A Choreographic View of Smart Contracts

Elvis Gerardin Konjoh Selabi
@GSSI & UniCam

Maurizio Murgia
@GSSI

António Ravara
@NOVA

Emilio Tuosto
@ GSSI

A tutorial @ FORTE 2025, Lille

Work partly supported by the PRIN 2022 PNRR project DeLiCE (F53D23009130001)

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 . . . . . Some tool support

What's up doc?

What's up doc?

Prologue . . . . . An inspiring initiative

Act I . . . . . A coordination framework

Act II . . . . . Some tool support

Act III . . . . . A little exercise

What's up doc?

What's up doc?

Prologue ..... An inspiring initiative

Act I ..... A coordination framework

Act II ..... Some tool support

Act III ..... A little exercise

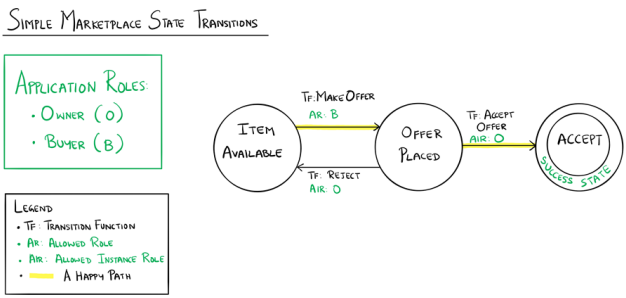
Epilogue ..... Work in progress

– Prologue –

[An inspiring initiative]

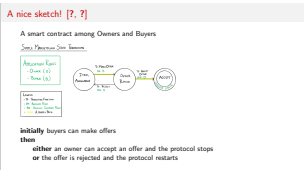
A nice sketch! [?, ?]

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?

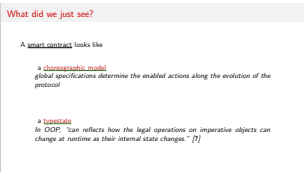
A smart contract looks like

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." [?]



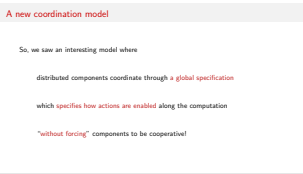
A new coordination model

So, we saw an interesting model where

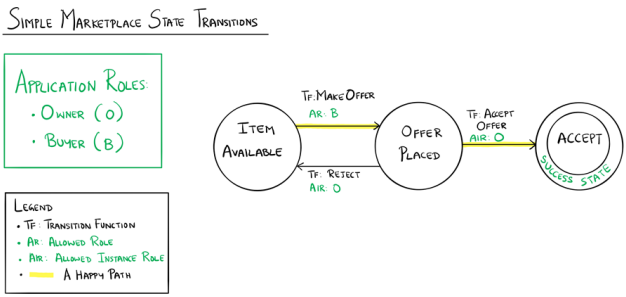
distributed components coordinate through a global specification

which specifies how actions are enabled along the computation

"without forcing" components to be cooperative!



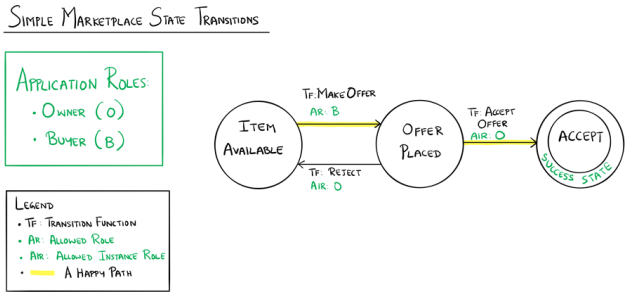
Let's look at our sketch again



Let's look at our sketch again

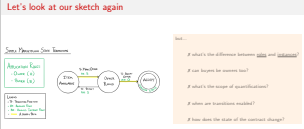
The diagram specifies a lot...

Let's look at our sketch again



but...

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



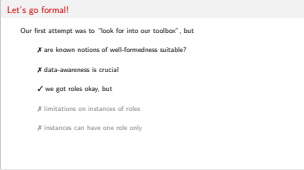
The diagram specifies a lot...

1. is the sketch giving semantics to roles and instances?
2. not forbidden...however what if we wanted to separate the roles?
3. from [**?**]: "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?
4. ok
5. should the price of the item remain unchanged when the owner rejects offers?

Let's go formal!

Our first attempt was to "look for into our toolbox", but

- ✗ are known notions of well-formedness suitable?
- ✗ data-awareness is crucial
- ✓ we got roles okay, but
- ✗ limitations on instances of roles
- ✗ instances can have one role only

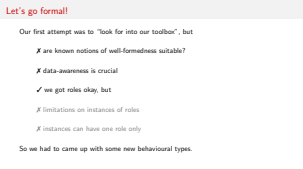


Let's go formal!

