entropy

MDPI

Article

Logic Programming with Post-Quantum Cryptographic Primitives for Smart Contract on Quantum-Secured Blockchain

Xin Sun [1], Piotr Kulicki [1,*] and Mirek Sopek [2]

[1] Department of the Foundations of Computer Science, The John Paul Catholic University of Lublin, 20-502 Lublin, Poland; xin.sun.logic@gmail.com
[2] MakoLab SA, 91-062 Lodz, Poland; sopek@makolab.com
* Correspondence: kulicki@kul.pl

Abstract: This paper investigates the usage of logic and logic programming in the design of smart contracts. Our starting point is the logic-based programming language for smart contracts used in a recently proposed framework of quantum-secured blockchain, called Logicontract (LC). We then extend the logic used in LC by answer set programming (ASP), a modern approach to declarative logic programming. Using ASP enables us to write various interesting smart contracts, such as conditional payment, commitment, multi-party lottery and legal service. A striking feature of our ASP implementation proposal is that it involves post-quantum cryptographic primitives, such as the lattice-based public key encryption and signature. The adoption of the post-quantum cryptographic signature overcomes a specific limitation of LC in which the unconditionally secure signature, despite its strength, offers limited protection for users of the same node.

Keywords: logic programming; quantum blockchain; smart contract

Citation: Sun, X.; Kulicki, P.; Sopek, M. Logic Programming with Post-Quantum Cryptographic Primitives for Smart Contract on Quantum-Secured Blockchain. *Entropy* **2021**, *23*, 1120. https://doi.org/10.3390/e23091120

Academic Editor: Peter Harremoës

Received: 23 July 2021
Accepted: 23 August 2021
Published: 28 August 2021

A blockchain is a distributed, transparent and append-only chain of cryptographically linked units of data (blocks) stored in a large decentralized network. Due to the mechanisms of introducing new data based on consensus, blockchain can be used by peers who do not trust each other. The data entries can be considered transactions, so a blockchain can be treated as a ledger. Peers in charge of updating the ledger (called often miners) have separated, identical copies of the ledger. This fact makes the system distributed and is crucial for its safety. Recently Bitcoin [1] and other cryptocurrencies made the blockchain technology widely known. Another important and still not fully utilized way of using blockchain is the implementation of smart contracts [2] on their basis. Smart contract is a piece of software that implements an agreement between parties in such a way that terms of the contracts are enforced automatically. Due to registering them in a blockchain, smart contracts are irrefutable, which makes them appropriate for mutually distrusting peers. The main advantage is that a trusted third party is not needed for the affirmation and enforcement of contracts.

The advantages of smart contracts reside in the fact that existing blockchain platforms provide an infrastructure for them. In most cases, procedural languages are supported for smart contracts. However, logic-based languages for smart contracts seem to provide some advantages over the procedural approach [3,4]. The main ones are the following:

- Logical programs in general and smart contracts among them are better suited for formal verification than procedural programs. In the case of procedural programs, a usual way to proceed is to construct a formal calculus with rigorous semantics and express a program as a set of expressions of that calculus [5,6]. Logic is formal calculus itself, so there is no need for translation to other systems, and the verification is easier.
- Logical contracts are usually more compact. In contrast to their procedural counterparts, they are limited only to what has to be done, without specifying how to achieve it.

- Expressing contracts in a logical language is less error prone [7], as they are much closer to the user-friendly specifications than the procedural programming languages.

This paper provides investigations on the usage of logic and logic programming in the design of smart contracts. Our starting point is the logic-based programming language for smart contracts used in the recently proposed framework of Logicontract (LC) [8]. We then extend the logic used in LC by answer set programming (ASP), a modern approach to truly declarative logic programming. Using answer set programming enables us to write various interesting smart contracts. Moreover, due to the well-defined and rigorous syntax and semantics, the contracts written in the ASP language are easier to understand and formally verify.

