)\ ,\ \mathbb{t}\end{array}}$$

Labelled Transition System

$$\mathbb{C}, \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$$

the **runtime** status of the **contract** (its current state and the pending events)

the **system's** global **clock**

the execution requires the **interaction with the external context**, highlighting the *open nature* of contract's behaviour

$$\mu \quad ::= \quad \tau \quad | \quad (\overline{A}, \overline{A_i} : \overline{v_i}^{i \in 1,..,n}) \quad | \quad A : \mathtt{f}(\overline{u})[\overline{v}] \quad | \quad v \rightarrow A \quad | \quad a \multimap A$$

observe the agreement:

- who is taking the legal responsibility for which contract's role,
- what are the terms of the contract, *i.e.*, the agreed initial values of the contract's fields.

observe the possibility (**at time t**) for the party *A* to call the function *f*

observe that (**at time t**) the party *A* can **receive** a *value*, resp. an *asset*

- **prohibitions** are observed through **impossibilities** to do an action
- **time progress is not observed**, but
- we can **shift forward the observation time**, observing the effects of **obligations**

Normative Equivalence

When two syntactically different contracts are **legally equivalent**?

When they **express the same legal binding**:
the parties using them cannot distinguish one from the other.

↕

the two contracts involve the **same parties**
observing the same interactions during the contracts' lifetime.

↕

***bisimulation-based* observation equivalence**

Normative Equivalence

captures the observation of prohibitions

abstracts away the ordering of the observations within the same time clock

Definition 1 (Normative Equivalence). *A symmetric relation $\mathcal{R}$ is a bisimulation between two configurations at time $\mathbb{t}$, written $\mathbb{C}_1, \mathbb{t}\ \mathcal{R}\ \mathbb{C}_2, \mathbb{t}$, whenever*

1. *if* $\mathbb{C}_1, \mathbb{t} \overset{\alpha}{\Longrightarrow} \mathbb{C}'_1, \mathbb{t}$ *then* $\mathbb{C}_2, \mathbb{t} \overset{\alpha'}{\Longrightarrow} \mathbb{C}'_2, \mathbb{t}$ *for some* α' *such that* $\alpha \sim \alpha'$ *and* $\mathbb{C}'_1, \mathbb{t}\ \mathcal{R}\ \mathbb{C}'_2, \mathbb{t}$;
2. *if* $\mathbb{C}_1, \mathbb{t} \overset{\mu_1}{\Longrightarrow} \cdots \overset{\mu_n}{\Longrightarrow} \mathbb{C}'_1, \mathbb{t} \longrightarrow \mathbb{C}'_1, \mathbb{t}+1$ *then there exist* $\mu'_1 \cdots \mu'_n$ *that is a permutation of* $\mu_1 \cdots \mu_n$ *such that* $\mathbb{C}_2, \mathbb{t} \overset{\mu'_1}{\Longrightarrow} \cdots \overset{\mu'_n}{\Longrightarrow} \mathbb{C}'_2, \mathbb{t} \longrightarrow \mathbb{C}'_2, \mathbb{t}+1$ *and* $\mathbb{C}'_1, \mathbb{t}+1\ \mathcal{R}\ \mathbb{C}'_2, \mathbb{t}+1$.

Let $\simeq$ *be the largest bisimulation, called* normative equivalence. *When the initial configurations of contracts* $\mathtt{C}$ *and* $\mathtt{C}'$ *are bisimilar, we simply write* $\mathtt{C} \simeq \mathtt{C}'$.

a transfer property that shifts the time of observation to the next time unit

- **abstracts away** the **ordering of messages** within the same time unit, and the contract's **internal names**
- does not overlook essential **precedence constraints**, which are important in legal contracts, e.g. a function delivering a service *can only be invoked after* a payment.
- allows to **garbage-collect events that cannot be triggered** anymore because the time for their scheduling is already elapsed.

Conclusions

the assimilation of *software-based* contracts to *legally binding* contracts
raises **both legal** and **technological** issues.

Interdisciplinary Assessment

- **usability** of legal programming languages,
- unveil partial or **erroneous interpretations of the law** embedded in technical artefacts,
- understand the **actual extent of the legal protection** provided by the *software normativity.*

Legal Calculi

may sheds some light on the digitalisation of legal texts

Effective **Implementation**

- the primitives can be implemented over a *centralized* or a *distributed* system
- **legally robust** management of *identities, agreement,* time in obligations, assets

Conclusions

the assimilation of software-based contracts to legally binding contracts raises **both legal** and **technological** issues.

Interdisciplinary Assessment

Lesson we learned:

- the **intrinsic open nature** of legal contracts, that is **incompatible with the automatic execution** of software-based rules claimed by the Code-Driven Law
    - The intervention of the law is particularly significant **to protect the weaker party** *(e.g. the worker in an employment contract or the consumer in an online purchase)*
- any software solution must provide an **escape mechanism** *(e.g. the Authority pattern in Stipula)* that allows a flexible, and legally valid, link between ***what is true*** off-line and on-line.