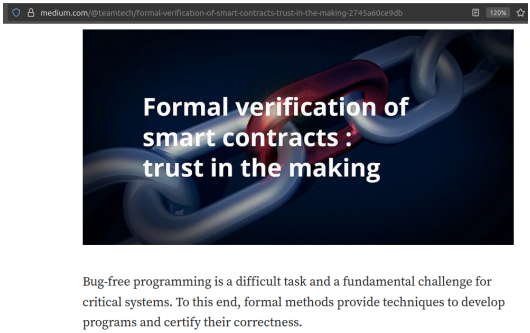
Our first attempt was to "look for into our toolbox", but

- ✗ are known notions of well-formedness suitable?
- ✗ data-awareness is crucial
- ✓ we got roles okay, but
- ✗ limitations on instances of roles
- ✗ instances can have one role only

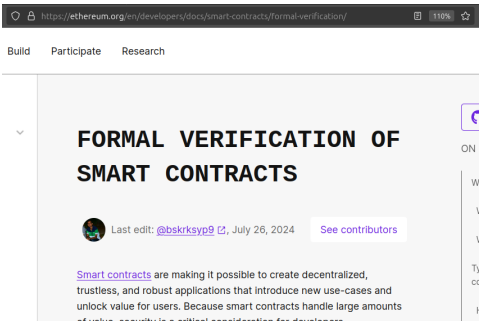
So we had to came up with some new behavioural types.



...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/

...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/

– Act I –

[A coordination framework]

Basic concepts and notation

Participants $p, p', \ldots$

Basic concepts and notation
Participants $p, p', \ldots$

Basic concepts and notation

Participants $p, p', \ldots$
have roles $R, R', \ldots$

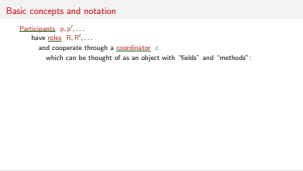
Basic concepts and notation
Participants $p, p', \ldots$
have roles $R, R', \ldots$

Basic concepts and notation

Participants $p, p', \ldots$
have roles $R, R', \ldots$
and cooperate through a coordinator c

Basic concepts and notation

Participants $p, p', \ldots$
have roles $R, R', \ldots$
and cooperate through a coordinator c
which can be thought of as an object with "fields" and "methods":



states of the coordinator determine which operations each roles is entitled to invoke

Basic concepts and notation

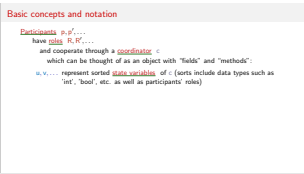
Participants $p, p', \ldots$

have roles $R, R', \ldots$

and 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)



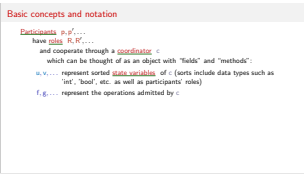
We assume that sorts can be inferred; **TRAC** instead requires to assign sorts explicitly

Basic concepts and notation

Participants $p, p', \ldots$
have roles $R, R', \ldots$
and 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)

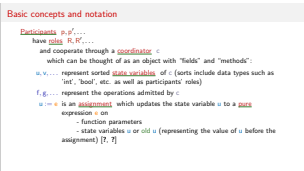
$f, g, \ldots$ represent the operations admitted by c



Basic concepts and notation

Participants $p, p', \ldots$
have roles $R, R', \ldots$
and 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
- $u := e$ is an assignment which updates the state variable u to a pure expression e on
  - function parameters
  - state variables u or old u (representing the value of u before the assignment) [**?**, **?**]



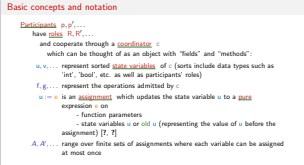
Expressions are standard but for state variables occurring in rhs e must have the old _ qualifier; this concept will be used in the definition of (progress for) well-formedness

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

Basic concepts and notation

Participants $p, p', \ldots$
have roles $R, R', \ldots$
and 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
- $u := e$ is an assignment which updates the state variable u to a pure expression e on
  - function parameters
  - state variables u or old u (representing the value of u before the assignment) [**?**, **?**]
- $A, A', \ldots$ range over finite sets of assignments where each variable can be assigned at most once



Data-Aware FSMs

A DAFSM c on roles $R_1, \ldots R_m$ and state variables $u_1, \ldots, u_n$ is a finite-state machine "instantiated" by a participant p whose transitions are decorated as follows[1]

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

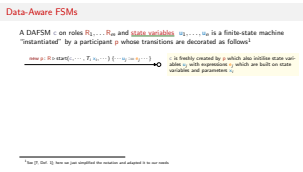
Data-Aware FSMs

A DAFSM c on roles $R_1, \ldots R_m$ and state variables $u_1, \ldots, u_n$ is a finite-state machine "instantiated" by a participant p whose transitions are decorated as follows[1]

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

c is freshly created by p which also initilise state variables u_j with expressions e_j which are built on state variables and parameters x_i

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

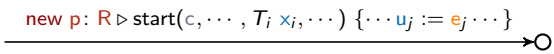


start is a "built-in" (and pleonastic) function name

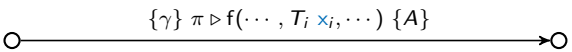
each state variable is declared and initialises with type-consistent expressions on state variables and parameters x_i

Data-Aware FSMs

A DAFSM c on roles $R_1, \ldots R_m$ and state variables $u_1, \ldots, u_n$ is a finite-state machine "instantiated" by a participant p whose transitions are decorated as follows[1]



c is freshly created by p which also initilise state variables u_j with expressions e_j which are built on state variables and parameters x_i