We assume that the underlying blockchain is a quantum-secured permissioned blockchain, such as LC. More specifically, we assume the existence of a quantum communication network. In this network, every node is a classical computer. It would be better if nodes were quantum computers. In fact, some interesting problems, such as voting, lotteries and auctions, can be solved in an unconditionally secure manner on a blockchain with quantum computers as nodes [9,10]. However, there are still plenty of interesting problems that can be solved on quantum-secured blockchain with classical computers. Nodes are connected by both classical and quantum channels such that unconditionally secure keys between each pair of nodes can be successfully established. Nodes are also the participants (sender or receiver) of transactions. Every transaction is signed by its sender, using a quantum key distribution-based signature scheme. Such a signature is unconditionally secure. Every node maintains a record of all transactions. The consensus algorithm of the blockchain ensures that different nodes have an identical record of transactions.

So far, in LC, there is only unconditionally secure signature (USS), but no public-key signature. While USS is good at protecting messages communicated between different nodes, it seems difficult—at least inconvenient—to protect different users on the same node. To overcome this limitation of USS, we embed post-quantum cryptographic primitives, such as lattice-based public-key signature [11–13], into ASP. This treatment allows a user of a node to identify himself from other users by his unique public key. The post-quantum cryptographic primitive makes our logical language even more powerful than ordinary ASP. Our approach is also practically feasible, as witnessed by the very recent work of Wang et al. [14], which experimentally demonstrated the efficiency of the quantum communication network with identity authenticated by post-quantum cryptography.

The structure of this paper is the following. In Section 1, we review the existing work on the programming language and formal models of smart contracts, with special interest in the programming language of LC. We then develop our update of LC in Section 2. Various examples are presented in this section to demonstrate the usage of our logical contract. We conclude this paper with future work perspectives in Section 3.

1. Background and Related Work: Programming Language and Formal Models of Bitcoin-like Smart Contract

In this section, we briefly review the existing work on the programming languages and formal models of Bitcoin-like smart contracts. The work on the programming languages and formal models of Ethereum-like smart contracts are also relevant to our paper. However, due to the limitation of space, we will not review them. The interested reader may find a survey of these works by Rouhani [15].

1.1. Timed Automata

Andrychowicz et al. [16] proposed a framework for modeling Bitcoin contracts using the timed automata (TA) in the UPPAAL model checker [17]. Their key idea is to use TA to model the behavior of each participant in a contract. The whole system is then modeled as the network constructed by composing these TAs, plus a TA that models the Bitcoin network. The security of the smart contract is then verified in UPPAAL

automatically, finding and correcting some subtle errors that are difficult to discover by the manual analysis.

1.2. Simplicity

Simplicity [18] is an alternative language for Bitcoin scripts. It is a typed, combinator-based, functional language without loops and recursion. For a formal verification of Simplicity programs, denotational semantics in Coq, a popular, general-purpose software proof assistant, is defined. An abstract machine constituting operational semantics for Simplicity is also provided. It is possible to statically estimate the resources (e.g., memory) required to execute contracts written in Simplicity. In Valliappan et al. [19], the authors connected Simplicity's primitives with a categorical model. This lifts the language to a more abstract level allowing for extending it by category theory models of computations.

1.3. BitML

Bartoletti and Zunino [20] expressed Bitcoin contracts in BitML—a simple process calculus. Contracts are three-phase processes:

1. Participants broadcast a contract advertisement, which specifies the content of the contract and its preconditions (e.g., depositing a certain amount of bitcoins).
2. Participants accept the contract and fulfill all the required preconditions. When all the needed participants commit to the contract, the contract is stipulated and can be executed.
3. Executing the contract eventually results in a transfer of the bitcoins deposited by the participants, according to the logic defined by the contract.

1.4. Logical Contract

The logical contract project http://logicalcontracts.com/(accessed on 22 August 2021) developed a logical representation of a legal document that is close to natural, human language and, at the same time, executable by computer. It can be used for the following:

- Monitor compliance of the parties to a contract;
- Enforce compliance, by automatically performing actions to fulfill obligations, and/or by issuing warnings and remedial actions to respond to violations of obligations;
- Explore logical consequences of hypothetical scenarios;
- Query and update the Ethereum blockchain.

