SMaRt Blockchain Distributed Workflow Management

Joerg Evermann[1] and Henry Kim[2]

[1] Memorial University of Newfoundland, St. John's, Canada
jevermann@mun.ca
[2] York University, Toronto, Canada
hkim@york.ca

Abstract. Interorganizational business processes are rapidly becoming the norm in many industries. At the same time, business partners are often in a state of "coopetition", or competitive cooperation. To support execution of business processes in this context requires distributed workflow management systems that can provide security, integrity, and verifiability of workflow states.
Blockchain technology provides a suitable infrastructure. Existing blockchain based workflow management systems are built on expensive, inefficient, public proof-of-work blockchains. In contrast, we believe that small-scale, efficient, private blockchains deployed for a particular workflow application are more appropriate in stable coopetition contexts with up to a few dozen well defined partners. In this paper, we present a prototype workflow management system that is built on state machine replication for Byzantine fault tolerant systems.

Keywords: Byzantine fault tolerance · blockchain · workflow management · interorganizational workflow · distributed workflow

1 Introduction

Inter-enterprise business processes may involve actors in adversarial relationships that nonetheless have to jointly complete process instances. In such a state of "coopetition" (cooperative competition), trust in the current state of a process instance and correct execution of activities by others may be lacking. Actors may be reluctant to accept infrastructure provided by central authority. Blockchain technology can provide a secure infrastructure that does not require a central authority and allows independent verification of the state of a workflow.

A blockchain cryptographically signs a series of blocks containing transactions, so that it is difficult or impossible to alter earlier blocks in the chain. In a distributed blockchain, actors independently validate transactions and add them to the blockchain. The independence of actors requires a protocol for achieving consensus regarding the validity and order of transactions and blocks. Actors must agree on the state of work as that determines the set of next valid activities in a process. Hence, it is natural to use blockchain transactions to describe workflow workflow instance states.

Workflow management systems (WfMS) can be implemented in different ways on blockchains. Prior work has focused on transaction ordering through proof-of-work consensus. In contrast, we use a consensus protocol based on Byzantine fault tolerant (BFT) state machine replication (SMR). We present a prototype WfMS as a proof-of-concept implementation for this architecture.

2 Innovation, Contribution, Novelty

Public proof-of-work based systems offer greater security at high cost, high latency, and high resource use, and are scalable to millions of actors. Private SMR BFT based systems do not scale as well, provide a lower level of resilience, require better networking connections among nodes, but offer low latency, finality of consensus, and are resource efficient. As such, they are suitable for a small set of pre-selected and fixed actors that may distrust each other but are not expressly malicious, and need to execute quick workflows. Our system is innovative in the following ways:

- It is one of only a few blockchain-based workflow management systems. As such, it provides information replication, verifiability, and cryptographically assured information integrity for distributed workflow management.
- It uses a novel method of SMR for BFT systems for implementation on a blockchain. We are not aware of any other distributed workflow management system using this method.
- It avoids the drawbacks of proof-of-work blockchains, offering lower latencies, finality of consensus, and higher efficiency than proof-of-work systems.
- It does not rely on smart contracts or specific virtual machines.
- A simple architecture enables future extensions and use of different workflow specification languages.

3 Short Introduction to Blockchains

A blockchain consists of blocks of transactions, which can be any kind of content. Information integrity is achieved by applying a hash function to the content of each block, which also contains the hash of the previous block in the chain. Hence, altering a block requires changes to all following blocks. In a distributed blockchain blocks are distributed to each node for independent validation and replicated storage and new transactions may originate on any node. The key challenge is to achieve a consensus on the validity and order of transactions and blocks, despite nodes that are characterized by "byzantine faults": they may not respond correctly, may respond unpredictably, may become altogether unresponsive, or may attempt to undermine the integrity of the chain.

Blockchain-based workflow management has only recently received attention [6]. A number of prototypes have been presented [3–5, 8], using smart contracts on the public, proof-of-work-based Ethereum blockchain. However, after examining different blockchain consensus mechanisms, [7] recommend BFT-based consensus for business process executions.

