TRAC: a tool for data-aware coordination (with an application to smart contracts)

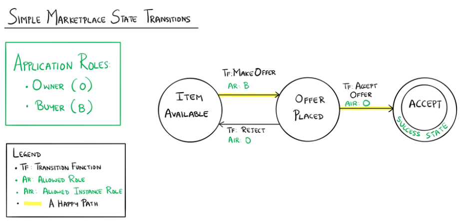
João Afonso[1], Elvis Konjoh Selabi[2,3], Maurizio Murgia[3], António Ravara[1], and Emilio Tuosto[3]

[1] NOVA School of Science and Technology
[2] Università di Camerino
[3] Gran Sasso Science Institute

Abstract. We propose TRAC, a tool for the specification and verification of coordinated multiparty distributed systems. Relying on finite-state machines (FSMs) where transition labels look like Hoare triples, TRAC can specify the coordination of the participants of a distributed protocol for instance an execution model akin blockchain smart contracts (SCs). In fact, the transitions of our FSMs yield guards, and assignments over data variables, and with participants binders. The latter allow us to model scenarios with an unbounded number of participants which can vary at run-time. We introduce a notion of *well-formedness* to rule out meaningless or problematic specifications. This notion is verified with TRAC and demonstrated on several case studies borrowed from the smart contracts domain. Then, we evaluate the performance of TRAC using a set of randomised examples, studying the correlations between the features supported and the time taken to decide well-formedness.

1 Introduction

We propose TRAC, a tool to support the coordination of distributed applications. The design of TRAC is inspired by the Azure initiative of Microsoft [29] which advocates the use of finite-state machines (FMSs) to specify the coordination of smart contract (SC for short). This idea is not formalised; in fact, Azure's FSMs are informal sketches aiming to capture the "correct" executions of SCs. For instance, the FSM for the simple market place (SMP) scenario borrowed from [30] (the textual description is ours):



The sketch declares the roles (Owner and Buyer) played by participants.
In the initial state `Item Available` the buyer is allowed to make an offer, moving the protocol to the `Offer Placed` state where two options are possible: the owner either accepts the offer (making the protocol reache the success state `Accept`) or rejects the offer (moving back the protocol to `Item Available`).
The labels of the transitions specify which role executes with operations to make the protocol progress.

The FSM informally specifies a protocol coordinating the participants enacting the roles owner and buyer, from a *global* standpoint; we call *coordination protocol* such specification. A coordination protocol can be regarded as *global view* –in the sense of choreographies [21,24]– where the state of the protocol determines which operations are enabled. This resembles the execution model of monitors [23]. In fact, as in monitors, coordination protocols encapsulate a state that –through an API– concurrent processes can have exclusive access to. The API is basically a set of operations guarded by conditions set to maintain an invariant on the encapsulated state (in the SMP scenario the operations are `MakeOffer`, `AcceptOffer`, and `Reject`). The key differences between coordination protocols and monitors [23] is that in the former (*i*) participants are distributed and do not share memory, (*ii*) the invocation of an operation whose guards is not valid in the current state is simply ignored without preempting the caller, and therefore (*iii*) processes do not have to be awaken.

We aim to refine the approach of Azure so to enable algorithmic verification of relevant properties of data-aware coordination of protocols. In fact, as for monitors, the interplay among the operations that modify the state and the guards in the API can lead to unexpected behaviours when informal specifications are used. We illustrate this problem with some examples on the SMP example.

1. The sketch of SMP does not clarify if a participant can play more roles simultaneously; for instance, it is not clear if an owner must be a different instance than buyers.
2. The labels distinguish roles and instances (`AR` and `AIR`): in fact, it is assumed that there can be many instances of a same role. Scope and quantification of roles is not clear; for instance, a requirement specified in [30] reads "The transitions between the `Item Available` and the `Offer Placed` states can continue until the owner is satisfied with the offer made." This sentence does not clarify if, after a rejection, the new offer can be made by a new buyer or it must be the original one;
3. The sketch specify neither the conditions enabling operations in a given state nor how operations change the state of the contract's variables; should the price of the item remain unchanged when the owner invokes the `Reject`?

Contributions & Structure Section 2 introduces *data-aware FSMs* (DAFSMs) to formalise coordination protocols. Roughly, DAFSMs allow specifications (*i*) to express conditions on how operations affect the state of the protocol and (*ii*) to explicitly declare the capabilities of participants. We propose *well-formedness* condition on DAFSMs to rule out erroneous coordination protocols.

The definition of DAFSMs is instrumental to our main contribution which is TRAC, a tool realising our model described in Section 3. We build on an initial proposal developed in [1].

The applicability of TRAC is demonstrated by showing how its features can specify and verify the SCs in [29]. Moreover, we discuss the performances of the TRAC with an experimental evaluation (cf. Section 4). The source code of TRAC and our experimental data is available at `https://github.com/loctet/TRAC`.

Related work and conclusions are given respectively in Sections 5 and 6.

2 Data-aware FSMs

In our model, protocols' *participants* cooperate through a *coordinator* according to their *role*. We let $\mathsf{p}, \mathsf{p}', \ldots$ denote *participant variables*, $\mathsf{R}, \mathsf{R}', \ldots$ denote *roles*, and $\mathsf{c}, \mathsf{c}', \ldots$ denote $\mathcal{C}$ *coordinator names*. Each coordinator name c has:

- A finite set $\mathcal{V}_{\mathsf{c}}$ of *data variables*; we let $\mathsf{c.x}, \mathsf{c.y}, \ldots$ range over $\mathcal{V}_{\mathsf{c}}$ and write $\mathsf{x}, \mathsf{y}, \ldots$ when the coordinator name is clear from the context. Each variables has an associated data type, e.g., Int, Bool, . . .; we also allow usual structured data types like arrays.
- A set of *function names*, ranged over by $\mathsf{c.f}, \mathsf{c.f}', \ldots$. Function parameters, ranged over by $\mathsf{x}, \mathsf{y}, \ldots$, can be either data or participants variables; we allow function calls with different parameters to a same function.

