Caterpillar Documentation (v2.0 – v2.1)

Caterpillar is a Business Process Management System (BPMS) that runs on top of Ethereum and that relies on the translation of process models into smart contracts. More specifically, Caterpillar accepts as input a process model specified in BPMN and generates a set of smart contracts that captures the underlying behavior. The smart contracts, written in Ethereum's Solidity language, can then be compiled and deployed to the public or any other private Ethereum network using standard tools. Moreover, Caterpillar exhibits a REST API that can be used to interact with running instances of the deployed process models.

Caterpillar also provides a set of modelling tools and an execution panel which interact with the underlying execution engine via the REST API. The latter can also be used by third party software to interact in a programmatic way via Caterpillar with the instances of business process running on the blockchain.

The full description of the version v2.0 can be accessed from:
https://arxiv.org/abs/1808.03517.

The paper describing the Role Dynamic Binding and Access Control can be accessed from:
https://arxiv.org/abs/1812.02909.

The source code of Caterpillar is distributed in two folders (v2.0-v2.1). The folder **caterpillar_core** includes the implementation of the core components (Off-chain Runtime + REST API, back end). The **execution_panel** consists of the code of the Web portal (fron-end) that serves to deploy, configure and keeping track of the execution state of process instances and to lets users check in process data. In v2.1, the source code is in the folder labelled as 'prototype'. Besides, the folder "Dynamic Binding Example" contains the binding policy and BPMN model used as running example in https://arxiv.org/abs/1812.02909.

1. Ganache cli

By default, the core of Caterpillar was configured to run on top of ***Ganache CLI*** which is a ***Node.js*** based ***Ethereum*** client for testing and development. It uses ethereumjs to simulate full client behavior and make developing Ethereum applications. All the instructions about the installation can be found here: https://github.com/trufflesuite/ganache-cli/ .However, the Ethereum Provider can be updated at the beginning of the source code in the controller "caterpillar-core/src/models/models.controller.ts" (check the comments in source code).

Note that Ganache CLI is written in JavaScript and distributed as a Node package via npm. Make sure you have Node.js (>= v6.11.5) installed. Besides, be aware to start the Ganache CLI server before running the applications Caterpillar Core. In that respect, you only need to open a terminal on your computer and run the command:

```
ganache-cli
```

2 MongoDB

The versions v2.0, v2.1 use a process repository to store and access metadata produced by Caterpillar when compiling the BPMN model into Solidity smart contracts. Currently, this repository is implemented on top of MongoDB which is a database that stores data as JSON-like documents. The instructions to install MongoDB Community Edition can be accessed from here: https://docs.mongodb.com/manual/administration/install-community/.

3. Caterpillar core

WARNING: Before running Caterpillar Core, make sure you installed ***gulp-cli*** running the command: ***npm install gulp-cli -g***. All the instructions about the ***glup-cli*** installation can be found here: https://www.npmjs.com/package/gulp-cli?activeTab=readme.

To set up and run the core, open a terminal in your computer and move into the folder ***caterpillar_core.***

To install the dependencies, run the commands:

```
npm install
gulp build
```

To start the application, you may use one of the following commands:

```
node ./out/www.js
gulp
```

By default, the application runs at http://localhost:3000.

WARNING: Make sure you have Ganache Cli running in your computer before starting the core. Besides, MongoDB client must be also running.

3.1 REST-API

The application provides a REST API to interact with the core of Caterpillar. The following tables summarize the mapping of resource-related actions[1]:

[1] A full escription of the Role Dynamic Binding and Access Control can be accessed from: https://arxiv.org/abs/1812.02909.

Verb:	POST	**URI:**	/models
Description	Registers a BPMN model (Triggers also code generation and Compilation).		

Request Body Parameters	{
	bpmn: “BPMN model to compile/register”
	}		

Response	{
	id: “(hash) Identifier of the root process in the process repository”,
	name: “Name of the root process”,
	solidity: “Code in Solidity of all the contracts generated from the model”
	}		

Verb:	GET	**URI:**	/models
Description	Retrieves the list of registered BPMN models.		
Request Body Parameters			
Response	[{ id: “(hash) Identifier of the root process in the process repository”, name: “Name of the root process”, bpmn: “source BPMN model”, solidity: “Code in Solidity of all the contracts generated from the model” }, …]		

Verb:	POST	**URI:**	/models/:bundleId
Description	Creates a new process instance from a given model. : bundleId (hash identifier of the process to instantiate)		

Request Body Parameters	{
	caseCreator: “Ethereum (public) address of the actor creating the instance”,
	creatorRole: “Role (String) of the actor creating the instance”
	}		

