A Choreographic View of Smart Contracts

Emilio Tuosto @ GSSI

joint work with

Elvis Gerardin Konjoh Selabi @GSSI & UniCam
Maurizio Murgia @GSSI
Antonio Ravara @NOVA

A tutorial @ FORTE 2025, Lille

What's up doc?

Prologue . . . . . An inspiring initiative

What's up doc?
Prologue . . . . . An inspiring initiative

What's up doc?

What's up doc?

Prologue . . . . . An inspiring initiative

Act I . . . . . A coordination framework

What's up doc?

What's up doc?

Prologue . . . . . An inspiring initiative

Act I . . . . . A coordination framework

Act II . . . . . A tool

What's up doc?

What's up doc?
Prologue ..... An inspiring initiative
Act I ..... A coordination framework
Act II ..... A tool
Act III ..... A little exercise

What's up doc?

What's up doc?

Prologue . . . . . An inspiring initiative

Act I . . . . . A coordination framework

Act II . . . . . A tool

Act III . . . . . A little exercise

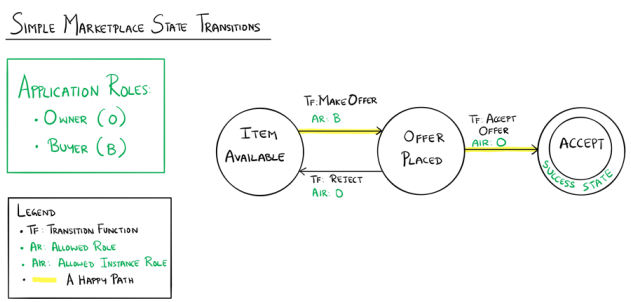
Epilogue . . . . . Work in progress

– Prologue –

[An inspiring initiative]

A nice sketch! [5, 6]

A smart contract among Owners and Buyers



initially buyers can make offers
then

either an owner can accept an offer and the protocol stops
or the offer is rejected and the protocol restarts

A nice sketch! [5, 6]
A smart contract among Owners and Buyers
initially buyers can make offers
then
either an owner can accept an offer and the protocol stops
or the offer is rejected and the protocol restarts

What did we just see?

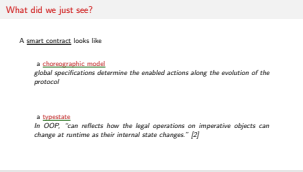
A smart contract looks like

a choreographic model

global specifications determine the enabled actions along the evolution of the protocol

a typestate

In OOP, "can reflects how the legal operations on imperative objects can change at runtime as their internal state changes." [2]



A new coordination model

So, we saw an interesting model where

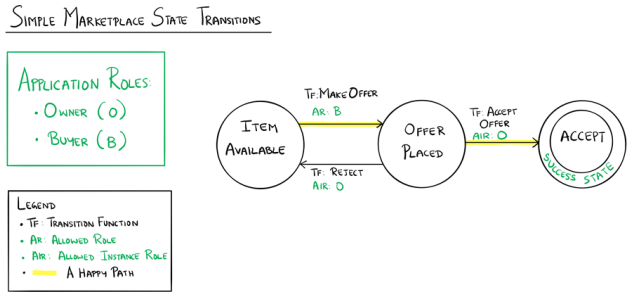
distributed components coordinate through a global specification

which specifies which actions enabled along the computation

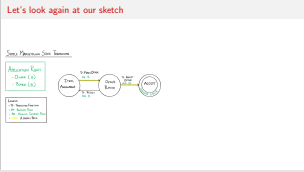
"without forcing" components to be cooperative!

A new coordination model
So, we saw an interesting model where
distributed components coordinate through a global specification
which specifies which actions enabled along the computation
"without forcing" components to be cooperative!

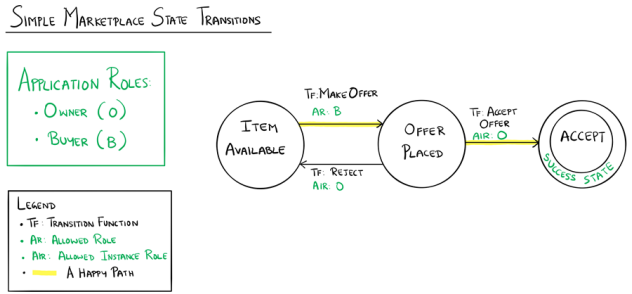
Let's look again at our sketch



Let's look again at our sketch

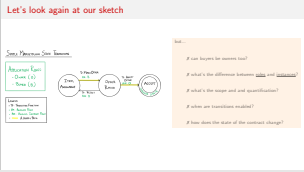


Let's look again at our sketch



but...

- ✗ can buyers be owners too?
- ✗ what's the difference between roles and instances?
- ✗ what's the scope and and quantification?
- ✗ when are transitions enabled?
- ✗ how does the state of the contract change?



ok

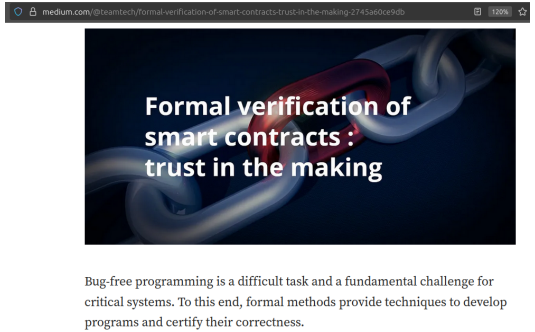
ok

from [6]: "The transitions between the Item Available and the Offer Placed states can continue until the owner is satisfied with the offer made." so, after a rejection, the new offer must be from the original buyer or a new one?