[Agree]

$$\frac{\texttt{assets}\ \overline{\texttt{h}} \in \texttt{C} \qquad \texttt{agreement}(\overline{\texttt{A}})\{\ \overline{\texttt{A}_1} : \overline{\texttt{x}_1} \ \cdots\ \overline{\texttt{A}_n} : \overline{\texttt{x}_n}\ \} \Rightarrow @\texttt{Q} \in \texttt{C}}{\texttt{C}(\_, \varnothing, \_, \_)\ ,\ \mathbb{t} \xrightarrow{(\overline{A},\, \overline{A_i : \overline{v_i}}^{\,i\in 1..n})} \texttt{C}(\texttt{Q}, [\overline{\texttt{A}} \mapsto \overline{A}, \overline{\texttt{x}_i} \mapsto \overline{v_i}^{\,i\in 1..n}, \overline{\texttt{h}} \mapsto \overline{0}], \_, \_)\ ,\ \mathbb{t}}$$

[Function]

$$\frac{\begin{array}{c} @\texttt{Q}\ \texttt{A} : \texttt{f}\,(\overline{\texttt{y}})\,[\overline{\texttt{k}}]\ (E)\,\{\,S\,W\,\} \Rightarrow @\texttt{Q}' \in \texttt{C} \\ \Psi\ ,\ \mathbb{t} \not\rightarrow \\ \ell(\texttt{A}) = A \quad \ell' = \ell[\overline{\texttt{y}} \mapsto \overline{u}, \overline{\texttt{k}} \mapsto \overline{v}] \quad [\![E]\!]_{\ell'} = \texttt{true} \end{array}}{\texttt{C}(\texttt{Q}, \ell, \_, \Psi)\ ,\ \mathbb{t} \xrightarrow{A:\texttt{f}(\overline{u})[\overline{v}]} \texttt{C}(\texttt{Q}, \ell', S\,W \Rightarrow @\texttt{Q}', \Psi)\ ,\ \mathbb{t}}$$

[State-Change]

$$\frac{[\![W\{^{\mathbb{t}}/_{\texttt{now}}\}]\!]_\ell = \Psi'}{\texttt{C}(\texttt{Q}, \ell, \_\,W \Rightarrow @\texttt{Q}', \Psi)\ ,\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}', \ell, \_, \Psi' \mid \Psi)\ ,\ \mathbb{t}}$$

[Event-Match]

$$\frac{\Psi = \mathbb{t} \gg @\texttt{Q}\ \{\ S\ \} \Rightarrow @\texttt{Q}' \mid \Psi'}{\texttt{C}(\texttt{Q}, \ell, \_, \Psi)\ ,\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}, \ell, S \Rightarrow @\texttt{Q}', \Psi')\ ,\ \mathbb{t}}$$

[Tick]

$$\frac{\Psi\ ,\ \mathbb{t} \not\rightarrow}{\texttt{C}(\texttt{Q}, \ell, \_, \Psi)\ ,\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}, \ell, \_, \Psi)\ ,\ \mathbb{t}+1}$$

[Value-Send]

$$\frac{[\![E]\!]_\ell = v \quad \ell(\texttt{A}) = A}{\texttt{C}(\texttt{Q}, \ell, E \to \texttt{A}\ \Sigma, \Psi)\ ,\ \mathbb{t} \xrightarrow{v \to A} \texttt{C}(\texttt{Q}, \ell, \Sigma, \Psi)\ ,\ \mathbb{t}}$$

[Asset-Send]

$$\frac{[\![E]\!]^{\mathsf{a}}_\ell = a \quad \ell(\texttt{A}) = A \quad [\![\texttt{h} - a]\!]^{\mathsf{a}}_\ell = a'}{\texttt{C}(\texttt{Q}, \ell, E \multimap \texttt{h}, \texttt{A}\ \Sigma, \Psi)\ ,\ \mathbb{t} \xrightarrow{a \multimap A} \texttt{C}(\texttt{Q}, \ell[\texttt{h} \mapsto a'], \Sigma, \Psi)\ ,\ \mathbb{t}}$$

[Field-Update]

$$\frac{[\![E]\!]_\ell = v}{\texttt{C}(\texttt{Q}, \ell, E \to \texttt{x}\ \Sigma, \Psi)\ ,\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}, \ell[\texttt{x} \mapsto v], \Sigma, \Psi)\ ,\ \mathbb{t}}$$

[Asset-Update]

$$\frac{\begin{array}{c} [\![E]\!]^{\mathsf{a}}_\ell = a \quad [\![\texttt{h} - a]\!]^{\mathsf{a}}_\ell = a' \quad [\![\texttt{h}' + a]\!]^{\mathsf{a}}_\ell = a'' \\ \ell' = \ell[\texttt{h} \mapsto a', \texttt{h}' \mapsto a''] \end{array}}{\texttt{C}(\texttt{Q}, \ell, E \multimap \texttt{h}, \texttt{h}'\ \Sigma, \Psi)\ ,\ \mathbb{t} \longrightarrow \texttt{C}(\texttt{Q}, \ell', \Sigma, \Psi)\ ,\ \mathbb{t}}$$

[Cond-true]

$$\frac{[\![E]\!]_\ell = \texttt{true}}{\begin{array}{l} \texttt{C}(\texttt{Q}, \ell, (\texttt{if}\,(E)\,\{\,S\,\}\,\texttt{else}\,\{\,S'\,\}\ \Sigma, \Psi)\ ,\ \mathbb{t} \\ \qquad\qquad \longrightarrow \texttt{C}(\texttt{Q}, \ell, S\ \Sigma, \Psi)\ , \end{array}}$$

Three sources of nondeterminism:

- the order of the execution of ready events’ handlers,
- the order of the calls of permitted functions, and
- the delay of permitted function calls to a later time (thus, possibly, after other event handlers)

The behaviour of a Stipula legal contract:

- the first action is always an agreement, which moves the contract to an **idle state**;
- in an idle state, if there is **a ready event** with a matching state, then its handler is completely executed, moving again to a (possibly different) idle state;
- in an idle state, if there is no event to be triggered, **either advance the system’s clock** or **call any permitted function** (*i.e.* with matching state and preconditions). A function invocation amounts to execute its body until the end, which is again an idle state.