



Empowering Business-Level Blockchain Users with a Rules Framework for Smart Contracts

Tara Astigarraga¹, Xiaoyan Chen², Yaoliang Chen³, Jingxiao Gu², Richard Hull¹✉, Limei Jiao², Yuliang Li⁴, and Petr Novotny¹

¹ IBM T.J. Watson Research Center, Yorktown Heights, USA
{asti,hull,p.novotny}@us.ibm.com

² IBM China Research Laboratory, Beijing, China
{xiaoyanc,gjxgu,jiaolm}@cn.ibm.com

³ Fudan University, Shanghai, China
yaoliangchen15@fudan.edu.cn

⁴ Megagon Labs, UC San Diego, San Diego, USA
yul206@eng.ucsd.edu

Abstract. The importance and adoption of Blockchain to support secure and trusted collaborations between businesses continues to grow. In today's practice, most Blockchain smart contracts (which capture the business processing logic) are written primarily by software developers. To enable widespread adoption of Blockchain, business analysts and subject matter experts will need to have direct access to the smart contract logic, including the abilities to understand, modify, and create substantial portions of that logic. This paper describes a fully functioning framework and system for specifying and executing smart contracts in which the core logic is specified by a controlled English, business-level rules language. The framework includes a browser-based smart editor for rules; a parser generator that enables substantial variation in the rules syntax; code generation that maps to a RETE based rules engine; and execution of the rules in either on-chain (using Hyperledger Fabric) or off-chain modes. The paper describes the rules framework and possible extensions, and identifies key aspects of Blockchain that impact the implementation.

1 Introduction

The shared ledger and Blockchain paradigms hold the promise of transforming the ways that businesses collaborate by enabling a single source of truth, and increased transparency through shared agreement about how business workflows will be conducted. In particular, “smart contracts”, i.e., the programs that guide the execution of transactions on Blockchain, are visible to and agreed upon by the participants in a Blockchain-enabled collaboration. As such, Blockchain for business collaboration opens new research challenges and industrial opportunities in service-oriented computing, in particular in the areas of new styles of business process management, distributed computing, and secure services. Today's smart

contracts are primarily created by software developers, using standard programming languages such as Golang, C++ or Java, and/or using domain-specific languages and frameworks such as Ethereum's Solidity [1] and Hyperledger's Composer [2]. However, because of the anticipated volume of business collaborations on Blockchain in coming years, and thus the volume of smart contracts to be created, it is paramount that business-level users be empowered to understand, create and modify smart contracts, or at least large portions of them. This paper describes a Business Collaboration Rules Language (BCRL) framework and implemented system, that enables business-level users to specify and maintain intricate business logic in Blockchain-enabled solutions. In particular, the framework enables the use of the same rules language that can be executed both as smart contracts on the Hyperledger Fabric [3], and in an off-chain rules engine, thereby enabling a more seamless experience for managing overall business collaboration solutions. This paper describes the rules framework, including illustrations of the current rules language, overview of the system architecture and extensibility, and discussion of the implications on the system related to implementation on top of the Hyperledger Fabric.

The importance of empowering business-level users in connection with smart contracts is highlighted in [4]. Recent articles have focused primarily on the business process level of smart contracts, e.g., by providing an implementation of BPMN on top of Ethereum [5,6] or describing how the business artifact approach can leverage the data-centric nature of Blockchain [7]. This paper complements that work by providing a framework for specifying intricate business logic through the use of business rules expressed in a controlled English. The integration of the BCRL framework with workflow-based smart contract frameworks is left as future work.

This paper illustrates the framework with a particular version of the rules language, called Business Collaboration Rules Language version 1.0 ("BCRL 1.0" or "BCRL" for short). This language is inspired largely from the BERL language of IBM's Operational Decision Manager (ODM) product [8]; this was a tactical design decision based in part on enabling more rapid creation of the first implementation. But the framework itself can support substantial extensions and variations to that language. For example, it could be adapted to follow the styles of other ODM languages, SBVR [9], the Oracle [10] and FlexRule [11] business rule languages, etc. One enabler for this is the use of the Business Domain Specific Language (BDSL) parser-generator, which is used to generate both the language parser and a browser-based smart editor; BDSL is an internal component [12] of the ODM product. BDSL includes multiple features specific to the creation of domain-specific languages (DSLs) that are based on controlled natural language, in particular around handling phrasing that would be ambiguous for a traditional LALR parser. A second enabler is the use of a lightweight, prototype, JavaScript-based rules engine called "nanoRETE", which supports the RETE algorithm for rules execution.

The rules engine framework has been implemented on the Hyperledger Fabric in two ways. One modality relies on the fact that Hyperledger Fabric v1.1

provides native support for JavaScript-based smart contracts. In this modality the nanoRETE rules engine and some integration modules are loaded directly into Fabric. The other modality takes advantage of the Hyperledger Composer [2] and runs on Hyperledger Fabric v1.0.

As shown by the examples in this paper, the BCRL framework enables business-level users to express and “own” large portions of the business logic underlying smart contracts. The language itself follows the spirit of other business-level rules languages and can be modified to fit popular syntactic styles, thereby enabling more rapid adoption. The primary contributions of the paper are the development of a framework that enables seamless use of such rules language both on- and off-chain, and the description of how the framework was brought into a fully implemented prototype system.

Section 2 provides an overview of the framework, illustrates BCRL 1.0, and discusses possible extensions. Section 3 overviews the system architecture and describes key components. Section 4 describes the main challenges that arise when embedding a rules framework onto Hyperledger Fabric and how the BCRL framework addresses them. Section 5 describes related languages and research, and Sect. 6 offers brief conclusions.

