A Choreographic View of Smart Contracts*

Emilio Tuosto[1]

Gran Sasso Science Institute, L'Aquila, Italy

Abstract. This tutorial concerns the use of a novel model of coordination of distributed systems. The model combines ideas from choreographic approaches and smart contracts. More precisely, application protocols regulating the coordination of a distributed application are rendered as *global views* that specify the expected behaviour of the system. Unlike in standard choreographic models though, participants are not necessarily obtained by projection from global views and can behave in completely unexpected ways. The adopted countermeasure to erroneous or malicious behaviour of participants is the one adopted in smart contracts: disabled interactions are just ignored.

1 Introduction

The design and implementation of distributed computations requires great attention to information and control flows. In fact, distributed computation require suitable mechanisms and protocols to properly share information about the state of the computation among the unit of computations.[1] Paramount questions for "correctness" in this context are:

1. what information should participants use to coordinate among themselves?
2. how is this information supposed to be shared among participants?
3. when should the information sharing happen?

How to address these questions depends on which properties one wants to enforce (e.g., classical properties such deadlock freedom, security such as confidentiality, etc.), on which assumptions are made on participants (e.g., if they are cooperative, competitive, malicious, etc.), and on which interaction mechanisms are adopted (e.g., message passing, remote procedure call, etc.).

Increasing attention has been recently paid to *choreographic approaches* [11] both in academia and industry [17,4,6,3] since choreographies stand out for a neat separation of concerns which (*i*) abstracts away local computations from participants' interactions and (*ii*) advocates a two-pronged description of computations consisting of a *global view* and a *local view* of systems.

* Research partly supported by the PRIN PNRR project DeLICE (F53D23009130001), by the MUR dipartimento di eccellenza 2023-2027

[1] We will call such units 'participants'; alternative terminology refers to such units as 'agents', 'processes', 'components'.

This tutorial hinges on a recent approach leveraging global views of choreographies to specify the coordination of participants through mechanisms inspired by execution models of *smart contracts* [19]. Our approach hinges on an attempt to formalise the informal model advocated in the Azure initiative of Microsoft [15] where models for the coordination of smart contract (SC for short) are given as finite-state machines (FMSs). Albeit commendable, this idea is only informally sketched; we illustrate this proposal using the simple marketplace (SMP) scenario.

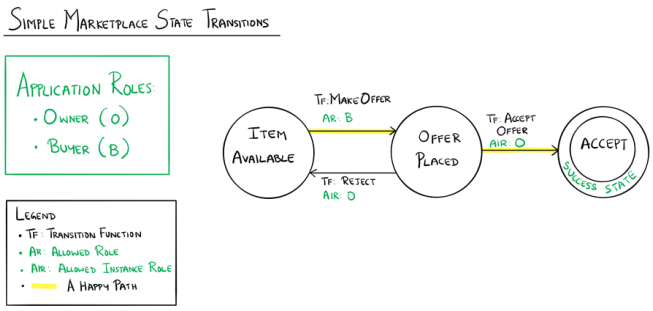


Fig. 1: The simple marketplace

Fig. 1 borrows the sketch of the SMP specification as given in [16]. The key elements of the sketch are *roles* (Owner and Buyer) played by participants of the protocol. The initial state Item Available enables the transition MakeOffer for participants acting as buyers since, according to the Azure notation, the green decoration on transitions specifies the roles allowed to invoke the operation transition function (in the Azure jargon). On successful completion of this invocation, the protocol moves to the next state Offer Placed; in this state the owner can choose either to Accept or Reject the offer. In the former case, the protocol reaches the success state Accept, otherwise the protocol moves back to Item Available. The expected behaviour (the Happy Path in Azure jargon) is the one highlighted in Fig. 1; in our example, the happy path can be interpreted as "an offer is eventually accepted".

As noted in [1], this idea is reminiscent of monitors [8,9] which encapsulate a state accessible through an API whose operations are guarded by conditions set to maintain an invariant on the state (in the SMP scenario the operations are MakeOffer, AcceptOffer, and Reject). Crucial aspects, however, tell apart monitors from this model. Firstly, our participants are distributed and do not share memory; secondly, callers of disabled operation do not suspend.