Proof-of-Work Consensus In proof-of-work consensus, new transactions are distributed to all nodes, are validated and added to each node's transaction pool. Each node can independently propose and distribute new blocks. Depending on network topology and speed, nodes may have different set of blocks and transactions, and hence may propose different blocks, leading to *side branches*. Each node considers the longest branch as its current main branch and proposes new blocks based on it. When a side branch becomes longer than the current main branch, the chain undergoes a *reorganization*: What was the side branch is validated and becomes the main branch. What was the main branch is considered invalid and becomes a side branch. Transactions no longer in the main branch are added back to the transaction pool to be included in other blocks. Consequently, different nodes can at times consider different blocks and transactions as valid.

Artificially limiting the rate of new blocks allows nodes to achieve eventual consensus on what constitutes the main branch, and prevents attackers from "overtaking" the creation of legitimate blocks with fraudulent one. For this, block proposers must solve a hard problem ("proof-of-work", "mining"). Assuming equal processing power for each node, the network needs $2f + 1$ total nodes to tolerate f faulty or malicious nodes.

Proof-of-work consensus is inefficient due do the mining, lacks final consensus due to chain reorganizations and induces significant latency for a new block to be accepted as valid.

State Machine Replication and BFT Consensus In response to the drawbacks of proof-of-work consensus, ordering algorithms for distributed systems have seen a resurgence in interest, mostly traceable to a seminal paper on practical byzantine fault tolerance (PBFT) [2]. PBFT orders requests for operations using a set of nodes that are fully connected by reliable messaging. Every ordering consensus is established by a specific set of nodes ("view") with a leader node. Tolerating up to f faulty nodes requires $3f + 1$ total nodes.

Clients send requests for operations to all nodes. The leader proposes a sequence number for the request and broadcasts a pre-prepare message. Upon receipt of a pre-prepare message, a node broadcasts a corresponding prepare message if it has itself received the request and has not already received another pre-prepare message for the same sequence number. Nodes then wait to receive $2f$ matching prepare messages from other nodes, indicating a majority is prepared to accept the proposed sequence number. When this occurs, a node broadcasts a commit message to all nodes. Each node then waits to receive $2f$ commit messages, indicating that a majority has accepted the proposed sequence number. Upon acceptance, the node executes the requested operation in order and sends the result to the client. The client in turn waits for $2f + 1$ replies, which indicates that an ordering consensus has been reached.

Consensus about operation sequencing is one aspect of state machine replication (SMR). Each node maintains a state that can be changed by the requested operations. When every node begins with the same state and executes operations in the same order, the state machine is replicated.

4 System Architecture and Implementation

We developed our prototype system in Java. It can be downloaded at joerg.evermann.ca/software.html and an introductory video can be found at the same URL. We use the BFT-SMART [1] library for state machine replication for byzantine fault tolerant systems. It can be configured to provide crash tolerance only, rather than byzantine fault tolerance, increasing its performance. Digital signatures for messages allow it to also tolerate malicious nodes.

Our system comprises three main services, ordering service, block service, and the workflow engine.

Ordering Service The ordering service receives transactions from the workflow engine, which is a request for an ordered operation. Each transaction represents a workflow instance state. The ordering service maintains as its state the latest block hash and block number, and a queue of transactions waiting to be collected into a block. When a sufficient number of transactions is available, it creates a new block, passes it to the block service, and clears the transaction queue.

Block Service The block service stores the blockchain, exchanges blocks with other nodes, and verifies the integrity of the blockchain. It uses a peer-to-peer network distinct from the network layer of BFT-SMART. Block exchange is required when a node begins operation or recovers after a restart. At that point, the ordering service state is first updated through the BFT-SMART state replication mechanism. The block service then compares its latest block to the latest hash from the ordering service. Verification of the blockchain then proceeds backwards from the head of the chain, i.e. the block with the latest hash. Missing blocks are requested from peers and verified prior to adding them.

Workflow Engine The workflow engine is notified by the block service when a new block is added to the chain. It reads all transactions in the block, updating its information about the state of each process instance and creating work items accordingly. It manages user interactions with work items and execution of external functions by work items. Upon work item completion, the engine generates a new transaction and passes it to the ordering service.