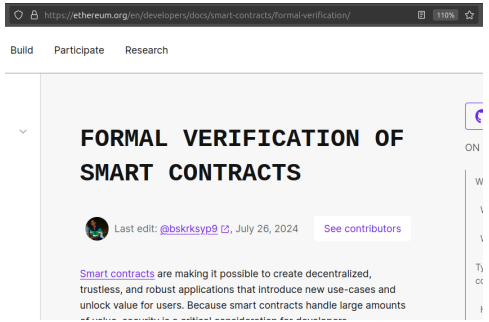
ok

should the price of the item remain unchanged when the owner invokes the Reject?

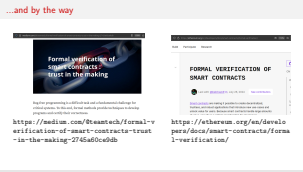
...and by the way



https://medium.com/@teamtech/formal-verification-of-smart-contracts-trust-in-the-making-2745a60ce9db



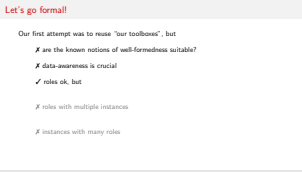
https://ethereum.org/en/developers/docs/smart-contracts/formal-verification/



Let's go formal!

Our first attempt was to reuse "our toolboxes", but

- ✗ are the known notions of well-formedness suitable?
- ✗ data-awareness is crucial
- ✓ roles ok, but
- ✗ roles with multiple instances
- ✗ instances with many roles



Let's go formal!

Our first attempt was to reuse "our toolboxes", but

- ✗ are the known notions of well-formedness suitable?
- ✗ data-awareness is crucial
- ✓ roles ok, but
- ✗ roles with multiple instances
- ✗ instances with many roles

So we had to came up with some new behavioural types.

Let's go formal!
Our first attempt was to reuse "our toolboxes", but
✗ are the known notions of well-formedness suitable?
✗ data-awareness is crucial
✓ roles ok, but
✗ roles with multiple instances
✗ instances with many roles
So we had to came up with some new behavioural types.

– Act I –

[A coordination framework]

Basic concepts and notation

Participants $\mathsf{p}, \mathsf{p}', \ldots$

Basic concepts and notation
Participants $\mathsf{p}, \mathsf{p}', \ldots$

We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitly

Basic concepts and notation

Participants $p, p', \ldots$

have roles $R, R', \ldots$

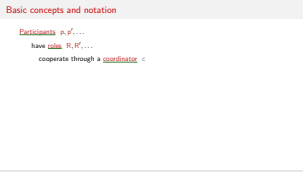
We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitly

Basic concepts and notation

Participants $p, p', \ldots$

have roles $R, R', \ldots$

cooperate through a coordinator c



We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitly

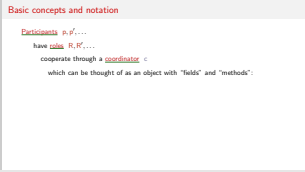
Basic concepts and notation

Participants $p, p', \ldots$

have roles $R, R', \ldots$

cooperate through a coordinator c

which can be thought of as an object with "fields" and "methods":



We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitly

Basic concepts and notation

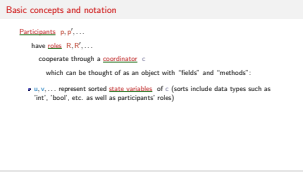
Participants $p, p', \ldots$

have roles $R, R', \ldots$

cooperate through a coordinator c

which can be thought of as an object with "fields" and "methods":

- $u, v, \ldots$ represent sorted state variables of c (sorts include data types such as 'int', 'bool', etc. as well as participants' roles)



We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitly

Basic concepts and notation

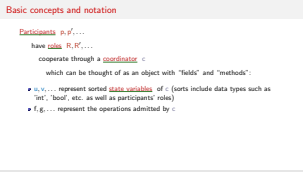
Participants $p, p', \ldots$

have roles $R, R', \ldots$

cooperate through a coordinator c

which can be thought of as an object with "fields" and "methods":

- $u, v, \ldots$ represent sorted state variables of c (sorts include data types such as 'int', 'bool', etc. as well as participants' roles)
- $f, g, \ldots$ represent the operations admitted by c



We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitly

Basic concepts and notation

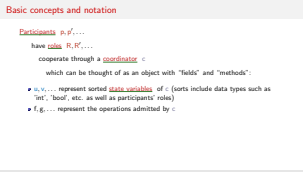
Participants $p, p', \ldots$

have roles $R, R', \ldots$

cooperate through a coordinator c

which can be thought of as an object with "fields" and "methods":

- $u, v, \ldots$ represent sorted state variables of c (sorts include data types such as 'int', 'bool', etc. as well as participants' roles)
- $f, g, \ldots$ represent the operations admitted by c



We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitly

Basic concepts and notation

Participants $p, p', \ldots$

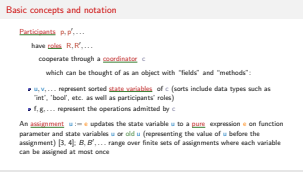
have roles $R, R', \ldots$

cooperate through a coordinator c

which can be thought of as an object with "fields" and "methods":

- $u, v, \ldots$ represent sorted state variables of c (sorts include data types such as 'int', 'bool', etc. as well as participants' roles)
- $f, g, \ldots$ represent the operations admitted by c

An assignment $u := e$ updates the state variable u to a pure expression e on function parameter and state variables u or old u (representing the value of u before the assignment) [3, 4]; $B, B', \ldots$ range over finite sets of assignments where each variable can be assigned at most once



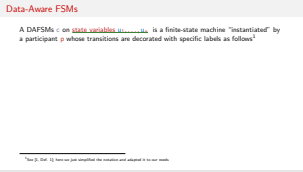
We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitlyExpressions are standard but for state variables occurring in rhs e must have the old _ qualifier; this concept will be used in the definition of (progress for) well-formedness

We adapt the mechanism based on the old keyword from the Eiffel language [4] which, as explained in [3] is necessary to render assignments into logical formulae since e.g., $x = x+1 \iff$ False.