2 Framework Overview and Illustrations

This section provides an overview of the rules framework, then illustrates it with an example taken from the domain of billing for Technical Service Support (TSS), and concludes with a discussion of some additional rule constructs that can be incorporated into the framework. More details about the architecture, language, and implementation are provided in subsequent sections.

As mentioned in the Introduction, the framework is focused on enabling business-level users to specify and execute possibly intricate rules logic in a Blockchain-enabled solution. The framework enables the use of the same rules engine in two different ways – one on-chain and the other off-chain – to provide a more seamless experience for business-level users who are working on a comprehensive Blockchain-enabled solution.

Figure 1 shows the high-level architecture of the rules framework. As shown in the upper left, the framework includes a template-based editor for specifying the domain model for a given solution. (In some contexts, the domain model might be defined elsewhere, in which case it can be imported as JSON into the solution.) The Smart Editor for Rules is shown in the upper right of the figure.

The primary function of the Code Generation component is to perform code generation of executable rules based on the business-level rules and domain model, and then to deploy them in rules engines both on-chain and off-chain. The on-chain rules are triggered by transaction invocations, and typically result in updates to the ledger and to the Worldstate (which in our architecture is maintained in CouchDB), and may yield notifications about the transaction outcome. For off-chain rules, rules can be loaded into the off-chain Rules Engine Container for execution. This engine acts primarily as a Policy Decision Point, but can also read and update an off-chain database (currently, Cloudant).

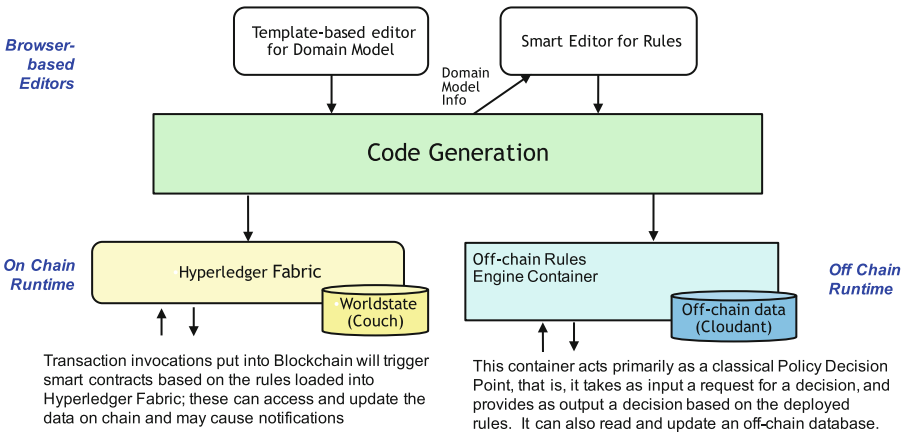


Fig. 1. High-level architecture of rules framework

2.1 Illustration of Rules in Billing Use Case

We now present a case study of the rules framework in action. The example was chosen in part to illustrate several different elements of the rules language currently supported by the framework. This is based on a real-world use case involving the generation of bills for a specific kind of Technical Support Services (TSS). In this example, IBM is providing maintenance for a client, called here ServerFarm, that operates numerous computer servers in data-centers spread across numerous countries. The payment for services may vary by country, and also by the level of service (in this case, either “Next Business Day” or “within 4 Hours”). The monthly bill for a given country and level of service is based on the number of machines being maintained. That number is in turn determined by examining the number of machines being maintained each week, and then taking the maximum weekly quantity for the month.

Figure 2 shows the two main process flows used to manage the Billing processing; these can be thought of as lifecycle models for the two primary entity types arising in the application, namely Weekly Usage Records and Monthly Billing Items. These lifecycle flows might be managed by a BPMN-based engine or other processing engine. We use here an informal, direct representation for these flows that includes constructs germane to Blockchain enabled solutions. This includes responding to events of a given type (black diamond), tasks performed off-chain (dashed line rounded box), and tasks performed on-chain (solid line rounded box). In our example some off-chain tasks are performed manually, and others performed automatically based on rule execution.

The key data sets are shown in Fig. 3. These are depicted more-or-less as tables, but in fact are collections of JSON documents that are stored in noSQL databases. The Install_Base table, which resides off-chain in a Cloudant database, holds data corresponding to contractual agreements between ServerFarm and its customers. Figure 4 shows a representative document, showing a hypothetical

event is shown in Fig. 5. This typically produces a record holding a weekly usage count for a given country and offering category. That data is used as input for a transaction request of type `insert_weekly_usage` sent to the Hyperledger Fabric, and results in the weekly usage data being loaded onto the Blockchain. Figure 6 shows a representative Hyperledger asset that would be written into the ledger as the outcome of that sequence of events.

Figure 9 illustrates the smart editor for rules and three of the rules used in response to a `compute_weekly_usage` event. The rules are in a controlled English, and the editor provides color coding for different syntactic elements.

```
{
  "customer_name": "GSG INC",
  "serial_number": "P3029287XA59288",
  "asset_sla": "NBD",
  "country_name": "China",
  "contract_start_date": "2016-11-22T17:00:00.000Z",
  "contract_end_date": "2017-11-21T07:00:00.000Z",
  "docType": "com.ibm.monthly_charge.Install_Base"
}
```

Fig. 4. Representative JSON document from `Install_Base`, which identifies one of the machines that `ServerHost` is maintaining for one of its customers. (Customer name is hypothetical.)

```
{
  "event_type": "compute_weekly_usage",
  "country_name": "China",
  "week_start_date": "2017-10-01T17:00:00.000Z",
  "week_end_date": "2017-10-08T17:00:00.000Z"
}
```

Fig. 5. Representative event, of type `compute_weekly_usage`, which would trigger an off-chain computation followed by an on-chain computation.