Every node in our system contains all three components. This allows the ordering service to quickly validate transactions using the local workflow engine, the blockservice to easily notify the workflow engine of new transactions and the workflow engine to easily submit new transactions to the ordering service.

Our workflow models are based on plain Petri nets. Each transition specifies a workflow activity. The workflow engine keeps track of the net markings and case data, and detects deadlocked and finished cases to remove them from the worklist. Each activity is associated with a single node. Each node can provide its own resource management by defining mechanisms for further work item allocation. External method calls are possible to static Java methods. The data perspective is implemented as a key–value store. We currently admit only simple Java types as we implement a GUI for these.

5 Maturity and Future Work

The current status of our system is that of a research and teaching prototype. We have used it to explore interfaces between blockchain infrastructure and workflow engines and to develop suitable software architectures. We are also using this system as a demonstration system for distributed, inter-organizational workflow management in one class of a process management course. Further work planned:

- Resource management needs further development using organizational units such as roles, departments, positions, and work item routing within each node must be based on this.
- Data management needs to be extended to complex types. This can remain on a Java basis, the challenge is to automatically generate user interfaces
- Additional operations need to be supported: case abortion, specification invalidation, specification versioning
- A richer control flow language,m such as BPMN or YAWL nets, is required.

One way to achieve all these objectives is to port an existing workflow system to our blockchain infrastructure. The simple interfaces between blockchain and workflow engine makes this a relatively easy task. Open-source systems such as YAWL are good candidates for this. Additionally, further work in evaluating the system in real use cases is needed. For this, we are currently identifying business partners and public organizations for case studies.

References

1. Bessani, A.N., Sousa, J., Alchieri, E.A.P.: State machine replication for the masses with BFT-SMART. In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, GA, USA. pp. 355–362. (2014)
2. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002)
3. Fridgen, G., Radszuwill, S., Urbach, N., Utz, L.: Cross-organizational workflow management using blockchain technology - towards applicability, auditability, and automation. In: 51st Hawaii International Conference on System Sciences. (2018)
4. Härer, F.: Decentralized business process modeling and instance tracking secured by a blockchain. In: Bednar, P.M., Frank, U., Kautz, K. (eds.) 26th European Conference on Information Systems ECIS. p. 55. AIS Electronic Library (2018)
5. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I.: Caterpillar: A blockchain-based business process management system. In: Proceedings of the BPM Demo Track co-located with 15th International Conference on Business Process Modeling. CEUR vol. 1920 (2017)
6. Mendling, J., Weber, I., van der Aalst, W.M.P., vom Brocke, J., Cabanillas, C., et al.: Blockchains for business process management - challenges and opportunities. ACM Trans. Management Inf. Syst. **9**(1), 4:1–4:16 (2018).
7. Viriyasitavat, W., Hoonsopon, D.: Blockchain characteristics and consensus in modern business processes. Journal of Industrial Information Integration (2018)
8. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted business process monitoring and execution using blockchain. In: Business Process Management - 14th International Conference, BPM, Proceedings. LNCS, vol. 9850, pp. 329–347. Springer (2016).

A Tutorial

A virtual machine appliance for use with VirtualBox[3] can be downloaded from https://joerg.evermann.ca/BlockchainDemo.html. The virtual machine image contains a ready to run setup as well as the Eclipse[4] integrated development environment with the source code project. The default user name is **ubuntu** and the login password is **password**.

A.1 Configuration

The application is installed in the folder ∼/`BlockchainWFMSBFTSmart`. Within this, is a `config` folder. This folder contains three configuration files, `hosts.config`, `system.config`, and `blockchain.config`. *These configuration files must be identical for all nodes of the system and must be manually distributed.*

Hosts The *hosts* configuration file defines all possible nodes for the system with their node number, their IP address and port used by the ordering service. Add as required or comment out lines with `#`.

```
#server id, address and port
#(the ids from 0 to n-1 are the service replicas)
0 127.0.0.1 11000
1 127.0.0.1 11010
2 127.0.0.1 11020
3 127.0.0.1 11030
4 127.0.0.1 11040
5 127.0.0.1 11050
6 127.0.0.1 11060
7 127.0.0.1 11070
8 127.0.0.1 11080
9 127.0.0.1 11090
```

System The *system* configuration file defines the general behaviour of the system. Only a few options should be adjusted for each use case.