Data-Aware FSMs

A DAFSMs c on state variables $u_1, \ldots, u_n$ is a finite-state machine "instantiated" by a participant p whose transitions are decorated with specific labels as follows[1]

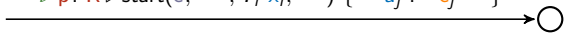
[1] See [1, Def. 1]; here we just simplified the notation and adapted it to our needs



Data-Aware FSMs

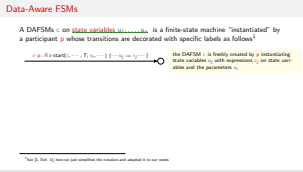
A DAFSMs c on state variables $u_1, \ldots, u_n$ is a finite-state machine "instantiated" by a participant p whose transitions are decorated with specific labels as follows[1]

$\nu\ p\colon R \triangleright \mathsf{start}(c, \cdots, T_i\ x_i, \cdots)\ \{\cdots u_j := e_j \cdots\}$



the DAFSM c is freshly created by p instantiating state variables u_j with expressions e_j on state variables and the parameters x_i

[1] See [1, Def. 1]; here we just simplified the notation and adapted it to our needs

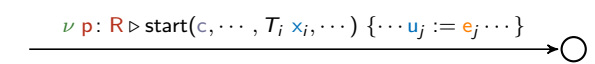


each state variable is declared and initialises with type-consistent expressions on state variables and parameters u_j

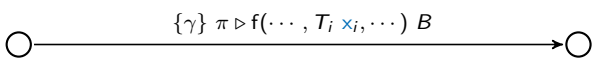
start is a "build-in" (and pleonastic) function name

Data-Aware FSMs

A DAFSMs c on state variables $u_1, \ldots, u_n$ is a finite-state machine "instantiated" by a participant p whose transitions are decorated with specific labels as follows[1]



the DAFSM c is freshly created by p instantiating state variables u_j with expressions e_j on state variables and the parameters x_i

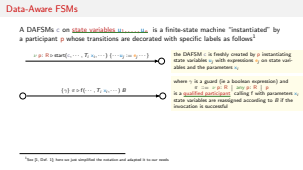


where γ is a guard (ie a boolean expression) and

$$\pi \ ::= \ \nu \ \mathsf{p}\colon \mathsf{R} \ \mid \ \text{any p}\colon \mathsf{R} \ \mid \ \mathsf{p}$$

is a qualified participant calling f with parameters x_i state variables are reassigned according to B if the invocation is successful

[1] See [1, Def. 1]; here we just simplified the notation and adapted it to our needs



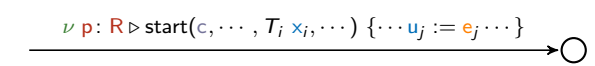
γ predicates over state variables and formal parameters of its transition; guards have to be satisfied for the invocation to succeed: an invocation that makes the guard false is rejected

ν p: R specifies that p must be a fresh participant with role R
any p: R qualifies p as an existing participant with role R
p we refer to a participant in the scope of a binder
invocations from non-suitable callers are rejected

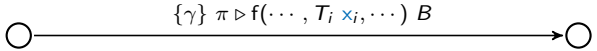
the variables occurring in the right-hand side of assignments in B are either state variables or parameters of the invocation

Data-Aware FSMs

A DAFSMs c on state variables $u_1, \ldots, u_n$ is a finite-state machine "instantiated" by a participant p whose transitions are decorated with specific labels as follows[1]



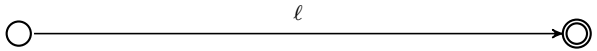
the DAFSM c is freshly created by p instantiating state variables u_j with expressions e_j on state variables and the parameters x_i



where γ is a guard (ie a boolean expression) and

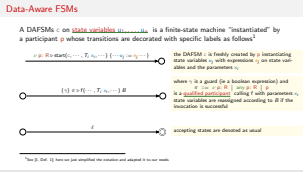
$$\pi \;::=\; \nu\ p\colon R \;\mid\; \text{any}\ p\colon R \;\mid\; p$$

is a qualified participant calling f with parameters x_i
state variables are reassigned according to B if the invocation is successful



accepting states are denoted as usual

[1] See [1, Def. 1]; here we just simplified the notation and adapted it to our needs



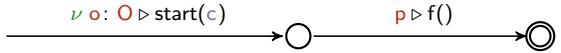
Exercise: modelling

Give a DAFSM for the protocol on slide 7 resolving the ambiguities listed there.

Exercise: modelling
Give a DAFSM for the protocol on slide 7 resolving the ambiguities listed there.

let them play with qualified participants

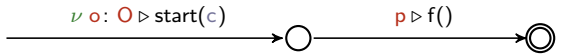
Not all DAFSMs “make sense”



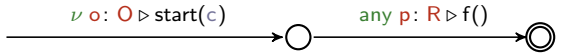
names’ freeness

Not all DAFSMs “make sense”
names’ freeness

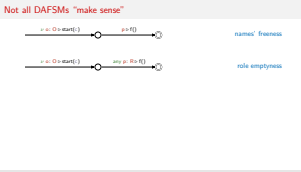
Not all DAFSMs “make sense”



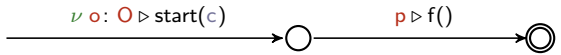
names’ freeness



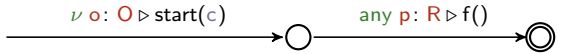
role emptyness



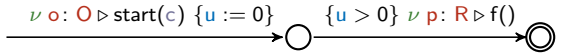
Not all DAFSMs “make sense”