Response	{
	address: “Ethereum address where running the root process contract”,
	gas: “Gas used to deploy the process contracts”,
	runtimeAddress: “Ethereum address of the access control contract”,
	runtimeGas: “Gas used to deploy the access control contract”,
	transactionHash: “transaction hash generated to register the case-creator”
	}		

Verb:	GET	**URI:**	/processes
Description	Retrieves the list of active process instances		
Request Body Parameters			

Response	[
	{
	id: “(hash) Identifier of the root process in the process repository”,
	name: “Name of the root process”,
	address: “Ethereum address where running the root process contract”
	}
	…
	]		

Verb:	GET	**URI:**	/processes/:procAddress
Description	Retrieves the current state of a process instance. :procAddress (Blockchain address of the process to query)		
Request Body Parameters			

Response	{
	bpmn: “BPMN model”
	workitems: “List of started workitems”,
	serviceTasks: “List of started service tasks”
	}
	One workitem is described by:
	{
	elementId: “Identifier (String) of the corresponding activity in the BPMN model”,
	elementName: Name (String) of the corresponding activity in the BPMN model,
	input: “List of parameters defined in the model for the given element”,
	bundleId: “(hash) Identifier of the root process in the process repository”,
	processAddress: “Ethereum address where running the root process contract”,
	pCases: “List with the addresses of each sub-process when started the element”,
	hrefs: “List of the URI’s to execute the workitem in each subprocess via Worklist”
	}		

Verb:	POST	**URI:**	/workitems/:worklistAddress/:reqId
Description	Checks-in a work item (i.e. user task) :worklistAddress (address of the worklist containing the workitem) :reqId (identifier (Integer) of the workitem)		
Request Body Parameters			

Response	{
	transactionHash: “transaction hash generated when executing the task”
	}		

In addition to the previous URI’s the version v2.1 introduces new ones to handle the Role Binding Operations:

Verb:	POST	**URI:**	/registry
Description	Deploys a new instance of the Runtime Registry		
Request Body Parameters			

Response	{
	address: “Ethereum address where running the runtime registry”,
	gas: “Gas used to deploy the registry smart contract”
	}		

Verb:	POST	**URI:**	/resources/policy
Description	Generates/Deploys the contracts from a given binding policy specification (BPF).		

Request Body Parameters	{
	model: “Binding Policy Specification[2]”
	}		

Response	{
	address: “Ethereum address where running the policy smart contract”,
	gas: “Gas used to deploy the policy smart contract”
	}		

Verb:	POST	**URI:**	/resources/task-role
Description	Generates/Deploys the contract mapping the roles to the tasks in a BPMN model.		

Request Body Parameters	{
	rootProc: “(hash) Identifier of the root process”
	}		

Response	{
	address: “Ethereum address where running the TaskRoleMap smart contract”,
	gas: “Gas used to deploy the TaskRoleMap smart contract”
	}		

Verb:	POST	**URI:**	/resources/nominate
Description	Nominates an actor into a role as defined in the BPF		

Request Body Parameters	{
	rNominator: “Role (String) of the nominator”,
	rNominee: “Role (String) of the nominee”,
	nominator: “Ethereum (public) address of the nominator (actor)”,
	nominee: “Ethereum (public) address of the nominee (actor)”,
	pCase: “Ethereum address of (sub-)process where the nomination takes place”
	}		

Response	{
	transactionHash: “transaction hash generated when nominating the actor”
	}		

[2] The grammar describing the Binding Policy Specification can be found at: https://github.com/orlenyslp/Caterpillar/blob/master/v2.1/prototype/caterpillar-core/src/models/dynamic_binding/antlr/binding_grammar.g4.

Verb:	POST	**URI:**	/resources/release
Description	Releases an actor from a role as defined in the BPF		

Request Body Parameters	{
	rNominator: “Role (String) of the nominator (role who will release)”,
	rNominee: “Role (String) of the nominee (role to release)”,
	nominator: “Ethereum (public) address of the actor performing the release”,
	pCase: “Ethereum address of (sub-)process where the release takes place”
	}		

Response	{
	transactionHash: “transaction hash generated when releasing the actor”
	}		

Verb:	POST	**URI:**	/resources/vote
Description	Accepts/Rejects the nomination/release of an actor as defined in the BPF		

Request Body Parameters	{
	rNominator: “Role (String) that performed the nomination/release”,
	rNominee: “Role (String) that received the nomination/release”,
	rEndorser: “Role (String) that is voting for the nomination/release”,
	endorser: “Ethereum (public) address of the actor that is voting”,
	pCase: “Ethereum address of (sub-)process where the nom./release was made”,
	onNomination: “True if voting a nomination, False if voting a release”,
	isAccepted: “True if accepting the nomination/release, False if rejecting”
	}		