An *assignment* takes the form $\mathsf{c.x} := \mathsf{e}$, where e is an expression;[4] the set $\mathcal{B}$ of assignments is ranged over by β while $B, B', \ldots$ range over finite subsets of β where each variable can be assigned at most once; moreover, we assume that all assignments in B are executed simultaneously. In an assignment $\mathsf{c.x} := \mathsf{e}$ data variables occurring in e must have the old qualifier to refer to the value of $\mathsf{c.x}$ before the assignment. The set of *guards* $\mathcal{G}$, ranged over by $\mathsf{g}, \mathsf{g}', \ldots$, consists of constraints (i.e., boolean expressions) over data variables and function parameters. Parameter *declarations* are written as $\mathsf{x} : T$ or $\mathsf{p} : \mathsf{R}$ to respectively assign data type T to x and role R to p; we let $\mathcal{D}$ be the set of all declarations and d to range over $\mathcal{D}$. Lists of declarations are denoted by $\mathbf{d}$ with the implicit assumption that the parameters in $\mathbf{d}$ are pairwise distinct.

The set $\mathcal{P}$ of *qualified participants* consists of the terms generated by

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

where both ν and @ are binders. Intuitively, $\nu\ \mathsf{p} \colon \mathsf{R}$ specifies that variable p represents a fresh participant with role R while $\mathsf{any}\ \mathsf{p} \colon \mathsf{R}$ qualifies p as an existing participant with role R. With p we refer to a participant in the scope of a binder.

Before its formal definition (cf. Definition 1), we give an intuitive account of our model. We use FSMs as coordination protocols with a single coordinator c. The transitions of an FSM represent the call to functions exposed by the coordinator c performed by participants. Such calls may update the current control state (by means of state transitions) and the state of data variables (by mean of assignments). Access to functions can be restricted to some participants (using participants variables and modifiers), and the availability of a function may depend on the current control or data states (using guards). A protocol starts in the initial state of the FSM specifying where the initial state of variables is set by the creator of the coordinator; intuitively, the creator may be thought of as an object in object-oriented programming created by invoking a constructor.

[4] We borrow from Z3 a substantial subset of expressions over variables and parameters (barred participants parameters) whose syntax is standard and therefore omitted. We assume that expressions do not have side effects.

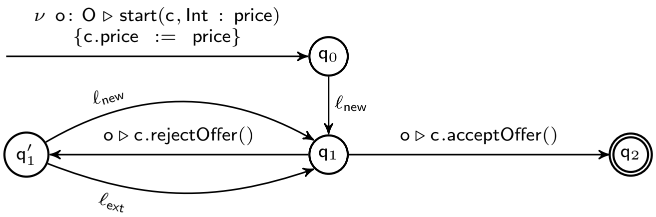
Definition 1. *Let $2^{\mathcal{B}}_{\mathsf{fin}}$ be the set of all finite subsets of $\mathcal{B}$ and $\mathcal{L} = \mathcal{G} \times \mathcal{P} \times \mathcal{F} \times \boldsymbol{\mathcal{D}} \times 2^{\mathcal{B}}_{\mathsf{fin}}$ be the set of* labels*, ranged over by ℓ. A* data-aware finite state machine *(DAFSM for short) is a tuple $\mathcal{S} = (\mathsf{S}, \mathsf{q}_0, \rightarrow, \mathsf{F}, \mathsf{c}, \nu\ \mathsf{p} \colon \mathsf{R}, \mathbf{d_0}, B_0)$ where:*

- $(\mathsf{S}, \mathsf{q}_0, \rightarrow, \mathsf{F})$ *is an FSM over $\mathcal{L}$ (namely, S is finite set of* states*, $\mathsf{q}_0 \in \mathsf{S}$ is the* initial state*, $\rightarrow \subseteq \mathsf{S} \times \mathcal{L} \times \mathsf{S}$, and $\mathsf{F} \subseteq \mathsf{S}$ is the set of* accepting *states);*
- $\mathsf{c} \in \mathcal{C}$ *is the coordinator name;*
- *for each transition label $(\mathsf{g}, \pi, \mathsf{f}, \mathbf{d}, B)$, if $\mathsf{c.x} := \mathsf{e} \in B$ then every data parameter occurring in e occurs in $\mathbf{d}$, e is well typed, and the data variables occurring in the guards of any of the transitions of $\mathcal{S}$ belong to $\mathcal{V}_{\mathsf{c}}$;*
- $\nu\ \mathsf{p} \colon \mathsf{R}$ *binds p to the participant creating the coordinator;*
- $\mathbf{d_0} \subseteq \boldsymbol{\mathcal{D}}$ *is the parameters list of the coordinator;*
- $B_0 \subseteq_{\mathsf{fin}} \mathcal{B}$ *is a set of assignments (setting the initial values of the state variables).*

A path *is a finite sequence of transitions $\mathsf{s}_0 \xrightarrow{\ell_1} \mathsf{s}_1 \cdots \mathsf{s}_n \xrightarrow{\ell_n} \mathsf{s}_{n+1}$ with $\mathsf{s}_0 = \mathsf{q}_0$.*

The next example introduces a convenient graphical notation for DAFSMs in which guards on transitions are in curly brackets for readability; this notation is reminiscent of Hoare triples (guards are not to be confused with sets).

Example 1. Let $\ell_{\text{new}} = \{\mathsf{offer} > 0\}\ \nu\ \mathsf{b} \colon \mathsf{B} \triangleright \mathsf{c.makeOffer}(\mathsf{Int} : \mathsf{offer})\ \{\mathsf{c.offer} := \mathsf{offer}\}$ and $\ell_{\text{ext}} = \{\mathsf{offer} > 0\}\ @\ \mathsf{b} \colon \mathsf{B} \triangleright \mathsf{c.makeOffer}(\mathsf{Int} : \mathsf{offer})\ \{\mathsf{c.offer} := \mathsf{offer}\}$. The DAFSMs below represents the SMP protocol of Section 1.



The initial state is q_0 and it is graphically represented by the source-less arrow entering it. The label[5] of this arrow represents the invocation from a new participant o with the owner's role O to the constructor for a coordinator c with a parameter price of type Int. The set of assignments is the singleton initialising the coordinator's variable $\mathsf{c.price}$ to price.

In q_0, the only enabled function is $\mathsf{c.makeOffer}(\mathsf{Int} : \mathsf{offer})$; the first buyer b invoking this function with a parameter offer satisfying the guard $\mathsf{offer} > 0$ moves the protocol to state q_1 while recording the new offer in the coordinator state with the assignment $\mathsf{c.offer} := \mathsf{offer}$. Contextually, the state of the coordinator records that the caller b plays role B.

From state q_1 only the owner o can make the protocol progress by either accepting or rejecting the offer. In the former case, the protocol reaches the

[5] We may omit writing guards when they are True and assignments when they are empty as in the transitions from q_1.

accepting state q_2 (graphically denoted with a doubly-circled node); in the latter case, the protocol reaches state q'_1 where either an existing buyer or a new one can make further offers. $\diamond$

Notably, the DAFSM of Example 1 refines the informal one in Section 1 by more precisely specifying that offers can arrive either from previous buyers or new ones (cf. item 2 in Section 1).