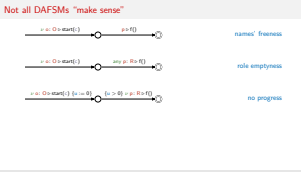
names’ freeness



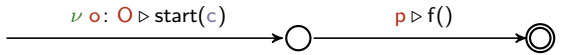
role emptyness



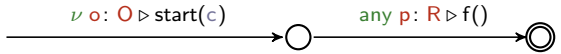
no progress



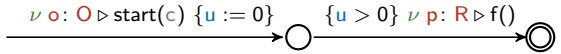
Not all DAFSMs “make sense”



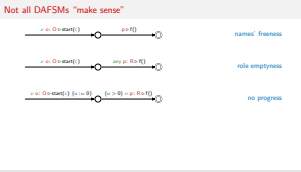
names’ freeness



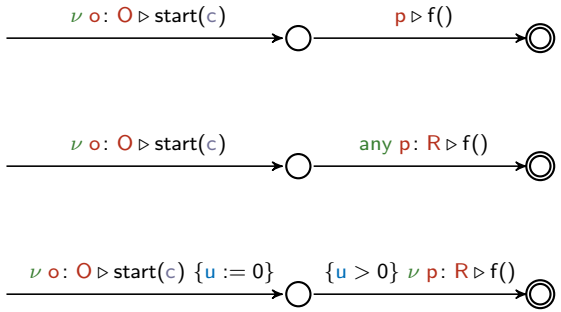
role emptyness



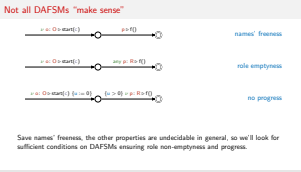
no progress



Not all DAFSMs “make sense”



Save names' freeness, the other properties are undecidable in general, so we'll look for sufficient conditions on DAFSMs ensuring role non-emptyness and progress.



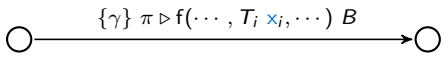
Closed DAFSMs

Binders: parameter declarations in function call, ν p: R, and any p: R

Closed DAFSMs
Binders: parameter declarations in function call, ν p: R, and any p: R

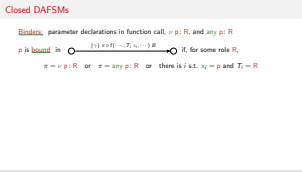
Closed DAFSMs

Binders: parameter declarations in function call, ν p: R, and any p: R

p is bound in 

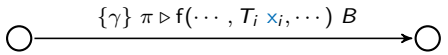
if, for some role R,

$\pi = \nu\ \mathsf{p}\colon \mathsf{R}$ or $\pi = \text{any}\ \mathsf{p}\colon \mathsf{R}$ or there is i s.t. $\mathsf{x}_i = \mathsf{p}$ and $T_i = \mathsf{R}$



Closed DAFSMs

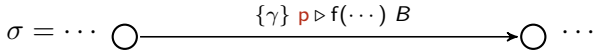
Binders: parameter declarations in function call, ν p: R, and any p: R

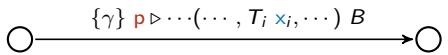
p is bound in 

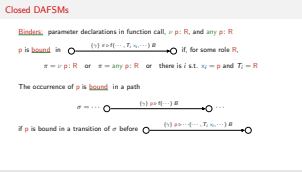
if, for some role R,

$\pi = \nu$ p: R or $\pi =$ any p: R or there is i s.t. $x_i =$ p and $T_i =$ R

The occurrence of p is bound in a path

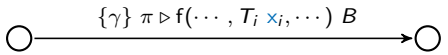


if p is bound in a transition of σ before 



Closed DAFSMs

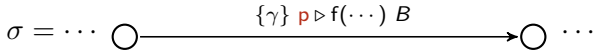
Binders: parameter declarations in function call, ν p: R, and any p: R

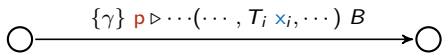
p is bound in 

if, for some role R,

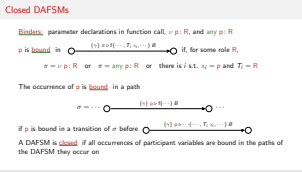
$$\pi = \nu\ \mathsf{p}\colon \mathsf{R} \quad \text{or} \quad \pi = \text{any}\ \mathsf{p}\colon \mathsf{R} \quad \text{or} \quad \text{there is } i \text{ s.t. } \mathsf{x}_i = \mathsf{p} \text{ and } T_i = \mathsf{R}$$

The occurrence of p is bound in a path



if p is bound in a transition of σ before 

A DAFSM is closed if all occurrences of participant variables are bound in the paths of the DAFSM they occur on



Roles non-emptyness

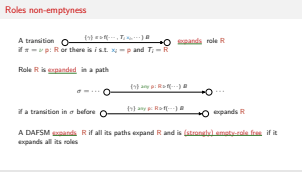
A transition $\bigcirc \xrightarrow{\{\gamma\}\ \pi \triangleright \mathsf{f}(\cdots, T_i\ \mathsf{x}_i, \cdots)\ B} \bigcirc$ expands role R
if $\pi = \nu\ \mathsf{p}\colon \mathsf{R}$ or there is i s.t. $\mathsf{x}_i = \mathsf{p}$ and $T_i = \mathsf{R}$