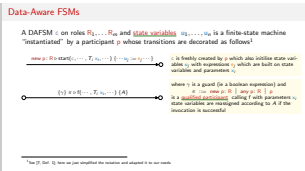


where γ is a guard (ie a boolean expression) and

$$\pi \ ::= \ \texttt{new}\ p\colon R \ \mid \ \texttt{any}\ p\colon R \ \mid \ p$$

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

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



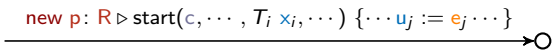
γ predicates over state variables and formal parameters of its transition; guards have to be satisfied for the invocation to succeed: an invocation that makes the guard false is rejected

- new 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 refers to a participant in the scope of a binder
- invocations from non-suitable callers are rejected

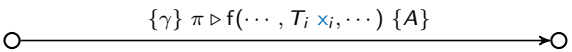
the variables occurring in the right-hand side of assignments in A are either state variables or parameters of the invocation

Data-Aware FSMs

A DAFSM c on roles $R_1, \dots R_m$ and state variables $u_1, \dots, u_n$ is a finite-state machine "instantiated" by a participant p whose transitions are decorated as follows[1]



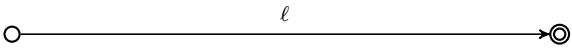
c is freshly created by p which also initilise state variables u_j with expressions e_j which are built on state variables and parameters x_i



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

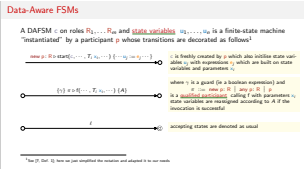
$$\pi \ ::= \ \text{new } p\colon R \ \mid \ \text{any } p\colon R \ \mid \ p$$

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



accepting states are denoted as usual

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

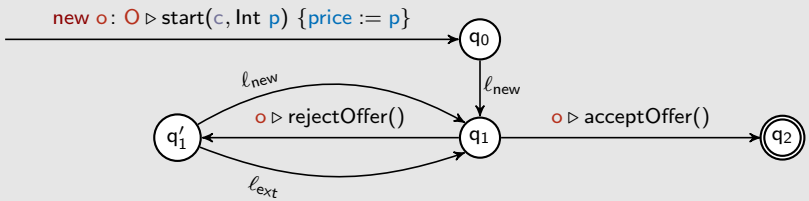


Exercise: modelling

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

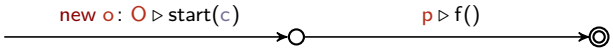
Exercise: modelling
Give a DAFSM for the protocol on slide 7 resolving the ambiguities discussed there.

Let $\ell_{\mathsf{new}} = \{\mathsf{newOffer} > 0\}$ new b: B ▷ makeOffer(Int newOffer) {offer := newOffer}
and $\ell_{\mathsf{ext}} = \{\mathsf{newOffer} > 0\}$ any b: B ▷ makeOffer(Int newOffer) {offer := newOffer}



A new participant o acts as owner O for a coordinator c. The set of assignments is the singleton initialising the state variable price to p.
In q_0, the only enabled function is makeOffer(Int offer); the first buyer b invoking this function with an actual parameter newOffer, satisfying the guard newOffer > 0, moves the protocol to state q_1 while recording the new offer in the coordinator state with the assignment offer := newOffer. Contextually, the state of the coordinator records that the caller b plays role B.

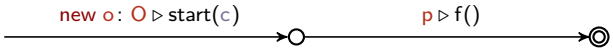
Not all DAFSMs “make sense”



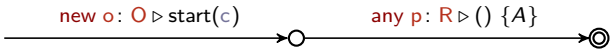
names’ freeness

Not all DAFSMs “make sense”
names’ freeness

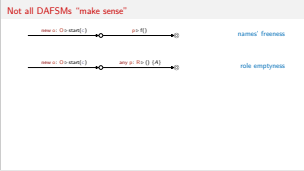
Not all DAFSMs “make sense”



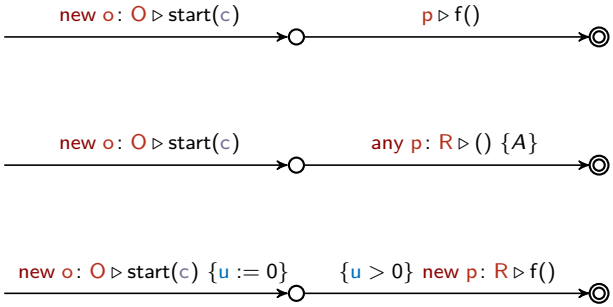
names’ freeness



role emptyness



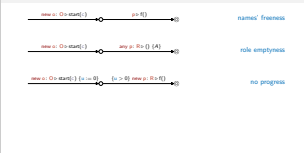
Not all DAFSMs “make sense”



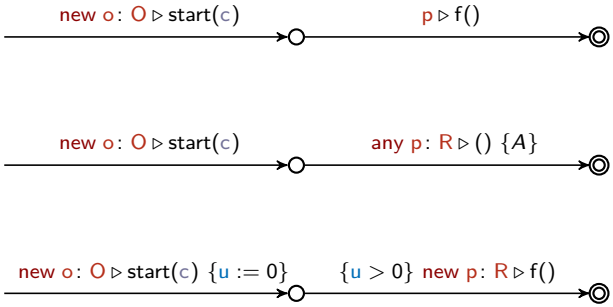
names’ freeness

role emptyness

no progress



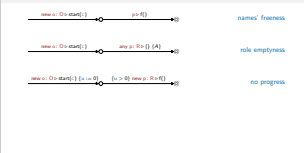
Not all DAFSMs “make sense”