2.1 Well-formedness of DAFSM

The restrictions in Definition 1 concern single transitions; however, DAFSMs can model meaningless and wrong behaviours, due to conditions spanning several transitions, e.g., free occurrences of participant variables, lack of participants of a role or inconsistent guards. Below we spell-out those constraints after motivating them with simple examples.

A first issue is the presence of free occurrences of participants names.

Example 2. The DAFSM $\xrightarrow{\nu\ \mathsf{o}\colon \mathsf{O} \triangleright \mathsf{start}(\mathsf{c})} \mathsf{s}_0 \xrightarrow{\mathsf{p} \triangleright \mathsf{c.f}()} \mathsf{s}_1$ is syntactically erroneous since the participant variable p is not bound. $\diamond$

In our model 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. In a DAFSM all occurrences of participant variable should be in the scope of a binder to be meaningful. Formally, we say that a transition $(\mathsf{s}_1, \mathsf{g}, \pi, \mathsf{c.f}, \mathbf{d}, \beta, \mathsf{s}_2)$ binds p iff:

$$\exists \mathsf{R} : \pi = \nu\ \mathsf{p}\colon \mathsf{R} \quad \vee \quad \pi = \mathsf{any}\ \mathsf{p}\colon \mathsf{R} \quad \vee \quad \mathsf{p} : \mathsf{R} \in \mathbf{d}$$

The occurrence of p in a path $\sigma = \sigma_1(\mathsf{s}_1, \mathsf{g}, \mathsf{p}, \mathsf{c.f}, \mathbf{d}, \beta, \mathsf{s}_2)\sigma_2$ is *bound in* σ if there is a transition in σ_1 binding p and it is *bound in a DAFSM* $\mathcal{S}$ if all the paths of $\mathcal{S}$ including the occurrence binding it. Finally, $\mathcal{S}$ is *closed* if all occurrences of participant variables are bound in $\mathcal{S}$.

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

Example 3. If we bind the occurrence of p in the DAFSM of Example 2 with the binder @, we obtain the closed DAFSM

$$\mathcal{S}_2 = \quad \xrightarrow{\nu\ \mathsf{o}\colon \mathsf{O} \triangleright \mathsf{start}(\mathsf{c})} \mathsf{s}_0 \xrightarrow{\mathsf{any}\ \mathsf{p}\colon \mathsf{R} \triangleright \mathsf{c.f}()} \mathsf{s}_1$$

However, we argue that $\mathcal{S}_2$ is ill-formed since R is necessarily empty in s_0. Hence no action is possible, and the execution gets stuck in the initial state. $\diamond$

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

A binder *expands* role R if it is a qualified participant of the form $\nu\, \mathsf{p} \colon \mathsf{R}$ or a parameter declaration of the form $\mathsf{p} : \mathsf{R}$. A role R is *expanded* in a path σ iff:

$$\sigma = \sigma_1(\mathsf{s}_1, \mathsf{g}, @\, \mathsf{p} \colon \mathsf{R}, \mathsf{c.f}, \mathbf{d}, \beta, \mathsf{s}_2)\sigma_2 \implies \exists \mathsf{t} \in \sigma_1 : \mathsf{t} \text{ expands } \mathsf{R}$$

A DAFSM $\mathcal{S}$ *expands* a role R if every path of $\mathcal{S}$ expands R. Finally, $\mathcal{S}$ is (*strongly*) *empty-role free* if $\mathcal{S}$ expands every role in $\mathcal{S}$.

Despite the quantification over the possibly infinite set of all paths, empty-role freedom can be decided by considering only *acyclic* paths, that is paths which contain at most one occurrence of each state. Clearly, there are only finitely many acyclic paths. Notice that $\mathcal{S}_2$ above is not empty-role free.

Finally, progress can be jeopardised if assignments falsify all the guards of the subsequent transitions.

Example 4. The DAFSM $\mathcal{S}_3$ below is both closed and empty-role free, as the caller of $\mathsf{c.f}$ is o which is bound by the constructor, and there are no @ modifiers.

$$\mathcal{S}_3 = \xrightarrow{\nu\, \mathsf{o} \colon \mathsf{O} \triangleright \mathsf{start(c)}\ \{\mathsf{c.x} := 0\}} (\mathsf{s}_0) \xrightarrow{\{\mathsf{c.x} > 0\}\ \mathsf{o} \triangleright \mathsf{c.f()}} ((\mathsf{s}_1))$$

Crucially, $\mathsf{c.x} > 0$ will never be satisfied at run-time because $\mathsf{c.x}$ is initialised to 0 and never changed. So again every execution gets stuck in state s_0. ⋄

Similarly to empty roles, detecting inconsistencies is undecidable at least for expressive enough constraints and expressions. We therefore devise a syntactic technique amenable of algorithmic verification. The idea is to check 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. Before formally introduce our notion of consistency, we need a few auxiliary definitions.

Definition 2. *For all states* s*, we define the* progress constraint $\mathsf{g_s}$ *as* True *when* s *is accepting, and as the disjunction of guards of the outgoing transitions of* s*. Let* $\mathsf{c.x} \notin B$ *mean that for all* $\mathsf{c.y} := \mathsf{e} \in B$*,* $\mathsf{c.y}$ *and* $\mathsf{c.x}$ *differ and the expression* e *is not* $\mathsf{c.x}$*. The* progress constraint *of an assignment* B *is*

$$\mathsf{g}_B = \bigwedge_{(\mathsf{c.x}:=\mathsf{e})\in B} \mathsf{c.x} = \mathsf{e} \wedge \bigwedge_{\mathsf{c.x}\notin B} \mathsf{c.x} = \mathsf{old\ c.x}$$

We define $\mathsf{dataParams}(\mathbf{d})$ *as the list of data parameter names occurring in* $\mathbf{d}$*.*

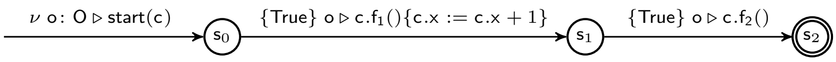
We can now define our notion of consistency.