Response	{
	transactionHash: “transaction hash generated when voting”
	}		

Verb:	GET	**URI:**	/resources/:role/:pAddr
Description	Retrieves the current state of an actor into a role in a process instance :role (role to query the state) :pAddr (Blockchain address of the process to check the role state)		
Request Body Parameters			

Response	{
	state: “State of the current role in the process instance”
	}		

4. Execution-panel (v2.1)

WARNING: Before running the Execution Panel, make sure that you installed angular-cli: https://github.com/angular/angular-cli/wiki

To set up and run the execution panel, open a terminal in your computer and move into the folder **execution-panel**.

To install the dependencies, run the command:

npm install

To run the application, use the command:

ng serve

Open a web browser and put the URL http://localhost:4200/. You should see a view as shown in Fig. 1. The execution panel interacts with the REST-API of Caterpillar. Accordingly, the request/results of each operation are printed in the terminal where running Caterpillar's core.

Fig. 1 Dashboard in Caterpillar's execution panel.

4.1. Runtime Registry

Caterpillar uses the "Runtime registry" to store each contract deployed as well as the relations among contracts. Thus, the first action to perform is create (deploy) a new runtime registry. To create the registry, expand "Configuration: Process Runtime Registry Operations" to display the view in Fig. 2, and click the button "Create Registry".

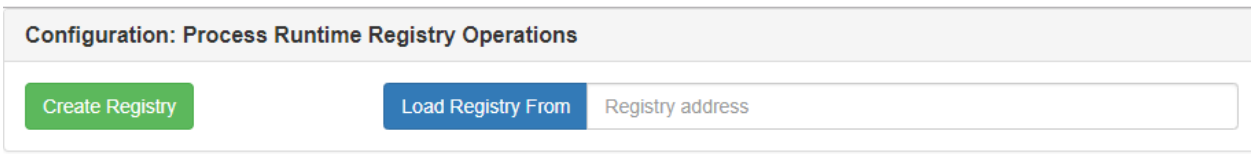


Fig. 2 Runtime Registry Deployment.

WARNING: The "Runtime Registry" must be deployed before any other operation in the execution panel. An error "**Registry NOT Found**" will be displayed in the terminal of the caterpillar core when trying to perform an operation before deploying the registry.

4.2. BPMN Models

As described in https://arxiv.org/abs/1808.03517, the process models in Caterpillar contain annotations in Solidity. Most of the annotations must be added in the documentation of the corresponding process/element. The decision gateways are the only exception as the annotations are added on the output arcs. For example, Fig. 3 corresponds to the root process that can be found in the Caterpillar's repository (v2.1). In the documentation of the process you will find the global declaration:

```
bool private poStatus;
```

This variable will be used to store whether a purchase order is accepted or not and used in the decision gateways. Besides, this variable is updated in the documentation of the task "Validate PO" that is annotated as:

```
@ Supplier @ () : (bool _poStatus) -> {poStatus = _poStatus;}
```

Be aware that in general the tasks are annotated as:

(data_to_export) : (data_to_import) -> {Operations_to_perform}

The data_to_export section defines which variables are read by the task from the process contract (i.e. input parameters of the task). The specifies the output parameters of the task, i.e. the data that the task obtains from the external resource. The Operations_to_perform section contains a set of Solidity operations to map the output parameters to the variables of the process. In the task "Validate PO", we are not exporting any data, and only requesting from an actor (with role Supplier) to accept (or not) the PO and accordingly the global variable "poStatus" is updated (as part of the Operations to perform in the task).

Also, from version v2.1 the user tasks must include between @@ the role of the actor designed to perform it. For example, @ Supplier @ in "Validate PO" means that a Supplier must perform the task, and that will be checked by the binding policy related to the BPMN model.

Another, mandatory annotation is the link of a call activity with a subprocess. For example, in Fig. 3 is shown how the call activity "Shipment" is linked to a sub-process that is represented by its hash identifier (blue rectangle).

WARNING: To deploy a process with call activities the subprocesses to link must exist in the process repository of Caterpillar, and hash must be stored in the Registry and linked to a factory (see section 4.3).