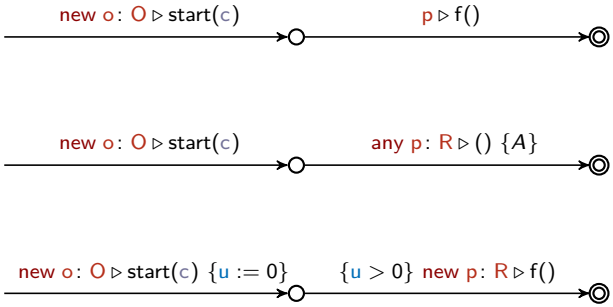
names’ freeness

role emptyness

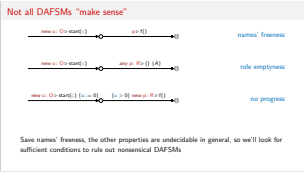
no progress



Not all DAFSMs “make sense”



Save names’ freeness, the other properties are undecidable in general, so we’ll look for sufficient conditions to rule out nonsensical DAFSMs



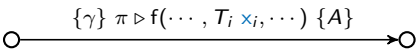
Closed DAFSMs

Binders: parameter declarations in function calls, new p : R, and any p : R

Closed DAFSMs
Binders: parameter declarations in function calls, new p: R, and any p: R

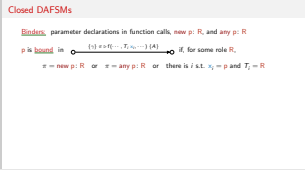
Closed DAFSMs

Binders: parameter declarations in function calls, new p: R, and any p: R

p is bound in 

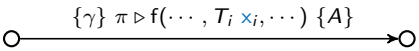
if, for some role R,

π = new p: R or π = any p: R or there is i s.t. x_i = p and T_i = R



Closed DAFSMs

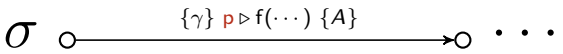
Binders: parameter declarations in function calls, new p: R, and any p: R

p is bound in 

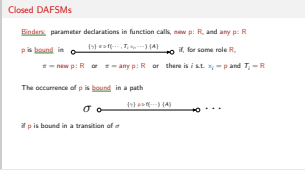
if, for some role R,

π = new p: R or π = 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

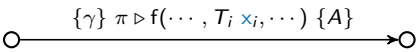


if p is bound in a transition of σ



Closed DAFSMs

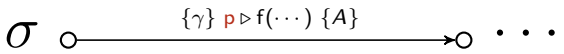
Binders: parameter declarations in function calls, new p: R, and any p: R

p is bound in 

if, for some role R,

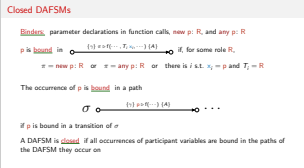
$\pi = \texttt{new p: R}$ or $\pi = \texttt{any p: R}$ or there is i s.t. $x_i = \texttt{p}$ and $T_i = \texttt{R}$

The occurrence of p is bound in a path



if p is bound in a transition of σ

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



Role emptyness

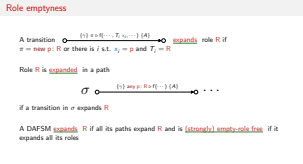
A transition $\xrightarrow{\{\gamma\}\ \pi \triangleright \mathsf{f}(\cdots, T_i\ \mathsf{x}_i, \cdots)\ \{A\}}$ expands role R if $\pi = \mathsf{new}\ \mathsf{p}\colon \mathsf{R}$ or there is i s.t. $\mathsf{x}_i = \mathsf{p}$ and $T_i = \mathsf{R}$

Role R is expanded in a path

$$\sigma \xrightarrow{\{\gamma\}\ \mathsf{any}\ \mathsf{p}\colon \mathsf{R} \triangleright \mathsf{f}(\cdots)\ \{A\}} \cdots$$

if a transition in σ expands R

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



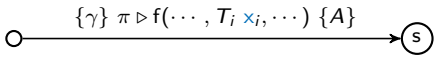
'expands' means register a new participant with that role in the protocol (the participant might already be registered with a different role)

Exercise: Role emptyness

todo

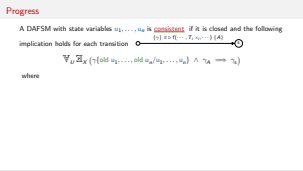
Exercise: Role emptyness
todo

Progress

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

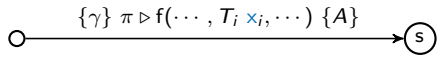
$$\mathbb{V}_U \, \mathbb{E}_X \left(\gamma\{\text{old } u_1, \ldots, \text{old } u_n / u_1, \ldots, u_n\} \ \wedge \ \gamma_A \implies \gamma_s \right)$$

where



for a finite set of symbols Z, $\mathbb{V}_Z\,(\_)$ and $\mathbb{E}_Z\,(\_)$ are the universal and existential closures of a logical formula on the symbols in Z