As said, the Azure approach is appealing but not formal. The sketch of the SMP scenario, for instance, blurs away some important details. For instance, the informal description hints that there is a single owner but, instead of being explicitly stated, this is implied by the use of Allowed Role and Allowed Role

Instance. Likewise, the description is unclear if an owner participant can also act as buyers. (Other ambiguities are described in [1].)

The model we use in this tutorial is based on a new class of symbolic finite-state machines formalising the Azure approach. Our model, at the same time, exposes the APIs of each participant of the protocol as well as the expected behaviour of the system. As global specifications such as global types [10], our model is instrumental to define a notion of *well-formedness*. Interestingly data-dependency, crucial in many contexts, can be explicitly represented. This is a quintessential aspect to handle properties that, like well-formedness, depend on the payloads of components' interactions such as those typical in smart contracts.

The natural application domain of our module are smart contracts; this tutorial shows this by covering the modelling and analysis of some smart contracts with a companion tool supporting our approach.

Structure of the paper Section 2 adapts the model presented in [1] to this tutorial (with minor adaptations to make the presentation lighter). Section 3 briefly discuss the intricacies of non-determinism in our model. Section 4 informally discusses our well-formedness conditions. Finally, Section 5 outlines our tool and the DSL it uses for the analysis.

2 Smart contract inspired coordination

In our model, protocols' *participants* $\mathsf{p}, \mathsf{p}', \ldots$ cooperate through a *coordinator* c according to their *role roles* $\mathsf{R}, \mathsf{R}', \ldots$. Each coordinator c has:

- A finite set of *state variables* $\mathsf{u}, \mathsf{v}, \ldots$; each state variable has an associated data type, e.g., $\mathsf{Int}, \mathsf{Bool}, \ldots$; we also admit usual structured data types like arrays.
- A set of *function names* $\mathsf{f}, \mathsf{g}, \ldots$ representing the operations available in c. Function parameters $\mathsf{x}, \mathsf{y}, \ldots$ can be either data or participants variables.

An *assignment* $\mathsf{u} := \mathsf{e}$ updates the state variable u to a *pure* expression e which may contain function parameters or the lexeme $\mathsf{old}\ \mathsf{u}$; the latter denotes the value of the state variable u before the assignment.[2] We let $B, B', \ldots$ range over finite sets of assignments where each variable can be assigned at most once. For simplicity, the syntax of expressions is kept implicit; the reader can assume that the syntax of expressions is standard (but for the use of the $\mathsf{old}\ \_$ qualifier).

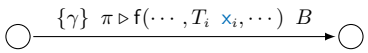
A *coordinator* c *on state variables* $\mathsf{u}_1, \ldots, \mathsf{u}_n$ is a finite-state machine "instantiated" by a participant p whose transitions are essentially of two kinds.[3] A transition like

$$\nu\ \mathsf{p}\colon \mathsf{R} \triangleright \mathsf{start}(\mathsf{c}, \cdots, T_i\ \mathsf{x}_i, \cdots)\ \{\cdots \mathsf{u}_j := \mathsf{e}_j \cdots\}$$

[2] We adapt the mechanism based on the old keyword from the Eiffel language [14] which, as explained in [13] is necessary to render assignments into logical formulae since e.g., $x = x + 1 \iff \mathsf{False}$.

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

allows participant p to create a fresh instance of coordinator c instantiating state variables u_j with expressions e_j on state variables and the parameters x_i. Transitions of the kind



are enabled when their *guard* γ, that is a boolean expression, holds true; in this case a *qualified participant* π defined by the following grammar:

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

can call f with parameters x_i which reassigns state variables as defined by the set of assignments B. Intuitively, a qualified participant ν p: R specifies that variable p represents a fresh participant with role R while any p: R qualifies p as an existing participant with role R.

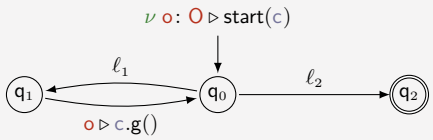
This model can be used to specify the conditions that enable the operations of a coordinator, how participants are expected to invoke the operations according to their role, and how successful invocation modify the state variables of the coordinator.