Rule 1.1 illustrates the overall structure of rules, which includes a “when” clause that refers to the type of event being processed, an “if” clause that includes conditions, and a “then” clause that holds one or more actions. In Rule 1.1 the “if” clause is testing whether the `offering_category` field of the incoming event is outside of the permitted values “NBD” or “4HR”.

Rule 1.2 provides a simple illustration of syntax checking by the smart editor. Here the keyword ‘is’ is missing from the “if” clause; this is indicated at the bottom of the screen, and also in a pop-up box if the user mouses over the erroneous text.

Rule 1.3 illustrates several features supported in BCRL 1.0. This includes a fourth building block for the rules, called “definitions”, that allows to define sets of records (shown in this figure) and to select individual records (see Fig. 8).

```

{
  "country": "China",
  "docType": "com.ibm.monthly_charge.Weekly_Usage_Record",
  "offering_category": "NBD",
  "quantity": 184,
  "week_start_date": "2017-10-01T17:00:00.000Z",
  "week_end_date": "2017-10-08T17:00:00.000Z",
  "weekly_usage_record_ID": "3b1f0eb0-2ad2-11e8-bb2b-2bd0e4e667c6"
}

```

Fig. 6. Representative value of Hyperledger asset (key-value pair) written onto the ledger as result of off-chain compute_weekly_usage event followed by on-chain insert_weekly_usage transaction

In Rule 1.3 a set designated with variable name “the usage_records” is built from the Install_Base database. A representative document from Install_Base is shown in Fig. 4. The rule builds “the usage_records” using a combination of ‘and’ (“all of the following are true”) and ‘or’ (“at least one of the following are true”) constructs. This rule also illustrates an action of creating a new record and writing it into a database. The built-in function “sum” is used to take a sum of values from the quantity field of the records in ‘the usage_records’.

```

-- Rule 2.4
when an insert_weekly_usage event occurs
if all of the following are true:
- the offering_category of this event is in ("NBD", "4HR")
- the country of this event is in
  ("China", "India", "Indonesia", "Japan", "Korea", "Philippines", "Singapore", "Taiwan", "Thailand")
- the quantity of this event is greater than 0
- there is no 'Weekly_Usage_Record' as 'the weekly_usage_record' where
  all of the following are true:
  - the week_start_date of 'the weekly_usage_record' is the week_start_date of 'this event'
  - the week_end_date of 'the weekly_usage_record' is the week_end_date of 'this event'
  - the offering_category of 'the weekly_usage_record' is the offering_category of 'this event'
  - the country of 'the weekly_usage_record' is the country of 'this event';
then
create a new 'Weekly_Usage_Record' as 'the weekly_usage_record'
set the week_start_date of 'the weekly_usage_record' to the week_start_date of 'this event'
set the week_end_date of 'the weekly_usage_record' to the week_end_date of 'this event'
set the offering_category of 'the weekly_usage_record' to the offering_category of 'this event'
set the country of 'the weekly_usage_record' to the country of 'this event'
set the quantity of 'the weekly_usage_record' to the quantity of 'this event';

```

Fig. 7. Rule for inserting a weekly usage record into blockchain

The outputs of firings of Rule 1.3 will be used as the payload for insert_weekly_usage transactions on the Hyperledger Fabric. Such transactions may result in the firing of Rule 2.4 shown in Fig. 7, (The first digit in the rule numbering scheme correspond to the different types of events/transactions that can lead to rule firing.) This rule shows another capability of BCRL, specifically the ability to perform a “not exists”, or said differently, to check that there are no records satisfying a certain property. In this case we check that there is no Weekly_Usage_Record asset already on-chain that corresponds to the same country, offering category, and week. (Updates to an existing Weekly_Usage_Record


```

-- Rule 5.1
when a compute_monthly_charge event occurs
definitions
  find all from 'Weekly_Usage_Record' as 'the weeks_in_month' where
    all of the following are true:
      - the week_start_date of 'the weeks_in_month' is between
        the month_start_date of 'this event' and the month_end_date of 'this event'
      - the offering_category of 'the weeks_in_month' is the offering_category of 'this event'
      - the country of 'the weeks_in_month' is the country of 'this event';
  find one from 'Rate_Table' as 'the category_rate_record' where
    the country of 'the category_rate_record' is the country of 'this event';
    the offering_category of 'the category_rate_record' is the offering_category of 'this event';
if all of the following are true:
  - 'the weeks_in_month' is not empty
  - 'the category_rate_record' is defined
then
  create a new 'Monthly_Billing_Item' as 'the monthly_billing_item'
  set the offering_category of 'the monthly_billing_item' to the offering_category of 'this event'
  set the country of 'the monthly_billing_item' to the country of 'this event'
  set the max_weekly_count of 'the monthly_billing_item' to the maximum value of the quantity of 'the weeks_in_month'
  set the offering_service_rate of 'the monthly_billing_item' to the offering_service_rate of 'the category_rate_record'
  set the monthly_charge of 'the monthly_billing_item' to the
    max_weekly_count of 'the monthly_billing_item' * the offering_service_rate of 'the monthly_billing_item'
  set the month_start_date of 'the monthly_billing_item' to the month_start_date of 'this event'
  set the month_end_date of 'the monthly_billing_item' to the month_end_date of 'this event'
  set the weekly_usage_records of 'the monthly_billing_item' to 'the weeks_in_month';