Progress

A DAFSM with state variables $\mathsf{u}_1, \ldots, \mathsf{u}_n$ is consistent if it is closed and the following implication holds for each transition 

$$\mathbb{\forall}_U \mathbb{\exists}_X \left(\gamma\{\mathsf{old}\ \mathsf{u}_1, \ldots, \mathsf{old}\ \mathsf{u}_n / \mathsf{u}_1, \ldots, \mathsf{u}_n\} \ \wedge\ \gamma_A \implies \gamma_\mathsf{s}\right)$$

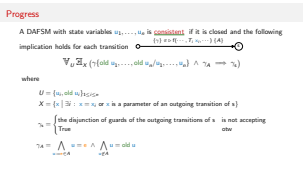
where

$$U = \{\mathsf{u}_i, \mathsf{old}\ \mathsf{u}_i\}_{1 \le i \le n}$$

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

$$\gamma_\mathsf{s} = \begin{cases} \text{the disjunction of guards of the outgoing transitions of s} & \text{is not accepting} \\ \text{True} & \text{otw} \end{cases}$$

$$\gamma_A = \bigwedge_{\mathsf{u} := \mathsf{e} \in A} \mathsf{u} = \mathsf{e} \ \wedge \ \bigwedge_{\mathsf{u} \notin A} \mathsf{u} = \mathsf{old}\ \mathsf{u}$$



for a finite set of symbols Z, $\mathbb{\forall}_Z(\_)$ and $\mathbb{\exists}_Z(\_)$ are the universal and existential closures of a logical formula on the symbols in Z

$\mathsf{u} \notin A$
iff
for all $\mathsf{v} := \mathsf{e} \in A$, $\mathsf{u} \neq \mathsf{v}$ and $\mathsf{old}\ \mathsf{u}$ does not occur in e

Exercise: Consistency

todo

Exercise: Consistency

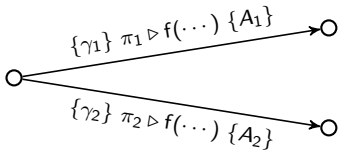
todo

Determinism

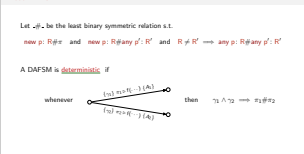
Let $\_\#\_$ be the least binary symmetric relation s.t.

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

A DAFSM is deterministic if

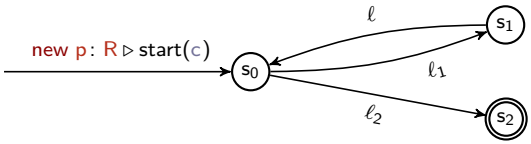
whenever 

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



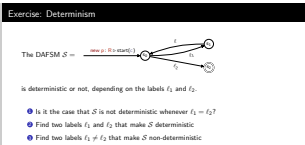
transitions from the same source state and calling the same function

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



1. no: eg for $\ell_1 = \ell_2 =$ new p : R $\mathcal{S}$ is deterministic
2. $\ell_1 = \ell_2 =$ new 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(Int x) and $\ell_2 = \{x \geq -1\}$ p ▷ f(Int x) 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

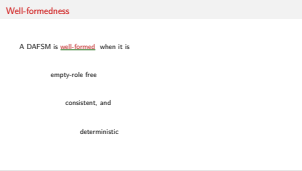
Well-formedness

A DAFSM is well-formed when it is

empty-role free

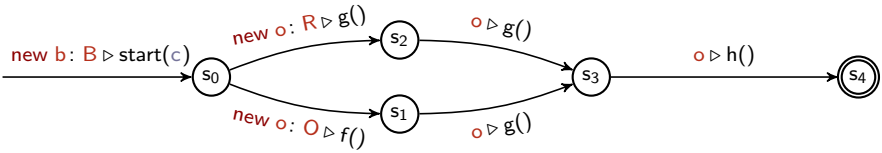
consistent, and

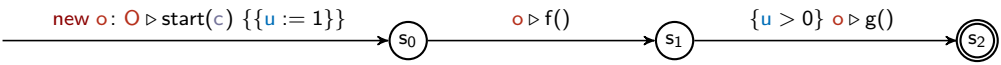
deterministic

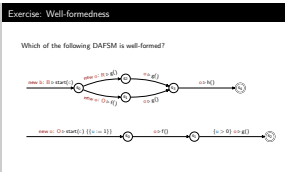


Exercise: Well-formedness

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 –

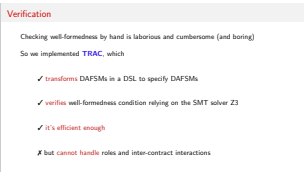
[A tool]

Verification

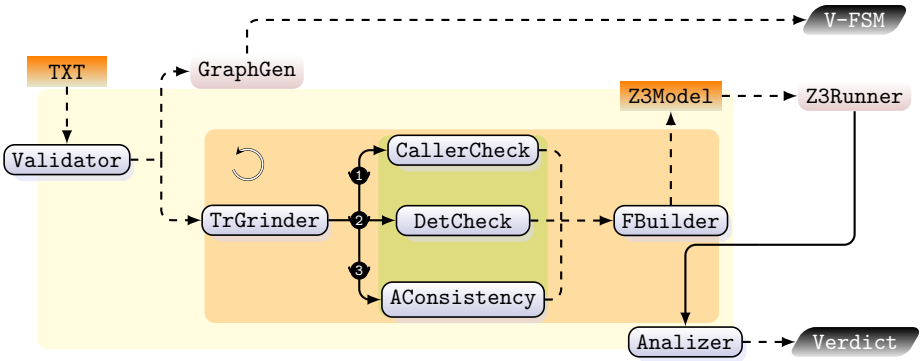
Checking well-formedness by hand is laborious and cumbersome (and boring)