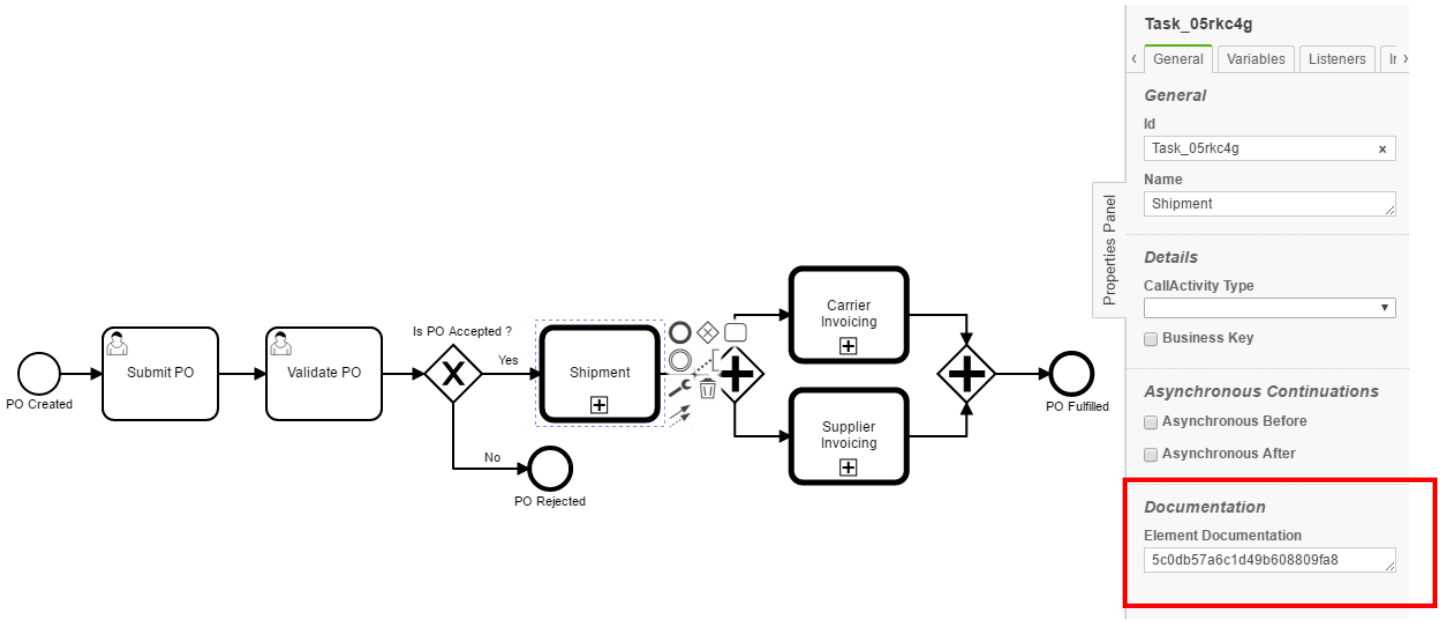


Fig. 3 Example of BPMN model.

4.3. Business Process Operations

To add a new a new process model, expand the "Business Process Operations" in the execution panel and click "Add Process Model" (Fig. 4). A new view will allow you to upload a model or start to draw a new one. Be sure of providing a valid ID and name to the process before trying to save the model because Caterpillar rejects models with no names.