The theoretic foundation of the logical contract project is a LPS (Logic-based Production System), which is a general-purpose computer language, developed by Kowalski, Sadri and Calejo [21–23].

1.5. Probabilistic Logic Programs for Blockchain and Smart Contracts

Azzolini et al. [24–26] applied probabilistic logic programs to model and analyze the safely and effectiveness of blockchain systems. They presented a method to translate smart contracts into probabilistic logic programs that can be used to analyze the expected values of several smart contract's utility parameters and obtain a quantitative idea on how smart contracts variables changes over time. They have used this method to study several real smart contracts deployed on the Ethereum blockchain.

1.6. Logicontract

The programming language for smart contracts on LC [8] is based on the script language of Bitcoin [5,20,27]. The definition of a formula of the language is as follows:

$$e ::= x \mid k \mid e + e \mid Hash(e)$$
$$\phi ::= e = e \mid e > e \mid Odd(e) \mid After(e) \mid \neg\phi \mid \phi \wedge \phi$$

where ϕ is a formula, e is an arithmetic expression, x is a variable ranging over natural numbers, $k \in \mathbb{N}$ is a constant natural number, $Hash$ is a collision-resistant hash function on natural numbers, and $Odd(e)$ means that e is an odd number. LC uses a global clock.

$After(e)$ means that the current time by the clock is later than e. Propositional operators of negation and conjunction are represented in a usual way by the symbols $\neg$ and $\wedge$, respectively. The formal definition of the transaction on LC is as follows.

Definition 1 (transaction [8]). *A transaction T is a tuple $(send, rece, sour, cert, prot)$ with the following:*

- *T is the name of the transaction.*
- *send stands for the sender of the transaction.*
- *rece stands for the set of ordered pairs and each of the pairs consists of a potential receiver of this transaction and the amount of the currency that the receiver will receive, i.e., $rece = \{(r_1, a_1), \ldots, (r_m, a_m)\}$.*
- *sour stands for the source of the transaction, which is a list of names of transactions $(T_1, \ldots, T_n)$ that are redeemed by T.*
- *prot stands for the protection, which is a list of ordered pairs in which each pair consists of a receiver from rece and a formula that has to be fulfilled by the receiver to redeem the transaction, i.e., $prot = \{(r_1, \phi_1), \ldots, (r_m, \phi_m)\}$. In order to redeem T, r_i has to provide ϕ_i as the certification of a following transaction.*
- *cert stands for the certification, which is a list of ordered pairs consisting of names of transactions and valuation functions that are supposed to satisfy protections of source transactions, i.e., $cert = \{(T_1, V_1), \ldots, (T_n, V_n)\}$. Valuation functions map variables to natural numbers. The list must provide a valuation function for each source transaction.*

A transaction T redeems source transactions if and only if the following holds:

1. The sender of T is one of the receivers in each of its source transactions.
2. The certification of T evaluates correctly the protections of all sources.
3. None of the sources has been already redeemed.

A transaction T is redeemed if one of its receivers has redeemed it.

Let us now see how the process of redeeming of one transaction by another on examples.

Example 1 (direct payment). *Bob receives the amount of 1 coin from Alice (see Figure 1).*

T_0	
send:	Alice
rece:	$\{(\text{Bob}, 1)\}$
sour:	$\ldots$
cert:	$\ldots$
prot:	$\emptyset$

T_1	
send:	Bob
rece:	$\ldots$
sour:	T_0
cert:	$\emptyset$
prot:	$\ldots$

Figure 1. Direct payment.

Example 2 (payment from multiple sources). *Bob receives payments from Alice and from Eve (1 coin from each) (see Figure 2).*

T_0	
send:	Alice
rece:	$\{(\text{Bob}, 1)\}$
sour:	...
cert:	...
prot:	$\emptyset$

T_1	
send:	Eve
rece:	$\{(\text{Bob}, 1)\}$
sour:	...
cert:	...
prot:	$\emptyset$

T_2	
send:	Bob
rece:	...
sour:	T_0, T_1
cert:	$\emptyset$
prot:	...