Role R is expanded in a path

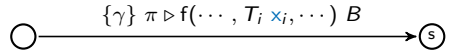
$$\sigma = \cdots \bigcirc \xrightarrow{\{\gamma\}\ \text{any p}\colon \mathsf{R} \triangleright \mathsf{f}(\cdots)\ B} \bigcirc \cdots$$

if a transition in σ before $\bigcirc \xrightarrow{\{\gamma\}\ \text{any p}\colon \mathsf{R} \triangleright \mathsf{f}(\cdots)\ B} \bigcirc$ expands R

A DAFSM expands R if all its paths expand R and is (strongly) empty-role free if it expands all its roles



Progress

A DAFSM with state variables $U = \{u_1, \ldots, u_n\}$ is consistent if it is closed and the following implication holds for each transition 

$$\forall(u, \text{old } u)_{u \in U}\ \exists(x)_{x \in X} :\ (\gamma\{\text{old } u/u\}_{u \in U}\ \wedge\ \gamma_B) \implies \gamma_s$$

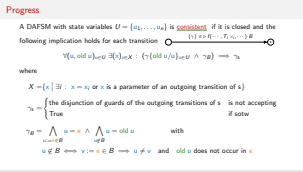
where

$$X = \{x \mid \exists i :\ x = x_i \text{ or } x \text{ is a parameter of an outgoing transition of } s\}$$

$$\gamma_s = \begin{cases} \text{the disjunction of guards of the outgoing transitions of } s & \text{is not accepting} \\ \text{True} & \text{if sotw} \end{cases}$$

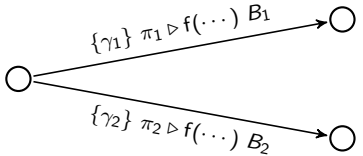
$$\gamma_B = \bigwedge_{u := e \in B} u = e\ \wedge\ \bigwedge_{u \notin B} u = \text{old } u \qquad \text{with}$$

$$u \notin B \iff v := e \in B \implies u \neq v \quad \text{and} \quad \text{old } u \text{ does not occur in } e$$



Determinism

A DAFSM is deterministic if

whenever 

then $\gamma_1 \wedge \gamma_2 \implies \pi_1 \# \pi_2$

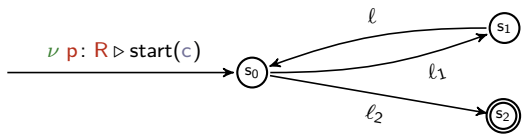
where $\_ \# \_$ is the least binary symmetric relation s.t.

$\nu\ p\colon R \# \pi$ and $\nu\ p\colon R \# \text{any } p'\colon R'$ and $R \neq R' \implies \text{any } p\colon R \# \text{any } p'\colon R'$

Determinism
A DAFSM is deterministic if
whenever
then
where $\_ \# \_$ is the least binary symmetric relation s.t.

transitions from the same source state and calling the same function

Exercise: Determinism

The DAFSM $\mathcal{S} =$ 

is deterministic or not, depending on the labels ℓ_1 and ℓ_2.

1. Is it the case that $\mathcal{S}$ is not deterministic whenever $\ell_1 = \ell_2$?
2. Find two labels ℓ_1 and ℓ_2 that make $\mathcal{S}$ deterministic
3. Find two labels $\ell_1 \neq \ell_2$ that make $\mathcal{S}$ non-deterministic

Exercise: Determinism
The DAFSM $\mathcal{S} =$
is deterministic or not, depending on the labels ℓ_1 and ℓ_2.
1. Is it the case that $\mathcal{S}$ is not deterministic whenever $\ell_1 = \ell_2$?
2. Find two labels ℓ_1 and ℓ_2 that make $\mathcal{S}$ deterministic
3. Find two labels $\ell_1 \neq \ell_2$ that make $\mathcal{S}$ non-deterministic

1. no: eg for $\ell_1 = \ell_2 = \nu$ p: R $\mathcal{S}$ is deterministic
2. $\ell_1 = \ell_2 = \ \nu$ p: R ▷ f($\cdots$, T_i x_i, $\cdots$) make $\mathcal{S}$ deterministic because the next state is unambiguously determined by the caller which is fresh on both transitions
3. $\ell_1 = \{x \leq 0\}$ p ▷ f(x : Int) and $\ell_2 = \{x \geq -1\}$ p ▷ f(x : Int) make $\mathcal{S}$ non-deterministic because the guards of ℓ_1 and of ℓ_2 are not disjoint therefore the next state is not determined by the caller

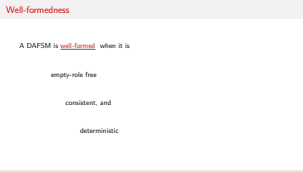
Well-formedness

A DAFSM is well-formed when it is

empty-role free

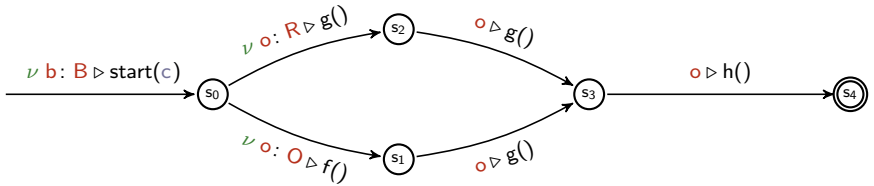
consistent, and

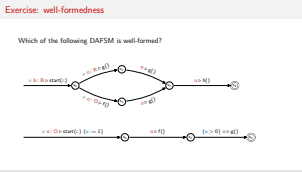
deterministic



Exercise: well-formedness

Which of the following DAFSM is well-formed?





yes: o is defined on paths it occurs on and the DAFSM is deterministic.

no: the transition from s_0 violates consistency since True does not imply $u > 0$ hinting that the protocol could get stuck in state s_1. However, this never happens because u is initially set to 1 and never changed, hence the transition from s_1 would be enabled when the protocol lands in s_1.