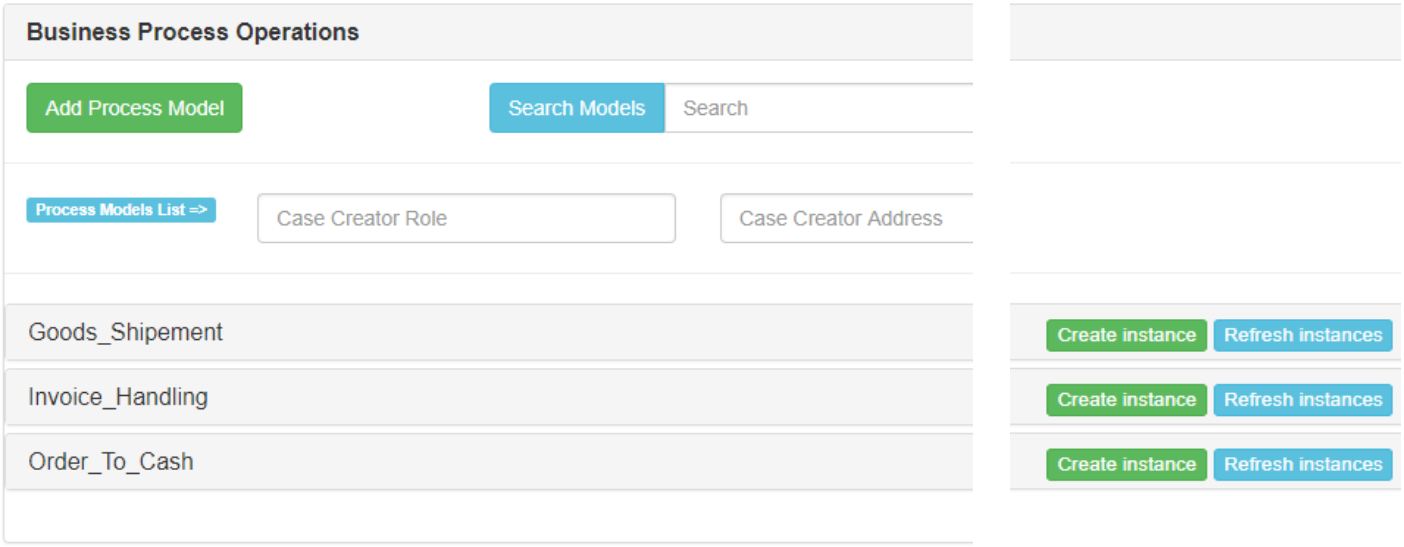


Fig 4. Business Process Operations

Once the model is saved, the execution panel would display a view as shown in Fig. 5 corresponding to the process in Fig. 4. For space reasons, we omitted the Solidity code of the contracts. When saving a process model in the dashboard, Caterpillar generates the process contracts, as well as the factories and worklists. Besides, the compilation metadata is stored in the process repository, and the hash of the entry will be the identifier of the process for future execution. Accordingly, Caterpillar updates the runtime registry. Note

how the red rectangle encloses the relation child-parent relying on the hashes of each sub-process. The format of the expression is (child) => (element-index), (parent), where index 0 represents the root process which is considered as its parent. Be aware that the call-activities must include the hash of the repository entry of the process to link, which accordingly should be registered before.

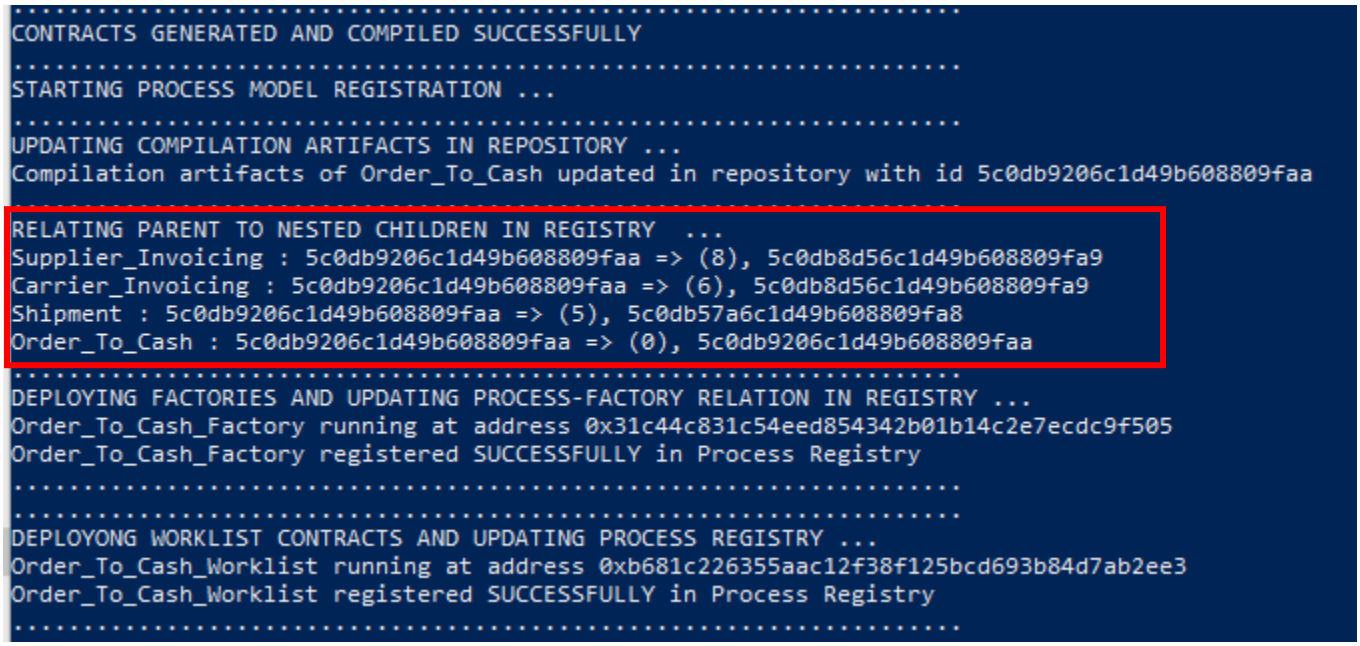


Fig. 5 Registering of the process model 'Order to Cash'.