- `system.communication.signatureAlgorithm = SHA512withECDSA`
  The default digital signature algorithm uses elliptic curves with a 512-bit SHA hash function. Alternatively, this can be changed to use an RSA prime factorization algorithm. When generating public and private keys for each node, the system config file is read and keys are generated based on this setting.
- `system.communication.signatureAlgorithmProvider = SunEC`
  This is the security provider installed in Java that provides the signature algorithm. The configured provider must offer the configured signature algorithm.

[3] https://www.virtualbox.org
[4] https://www.eclipse.org

- `system.servers.num = 1`
  This is the total number of nodes in the original (startup) view. The system will not be able to process operations until this number of nodes have started and joined the view. Note that in BFT (byzantine fault tolerant, see below) mode, at least three nodes are required, while in CT (crash tolerant) mode, a single node may be used.
- `system.servers.f = 0`
  This is the maximum number of faulty servers that the system will be able to tolerate. In BFT mode, at least $3f + 1$ total nodes (`system.servers.num`). Example: To tolerate 1 faulty node, at least 4 nodes are required. In CT mode, at least $f + 1$ total nodes are required.
- `system.initial.view = 0,1,2,3`
  This specifies the set of nodes in the initial view in comma-separated form. The system will not process operations until all specified nodes have started and joined the view. The number of nodes specified here must match the number in `system.servers.num`.
- `system.ttp.id = 9`
  This parameter specifies the node ID of the "trusted third party" (TTP). This node is allowed to issue requests for view changes, to add or remove a node from the current view or to change the f of the current view. In our system, each node can assume this special node ID and issue reconfiguration requests. Ensure that this ID is defined in `hosts.config` and a private/public key pair is generated.
- `system.bft = false`
  This parameter specifies whether the system should run in BFT (byzantine fault tolerant) mode. If this is set to `false`, the system will operate in CT (crash tolerant) mode only.

Blockchain The *blockchain* configuration file defines the behaviour of the blockchain and the workflow management components of the system.

- `maxConnections = 2`
  This parameter specifies the number of connections that each node maintains on the peer-to-peer network that is used for block exchange. This is distinct from the fully connected network maintained by the ordering service.
- `numTransactionsPerBlock = 2`
  This parameter specifies the number of transactions for each block. As there is no mining expense with our system, this parameter can be kept quite small (even a value of 1 is possible). There is little to no overhead for block creation and creating blocks sooner reduces the latencies in the workflow applications.
- `textFieldWidth = 20`
  This parameter determines the width of text entry fields for the automatically generated GUI when interacting with work items.
- `portDiff = 2`
  This parameter specifies the offset for port numbers for the p2p network from

those specified in hosts.config. This must be chosen so as not to conflict with other applications. Note also that the ordering service uses the immediately adjacent port to the one specified in `hosts.config` for client communications. For example, when `hosts.config` specifies a port number N, this is the port used for server-server communication by the ordering service, and port N+1 is used for client-server communication by the ordering service.

A.2 Key Generation

Nodes communicate with each other using messages signed with digital signatures. These signatures are based on public and private keys, which must be generated for the chosen signature algorithm and signature provider. To generate keys for a particular node, run the following command from the application directory (where `<n>` is the node number):

```
java -jar CertificateAuthority.jar <n>
```

This command will read the `hosts.config` and `system.config` configuration files and generate keys based on that configuration. Elliptic curve based keys are generated in the folder `./config/ecdsakeys` while prime factorization (RSA) keys are generated in the folder `./config/keys`. Important:

- You must generate a key-pair for every node defined in `hosts.config`, even if this node is not joining the ordering view.
- You must manually distribute the public keys to all nodes in the system and you must manually distribute the private keys to each respective node