– Act II –

[A tool]

Verification

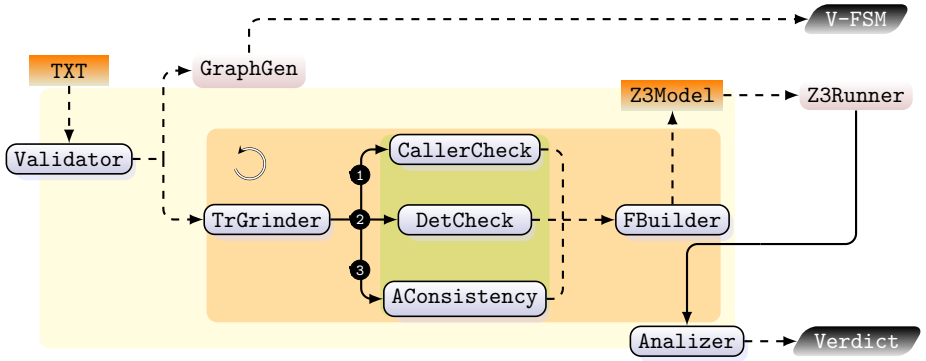
Checking well-formedness by hand is laborious and cumbersome

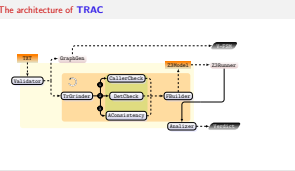
So we implemented **TRAC**, which

- ✓ transforms DAFSMs in a DSL to specify DAFSMs
- ✓ verifies well-formedness condition relying on the SMT solver Z3
- ✓ it's efficient enough
- ✗ but cannot handle roles and inter-contract interactions

Verification
Checking well-formedness by hand is laborious and cumbersome
So we implemented TRAC, which
✓ transforms DAFSMs in a DSL to specify DAFSMs
✓ verifies well-formedness condition relying on the SMT solver Z3
✓ it's efficient enough
✗ but cannot handle roles and inter-contract interactions

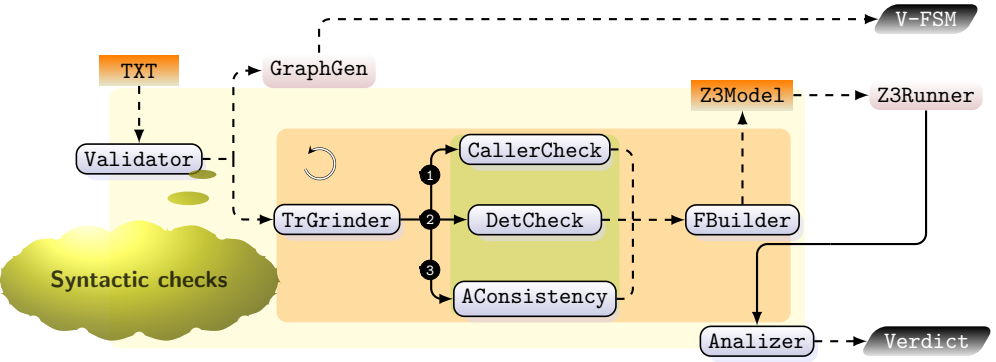
The architecture of **TRAC**

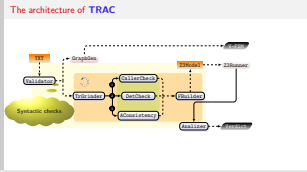




the architecture of **TRAC** is compartmentalised into two principal modules: parsing and visualisation (yellow box) and **TRAC**'s core (orange box). The latter module implements well-formedness check (green box). Solid arrows represent calls between components while dashed arrows data IO.

The architecture of **TRAC**

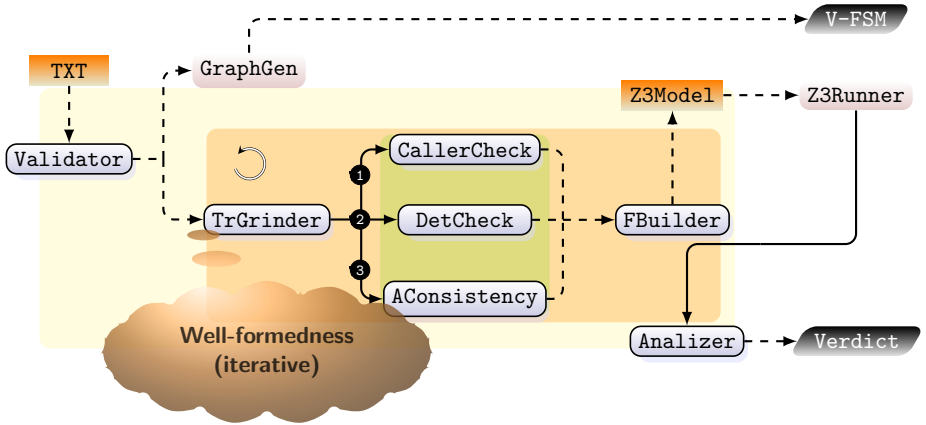


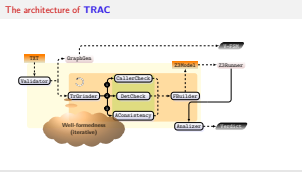


basic syntactic checks on a DSL representation of DAFSMs and transforming the input in a format that simplifies the analysis of the following phases:

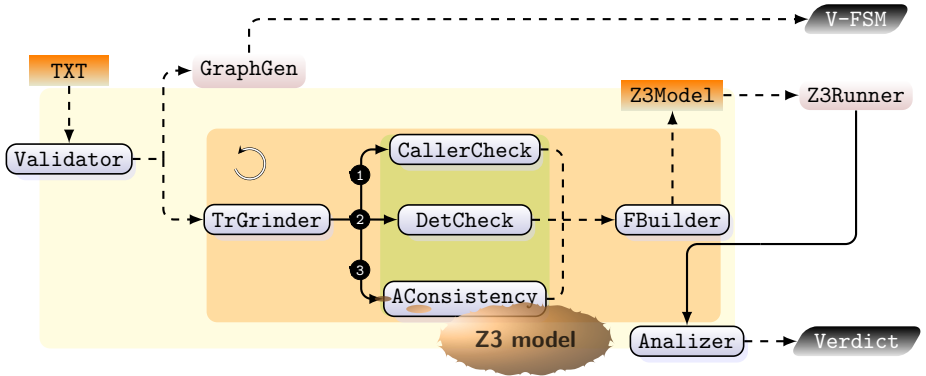
- passed to `GraphGen` for visual representation of DAFSMs (`V-FSM` output)
- passed to the `TrGrinder` component (orange box) for well-formedness checking.

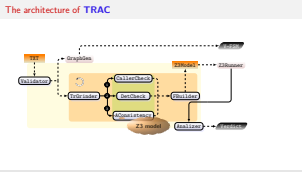
The architecture of **TRAC**





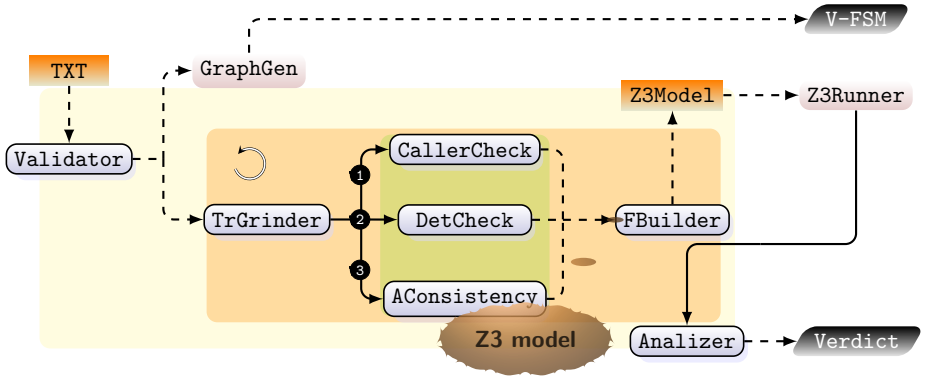
The architecture of **TRAC**

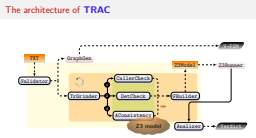




`AConsistency` (arrow ❸ to generate a Z3 formula which holds if, and only if, the transtion is consistent.

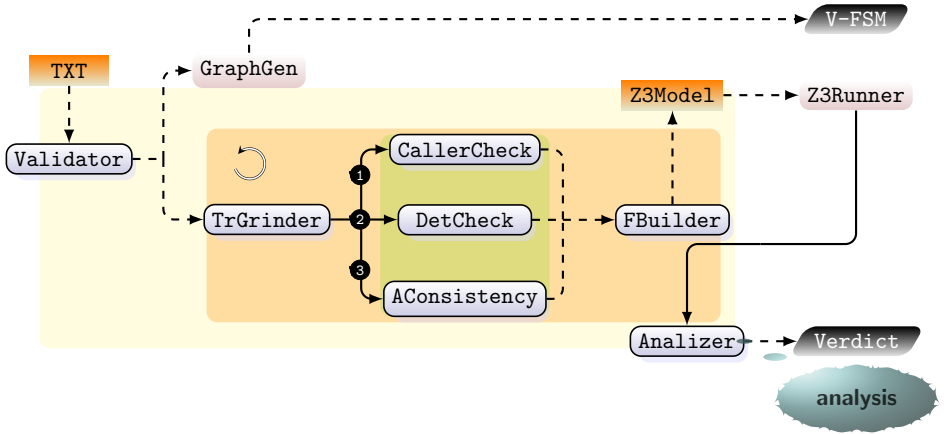
The architecture of **TRAC**

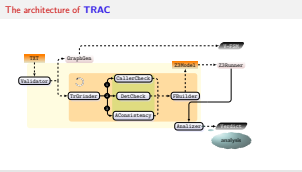




computes the z3 f.la equivalent to the conjunction of the outputs which is then passed to a Z3 engine to check its satisfiability

The architecture of **TRAC**





Finally, the `Analizer` component that diagnoses the output of Z3 and produces a `Verdict` which reports (if any) the violations of well-formedness of the DAFSM in input.

Installation

Detailed instructions at `https://github.com/loctet/TRAC`

Dependencies: Java RE (to render DAFSM graphically) & Python 3.6 or later

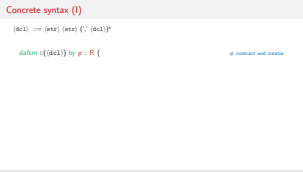
```
$ pip install z3-solver matplotlib networkx
```


Installation
Detailed instructions at https://github.com/loctet/TRAC
Dependencies: Java RE (to render DAFSM graphically) & Python 3.6 or later
$ pip install z3-solver matplotlib networkx

Concrete syntax (I)

⟨dcl⟩ ::= ⟨str⟩ ⟨str⟩ (',' ⟨dcl⟩)*

dafsm c(⟨dcl⟩) by p : R { # contract and creator



recall that e and γ are SMT-Lib2 syntax for expressions and boolean expressions respectively

Concrete syntax (I)

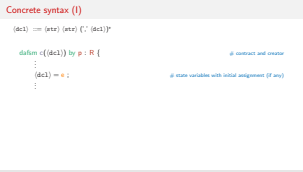
⟨dcl⟩ ::= ⟨str⟩ ⟨str⟩ (',' ⟨dcl⟩)*

dafsm c(⟨dcl⟩) by p : R { # contract and creator

⋮

⟨dcl⟩ = e ; # state variables with initial assignment (if any)