```

Fig. 8. Rule used to compute monthly charge on blockchain

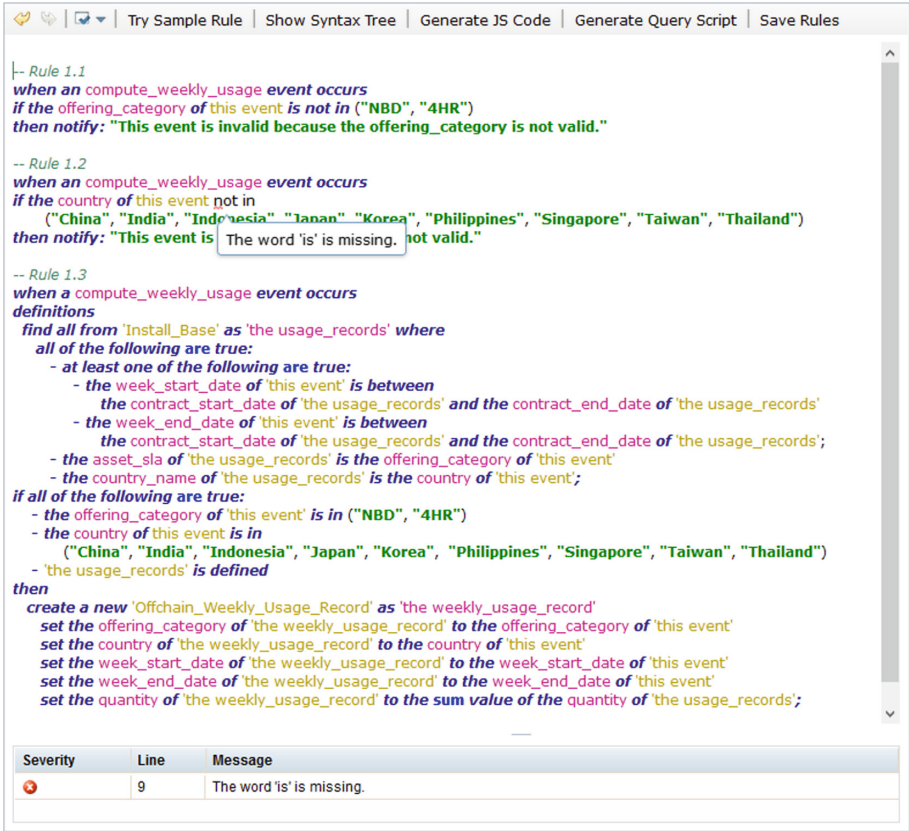
asset can be made using an update_weekly_usage transaction request.) As noted above, Fig. 6 shows a representative Hyperledger asset that will be written onto the ledger after Rules 1.3 and 2.4 have fired.

We pause to comment on the “when” clause in BCRL rules. This is used primarily to provide a clustering and modularity for the overall set of rules. In essence, if an event of a given type is pushed into the rules engine, then all rules with “when” clause referring to that type are eligible for firing, including as the result of rule chaining.

Finally, we describe Rule 5.1 in Fig. 8, which computes Monthly_Billing_Item assets. The rule focuses on all weeks which start within the month. This rule illustrates the construct for selecting a single element from the Rate_Table, using the “find one from” construct and also checking that the resulting record is “defined”, which in this case includes a check that only one record was found. The “then” clause illustrates the use of built-in arithmetic functions and the aggregate operator “max”. Note also that the definition of one field value of ‘the monthly_billing_item’, specifically the “monthly_charge” value, can refer to previously defined field values of ‘the monthly_billing_item’. A representative asset produced by this rule is shown in Fig. 10.

2.2 Discussion

As illustrated in the examples above, BCRL 1.0 provides business-level users with the ability to express a broad variety of data manipulations, mainly in the area of accessing and manipulating documents (off-chain) and assets (on-chain), and also lists of such objects. Although not highlighted in the above examples, it is important to note that the code generation maps the rules into the nanoRETE engine, which can support rich chaining of the rules.



```

|-- Rule 1.1
when an compute_weekly_usage event occurs
if the offering_category of this event is not in ("NBD", "4HR")
then notify: "This event is invalid because the offering_category is not valid."

-- Rule 1.2
when an compute_weekly_usage event occurs
if the country of this event not in
  ("China", "India", "Indonesia", "Japan", "Korea", "Philippines", "Singapore", "Taiwan", "Thailand")
then notify: "This event is The word 'is' is missing. not valid."

-- Rule 1.3
when a compute_weekly_usage event occurs
definitions
  find all from 'Install_Base' as 'the usage_records' where
    all of the following are true:
    - at least one of the following are true:
      - the week_start_date of 'this event' is between
        the contract_start_date of 'the usage_records' and the contract_end_date of 'the usage_records'
      - the week_end_date of 'this event' is between
        the contract_start_date of 'the usage_records' and the contract_end_date of 'the usage_records';
    - the asset_sla of 'the usage_records' is the offering_category of 'this event'
    - the country_name of 'the usage_records' is the country of 'this event';
  if all of the following are true:
  - the offering_category of 'this event' is in ("NBD", "4HR")
  - the country of this event is in
    ("China", "India", "Indonesia", "Japan", "Korea", "Philippines", "Singapore", "Taiwan", "Thailand")
  - the usage_records' is defined
then
  create a new 'Offchain_Weekly_Usage_Record' as 'the weekly_usage_record'
  set the offering_category of 'the weekly_usage_record' to the offering_category of 'this event'
  set the country of 'the weekly_usage_record' to the country of 'this event'
  set the week_start_date of 'the weekly_usage_record' to the week_start_date of 'this event'
  set the week_end_date of 'the weekly_usage_record' to the week_end_date of 'this event'
  set the quantity of 'the weekly_usage_record' to the sum value of the quantity of 'the usage_records';

```

Severity	Line	Message
✖	9	The word 'is' is missing.