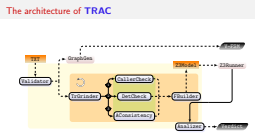
So we implemented **TRAC**, which

- ✓ 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



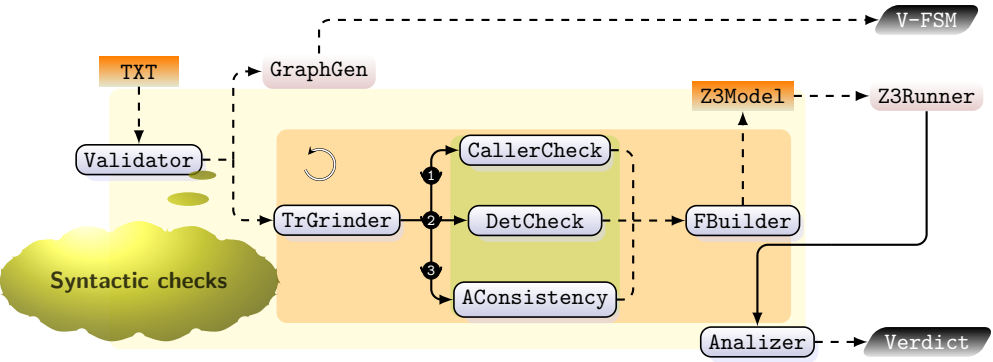
The architecture of **TRAC**

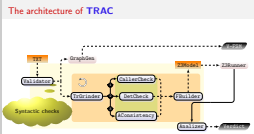




the architecture of **TRAC** is compartmentalised into two principal modules: parsing and visualisation (yellow box) and **TRAC**'s core (orange box). The latter module implements well-formedness check (green box). Solid arrows represent calls between components while dashed arrows data IO.

The architecture of **TRAC**

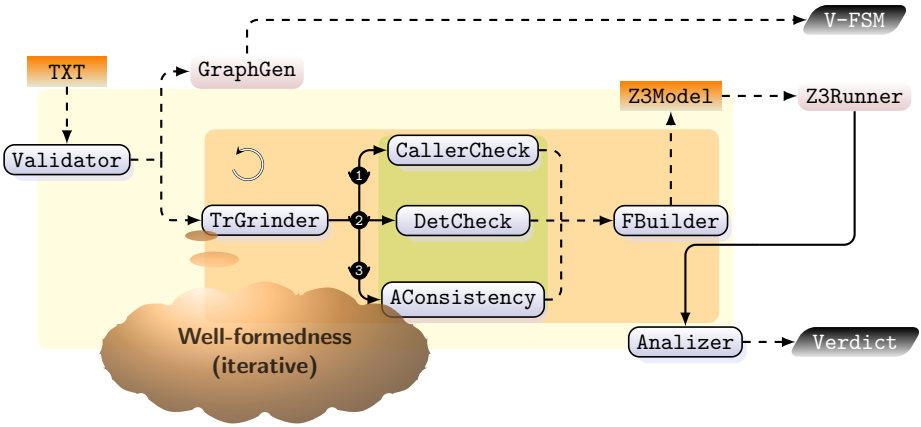


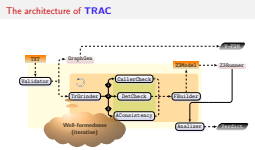


basic syntactic checks on a DSL representation of DAFSMs and transforming the input in a format that simplifies the analysis of the following phases:

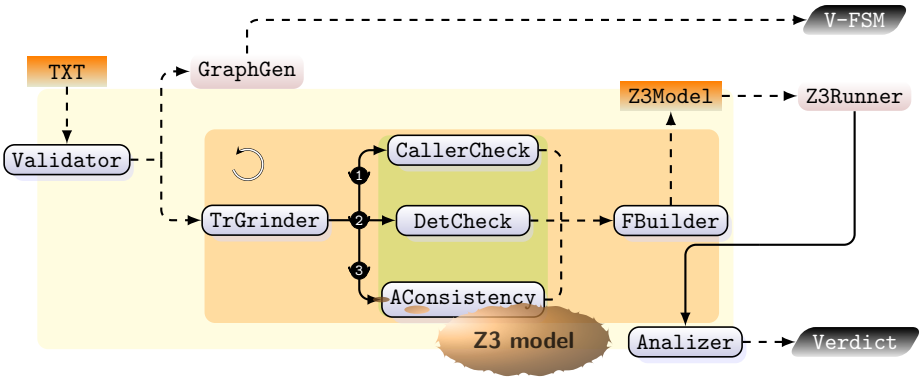
- passed to `GraphGen` for visual representation of DAFSMs (`V-FSM` output)
- passed to the `TrGrinder` component (orange box) for well-formedness checking.