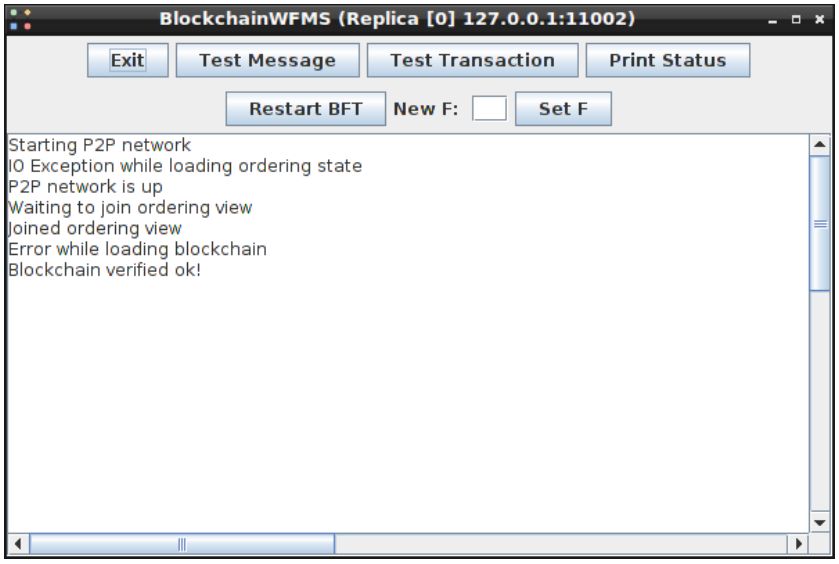
A.3 Launching the Application

Important The ordering service maintains a record of the last view. If you want to start the system with a fresh configuration, you must delete the file `currentView` in the `config` folder.

After configuration and key distribution, the configuration can be launched with the following command from the application directory (where `<n>` is the node number):

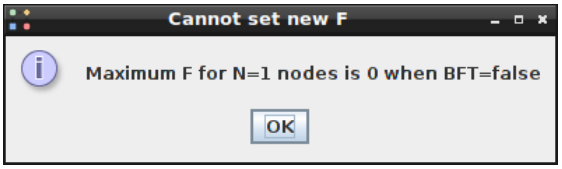
```
java -jar BlockchainWFMSBFTSmart.jar <n>
```

After a few seconds, the application will show two windows. The console window in the following screenshot is currently used for testing and experimentation and will be removed in future stable versions.