Definition 3. *Let* $\mathsf{g}\{\mathbf{y}/\mathbf{x}\}$ *be the guard obtained from* g *after the simultaneous substitution of variables* $\mathbf{x}$ *with* $\mathbf{y}$*. A transition* $\mathsf{t} = (\mathsf{s}, \mathsf{g}, \pi, \mathsf{c.f}, \mathbf{d}, \beta, \mathsf{s}')$ *is* consistent *if:*

$$\forall \mathbf{c.x}, \mathbf{old\ c.x} : \exists \mathsf{dataParams}(\mathbf{d}) : (\mathsf{g}\{\mathbf{old\ c.x}/\mathbf{c.x}\} \wedge \mathsf{g}_B) \implies \mathsf{g}_{\mathsf{s}'}$$

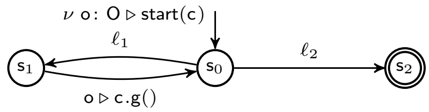
A DAFSM $\mathcal{S}$ *is* consistent *if so is every transition of* $\mathcal{S}$*.*

Example 5. The DAFSM $\mathcal{S}_4$ below shows the importance of renaming variable with old. The `AConsistency` formula of $\mathcal{S}_4$ for the transition from s_0 to s_1 is $\forall \mathsf{c.x} : \mathit{True} \;\; \& \;\; \mathsf{c.x} = \mathsf{c.x} + 1 \Rightarrow \mathit{True}$. The latter formula is evaluated as $\mathit{False} \implies \mathit{True}$ which is True. We don't want this inconsistency case, therefore, by replacement, the `AConsistency` formula of $\mathcal{S}_4$ becomes $\forall \mathsf{c.x}, \mathsf{c.x_{old}} : \mathit{True} \;\; \& \;\; \mathsf{c.x} = \mathsf{c.x_{old}} + 1 \Rightarrow \mathit{True}$.

$\mathcal{S}_4 =$ 

$\diamond$

Non-determinism could be useful for some applications, most of the time *determinism* is a desirable property (e.g., SCs are usually required to be deterministic [7]). Before the formal definition, we give a few examples illustrating how non-determinism may arise in DAFSMs.

Example 6. The DAFSM $\mathcal{S}$ = 

is deterministic or not, depending on the labels ℓ_1 and ℓ_2. Let us consider some cases.

$\ell_1 = \ell_2 = \mathsf{o} \triangleright \mathsf{c.g()}$ $\mathcal{S}$ is non-deterministic because a call to function $\mathsf{c.f}$ by o can lead either to s_1 or to s_2.

$\ell_1 = \nu\, \mathsf{p} \colon \mathsf{R} \triangleright \mathsf{c.g()}$ **and** $\ell_2 = @\, \mathsf{p} \colon \mathsf{R} \triangleright \mathsf{c.g()}$ $\mathcal{S}$ is deterministic intuitively because the next state is unambiguously determined by the caller of $\mathsf{c.g}$: the protocol moves to s_1 or s_2 depending whether the call is performed by an existing or a new participant.

$\ell_1 = \{\mathsf{x} \le 10\}\; \mathsf{o} \triangleright \mathsf{c.g(x : Int)}$ **and** $\ell_2 = \{\mathsf{x} > 10\}\; \mathsf{o} \triangleright \mathsf{c.g(x : Int)}$ $\mathcal{S}$ is deterministic because guard $\mathsf{x} \le 10$ leading to s_1 and guard $\mathsf{x} > 10$ leading to s_2 are disjoint; therefore the next state is determined by the value of the parameter x, and every value enables at most one transition.

Also, taking ℓ_1 as in the latter case and $\ell_2 = \{\mathsf{x} \ge 10\}\; \mathsf{o} \triangleright \mathsf{c.g(x : Int)}$ would 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 of $\mathsf{c.g}$. $\diamond$

We now define a notion of *strong determinism*, which is decidable and can be efficiently established. To this aim, we first define the binary relation $\# \subseteq \mathcal{P} \times \mathcal{P}$ as the least symmetric relation satisfying:

$$\nu\, \mathsf{p} \colon \mathsf{R} \;\#\; \mathsf{p}', \quad \nu\, \mathsf{p} \colon \mathsf{R} \;\#\; @\, \mathsf{p}' \colon \mathsf{O}, \quad \text{and} \quad \mathsf{R} \neq \mathsf{O} \implies @\, \mathsf{p} \colon \mathsf{R} \;\#\; @\, \mathsf{p}' \colon \mathsf{O}$$

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

We now define strong determinism.

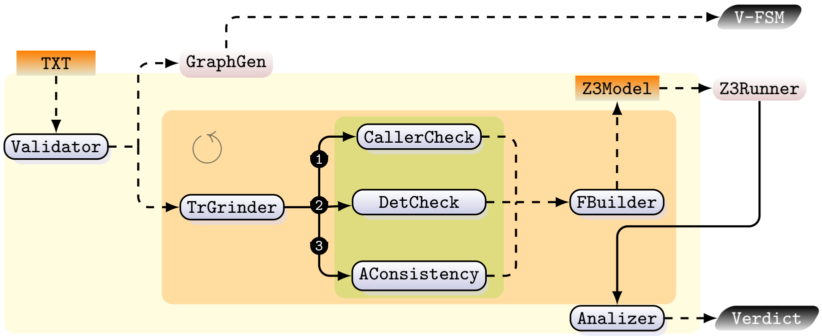


Fig. 1: The architecture of TRAC

Definition 4. *A DAFSM $\mathcal{S}$ is* (strongly) deterministic *if, for all transitions* $\mathtt{t}_1 \neq \mathtt{t}_2$ *in* $\mathcal{S}$ *such that* $\mathtt{t}_1$ *and* $\mathtt{t}_2$ *have the same source state and the same function then:*

$$(\mathsf{g}_1 \wedge \mathsf{g}_2 \implies \mathsf{False}) \quad \vee \quad \pi_1 \# \pi_2$$

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

A DAFSM is *well-formed* when empty-role free, consistent, and deterministic.

3 The Tool

We implement our model in TRAC. Specifically, TRAC renders DAFSMs in terms of a DSL to specify DAFSMs and verify the well-formedness condition defined in Section 2 relying on the SMT solver Z3. We present the architecture of TRAC in Section 3.1 and some implementation details in Section 3.2.

3.1 Architecture

Fig. 1 represents the architecture of TRAC which, for convenience, is compartmentalised into two principal modules: DAFSM 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 flow starts `Validator` performing basic syntactic checks on a textual representation[6] of DAFSMs and transforming the input in a format that simplifies the analysis of the following phases. Specifically, the output of `Validator` can be passed (*i*) to `GraphGen`, a component yielding a visual representation of DAFSMs (`V-FSM` output) and (*ii*) to the "transitions Grinder" `TrGrinder` component (orange box) for well-formedness checking.