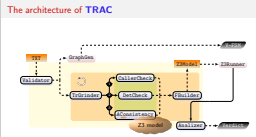
The architecture of **TRAC**





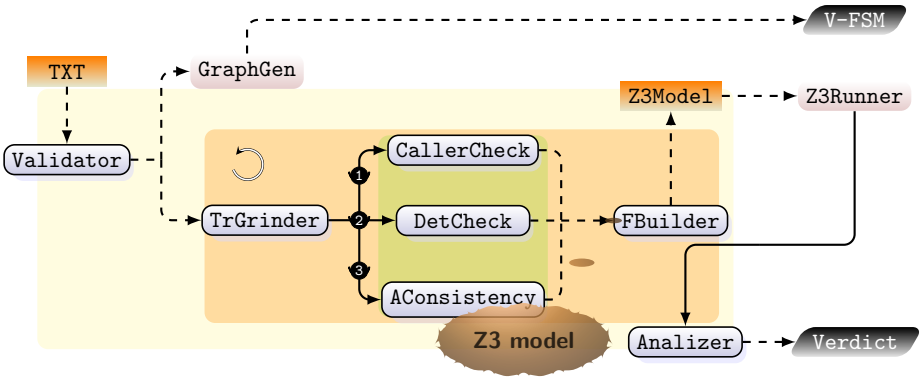
The architecture of **TRAC**

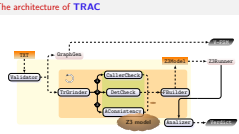




`AConsistency` (arrow ❸) to generate a Z3 formula which holds if, and only if, the transtion is consistent.

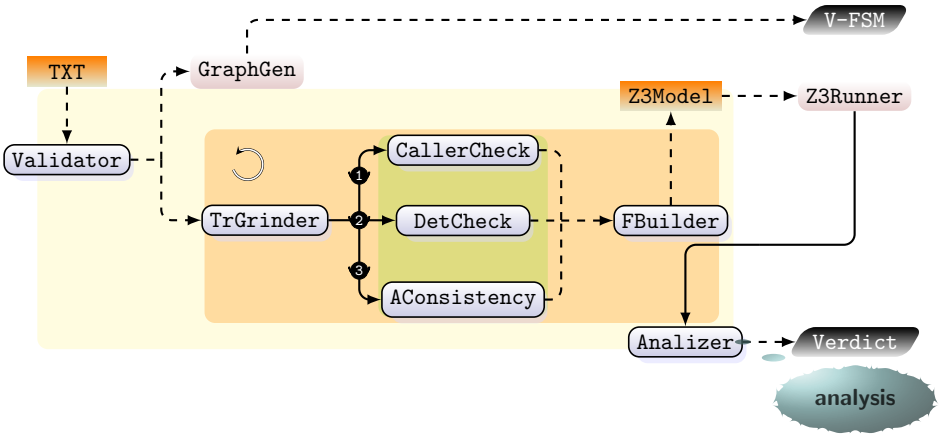
The architecture of **TRAC**

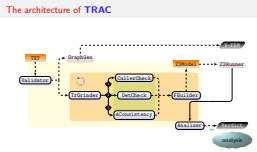




computes the z3 f.la equivalent to the conjunction of the outputs which is then passed to a Z3 engine to check its satisfiability