Figure 2. Payment from multiple sources.

Example 3 (conditional payment). *Bob receives the amount of 1 coin from Alice if Bob presents a number larger than* 10 *as the value of variable x (see Figure 3).*

T_0	
send:	Alice
rece:	$\{(\text{Bob}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Bob}, x > 10)\}$

T_1	
send:	Bob
rece:	...
sour:	T_0
cert:	$\{(T_0, V(x) = 11)\}$
prot:	...

Figure 3. Conditional payment.

Example 4 (commitment). *Alice commits a secret number x to Bob (the hash value of x is in the example 1234) and makes a deposit (1 coin). If she reveals this secret before a certain time (20211230 in the example), then she redeems the deposit. Otherwise Bob redeems the deposit after the agreed upon time (see Figure 4).*

T_0	
send:	Alice
rece:	$\{(\text{Alice}, 1), (\text{Bob}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Alice}, Hash(x) = 1234), (\text{Bob}, After(20211230))\}$

T_1	
send:	Alice
rece:	...
sour:	T_0
cert:	$\{(T_0, V(x) = Hash^{-1}(1234))\}$
prot:	...

Figure 4. Commitment.

2. Logic Programming for Smart Contracts

The logic used to specify the protection in LC has strong limits on functions and predicates. It can be straightforwardly generalized to involve more expressive logical formulas from logic programming. More specifically, we will use answer set programming [28–30], a modern approach to logic programming, as our underlying logic. In the following, we will first review classical logic programming, then review the answer set programming as an extension of classical logic programming.

2.1. Classical Logic Programming

Now, we give a formal definition of notions, such as formula, valuation, satisfactions and entailment. All these notions can be found in the literature of logic programming [31,32].

Let $X, Y, Z, \ldots$ stand for variables, $a, b, c, \ldots$ for constants $p, q, \ldots$ for predicate symbols and $f, g, h, \ldots$ for function symbols. Constants can be treated as special cases of functions that have no arguments. A language $\mathfrak{L}$ of logic programs is determined by the set of its predicates and functions (and constants).

A term is defined inductively as follows: any variable and any constant is a term, and if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. If no variable occurs in a term, then the term is ground. The set of all ground terms that can be formed with the functions and constants of a language in $\mathfrak{L}$ is called its Herbrand universe $U_{\mathfrak{L}}$.

An atomic formula (atom in short) is built from an n-ary predicate and n terms as its arguments, e.g., $p(t_1, \ldots, t_n)$. An atom is ground if all terms t_i are ground. The set of all ground atoms that can be formed in $\mathfrak{L}$ is called Herbrand base $B_{\mathfrak{L}}$. Any atom and any negation of an atom is called a literal.

A rule of the following form is called a Horn clause:

$$A_0 \leftarrow A_1, \ldots, A_n \quad (n \geq 0),$$

where each A_i $(0 \leq i \leq n)$ is an atom. A_0 is called the head of the clause, and the part on the right of $\leftarrow$ is called its body. When $i = 0$, the body of a rule is empty; such a rule, taking the form $A_0 \leftarrow$, is called a fact. A classical logic program is a finite set of Horn clauses. Clauses (including facts) and logic programs containing no variables are called ground.

For any logic program P, we can define the language $\mathfrak{L}(P)$ that consists of the predicates, functions, and constants occurring in P. If there are no constants in P, we add to $\mathfrak{L}(P)$ a constant to avoid the empty domain. For simplicity, instead of $U_{\mathfrak{L}(P)}$ and $B_{\mathfrak{L}(P)}$, we will write U_P and B_P, respectively. A Herbrand interpretation of P is any subset $I \subseteq B_P$ of its Herbrand base. Intuitively, the atoms in I are true, and all others are false. A Herbrand interpretation of P such that for each rule $A_0 \leftarrow A_1, \ldots, A_m$ in P, this interpretation satisfies the logical formula $\forall X((A_1 \wedge \ldots \wedge A_m) \rightarrow A_0)$ (X being the list of all variables occuring in the rule) is called the Herbrand model of P.