Fig. 9. Illustration of smart rules editor

While BCRL can be used as-is to provide decision support for a variety of applications, there are multiple simplifications and extensions that would be beneficial; we mention some of these here.

BCRL 1.0 is quite verbose. This was an appropriate first step because it provides a completely explicit way for expressing the constructs, a feature that will be useful in contexts where more abbreviated variants may be confusing for some users.

An important streamlining currently underway is to enable the parser and code generation to take advantage of the meta-data about the data sets and the event signatures. For example, in Rule 1.3 (Fig. 9) this would allow for replacing the phrase “the week_start_date of ‘this event’” by “the week_start_date”, since the only relevant object with that attribute is ‘this event’.

In terms of extensions, we see considerable value in enabling a construct of form “then for each <variable> in <defined list>”. Inside that would be a

full “definitions-if-then” block, which is to be executed for each element of the defined list.

Another extension would be to permit richer modularity in the specification of rule sets. For example, we could imitate a paradigm found in ODM, which is to allow the specification of several sets of rules that are connected into a flowchart.

3 Implementation of Rules Framework

This section describes the primary components of the BCRL framework, including the smart editor, the parsing, the code-generation, and the deployment both on-chain and off-chain. Some technical considerations specific to operating on Hyperledger Fabric are deferred until the next section.

The rules framework architecture separates execution code of business rules from the other application components, such as rule scheduling, access control and business data storage. This separation helps reduce the costs of application maintenance by allowing the business users to modify the rules as necessary without the need for other code changes. It also allows for maximum re-use of components across on-chain and off-chain rules specification, deployment and execution.

```
{
  "country": "China",
  "docType": "com.ibm.monthly_charge.Monthly_Billing_Item",
  "max_weekly_count": 187,
  "month_end_date": "2017-10-31T17:00:00.000Z",
  "month_start_date": "2017-10-01T17:00:00.000Z",
  "monthly_billing_item_ID": "7cf06c30-2ad2-11e8-bb2b-2bd0e4e667c6",
  "monthly_charge": << omitted -- proprietary >>
  "offering_category": "NBD",
  "offering_service_rate": << omitted -- proprietary >>
  "weekly_usage_records": [
    {
      "country": "China",
      "docType": "com.ibm.monthly_charge.Weekly_Usage_Record",
      "offering_category": "NBD",
      "quantity": 184,
      "week_end_date": "2017-10-08T17:00:00.000Z",
      "week_start_date": "2017-10-01T17:00:00.000Z",
      "weekly_usage_record_ID": "3b1f0eb0-2ad2-11e8-bb2b-2bd0e4e667c6"
    },
    {
      "country": "China",
      "docType": "com.ibm.monthly_charge.Weekly_Usage_Record",
      "offering_category": "NBD",
      "quantity": 187,
      "week_end_date": "2017-10-15T17:00:00.000Z",
      "week_start_date": "2017-10-08T17:00:00.000Z",
      "weekly_usage_record_ID": "51cb5920-2ad2-11e8-bb2b-2bd0e4e667c6"
    },
    ...
  ]
}
```

Fig. 10. Part of representative asset written onto the ledger as a result of a `compute_monthly_charge` transaction. (The rate and monthly charge are proprietary so omitted.)

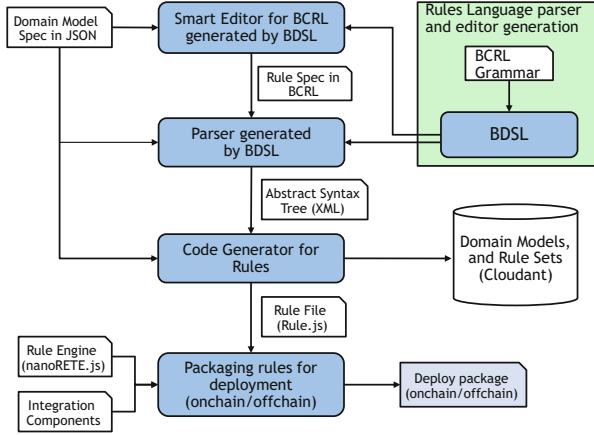


Fig. 11. Components that support the rules language

An overview of the rule system generation architecture is shown in Fig. 11. The four main components (forming a vertical column in the center of the figure) act as a pipeline that maps BCRL rules into executable code.

The browser-based smart editor (top-most component) and the parser (just below that) were created using the BDSL plugin [12] from IBM’s ODM product [8]. BDSL can support grammars that capture highly flexible domain-specific languages based on controlled English (or other natural languages). The smart editor was illustrated in Fig. 9 above. The Smart Editor also incorporates information about the domain model and types of events.