⋮



recall that e and γ are SMT-Lib2 syntax for expressions and boolean expressions respectively

Concrete syntax (I)

⟨dcl⟩ ::= ⟨str⟩ ⟨str⟩ (',' ⟨dcl⟩)*

```
dafsm c(⟨dcl⟩) by p : R {                    # contract and creator
    ⋮
    ⟨dcl⟩ = e ;                              # state variables with initial assignment (if any)
    ⋮
    if γ                                     # initial guard (this clause can be omitted)
}
```


Concrete syntax (I)
⟨dcl⟩ ::= ⟨str⟩ ⟨str⟩ (',' ⟨dcl⟩)*
dafsm c(⟨dcl⟩) by p : R { # contract and creator
⋮
⟨dcl⟩ = e ; # state variables with initial assignment (if any)
⋮
if γ # initial guard (this clause can be omitted)
}

recall that e and γ are SMT-Lib2 syntax for expressions and boolean expressions respectively

Concrete syntax (I)

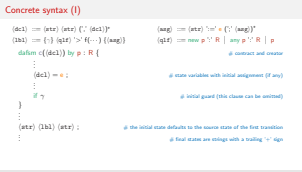
⟨dcl⟩ ::= ⟨str⟩ ⟨str⟩ (',' ⟨dcl⟩)*

⟨lbl⟩ ::= {γ} ⟨qlf⟩ '>' f(···) {⟨asg⟩}

⟨asg⟩ ::= ⟨str⟩ ':=' e (';' ⟨asg⟩)*

⟨qlf⟩ ::= new p ':' R | any p ':' R | p

```
dafsm c(⟨dcl⟩) by p : R {                          # contract and creator
    ⋮
    ⟨dcl⟩ = e ;                                    # state variables with initial assignment (if any)
    ⋮
    if γ                                           # initial guard (this clause can be omitted)
}
⋮
⟨str⟩ ⟨lbl⟩ ⟨str⟩ ;                                # the initial state defaults to the source state of the first transition
⋮                                                  # final states are strings with a trailing '+' sign
```

recall that e and γ are SMT-Lib2 syntax for expressions and boolean expressions respectively
in gray optional lexemes

Concrete syntax (I)

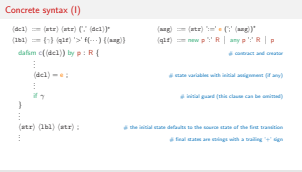
⟨dcl⟩ ::= ⟨str⟩ ⟨str⟩ (',' ⟨dcl⟩)*

⟨lbl⟩ ::= {γ} ⟨qlf⟩ '>' f(···) {⟨asg⟩}

⟨asg⟩ ::= ⟨str⟩ ':=' e (';' ⟨asg⟩)*

⟨qlf⟩ ::= new p ':' R | any p ':' R | p

```
dafsm c(⟨dcl⟩) by p : R {                          # contract and creator
    ⋮
    ⟨dcl⟩ = e ;                                    # state variables with initial assignment (if any)
    ⋮
    if γ                                           # initial guard (this clause can be omitted)
}
⋮
⟨str⟩ ⟨lbl⟩ ⟨str⟩ ;                                # the initial state defaults to the source state of the first transition
⋮                                                  # final states are strings with a trailing '+' sign
```

recall that e and γ are SMT-Lib2 syntax for expressions and boolean expressions respectively
in gray optional lexemes

Exercise: **TRAC** syntax (I)

Edit a `.trac` file for the DAFSM on slide **??**.

use basic_provenance.txt ?

Exercise: **TRAC** syntax (I)
Edit a `.trac` file for the DAFSM on slide **??**.

Concrete syntax (II)

The syntax of expressions (and hence of guards) follows the z3py API (at ??)

Concrete syntax (II)
The syntax of expressions (and hence of guards) follows the z3py API (at ??)

Exercise: **TRAC** syntax (II)

Edit a `.trac` file for the DAFSM on slide **??**.

Exercise: **TRAC** syntax (II)
Edit a `.trac` file for the DAFSM on slide **??**.

– Act III –

[A little exercise]

https://github.com/blockchain-unica/rosetta-smart-contracts/tree/main/contracts/vesting

– Epilogue –

[Work in progress]

Thank you

References I

[1] J. Afonso, E. Konjoh Selabi, M. Murgia, A. Ravara, and E. Tuosto. TRAC: A tool for data-aware coordination - (with an application to smart contracts).
In I. Castellani and F. Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *LNCS*, pages 239–257. Springer, 2024.

[2] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming.
ACM Trans. Program. Lang. Syst., 36(4), Oct. 2014.

[3] B. Meyer. *Introduction to the Theory of Programming Languages*.
Prentice-Hall, 1990.

References I

[1] J. Afonso, E. Konjoh Selabi, M. Murgia, A. Ravara, and E. Tuosto. TRAC: A tool for data-aware coordination - (with an application to smart contracts).
In I. Castellani and F. Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *LNCS*, pages 239–257. Springer, 2024.

[2] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming.
ACM Trans. Program. Lang. Syst., 36(4), Oct. 2014.

[3] B. Meyer. *Introduction to the Theory of Programming Languages*.
Prentice-Hall, 1990.

References II

[4] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[5] Microsoft. The blockchain workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench, 2019.

[6] Microsoft. Simple marketplace sample application for azure blockchain workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/simple-marketplace, 2019.

References II

[4] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[5] Microsoft. The blockchain workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench, 2019.

[6] Microsoft. Simple marketplace sample application for azure blockchain workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/simple-marketplace, 2019.