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

[6] Our DSL is immaterial here; it is described in the accompanying artefact submission.

– `CallerCheck` (arrow ❶) that returns a boolean which is true if, and only if, the DAFSM is closed and strongly empty-role free;
– `DetCheck` (arrow ❷) that builds a Z3 formula which is true if, and only if, the state is strongly deterministic;
– `AConsistency` (arrow ❸) to generate a Z3 formula which holds if, and only if, the transtion is consistent.

The component `FBuilder` computes the conjunction of the output of the components above, yelding a `Z3Model`, which is then executed by the `Z3Runner`.

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 DAFSM in input.

3.2 Implementation

We now give some implementation details on the main features of TRAC; we first consider each component of TRAC's architecture.

The `Validator` processes the input which essentially lists transitions of a DAFSM expressed in the format of our DSL. For instance, the transition to make offers of Example 1 is rendered in our DSL as

```
S0 {_offer > 0} b:B > c.makeOffer(int _offer) {offer := _offer} S1
```

Basically, `Validator` reads each transition in the file and extract participants, actions, states, preconditions, assignments and input parameters of the action. To inspect the DAFSM in input TRAC relies on `GraphGen`[7] which creates a visual representation of graphs.

Component `TrGrinder` transforms the DAFSM obtained by `Validator` in an internal format suitable for the analysis. Next, `TrGrinder` iterates on the transitions, invokes different checker component by supplying them with the necessary data.

The first component invoked by `TrGrinder` is `CallerCheck` which takes in input a transition `t` and the (internal representation of the) DAFSM. If the caller of `t` is of the form p or $\mathsf{any}\ \mathsf{p}: \mathsf{R}$, `CallerCheck` retrieves all acyclic paths[8] that, from the initial state, lead to `t`'s source state, and then checks that every such path contains $\nu\ \mathsf{p}: \mathsf{R}$ or $\mathsf{any}\ \mathsf{p}: \mathsf{R}$ for some R (if the caller was p), or contains $\nu\ \mathsf{p}: \mathsf{R}$ (if the caller was $\mathsf{any}\ \mathsf{p}: \mathsf{R}$). As soon as a path violates that condition, `CallerCheck` halts returning False otherwise True is returned. To avoid checking again a same path, the formula is saved and just retrieved when transitions with same same source and caller as `t` are considered.

The component `DetCheck` takes as inputs a transition `t` and the list of transitions with source the target `t`. The list is partitioned by grouping transitions

[7] `GraphGen` is a wrap component that uses GraphStream [32] to generate the visual FSM (`V-FSM`).

[8] Crucially, the internal format produced by `TrGrinder` is instrumental for extracting acyclic paths using the `networkx` library [22].

with the same function name and callers not related by # (cf. Section 2). For non-singleton partitions, `DetCheck` builds a Z3 formula which is true if, and only if, whenever the guard of a transition is true the others are false. Let $\mathcal{T}$ be the set of all transitions and $\mathcal{F}$ be the partition of $\mathcal{T}$ as described above. The formula returned by `DetCheck` is the Z3 correspondent of $\Phi_{\texttt{DetCheck}} = \bigwedge_{F \in \mathcal{F}} \Phi(F)$ where, assuming that $\mathsf{g_t}$ is the guard of a transition `t`, we set

$$\Phi(F) = \bigwedge_{t \in F} \left(\mathsf{g}_t \implies \bigwedge_{t' \in F,\ t' \neq t} \neg \mathsf{g}_{t'} \right)$$

Double checking is avoided by keeping track of checked states.

Component `AConsistency` implements Definition 3. Using the formula of the formal definition "as is" however would be inefficient, because of the presence of universal quantification and many unnecessary variables and equations (those of the form $\mathsf{c.x} = \mathsf{old}\ \mathsf{c.x}$). Universal quantification, as usual with SMT solvers, is dealt with by just removing quantifiers and negating the formula. The result of the checker will be negated again at the end. Unnecessary equations are removed as follows. Given a transition `t` and a list of outgoing transitions from its source, `AConsistency` scrutinises the pre-conditions and post-conditions for shared state variables. When a variable is used in both conditions, `AConsistency` rename the occurrences of the variable in the pre-conditions by adding the _old suffix. Likewise, the suffix _old is added to state variables x occurring in the right-hand side of an assignment of the post-condition if x is assigned in the post-conditions. Subsequently, the assignments in the post-conditions are transformed in a conjunction of equations representing the state update. Finally, `AConsistency` constructs a Z3 formula which ensures that given the pre-conditions and post-conditions bounded by input variables, at least one precondition of the outgoing transitions should be met.

From the outputs of each of the above components, `FBuilder` generates a single formula composed of the conjunction of all the formulae for each transition. After going through all transitions, `FBuilder` compiles all the generated Z3 formulas to build the `Z3Model`. After processing all transitions, `FBuilder` outputs a Python file containing the set of Z3 formulae, referred in Fig. 1 as the `Z3Model`. This model includes all the necessary libraries, variable declarations, and solver configurations to run the model and determine its satisfiability.

The component `Z3Runner` takes this Python file, executes it, and forwards the results to the `Analizer`. If the `Z3Model` is found to be satisfiable, it indicates that the DAFSM is well-formed; otherwise, it is deemed non-well-formed. This final output is the `Verdict`.

Finally, we remark that TRAC operates in two modes: a `non-stop mode`, which builds and evaluates the entire model (used for our experimental evaluation) and a `stop mode`, which halts immediately as soon as a violation of well-formedness if found.

Table 1: Features in the Azure benchmark

	ICI	BI	PP	RR	MPR
Hello Blockchain	⊖	✓	⊖	⊖	⊖
Bazaar	×	✓	⊖	⊖	⊖
Ping Pong	×	✓	⊖	⊖	⊖
Defective Component Counter	⊖	✓	✓	⊖	⊖
Frequent Flyer Rewards Calculator	⊖	✓	✓	⊖	⊖
Room Thermostat	⊖	⊖	✓	⊖	⊖
Simple Marketplace	⊖	✓	⊖	↗	⊖
Asset Transfer	⊖	✓	✓	↗	⊖
Basic Provenance	⊖	✓	✓	↗	⊖
Refrigerated Transport	⊖	✓	✓	↗	↗
Digital Locker	⊖	✓	✓	↗	↗

Legend
✓ : feature present in the example and TRAC successfully handles it
× : feature present but not supported by TRAC
⊖ : feature not present int to the example
↗ : feature present and TRAC supports it with some workarounds

4 Evaluation

We evaluate DAFSMs expressiveness and TRAC performance using two benchmarks. The first consists of the examples from the Azure BC workbench [29], showing how the DAFSMs (and the current version of TRAC) deals with simple, yet realistic, SCs also used in related work (e.g., [20]); the second contains randomly generated large examples to stress-test TRAC.