Exercise 1. *Give a contract for the SMP protocol in Section 1 resolving the ambiguities discussed there.*

3 A note on non-determinism

Sometimes determinism is required; for instance, smart contracts are usually required to behave deterministically [5]). Introducing a notion of deterministic coordinator requires some care as the labels of our automata use guards and symbolic expressions.

Exercise 2. *Consider the coordinator below*



and say if it is deterministic in each of the following cases:

- $\ell_1 = \ell_2 = \mathsf{o} \triangleright \mathsf{c}.\mathsf{g}()$
- $\ell_1 = \nu\ \mathsf{p}\colon \mathsf{R} \triangleright \mathsf{c}.\mathsf{g}()$ *and* $\ell_2 = \mathsf{any}\ \mathsf{p}\colon \mathsf{R} \triangleright \mathsf{c}.\mathsf{g}()$
- $\ell_1 = \{\mathsf{x} \leq 10\}\ \mathsf{o} \triangleright \mathsf{c}.\mathsf{g}(\mathsf{x} : \mathsf{Int})$ *and* $\ell_2 = \{\mathsf{x} > 10\}\ \mathsf{o} \triangleright \mathsf{c}.\mathsf{g}(\mathsf{x} : \mathsf{Int})$

We now introduce a binary relation instrumental to the definition of determinism. Let $\# \subseteq \mathcal{P} \times \mathcal{P}$ be the least symmetric relation such that

– $(\nu\ \mathsf{p}\colon \mathsf{R})\,\#\,\mathsf{p}'$,
– $(\nu\ \mathsf{p}\colon \mathsf{R})\,\#(\mathsf{any}\ \mathsf{p}'\colon \mathsf{O})$, and
– $\mathsf{R} \neq \mathsf{O} \implies (\mathsf{any}\ \mathsf{p}\colon \mathsf{R})\,\#(\mathsf{any}\ \mathsf{p}'\colon \mathsf{O})$.

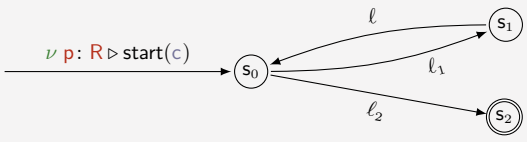
Intuitively, if $\pi_1 \ \# \ \pi_2$, then the participants in π_1 and π_2 *differ*. Indeed, the first two items just say that a new participant is necessarily different from an existing one. The third item says that two participants with different roles are necessarily different (since we require that every participant can have at most one role).

We now define *strong determinism* which basically ensures that different transitions calling the same function from a same participant have mutually exclusive guards. A coordinator is *(strongly) deterministic* if for all of its transitions $\mathtt{t}_1 \neq \mathtt{t}_2$ from the same source state and calling the same function we have:

$$(\mathtt{g}_1 \wedge \mathtt{g}_2) \implies (\pi_1 \,\#\, \pi_2)$$

where, for $i \in \{1, 2\}$, $\mathtt{g}_i$ is the guard of $\mathtt{t}_i$ and π_i is the qualified participant of $\mathtt{t}_i$.

Exercise 3. *Consider the coordinator*

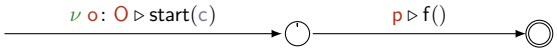


1. *Find two labels $\ell_1 = \ell_2$ such that the coordinator is deterministic.*
2. *Find two labels $\ell_1 \neq \ell_2$ that make the coordinator non-deterministic.*

4 Do all coordinators make sense?

In our model care is necessary to avoid specifying nonsensical coordinators. We consider a few cases that are evidently problematic.

Qualified participants of the form $\nu\ \mathsf{p}\colon \mathsf{R}$ and $\mathsf{any}\ \mathsf{p}\colon \mathsf{R}$, and parameter declarations of the form $\mathsf{p} : \mathsf{R}$ act as binders. Therefore, we should rule out coordinators like (1) below

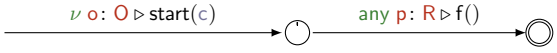


(1)

since p is a free occurrence which cannot denote any of the participants of the protocol. This can be simply attained by imposing the following property.