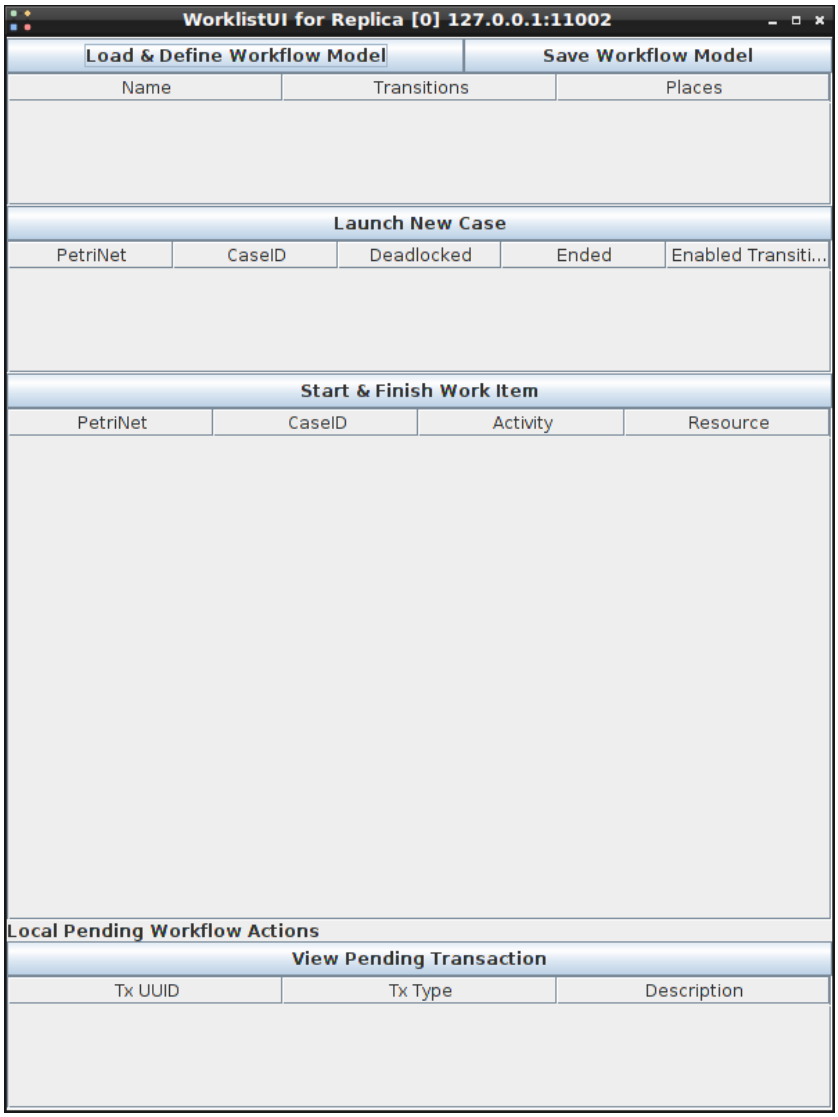


The example above shows a new node being started in a view of one in CT mode. The first line indicates that the P2P network is being started. This is followed by a message indicating that the state of the ordering service could not be loaded from an existing file, which can be ignored for now. Line 3 indicates that the P2P network is running, followed in line 4 by a message that the ordering service is waiting to join the current view. In this case, the initial view is configured with just this node, so that the system immediately joined the ordering view (line 5). Next, the application attempts to load existing blocks of the blockchain, in this case unsuccessfully (line 6) and verifies the blockchain. The empty blockchain verifies as "ok" (line 7).

The "Test Message" sends a ping message over the peer-to-peer network to connected nodes, the "Test transaction" message generates a non-workflow transaction and adds it to the ordering service transaction pool. The "Print Status" button prints the current blockchain and ordering service state to the console. The "Restart BFT" button will simulate a node crash and restart the ordering service. The ordering service will fetch state from other nodes in the current view and re-join the ordering system. A new level of fault tolerance can be set by entering a value for f and pushing the "Set F" button. The system will issue an error when the new value is too high for the current view (the current number of nodes).



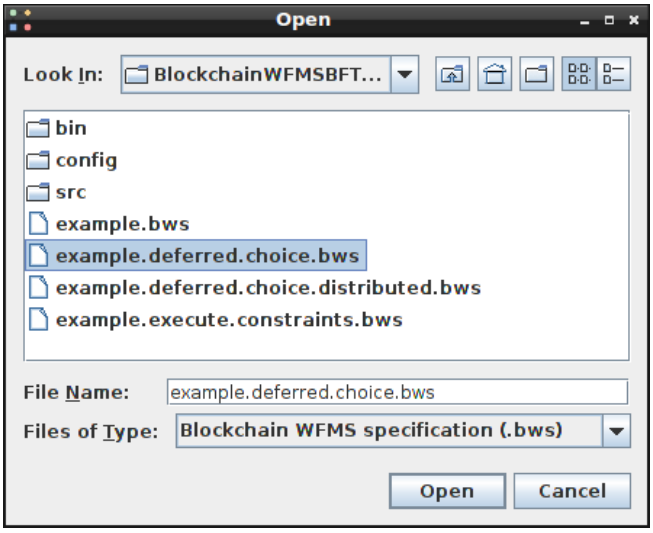
The second window shows the worklist. This includes information about the workflow specifications on the blockchain, about running cases, about tasks in the local nodes' worklist and about pending transactions (transactions created by the local node that are waiting to be included in a block).



Important Depending on how the initial view is configured in `system.config` and how many other nodes are running and connected, the application may wait at the "Waiting to join ordering view" message until all required nodes are running and have joined the view. The worklist window is only available once the ordering view is ready to process requests.