These examples exhibit a variety of features that are essential for the representation of SCs. We consider a significant range of features in our analysis, including inter-contracts interactions (ICI), joining of new participants by-invocation (BI) or by participant passing (PP), role revocation (RR), and the possibility for a participant to assume multiple roles (MPR). Our aim is to assess to what degree TRAC can model these features, present in illustrative expressive examples in the literature on SCs. Our findings are outlined in Table 1.[9] Notably, TRAC covers most of the features and the only limitation is that TRAC does not support yet inter-contracts interactions (ICI column). Notably, we could approximately model the examples with RR and MPR using some workarounds. In particular, in all the examples featuring RR revocation was performed on singleton roles, that is roles that can be played by at most one participant at a time. Moreover, every revocation is followed by a re-assignment of the role to a participant. We therefore modelled this situation using a participant variable for the role. So, $\nu\, \mathsf{p} \colon \mathsf{R}$ simultaneously assigns role to p, and revokes R from the previous participant holding it. This has the drawback that the role cannot be reassigned to a participant formerly holding it. For MPR, in the examples con-

[9] Commonplace features such as multiple participants and multiple roles are present in all the examples and supported by TRAC.

sidered the participant with multiple roles was at most one. We could therefore add explicit moves for that participant only to emulate it having two roles.

We now turn our attention to the performance of TRAC, using a benchmark of randomly generated DAFSMs. More precisely, we evaluated TRAC using a data set of 135 DAFSMs [10] randomly generated according to the following process.[11] Let $\mathsf{rand}(i, j)$ be a random number between number i and $j \geq i$ (we let $\mathsf{rand}(i) = \mathsf{rand}(1, i)$). We fix a maximal number of participants $p \in \mathsf{rand}(2, 10)$, of functions $f \in \mathsf{rand}(10, 20)$, and of data variables $v \in \mathsf{rand}(50)$. For each $s \in \{10, 20, 30\}$ and for each $s \leq t \leq 3s$ such that $t \mod 5 = 0$ we generate five DAFSMs, each having s states, by iterating the following steps until all nodes are connected and t transitions have been generated:

- create $\mathsf{rand}(2, 5)$ transitions with source the current state and randomly selected target nodes not connected yet (if any, otherwise the targets are selected randomly on the whole set of nodes);[12]
- for each of the transitions a qualified participants and an operation with a number of parameters are randomly selected according to $\mathsf{rand}(p)$, $\mathsf{rand}(0, f)$, and $\mathsf{rand}(0, v)$.

We measured the performance of `DetCheck`, `AConsistency`, and `CallerCheck` by averaging the running time over ten executions of each generated DAFSM. The experiments were conducted on a Dell XPS 8960, 13th Gen Intel Core (9-13900K) with 32 cores and 32GB RAM running Linux 6.5.0-17-generic (Ubuntu 23.10, 64bit). The results are reported in the following plots that we now discuss.

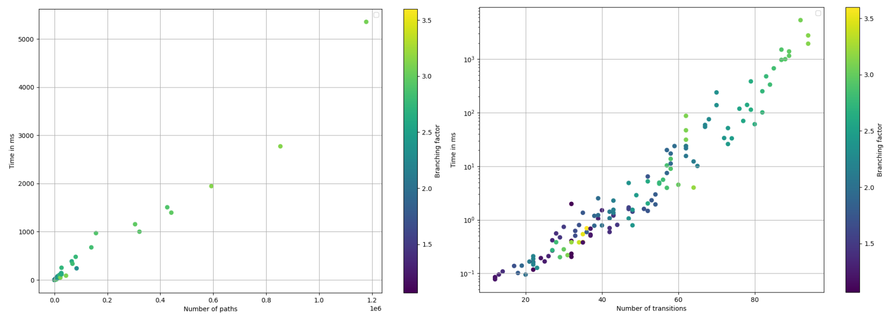


Fig. 2: `CallerCheck` time against of number paths (left) and transitions (right, y-axis in logaritmic scale)

[10] Number fixed to obtain graphs with a few dozens states but increasing number of transitions and qualified participants.

[11] The parameters (fixable as script inputs) were set in order to obtain sufficiently large DAFSMs (covering cases with millions of paths) while maintaining the execution time below one hour.

[12] We do not spell out the details of the random generation of guards and assignments – they are immaterial for the performance of `DetCheck`, `AConsistency`, and `FBuilder`.

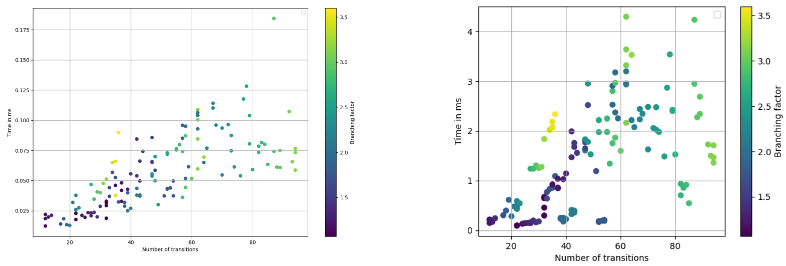


Fig. 3: `DetCheck` (left) `AConsistency` (right) time against number of transitions

The evaluation results presented in Fig. 2 shows the time taken for `CallerCheck` against the number of paths in the model: as the number of paths increases with the number of transitions of the FSM, the outcome confirms that times to check closedness and empty-role freeness is exponential in the number of transitions. Interestingly, even in the cases with more that 10^6 paths, `CallerCheck` terminates the analysis in less that 6 seconds.[13]

Fig. 3 shows the results of our analysis for `DetCheck` and `AConsistency`. The left plot in Fig. 3 hints that the execution time of `DetCheck` linearly grows with the number of transitions. Fig. 3 right shows that the time to run `AConsistency` is too low to allow any conclusion. Further analysis is require to correlated number of transitions and time.

These plots result from initial investigations of the performances of the main components of TRAC.[14] The complexity of checking well-formedness is dominated by `CallerCheck`, which is exponential in the number of transitions since `CallerCheck` has to check a path property. However that TRAC shows good performances even for experiments with high number of paths (cf. Fig. 2).