Name freeness: for each path of a coordinator, all free occurrences of participant variables should be after a quantifier.

Another problem arises when the role of a qualified participant is empty; consider



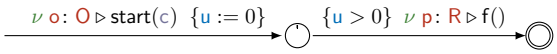
(2)

Arguably, the coordinator (2) is ill-formed since R is necessarily empty in s_0 and therefore the execution gets stuck in the initial state since no action is possible.

Role emptyness: all the roles in a path of a coordinator must appear in a transition with a freshly qualified participant.

We adopt a simple syntactical check that avoids the problem of empty roles. In fact, a sound and complete procedure for empty-roles detection subsumes reachability which, depending on the chosen expressivity of constraints and expressions, may be undecidable.

Progress can also be prevented in cases like the following



(3)

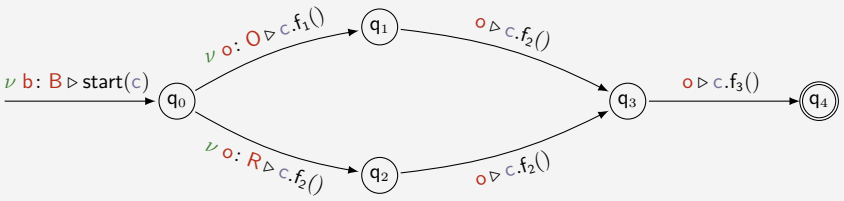
where the assignment in the start-transition of the coordinator (3) falsifies the guard of the transition from the initial state.

Consistency: the disjunction of the guards of the outgoing transitions of each state should not be a contradiction.

Similarly to empty roles, no-progress is undecidable in general. Our algorithmic verification checks that every transition t, regardless of the "history" of the current execution, leads to a state which is either accepting or it has at least a transition enabled. This is intuitively accomplished by checking that the guard of t, after being updated according to the assignments of t, implies the disjunction of the guards of the outgoing transitions from the target state of t.

Finally, we restrict to *well-formed* coordinators, that is coordinators that are deterministic closed, empty-role free, and consistent.

Exercise 4. *Is the coordinator*



well-formed? Justify your answer.

Our notion of well-formedness is not complete as shown by reflecting on the solution of the following exercise.

Exercise 5. *Explain why the following coordinator*

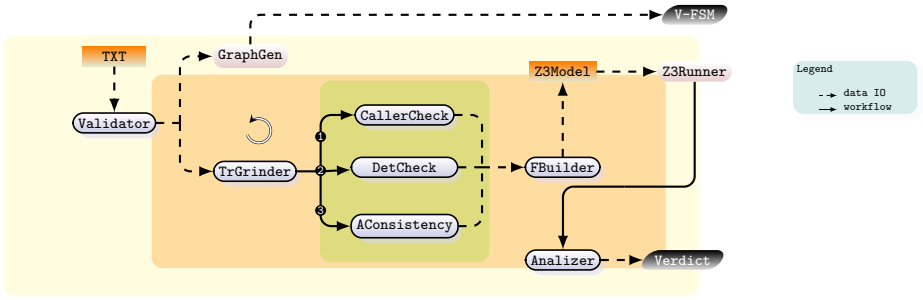


Fig. 2: Architecture of TRAC

is not well-formed.

5 Checking well-formedness

Fig. 2 borrows from [1] the architecture of our *t*ool for *r*esource-*a*ware *c*oordination (TRAC). The architecture of TRAC consists of two principal modules: one for parsing and visualisation (yellow box) and a module for the core functionalities (orange box). The latter module implements well-formedness check (green box).

The `Validator` transforms a textual representation of the coordinator in the `V-FSM` format suitable for visualisation through `GraphGen` or in an internal format suitable for our analysis.