The creation of a new instance of a process is made via the button "Create Instance" of the corresponding process in the dashboard. However, to create an instance of the process, a BindingPolicy, and a RoleTaskMap must be deployed before (see section 4.4 in this document and https://arxiv.org/abs/1812.02909 for further details). Whether the policy is already deployed, then the role of the case creator, and their Ethereum (public) address must be provided (see Fig. 4) to create the process instance. Note that the text-fields to provide case-creator role and address are reused for all the processes in the dashboard.

You must use the button "Refresh instances" to update the instances running, and then click one of the displayed addresses to proceed with the process execution (Fig. 6). Then by pressing the button "Display Model" in the new view, you can see the started activities visualized in dark green. To perform any started activity, click on it and fill the parameter info if required (see Fig. 7). Also, to perform the task, an actor referred to as "Performer User Account Address" must be bound to the role related to the task, what is controlled by the binding policy. Note that the binding policies use the public key as the identifier of an actor, that once accepted (if they fulfill the binding policy statements) must sign the transaction using his private key.

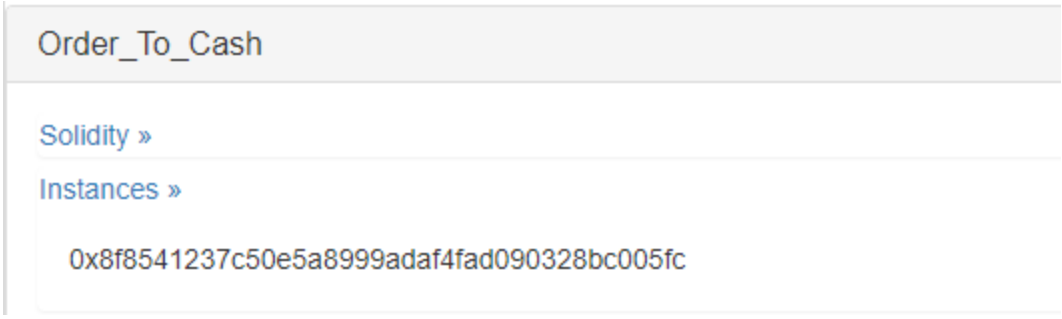


Fig. 6. Click the address to proceed with the process execution in Fig. 7.