5 Related works

The literature on models of coordination is vast. We restrict our comparison to tool-supported approaches within three categories: FSM-based models, formal language models, and domain-specific languages for SCs. We compare DAFSMs with other coordination models as well as with approaches specific to SCs.

Coordination models of distributed systems based on extensions of FSMs with (fragments) of first-order logic have appeared in the literature. Notably

[13] To improve the readability of the left plot of Fig. 2 we did not include two DAFSMs whose number of paths was higher more than a factor of 20 that other instances. This is not necessary for plot on the right, which in fact includes all the generated DAFSMs, since there we use a logarithmic scale.

[14] Further, more systematic, experiments are needed to lead to broader conclusions.

data-aware version of BIP and REO have been studied in [15,34]. As in DAFSMs, data that can be accessed and modified as part of an interaction in both BIP and REO. A difference with our model is that interactions can involve more participants and updates are local to the participants of interactions. This also applies to recent models based on asserted communicating finite-state machines [33,35].

Choreography automata [2] and their extension with assertions [18] are global specifications for communicating systems behind Corinne [31] and CAScr [19]. Both these tools are designed to check well-formedness conditions different than ours (resp. those in [2] and in [18]) and neither of them supports multiple instances of roles. Assertions in CAScr are not guards; they express rely-guarantee conditions between the sender and the receiver of interactions. In the same vein, CAT [4] is an automata-based tool for the verification of communication protocols. Based on *contract automata* [3,5,6], CAT is not data-aware and its contracts purely regard the communication interface of participants (which are also fixed).

Protocol languages that advocates a programming style based on FSMs to specify SCs are FSolidM framework [27,28] and SmartScribble [16]. The former relies on model checking CTL formulae to verify safety and liveness properties (including deadlock-freedom). The automata have a global state, represented by contract, input, and output variables, and transitions are guarded by boolean conditions on these variables. The tool has been extended to feature code generation and interaction verification between multiple SCs [25]. This progress marked a substantial improvement in detecting common vulnerabilities such as re-entrancy attacks and fallback errors.

The interaction patterns that can be programmed with SmartScribble [16] correspond to FSMs. The tool extracts Plutus code[15] from valid protocol descriptions, leaving to the developer the task to fill in the application logic. The automatic generation of code (a feature we aim to) greatly accelerates developing time, and guarantee correct-by-construction code (in what concerns the interaction patterns).

Participants are first-class citizens in DAFSMs while FSolidM encodes them with variables and SmartScribble identifies participants with roles which are fixed statically. Also, SmartScribble does not support assertions.

An application of Event-B to SCs generating automatically Solidity code appeared in [36]. There is no report on the validation of the tool with benchmarks.

A parallel line of research explores the use of BC technology to audit choreographic programs [10,11]. Roughly, the idea is to generate Solidity contracts from models expressed as BPMN [21], so that the contracts' trace the execution of their choreography. Notably, [12] is an extension of [11] which allows multiple participants to play the same role. This line of work has fairly different goals than ours: its aim is to exploit BC immutability to record the execution in a secure way. Our approach instead concerns modelling and verifying distributed applications coordinated by a FSM, possibly implemented as an SC.

[15] Plutus is the programming language to develop SCs to the Cardano BC.

The previous tools take a "top-down approach" – propose an abstraction to (rigorously) define formal models of computations. In several cases, SCs code is automatically generated from (correct) specifications. The next proposals, Obsidian [9] and Stipula [13,14,26], embed the definition of the FSM in the contracts' programming language. Both are inspired by *typestates* [37] and their use in programming languages [17]: states are explicit entities with a defined API; invocations to an operation of the API of a state possibly update of the variables of the program and yield to (possibly) another state. In this respect, the execution model of both languages is quite similar to DAFSMs.

Obsidian uses typestates and linear types [38] to control "assets" (critical resources of SCs). Safe, yet flexible, aliasing is ensured with an ownership type system [8]. Two case studies established the usability of the language in "real-world" scenarios.

Stipula focuses on legal contracts and provides a strict discipline to guarantee *liquidity*: no asset remains frozen forever.

The Azure repository has been used as a benchmark in [20] where solidity code is annotated with assertions. There the Contractor toolkit extracts from the annotate code data-aware abstractions (akin to FSMs). Such abstraction can then be validate with respect to user-defined properties. The main strength of Contractor is the possibility to automatically construct sound models, while its main drawback is that it does not directly support multi object protocols.

6 Conclusions & Future works

This paper proposes DAFSMs, a data-aware coordination model for orchestrated computation applicable to the description of multiparty protocols. The key novelties are: 1. the support for multiple participants, organised by roles, which can dynamically join a protocol; 2. the use of assertions to describe a protocol state and control how (parametrised) actions change it (in a style akin to Hoare triples); 3. a notion of well-formed models and a checking algorithm; 4. a tool for describing systems with DAFSMs, visualising them as FSMs, and checking their well-formedness.

In scope of future work is to define a model-checking approach to support safety and liveness property analysis. We also plan generalisations of the model to allow role revocation and code generation. An interesting line of work extract DAFSMs from actual SC programs. A deeper analysis of the performances should also be conducted on more case studies and possibly refining the random generation of DAFSMs.

References

1. Afonso, J.: Mechanisms for Modeling and Validation of Smart Contracts. Master's thesis, Departamento de Informática, NOVA School of Science and Technology (2023), advisor: António Ravara