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
```


Concrete syntax (I)

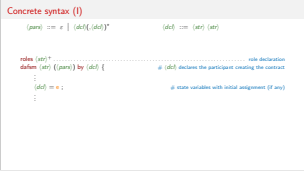
$\langle pars \rangle \ ::= \ \varepsilon \ \mid \ \langle dcl \rangle (, \langle dcl \rangle)^\star$ $\qquad$ $\langle dcl \rangle \ ::= \ \langle str \rangle \ \langle str \rangle$

roles $\langle str \rangle^+$ . . . . . role declaration
dafsm $\langle str \rangle$ ($\langle pars \rangle$) by $\langle dcl \rangle$ { # $\langle dcl \rangle$ declares the participant creating the contract

Concrete syntax (I)

$\langle pars \rangle$ ::= ε | $\langle dcl \rangle (,\langle dcl \rangle)^\star$ $\quad\quad$ $\langle dcl \rangle$ ::= $\langle str \rangle$ $\langle str \rangle$

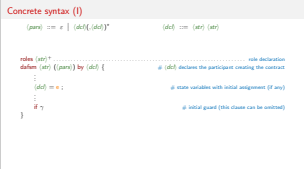
roles $\langle str \rangle^+$ . . . . . role declaration
dafsm $\langle str \rangle$ ($\langle pars \rangle$) by $\langle dcl \rangle$ { # $\langle dcl \rangle$ declares the participant creating the contract
⋮
$\langle dcl \rangle$ = e ; # state variables with initial assignment (if any)
⋮



Concrete syntax (I)

$\langle pars \rangle \ ::= \ \varepsilon \ \mid \ \langle dcl \rangle (,\langle dcl \rangle)^\star$ $\qquad$ $\langle dcl \rangle \ ::= \ \langle str \rangle \ \langle str \rangle$

roles $\langle str \rangle^+$ . . . . . role declaration
dafsm $\langle str \rangle$ ($\langle pars \rangle$) by $\langle dcl \rangle$ { # $\langle dcl \rangle$ declares the participant creating the contract
    ⋮
    $\langle dcl \rangle$ = 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)

$\langle pars \rangle ::= \varepsilon \mid \langle dcl \rangle (,\langle dcl \rangle)^\star$ $\quad\quad$ $\langle dcl \rangle ::= \langle str \rangle\ \langle str \rangle$

$\langle lbl \rangle ::= \{\gamma\}\ \pi > \langle str \rangle (\langle pars \rangle)\ \{\langle asgs \rangle\}$

$\langle asgs \rangle ::= \varepsilon \mid \langle asg \rangle (;\langle asg \rangle)^\star$ $\quad\quad$ $\langle asg \rangle ::= \langle str \rangle := \langle expr \rangle$

roles $\langle str \rangle^+$ . . . . . role declaration

dafsm $\langle str \rangle$ ($\langle pars \rangle$) by $\langle dcl \rangle$ { # $\langle dcl \rangle$ declares the participant creating the contract

⋮

$\langle dcl \rangle$ = e ; # state variables with initial assignment (if any)

⋮

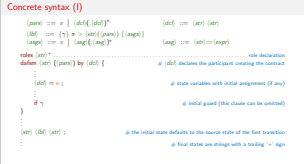
if γ # initial guard (this clause can be omitted)

}

⋮

$\langle str \rangle\ \langle lbl \rangle\ \langle str \rangle$; # 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

Concrete syntax (I)

$\langle pars \rangle ::= \varepsilon \mid \langle dcl \rangle (,\langle dcl \rangle)^\star$ $\quad\quad$ $\langle dcl \rangle ::= \langle str \rangle\ \langle str \rangle$

$\langle lbl \rangle ::= \{\gamma\}\ \pi > \langle str \rangle (\langle pars \rangle)\ \{\langle asgs \rangle\}$

$\langle asgs \rangle ::= \varepsilon \mid \langle asg \rangle (;\langle asg \rangle)^\star$ $\quad\quad$ $\langle asg \rangle ::= \langle str \rangle := \langle expr \rangle$

roles $\langle str \rangle^+$ . . . . . role declaration

dafsm $\langle str \rangle$ ($\langle pars \rangle$) by $\langle dcl \rangle$ { # $\langle dcl \rangle$ declares the participant creating the contract

⋮

$\langle dcl \rangle$ = e ; # state variables with initial assignment (if any)

⋮

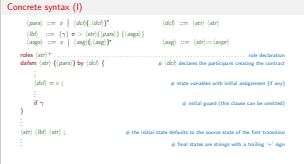
if γ # initial guard (this clause can be omitted)

}

⋮

$\langle str \rangle\ \langle lbl \rangle\ \langle str \rangle$; # 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

Exercise: TRAC usage (I)

Edit a `.trac` file for the contract specified at
`https://github.com/Azure-Samples/blockchain/blob/master/blockchain-workbench/application-and-smart-contract-samples/basic-provenance/readme.md`

Exercise: TRAC usage (I)
Edit a .trac file for the contract specified at
https:
//github.com/Azure-Samples/blockchain/blob/master/blockchain-workbench/application-and-smart-contract-samples/basic-provenance/readme.md

```
roles Owner Conterparty
dafsm basicProvenance(Owner o) by cp : Conterparty {}
q0 cp > TransferResponsibility(Conterparty cp) {} q1
q1 any cp: Conterparty > TransferResponsibility(Conterparty cp) {} q1
q1 o > Complete() {} q2+
```

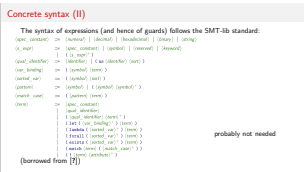
Concrete syntax (II)

The syntax of expressions (and hence of guards) follows the SMT-lib standard:

⟨spec_constant⟩ ::= ⟨numeral⟩ | ⟨decimal⟩ | ⟨hexadecimal⟩ | ⟨binary⟩ | ⟨string⟩
⟨s_expr⟩ ::= ⟨spec_constant⟩ | ⟨symbol⟩ | ⟨reserved⟩ | ⟨keyword⟩
| (⟨s_expr⟩*)
⟨qual_identifier⟩ ::= ⟨identifier⟩ | (as ⟨identifier⟩ ⟨sort⟩)
⟨var_binding⟩ ::= (⟨symbol⟩ ⟨term⟩)
⟨sorted_var⟩ ::= (⟨symbol⟩ ⟨sort⟩)
⟨pattern⟩ ::= ⟨symbol⟩ | (⟨symbol⟩ ⟨symbol⟩+)
⟨match_case⟩ ::= (⟨pattern⟩ ⟨term⟩)
⟨term⟩ ::= ⟨spec_constant⟩
| ⟨qual_identifier⟩
| (⟨qual_identifier⟩ ⟨term⟩+)
| (let (⟨var_binding⟩+) ⟨term⟩)
| (lambda (⟨sorted_var⟩+) ⟨term⟩)
| (forall (⟨sorted_var⟩+) ⟨term⟩)
| (exists (⟨sorted_var⟩+) ⟨term⟩)
| (match ⟨term⟩ (⟨match_case⟩+))
| (! ⟨term⟩ ⟨attribute⟩+)

probably not needed

(borrowed from [?])



https://smt-lib.org/papers/smt-lib-reference-v2.6-r2021-05-12.pdf
http://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf

Exercise: TRAC syntax (II)

Edit a `.trac` file for the DAFSM on slide 13.

Exercise: TRAC syntax (II)
Edit a .trac file for the DAFSM on slide 13.

TODO

– Act III –

[A little exercise]

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

– Epilogue –

[Work in progress]

Work in progress

Work in progress

Thank you

References I

References I