The notions of a Herbrand interpretation and Herbrand model can be generalized, in a natural way, to infinite sets of clauses. Let P be an arbitrary (finite or infinite) set of ground clauses. P defines an operator $T_P : 2^{B_P} \mapsto 2^{B_P}$, where 2^{B_P} denotes the set of all Herbrand interpretations of P, by the following:

$$T_P(I) = \{A_0 \in B_P \mid P \text{ contains a rule } A_0 \leftarrow A_1, \ldots, A_m \text{ such that } \{A_A, \ldots, A_m\} \subseteq I \text{ holds}\}$$

This operator is called the immediate consequence operator; intuitively, it yields all atoms that can be derived by a single application of some rule in P, given the atoms in I. Since T_P is monotone, by the Knaster–Tarski theorem, it has the least fixed point, denoted by T_P^∞. It can be proven that T_P^∞ is the limit of the sequence $T_P^0 = \emptyset$, $T_P^{i+1} = T_P(T_P^i)$. A ground atom A is called a consequence of a set P of clauses if $A \in T_P^\infty$ (we write $P \models A$). Additionally, we say that a negated ground atom $\neg A$ is a consequence of P and write $P \models \neg A$ if $A \notin T_P^\infty$.

The semantics of a set P of ground clauses, denoted as $\mathcal{M}(P)$, is defined as the following set consisting of atoms and negated atoms.

$$\mathcal{M}(P) = \{A \mid P \models A\} \cup \{\neg A \mid P \models \neg A\}.$$

For a ground formula ϕ, built from literals with connective $\wedge, \vee$, we define $P \models \phi$ if $\mathcal{M}(P) \models \phi$ according to the semantics of classical logic. The semantics of logic programs is now defined as follows. Let the grounding of a clause r in a language $\mathfrak{L}$, denoted as $ground(r, \mathfrak{L})$, be the set of all clauses obtained from r by all possible substitutions of elements of $U_{\mathfrak{L}}$ for the variables in r. For any logic program P, we define the following:

$$ground(P, \mathfrak{L}) = \bigcup_{r \in P} ground(r, \mathfrak{L}),$$

and we write $ground(P)$ for $ground(P, \mathfrak{L}(P))$. The operator $T_P : 2^{B_P} \mapsto 2^{B_P}$ associated with P is defined by $T_P = T_{ground(P)}$. Accordingly, $\mathcal{M}(P) = \mathcal{M}(ground(P))$.

2.2. Answer Set Programming

While a rule in classical logic programming is of the form $A_0 \leftarrow A_1, \ldots, A_m$, in answer set programming a rule has the following form:

$$L_1 \vee \ldots \vee L_k \vee \sim L_{k+1} \vee \ldots \vee \sim L_l \leftarrow L_{l+1} \wedge \ldots \wedge L_m \wedge \sim L_{m+1} \wedge \ldots \wedge \sim L_n,$$

where all L_i, $1 \leq i \leq n$ are literals (i.e., atoms or the negation of atoms) and $0 \leq k \leq l \leq m \leq n$. Here, $\sim$ is the default negation (negation as failure). For a rule r of the above form, $\{L_1, \ldots, \sim L_l\}$ is the *head* of r and $\{L_{l+1}, \ldots, \sim L_n\}$ is the body of r. We use $Head(r)$ and $Body(r)$ to denote the head and body of r, respectively.

The notion of an answer set is defined first for ground programs, which do not contain default negation. Let P be such a program and M be a consistent set of literals. We say that M is closed under P if for every rule $r \in P$, $Head(r) \cap M \neq \emptyset$ whenever $Body(r) \subseteq M$. M is an answer set for P if M is minimal (relatively to set inclusion) among the sets of literals that are closed under P.