The parser produces an Abstract Syntax Tree (AST) based on BCRL. This AST, along with the domain model information, serves as the input for the code generator. This is the key component in this architecture, which generates runnable JavaScript rule objects from the AST. In order to make the generated rule objects that can be run on both on-chain and off-chain environments, a utility interface is abstracted to isolate the difference between on-chain and off-chain. The main difference is related to database operations. The database operations for on-chain are using Fabric APIs, such as `getState`, `putState`, `getQueryResult`, to access the Hyperledger Fabric Worldstate, which in our case is maintained by a CouchDB instance. (See also Sect. 4.) In contrast, the database operations for off-chain are using Cloudant APIs, such as `insert` and `find`, to access the Cloudant database. Therefore, we use a variable named handler to represent the utility interface in generated JavaScript rule objects. Both on-chain and off-chain provide their own utility implementation in runtime.

The code generation process includes the following steps: (1) Traverse the input AST to construct the rule set using the internal rule structure. (2) Generate JavaScript rule objects from the rule sets using templates. (3) Store the generated JavaScript rule objects into a file named `rules.js` for deployment into the rules

engine. (4) Store the generated JavaScript rule objects into the Cloudant if required.

For the rule engine itself we use nanoRETE, a lightweight prototype JavaScript engine that supports the RETE algorithm. (This can run natively on Hyperledger Fabric v1.1, and can also run on Hyperledger Fabric v1.0 with Composer and some integration modules.) The rule engine runs in a data-driven approach. The context of an event changes, which may result in one or more rules being concurrently eligible and scheduled for execution. The rule execution may include querying business data, making computations and finally arriving at some conclusions, possibly including notifications and/or database updates.

The bottom component is in charge of generating deployable packages. The deployable packages include not only the generated rule file, but also supported components, such as the rule engine and utility for database operations. According to on-chain and off-chain requirements, the deployment packaging will choose the right components for packaging. The output packages can be deployed to on-chain or off-chain environments.

The framework also provides a family of straightforward APIs to interact with the deployed on-chain and off-chain rule engines. This includes APIs to deploy rules and also to invoke rules processing. In the on-chain runtime, the APIs communicate with Fabric Client SDK by gRPC and invoke smart contracts on Hyperledger Fabric Blockchain. In the off-chain runtime, the APIs invoke the business rules code directly by gRPC.

4 Implications of Execution on Hyperledger

In this section, we describe three aspects of the Hyperledger Fabric that must be considered, when embedding a rule engine into smart contracts. We first provide a high-level overview of the Fabric architecture and of the transaction processing flow within which the Smart contract is executed. We then highlight the implications and challenges of these to the design of the BCRL framework.

4.1 Fabric Architecture Overview and Transaction Processing Flow

Hyperledger Fabric provides the combination of the immutable shared ledger, high performance of transaction processing, and security and privacy features of private Blockchain. It is architected as a highly distributed system network consisting of several specialized types of components. Note that this architecture differs from standard Blockchain networks such as Ethereum, where all the functionality is integrated into the peer. The center element is the ledger consisting of a chain of blocks, which store the history of transactions along with the identity of the transactions' submitters and endorsers.

The submission of a transaction into the Fabric follows a precise sequence of steps. Initially, a client node submits a "transaction proposal" containing the arguments and name of the function to invoke in the smart contract, to

the endorsing peers (i.e., peers which aside from the ledger, host the smart contract). Upon reception, a peer invokes the smart contract in context of the peer’s “Worldstate”. The Worldstate (similarly as the “state trie” in Ethereum) provides fast and efficient access to the data recorded in the ledger. Next, the client node collects the response along with an endorsement (i.e., the digital signature of the result) from each of the endorsing peers. Note that no modifications of the ledger or Worldstate were made at this point. The client node then validates that the responses are matching and that the collected endorsements adhere to the endorsement (i.e., consensus) policy of the Blockchain network, and submits them to the “ordering service”. The ordering service is a centralized component, which receives endorsed transaction proposals, orders them into blocks, and submits the blocks to all peers of the Blockchain network. Upon reception of a block, the peer validates the transactions of the block based on the endorsement policy, commits the block into its ledger and projects the content of the block’s transactions onto its Worldstate. Only at this point, the transaction and the state modifications it introduces are added to the ledger.

In the context of the transaction processing flow, a smart contract of our Rule-based framework is hosted on the endorsement peers of the network and executed in response to the transaction proposals submitted by the client nodes. The distributed nature of the network and the sequence of the transaction processing steps has implications on the implementation of the of the smart contract functions and the availability of the data during the smart contract execution. In the following, we outline the key points.

4.2 Eliminating Non-determinism from Smart Contracts

The consensus in the Fabric network is centered on the comparison of the results of execution of the smart contract on multiple peers. Only if the results of the execution match among the required peers, the resultant transaction will be added to the ledger. Thus, to produce a matching outcome, the execution of the smart contract must lead to a deterministic result (i.e., given a set of arguments and a state of the ledger) the smart contract hosted on any of the network peers, must produce the same result.

In Hyperledger Fabric, the smart contract programming infrastructure of golang and JavaScript languages does not prevent the execution of any functions including those, which may produce a non-deterministic output (e.g., `new Date()`, `process.hrtime()`, `Math.random()`, etc.). Therefore, unaware or mistaken use of these functions may lead to a faulty behavior of the smart contract. It is thus the responsibility of the software engineer to ensure that the use of these functions is avoided.

Our Rule-based Framework for Smart contracts prevents producing non-deterministic results similarly as the Solidity language of Ethereum; by eliminating the non-deterministic functions from the set of features available in the grammar. Furthermore, the execution instructions produced by the Codegen are based only on deterministic functions.