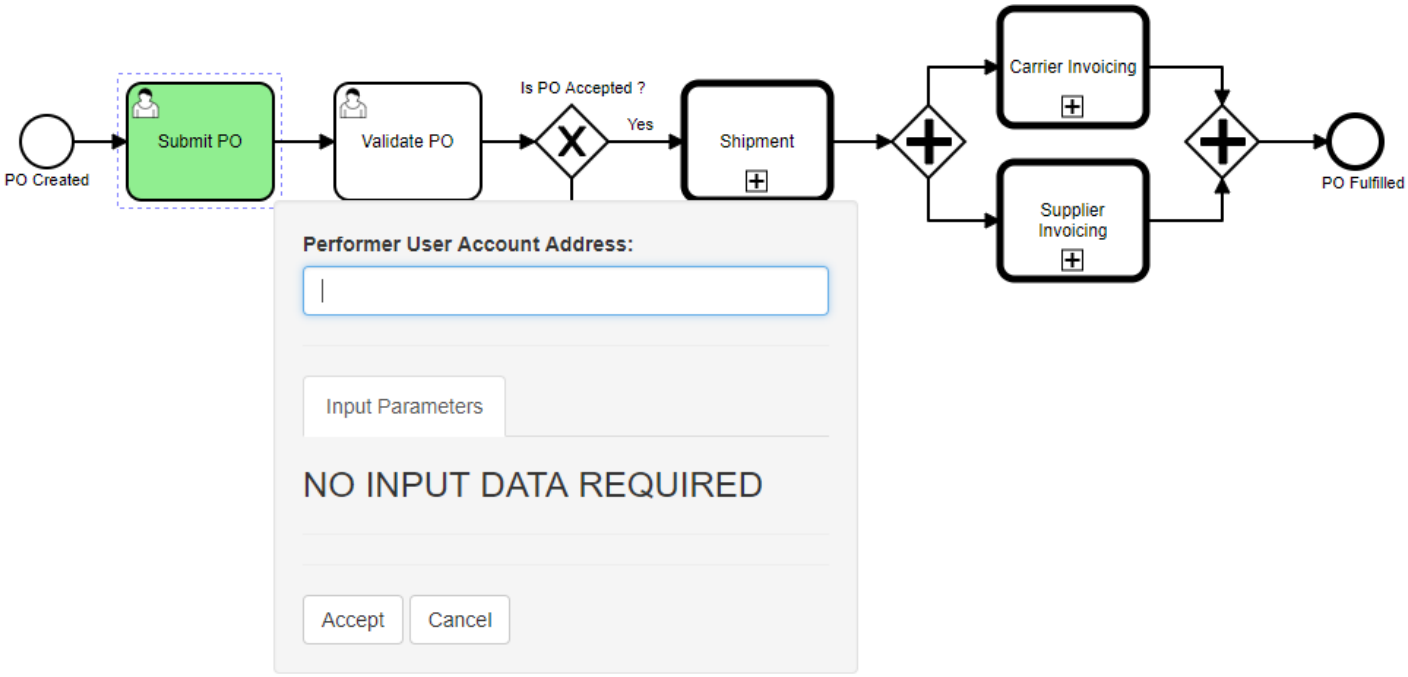


Fig. 7. Execution of a task.

Once a call activity is reached in the control flow, Caterpillar internally instantiates the contracts. However, to execute the corresponding subprocess, return to the dashboard, click the “Refresh instances” of the process linked to the call activity and proceed with the execution as described above.

4.4. Binding Policy: Static Configuration

Before creating new instances of a process, a BindingPolicy and a RoleTaskMap must be deployed to control the execution. An example of a policy specification can be found at:

https://github.com/orlenyslp/Caterpillar/blob/master/v2.1/Dynamic%20Binding%20Example/Models/BindingPolicySpecification.txt.

Besides, the deployment of binding policy also requires the runtime registry is running. Otherwise, the error "Registry NOT found" will be displayed. The binding policy can be deployed before or after the process model registration. Caterpillar allows uploading a specification from a file or design one in a simple editor. The smart contract generation and deployment occur after clicking the button "Deploy Binding Policy" (Fig. 8).

Finally, the deployment of the contract TaskRoleMap requires that the policy is already deployed, and the process model is already updated in the registry. Then, by providing the hash of the process, Caterpillar tries to relate the process model with the last policy deployed, via the TaskRoleMap by clicking the button "Deploy TaskRoleMap from". Caterpillar also validates that the policy is consistent with the model (i.e. it will not produce deadlocks). Inconsistent policies are rejected.

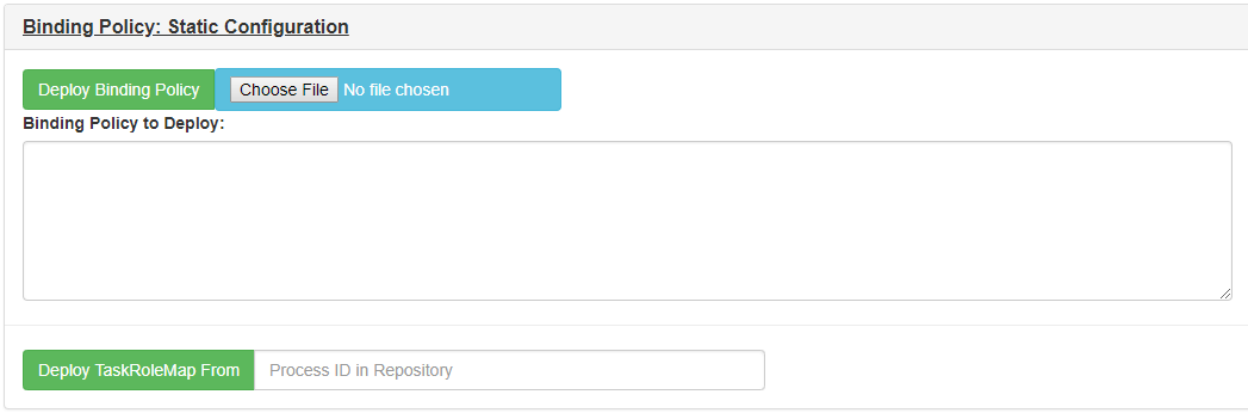


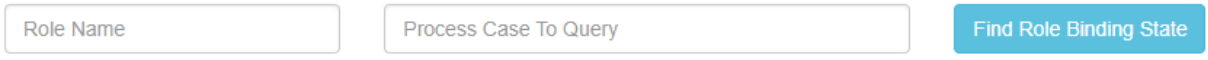
Fig. 8. Binding Policy: Static Configuration.