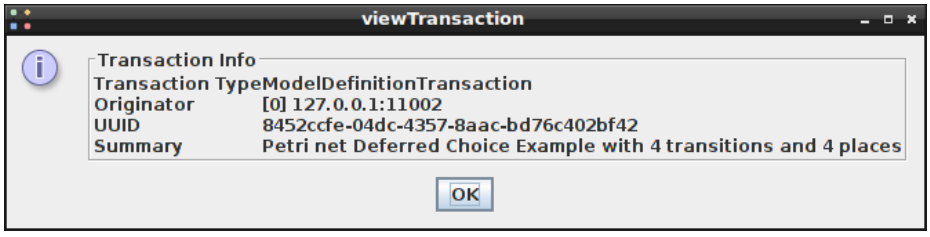
A.4 Loading Workflow Specifications

Using the button "Load & Define Workflow Model", a workflow specification can be loaded from a file and stored on the blockchain. The system comes with a few short pre-defined workflow specifications. Workflow specifications are XML files with the ending ".bws".



Defining a new workflow specification on the blockchain is a model creation transaction. This transaction may be pending until a new block is created. Until then, the workflow specification will not be available in the list of workflow specifications at the top of the worklist window. Instead, the pending transaction in the list at the bottom of the window may be selected and inspected using the "View Pending Transaction" button.

Tip If you need other transactions to fill a block, use the "Test Transaction" button in the console window.



Once a workflow specification is stored on the blockchain, it can be saved to a file or used to launch a new case against the specification.

A.5 Workflow Specifications

Workflow specifications are XML files. They define the basic structure of the Petri net

- Places
- Transitions
- Presets of transitions
- Postsets of transitions
- Source place
- Sink place

Additionally, they specify workflow-relevant information:

- Variables (name and datatype)
- Data constraints
- For each transition
  - Node (host and port)
  - Organizational role (not currently used)
  - Input variables
  - Output variables
  - External methods (Java class and static method)

The following are excerpts from a workflow definition file to illustrate the different aspects. The first fragment shows the XML declaration. The outer element "WorkflowSpecification" defines the name of the specification as an attribute. This name will be unique on the blockchain. Next, four places are defined with unique identifiers. *The identifiers must be valid Java UUIDs*

```
<?xml version="1.0" encoding="UTF-8"?>
<WorkflowSpecification name="Deferred Choice Example">
  <Places>
    <Place id="9185ac9c-69c8-4c44-b5db-c9f234d6475b"/>
    <Place id="d6eaceb8-8a0d-4bc4-b1de-86e4bcfb8b72"/>
    <Place id="f3e70f8f-f39d-49da-addc-5bfc805b8137"/>
    <Place id="7851738e-ce00-4b66-9e75-5f9cffc0dbcb"/>
  </Places>
  <Source id="9185ac9c-69c8-4c44-b5db-c9f234d6475b"/>
  <Sink id="d6eaceb8-8a0d-4bc4-b1de-86e4bcfb8b72"/>
```

The next fragment shows how transitions are defined. Transition with name "A" is scheduled for the node running at IP address 127.0.0.1 on port 11002. It has the variable with name "var1" as an output variable. Transition with name "B" is also scheduled for 127.0.0.1:11002 and has the variable with "input" as an input and the variable named "result" as an output. Rather than being scheduled for manual execution by the workflow system user, it automatically executes an external method call to class "java.lang.Math" and method "log".

```
<Transitions>
  <Transition name="A">
    <Host>127.0.0.1</Host>
    <Port>11002</Port>
    <Role>any</Role>
    <Outputs>
      <Output>var1</Output>
    </Outputs>
  </Transition>
  <Transition name="B">
    <Host>127.0.0.1</Host>
    <Port>11002</Port>
    <Role>any</Role>
    <Inputs>
      <Input>input</Input>
    </Inputs>
    <Outputs>
      <Output>result</Output>
    </Outputs>
    <Execute>
      <ExecClass>java.lang.Math</ExecClass>
      <ExecMethod>log</ExecMethod>
    </Execute>
  </Transition>
  <!-- more transitions here ... -->
</Transitions>
```