This implementation leads to minor restrictions on the smart contract capability. When an output of a non-deterministic function is needed to be included as part of a transaction; the function is executed once, prior to execution of the smart contract. The non-deterministic function execution can be implemented either in the middleware components prior to submitting a transaction proposal to the peers or in the client node invoking the middleware. In either case, the output of the non-deterministic function becomes an argument of the execution of the smart contract, and the same value is submitted to all peers of the network.

4.3 Blockchain Phantom Reads

The invocation of a smart contract triggers a “simulation” of transaction execution. During the processing, invocations of the functions modifying the state of the ledger are recorded into the WriteSet of the resultant transaction and do not impact the state of the ledger and the Worldstate. In effect, during execution of a single transaction, after the modification of an asset (i.e., key and value pair), the subsequent request for the value of the asset will return the unmodified asset value (i.e., phantom read).

The Rule-based framework is designed for smart contracts consisting of long sequences of rules repeatedly modifying and accessing the same assets, incrementally working towards the final result of the transaction. Thus, due to the phantom read behavior, using the native functions returning unmodified values would lead to faulty results. To support the needed functionality we are now encapsulating the Fabric API such that the Worldstate is integrated with an additional caching mechanism. This will allow for accessing the latest modifications necessary for incremental result building. At the same time, this mechanism allows obtaining the currently valid, unmodified values stored in the Worldstate with additional functions available in smart contract processing when needed.

4.4 Worldstate Indexes

In many smart contract contexts the queries against the Worldstate are focused on requesting data based on a key of an asset. However, it is typical of many rules to include associative queries against the Worldstate, that retrieve assets based on properties and ranges of different attributes. The performance of these queries is bound to the number of assets stored in the Worldstate and declines with increasing size of the ledger.

To improve the speed of the asset retrieval, Fabric provides the ability to define custom indexes along with the smart contract. However, it depends on the skill of the smart contract designer, whether and which indexes will be defined as well as how well these will support the execution of the smart contract queries.

The Rule-based grammar provides the opportunity for automated static code analysis of the smart contract, necessary to automatically determine the indexes needed for efficient execution of the associative queries in the rules. This is because the Rule-based grammar leads to a clear expression of the sequences

of data access operations as well as it explicitly includes the names of the keys (either the unique keys of the assets or the paths within the assets when the assets contain JSON) used to search for the data.

The index generating algorithm would have two main steps:

1. Identification of candidate keys - in this step the data access operations are analyzed to determine which keys are good candidates for indexes.
2. Generation of indexes - in this step the candidate keys transformed into definitions of the database indexes.

The generation of the indexes can be executed as part of the Codegen and will allow for transparent optimization of the overall performance of the rule-based smart contract.

5 Related Work

In the early days, all smart contracts were programmed using full-purpose programming languages such as Golang, C++ or Javascript.

Solidity [1] is more specifically targeted for the creation of smart contracts on Ethereum. It is essentially a Turing-complete language based on C++ and JavaScript-like syntax. Therefore, the language targets professional developers rather than business users. In particular, expressing rules-style logic in Solidity involves the use of ‘if’ and ‘else’ statements along with a set of braces containing the rule specific code. Moreover, the chaining of rules has to be coded explicitly. In contrast, the BCRL framework supports a business level DSL based on controlled English, and the rules chaining is supported implicitly with a general-purpose RETE engine.

Some of the recent and emerging DSLs and environments for smart contracts are intended for the software developer community. This includes the Hyperledger Composer [2], which includes abstractions for “assets” (business-relevant entities that whose representations are manipulated onchain), “participants” and “transactions” (which manipulate the assets). As noted in [7], it would be natural to extend the Composer notion of assets with lifecycle models, in the sense of Business Artifacts and Case Management. The Obsidean language [13] focuses on abstractions for linear types and state-machine-based object lifecycles, in part to reduce errors in smart contracts and to facilitate verification.

The R3 Corda initiative is developing an approach designed to support the creation of smart contracts for Financial Services [14].

Citation [5] focuses on empowering business-level users to create and understand smart contracts, by showing how the BPMN standard can be implemented on top of Ethereum. This has led to the open-source Caterpillar system [6].

There is a long tradition of extending traditional business process management systems with rules engines, to provide more flexibility to the business-level users who maintain the processing logic. In a recent development the IBM ODM rules engine has been integrated with Hyperledger, enabling on-chain smart contracts to use REST APIs to invoke an ODM engine for decision support [15].

While the ODM engines are not running directly within the smart contract, they are resident on the peer nodes that are executing the smart contracts, and can thus be tightly integrated with the immutability, privacy and consensus-based features of the Hyperledger Fabric.

The BCRL language presented here is inspired from the BERL rules language, one of several supported by the ODM product. Another Controlled-English rules language is found within the Semantics of Business Vocabulary and Rules (SBVR) model [9], an Object Management Group (OMG) standard that provides the vocabulary and syntax for documenting the semantics of business vocabularies, business facts, and business rules. As noted in the introduction, BCRL could be adapted to follow the style of SBVR and/or commercially available business rule languages [8, 10, 11]. The semantics of SBVR is a version of higher-order logic that can be machine-processed and automatically analyzed [16]. The expressive powers of BCRL and SBVR are not comparable. SBVR supports deontic logic operators for expressing business rules in the style of obligations and permissions; these are not supported by BCRL. On the other hand, SBVR does not have an operational semantics (e.g. to express updates to the business facts) so it is not directly executable on Blockchain as allowed by BCRL. We believe that the BCRL framework could be extended to support most or all SBVR's expressive power.