4.5. Binding Policy: Runtime Operations

The Runtime operations allow the nomination, release, and endorsement as described in https://arxiv.org/abs/1812.02909. Actors, which are identified by their Ethereum (public) addresses, are binded to roles in a process instance (it may be the root process or any child sub-process). In other words, to bind a role to an actor, a process instance must be assigned which also sets the scope where such actor can participate. As some nominations may require endorsement, we consider 4 states for a role in a process instance (UNBOUND, NOMINATED, BOUND, RELEASING). Accordingly, to execute a task, the actor must be BOUND to the related role in the corresponding process instance (or a parent in the processes hierarchy).

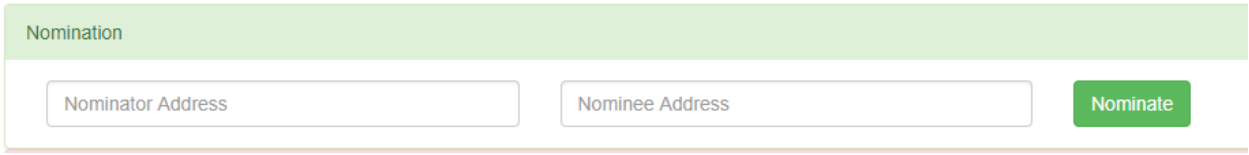
Fig. 9 (a) shows how to query the state of a role in a process instance. To nominate/release or endorse and actor, the three parameters in Fig. 9 (b) must be filled (Nominator role, nominee role, and process case). Then to proceed with the nomination (Fig. 9 (c)), the Ethereum public address of the nominator and nominee must be provided. Note that the nominator is who performs the nomination operation, then he must sign the corresponding

transaction in the blockchain. Similarly, the release operation Fig. 9 (d) requires the parameters in Fig. 9 (b) but also adds the public address of the actor trying to release. Note that here the addresses of the nominator and the nominee are not required as the release operation occurs on roles already nominated. Thus, that information is previously stored in the blockchain. Finally, a nomination/release operation is only completed whether the endorsements defined by the policy are provided. Then, an endorser must give the roles of the nominator, nominee, and the process case he is trying to endorse (Fig. 9 (b)), as well as their role and Ethereum address (Fig. 9(e)). Besides, the endorser must specify the type of operation he is voting, i.e. nomination/release, and whether he is accepting or not such operation.

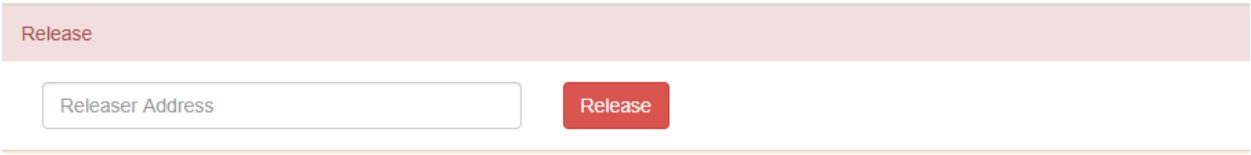


(a) Querying the state of a Role.

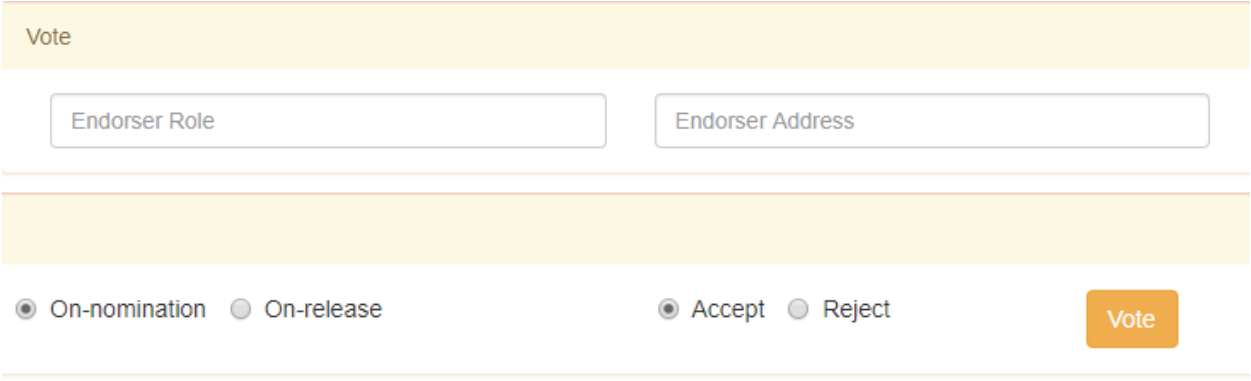
(b) Common parameters on the operations of nominate/release/vote.



(c) Nominate operation



(d) Release operation



(e) Vote operation

Fig. 9 Binding Policy: Runtime Operations.