2. Barbanera, F., Lanese, I., Tuosto, E.: Choreography automata. In: Bliudze, S., Bocchi, L. (eds.) Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020. LNCS, vol. 12134, pp. 86–106. Springer (2020). https://doi.org/10.1007/978-3-030-50029-0_6
3. Basile, D., ter Beek, M.: Contract automata library. Sci. Comput. Program. **221**, 102841 (2022). https://doi.org/10.1016/J.SCICO.2022.102841
4. Basile, D., Degano, P., Ferrari, G.L., Tuosto, E.: Playing with our CAT and communication-centric applications. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Networked and Distributed Systems. LNCS, vol. 9688, pp. 62–73. Springer (2016). https://doi.org/10.1007/978-3-319-39570-8_5
5. Basile, D., ter Beek, M.: A runtime environment for contract automata. In: Chechik, M., Katoen, J., Leucker, M. (eds.) Formal Methods - 25th International Symposium, FM 2023, Proceedings. LNCS, vol. 14000, pp. 550–567. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_31
6. Basile, D., ter Beek, M., Pugliese, R.: Synthesis of Orchestrations and Choreographies: Bridging the Gap between Supervisory Control and Coordination of Services. Logical Methods in Computer Science **16**(2) (2020). https://doi.org/10.23638/LMCS-16(2:9)2020
7. Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform. https://ethereum.org/whitepaper (2014)
8. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification, Lecture Notes in Computer Science, vol. 7850, pp. 15–58. Springer (2013). https://doi.org/10.1007/978-3-642-36946-9_3
9. Coblenz, M.J., Oei, R., Etzel, T., Koronkevich, P., Baker, M., Bloem, Y., Myers, B.A., Sunshine, J., Aldrich, J.: Obsidian: Typestate and assets for safer blockchain programming. ACM Trans. Program. Lang. Syst. **42**(3), 14:1–14:82 (2020). https://doi.org/10.1145/3417516
10. Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., Tiezzi, F.: Chorchain: A blockchain-based framework for executing and auditing BPMN choreographies. In: BPM (PhD/Demos). CEUR Workshop Proceedings, vol. 3216, pp. 132–136. CEUR-WS.org (2022)
11. Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., Tiezzi, F.: Engineering trustable and auditable choreography-based systems using blockchain. ACM Trans. Manag. Inf. Syst. **13**(3), 31:1–31:53 (2022)
12. Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., Tiezzi, F.: Blockchain-based execution of bpmn choreographies with multiple instances. Accepted for publication in Distributed Ledger Technol. Res. Pract. (2023)
13. Crafa, S., Laneve, C.: Liquidity analysis in resource-aware programming. In: Tarifa, S.L.T., Proença, J. (eds.) Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13712, pp. 205–221. Springer (2022). https://doi.org/10.1007/978-3-031-20872-0_12, https://doi.org/10.1007/978-3-031-20872-0_12
14. Crafa, S., Laneve, C., Sartor, G., Veschetti, A.: Pacta sunt servanda: Legal contracts in *Stipula*. Science of Computer Programming **225** (2023). https://doi.org/10.1016/J.SCICO.2022.102911

15. Dokter, K., Jongmans, S.S., Arbab, F., Bliudze, S.: Combine and conquer: Relating BIP and Reo. Journal of Logical and Algebraic Methods in Programming **86**(1), 134–156 (2017). https://doi.org/https://doi.org/10.1016/j.jlamp.2016.09.008
16. Falcão, A., Mordido, A., Vasconcelos, V.T.: Protocol-based smart contract generation. In: Financial Cryptography and Data Security. FC'22 International Workshops - CoDecFin, DeFi, Voting, WTSC, Revised Selected Papers. LNCS, vol. 13412, pp. 555–582. Springer (2022). https://doi.org/10.1007/978-3-031-32415-4_34
17. Garcia, R., Tanter, É., Wolff, R., Aldrich, J.: Foundations of Typestate-Oriented Programming. ACM Transactions on Programming Languages and Systems **36**(4), 12 (2014). https://doi.org/10.1145/2629609
18. Gheri, L., Lanese, I., Sayers, N., Tuosto, E., Yoshida, N.: Design-By-Contract for Flexible Multiparty Session Protocols. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming (ECOOP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 222, pp. 8:1–8:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.8
19. Gheri, L., Lanese, I., Sayers, N., Tuosto, E., Yoshida, N.: Design-by-Contract for Flexible Multiparty Session Protocols (Artifact). Dagstuhl Artifacts Series **8**(2), 21:1–21:5 (2022). https://doi.org/10.4230/DARTS.8.2.21
20. Godoy, J., Galeotti, J.P., Garbervetsky, D., Uchitel, S.: Predicate abstractions for smart contract validation. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS'22. pp. 289–299. ACM (2022). https://doi.org/10.1145/3550355.3552462
21. Object Management Group: Business Process Model and Notation, http://www.bpmn.org
22. Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using networkx. In: Varoquaux, G., Vaught, T., Millman, J. (eds.) Proceedings of the 7th Python in Science Conference. pp. 11 – 15. Pasadena, CA USA (2008)
23. Hansen, P.: Operating System Principles. Prentice-Hall (1973)
24. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y.: http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217, working Draft 17 December 2004
25. Keerthi Nelaturu, A.M., et al.: Correct-by-design interacting smart contracts and a systematic approach for verifying erc20 and erc721 contracts with verisolid. IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING **20** (2023)
26. Laneve, C.: Liquidity analysis in resource-aware programming. Journal of Logical and Algebraic Methods in Programming (2023)
27. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. In: Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10957, pp. 523–540. Springer (2018). https://doi.org/10.1007/978-3-662-58387-6_28
28. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: 23rd International Conference onFinancial Cryptography and Data Security, FC 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11598, pp. 446–465. Springer (2019). https://doi.org/10.1007/978-3-030-32101-7_27
29. Microsoft: The blockchain workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench (2019)

30. 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)
31. Orlando, S., Pasquale, V.D., Barbanera, F., Lanese, I., Tuosto, E.: Corinne, a tool for choreography automata. In: Salaün, G., Wijs, A. (eds.) Formal Aspects of Component Software - 17th International Conference, FACS 2021,. LNCS, vol. 13077, pp. 82–92. Springer (2021). https://doi.org/10.1007/978-3-030-90636-8_5
32. Pigné, Y., Dutot, A., Guinand, F., Olivier, D.: Graphstream: A tool for bridging the gap between complex systems and dynamic graphs (2008)
33. Pombo, C., Suñé, A., Tuosto, E.: A dynamic temporal logic for quality of service in choreographic models. In: Ábrahám, E., Dubslaff, C., Tarifa, S. (eds.) Theoretical Aspects of Computing – ICTAC 2023. pp. 119–138. Springer (2023). https://doi.org/https://doi.org/10.1007/978-3-031-47963-2_9
34. Qin, X., Bliudze, S., Madelaine, E., Hou, Z., Deng, Y., Zhang, M.: SMT-based generation of symbolic automata. Acta Informatica **57**, 627–656 (May 2020). https://doi.org/10.1007/s00236-020-00367-6
35. Senarruzza Anabia, D.: Bisimulación de Data-aware Communicating Finite State Machines con propiedades en las acciones. Master's thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires (2023), advisors: Carlos G. Lopez Pombo and Hernán C. Melgratti
36. Singh, N.K., Fajge, A.M., Halder, R., Alam, M.I.: Formal verification and code generation for solidity smart contracts. In: Distributed Computing to Blockchain, pp. 125–144. Elsevier (2023)
37. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering **SE-12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
38. Wadler, P.: Linear types can change the world! In: Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods. p. 561. North-Holland (1990)