6 Conclusions

This paper presents the BCRL framework that enables business-level users to specify and deploy business rules as smart contracts on the Hyperledger Blockchain fabric. The framework considers the use of Blockchain in the larger context of business collaborations, and enables use of the same rules language to be executed on-chain in smart contracts, and executed off-chain in business processes that are hosted by individual collaboration stakeholders. As such, the paper provides a key building block for empowering business-level users to program and manage intricate business logic for business collaborations, that are supported in a secure, distributed, service-oriented manner. Section 4 discussed key aspects of Blockchain that our rules implementation needs to address.

The BCRL framework as presented here is focused largely on decision support and associated updates to persistent data (both on-chain and off-chain). It will be valuable to explore approaches for integrating BCRL with BPMN-oriented and business artifact-oriented smart contract frameworks, to provide them with rich decision support.

Verification of smart contracts has become a major desired feature in Blockchain platforms like Ethereum [17, 18] due to the high impact of vulnerabilities in smart contracts (e.g. \$50 million USD caused by the DAO hack [19]). Specifying smart contracts in a rule language like BCRL not only enables smart contract programming by business users but also introduces new opportunities for formal verification. This is because unlike common smart contract languages which are Turing-complete in general (so verification is undecidable),

the expressive power of BCRL is close to first-order logic (FOL), making formal verification more feasible. In fact, the research on automatic verification of data-centric business processes (see [20] for a survey) has followed this trend and shown decidability results on various expressive specification models based on FOL. Implementation of a verifier for data-centric business processes was recently shown successful [21]. These results indicate that BCRL can be a good starting point for future research on smart contract verification.

Acknowledgements. The authors are grateful to Stephane Mery, Philippe Bonnard and Jean Michel Bernelas from the ODM product group at IBM for their inspiration around rules languages and Blockchain, and also for assistance with the nanoRETE rules engine and the BDSL parser-generator. The authors are grateful to Jerome Simeon for his numerous insights, especially in connection with the spectrum of design and engineering issues that arise in Domain-Specific Languages. The authors are also grateful to the team at IBM working on Blockchain for Technical Support Services, including Saurabh Sinha, Nerla JeanLouis, and Shu Tao, for providing an environment and grounded use cases for the exploration of business rules for smart contracts.

References

1. Ethereum: Solidity (2018). <https://solidity.readthedocs.io/en/v0.4.21/>. Accessed 17 Mar 2018
2. IBM: Hyperledger Composer Home Page. <https://www.hyperledger.org/projects/composer>. Accessed 17 Mar 2018
3. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, p. 30. ACM (2018)
4. Mendling, J., et al.: Blockchains for business process management-challenges and opportunities. *ACM Trans. Manag. Inf. Syst. (TMIS)* **9**(1), 4 (2018)
5. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted business process monitoring and execution using blockchain. In: La Rosa, M., Loos, P., Pastor, O. (eds.) *BPM 2016*. LNCS, vol. 9850, pp. 329–347. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_19
6. López-Pintado, O., et al.: Caterpillar: a blockchain-based business process management system. In: Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling, *BPM 2017*, Barcelona, Spain, 13 September 2017
7. Hull, R., Heath III, F.F.T., Vianu, V., Batra, V.S., Chen, Y.M., Deutsch, A.: Towards a shared ledger business collaboration language based on data-aware processes. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) *ICSOC 2016*. LNCS, vol. 9936, pp. 18–36. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46295-0_2
8. IBM: IBM Operational Decision Manager. <https://www.ibm.com/automation/software/business-rules-management>. Accessed 17 Mar 2018
9. OMG: Semantics of business vocabulary and business rules (SBVR), version 1.4 (2017). <http://www.omg.org/spec/SBVR/1.4/PDF>. Accessed 17 Mar 2018
10. Oracle: Fusion middleware designing business rules with Oracle business process management (2016). <https://docs.oracle.com/middleware/12212/bpm/rules-user/toc.htm>. Accessed 4 Aug 2018

11. Aghlara, A.: Business rule language (of FlexRule) (July 2015). <http://www.flexrule.com/archives/business-rule-language/>. Accessed 4 Aug 2018
12. IBM: IBM Operational Decision Manager Version 8.6.0: Business Rules Embedded developer guide (2014). <http://www-01.ibm.com/support/docview.wss?uid=swg21660485&aid=1>. Accessed 17 Mar 2018
13. Coblenz, M.: Obsidian: a safer blockchain programming language. In: Companion Proceedings of the 39th International Conference Software Engineering, ICSE Companion 2017 (2017)
14. Brown, R.G., et al.: Corda: an introduction. R3 CEV, August 2016
15. Mery, S., et al.: Make your blockchain smart contracts smarter with business rules. <http://ibm.biz/odm-blockchain>. Accessed 17 Mar 2018
16. Marinos, A., Krause, P.: An SBVR framework for RESTful web applications. In: Governatori, G., Hall, J., Paschke, A. (eds.) RuleML 2009. LNCS, vol. 5858, pp. 144–158. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04985-9_15
17. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96. ACM (2016)
18. Amani, S., et al.: Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: CPP. ACM (2018, To appear)
19. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
20. Deutsch, A., et al.: Automatic verification of database-centric systems. ACM SIGLOG News **5**(2), 37–56 (2018)
21. Deutsch, A., et al.: Verifas: a practical verifier for artifact systems. In: VLDB (2017)