The component `TrGrinder` relays each transition of the coordinator in input to the components in the green box that perform the verification of well-formedness according to Section 2; more precisely:

- `CallerCheck` (arrow ❶) checks for closedness and role emptyness;
- `DetCheck` (arrow ❷) builds a Z3 formula equivalent to strong determinism;
- `AConsistency` (arrow ❸) generates a Z3 formula equivalent to consistency.

These three outputs form the conjuncts of another Z3 formula built by the component `FBuilder` which finally yields a `Z3Model` passed to the `Z3Runner` component. The verification process ends with the `Analizer` component that diagnoses the output of Z3 and produces a `Verdict` which reports (if any) the violations of well-formedness of the coordinatorin input.

We now give some details on TRACinstrumental for this tutorial; further details can be found in [2] while the installation instructions are at [12].

To invoke TRACone has to prepare a `.trac` text-file to pass to the `Validator`. The content of the file is a sugared lists of transitions of a coordinator preceded by initial assignments of its state variables and a guard. The general form of a coordinator is given on the left below according to the DSL whose syntax is given by the grammar on the right below (where non-terminal symbols are in angled brackets):

```
dafsm c(⟨dcls⟩) by p : R {
    ⋮
    ⟨dcl⟩ :=e ;
    ⋮
    if γ
}
⋮
⟨str⟩ ⟨lbl⟩ ⟨str⟩ ;
⋮
```

```
⟨dcls⟩ ::= ε | ⟨dcl⟩(,⟨dcl⟩)*
⟨dcl⟩ ::= ⟨str⟩ ⟨str⟩

⟨lbl⟩ ::= {γ} ⟨qlf⟩ > ⟨str⟩(⟨dcls⟩) {⟨asg⟩}
⟨qlf⟩ ::= new p : R | any p : R | p
⟨asg⟩ ::= ⟨str⟩:=⟨expr⟩
⟨asgs⟩ ::= ε | ⟨asg⟩(;⟨asg⟩)*
```

Exercise 6. *Edit a* `.trac` *file for the SMP protocol.*
You can inspect the coordinator you provide using `GraphGen`*.*

Solution.


todo


References

1. João Afonso, Elvis Konjoh Selabi, Maurizio Murgia, António Ravara, and Emilio Tuosto. TRAC: A tool for data-aware coordination - (with an application to smart contracts). In Ilaria Castellani and Francesco 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. Joao Afonso, Elvis Konjoh Selabi, Maurizio Murgia, Emilio Tuosto, and Antonio Ravara. Artefact submission for paper #8 of COORDINATION 2024, April 2024.
3. Marco Autili, Paola Inverardi, and Massimo Tivoli. Automated synthesis of service choreographies. *IEEE Softw.*, 32(1):50–57, 2015.
4. Jonas Bonér. *Reactive Microsystems - The Evolution Of Microservices At Scale*. O'Reilly, 2018.
5. Vitalik Buterin. Ethereum: a next generation smart contract and decentralized application platform. https://ethereum.org/whitepaper, 2014.
6. Leonardo Frittelli, Facundo Maldonado, C. Hernán Melgratti, and Emilio Tuosto. A Choreography-Driven Approach to APIs: the OpenDXL Case Study. In *COORDINATION*, volume 12134 of *LNCS*. Springer, June 2020.
7. Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.
8. P. Brinch Hansen. *Operating System Principles*. Prentice-Hall, 1973.
9. Per Brinch Hansen. Monitors and concurrent pascal: a personal history. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 1–35, New York, NY, USA, 1993. Association for Computing Machinery.

10. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *JACM*, 63(1):9:1–9:67, 2016. Extended version of a paper presented at POPL08.
11. Nickolas Kavantzas, Davide Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217. Working Draft 17 December 2004.
12. Elvis Gerardin Konjoh Selabi. TRAC: a tool for data-aware coordination. Available at https://github.com/loctet/TRAC, 2024.
13. Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990.
14. Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.
15. Microsoft. The blockchain workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench, 2019.
16. 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.
17. Object Management Group. Business Process Model and Notation. http://www.bpmn.org.
18. Yoann Pigné, Antoine Dutot, Frédéric Guinand, and Damien Olivier. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs, 2008.
19. Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Comput. Surv.*, 54(7), July 2021.