The flow relation of the Petri net is defined by specifying pre-sets and post-sets for transitions, as shown in the following fragment.

```
<Preset transition="A">
  <Place id="9185ac9c-69c8-4c44-b5db-c9f234d6475b"/>
</Preset>
<Postset transition="A">
  <Place id="f3e70f8f-f39d-49da-addc-5bfc805b8137"/>
</Postset>
<Preset transition="B">
  <Place id="f3e70f8f-f39d-49da-addc-5bfc805b8137"/>
</Preset>
<Postset transition="B">
  <Place id="7851738e-ce00-4b66-9e75-5f9cffc0dbcb"/>
</Postset>
  <!-- more presets and postsets here ... -->
```

Finally, variables for the entire Petri net are defined by specifying type, initial value and name. Constraints can be specified in the form of Java boolean expressions. These will be evaluated when the transaction is being validated.

```
<Variables>
  <Variable type="java.lang.String" init="I am a String">
    var1</Variable>
```

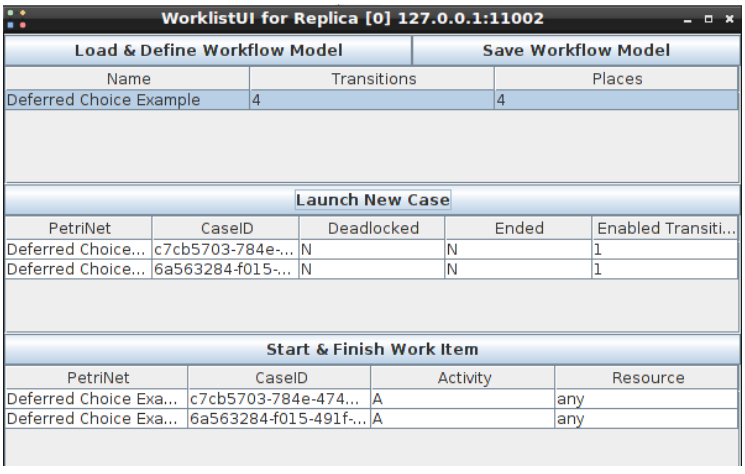
```
    <Variable type="java.lang.Double" init="1.0">
      input</Variable>
    <Variable type="java.lang.Double" init="0.0">
      result</Variable>
  </Variables>
  <Constraints>
    <Constraint>
      java.lang.Math.abs(result) != java.lang.Math.abs(input)
    </Constraint>
  </Constraints>
</WorkflowSpecification>
```

Important The system does *no sanity checking* on the XML specifications. You are responsible to make sure that:

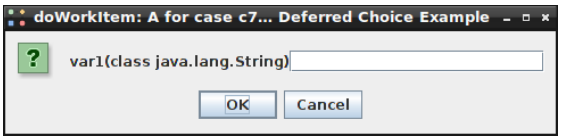
- All required elements are present and the XML has the correct structure
- Source, sink, pre-sets and post-sets refer to defined places
- Place identifiers are valid UUIDs
- Variable types are simple Java types
- Initial values are of the appropriate type
- Inputs and outputs of transitions refer to defined variables
- Executed methods are available on the Java classpath, take the appropriate input type(s) and produce the appropriate output type
- Constraints can be compiled at runtime and executed

A.6 Working with Cases and Work Items

To launch a case, select a specification and push the "Launch New Case" button. Launching a case will create an instance state transaction that is submitted to the ordering service. The transaction will remain pending until included in a new block. Once it is included in a block, the list of cases in the worklist window will be updated. If any enabled activities are scheduled for this node, they will show in the work item list.



To execute an activity, select the activity from the list and push the "Start & Finish Work Item" button. The workflow system will generate an appropriate user interface that displays any input values for this work item, and on which you can enter any output values.



Important The system does not provide any sanity checking on user input. If the entered value cannot be parsed to the appropriate type the output variable will be assigned a default (0 for numeric types).

Limitations For demonstration purposes, the system does not remove completed or deadlocked cases from the worklist UI. Also, workflow specifications cannot be made invalid and be removed from the list of available specifications. Both are simple future extensions.