Now, we extend the definition of an answer set to ground programs with default negation. Let P be an arbitrary program and M a consistent set of literals. The reduct P^M of P relative to M is the set of the following rule:

$$L_1 \vee \ldots \vee L_k \leftarrow L_{l+1} \wedge \ldots \wedge L_m$$

for all rules $L_1 \vee \ldots \vee L_k \vee \sim L_{k+1} \vee \ldots \vee \sim L_l \leftarrow L_{l+1} \wedge \ldots \wedge L_m \wedge \sim L_{m+1} \wedge \ldots \wedge \sim L_n$ in P such that M contains all literals $L_{k+1}, \ldots, L_l$ but does not contain any of $L_{m+1}, \ldots, L_n$. Thus, P^M is a ground program without default negation. We say that M is an answer set for P if M is an answer set of P^M.

Finally, for a non-ground program P, the answer set for P is the answer set for $ground(P)$. It can be verified that if a program P in answer set programming is also a classical logic program, then there is a unique answer set of P, and it coincides with the least fixed point of T_P. A ground literal L is entailed by a program P, denoted by $P \models L$, if L is contained in all answer sets of P. For a ground formula ϕ built from literals with connective $\wedge, \vee$, $P \models \phi$ is defined by interpreting $\wedge, \vee$ in the same way as in classical logic.

2.3. Transaction and Smart Contracts

Now, we are ready to define a new notion of transaction in which protection and certification are specified by logical programs and its semantics.

Definition 2 (transaction). *A transaction T is a tuple $(send, rece, sour, cert, prot)$, where $send, rece, sour$ are defined in the same way as in LC, and the following:*

- *$prot$ is the protection, which is a list of triples of logic programs, formulas and time locks. The number of triples must be the same as the number of receivers. Formally, $prot = \{(r_1, (P_1, \phi_1, tilo_1)), \ldots, (r_m, (P_m, \phi_m, tilo_m))\}$. P_i is a logical program of answer set programming. ϕ_i is either the logical truth $\top$ or a non-ground formula ϕ built from literals with connective $\wedge, \vee$. $tilo_i$ is the time lock, which is of the form $after(k)$ for some natural number k.*
- *$cert$ is the certification, which is a set of ordered pairs of which the first component is the name of a transaction and the second component is either a set of literals or a valuation function that maps variables to constants. Formally, $cert = \{(T_1, V_1), \ldots, (T_n, V_n)\}$.*
  V_i satisfies $(P_i, \top, tilo_i)$ if V_i is an answer set for P_i and the time lock is satisfied by the global clock. V_i satisfies $(P_i, \phi_i, tilo_i)$, where ϕ_i is a non-ground formula ϕ built from literals with connective $\wedge, \vee$, if $P_i \models V_i(\phi_i)$ and the time lock is satisfied by the global clock. When the time lock is $\emptyset$, it is vacuously satisfied by the global clock.

Just like in LC, a transaction T redeems its sources if and only if the following holds:

1. The sender of T is one of the receivers in each of its source transactions.
2. The certification of T evaluates the protections of all its sources to be true.

3. None of its source transactions has been redeemed.

A transaction T is redeemed if one of its receivers has redeemed it.

We typically include a special predicate $After(\cdot)$ in our language. The atom $After(t)$ is true when the global clock has passed time t. Some primitives of the post-quantum public key infrastructure [33] are also involved in our language. At the current stage, we do not specify which post-quantum algorithm is to be used because the standardization of post-quantum cryptography is still an ongoing procedure. Instead, we give an abstract description of some primitives of post-quantum cryptography in ASP. Those primitives include the encryption, signature and hash function.

- For encryption, a secret key sk is a constant. The function $\mathbf{pk}(\cdot)$ maps sk to the corresponding public key $\mathbf{pk}(sk)$. The encryption function $\mathbf{enc}(\cdot,\cdot)$ maps a plain message m and public key $pk = \mathbf{pk}(sk)$ to an encrypted message $\mathbf{enc}(m, pk)$.
- For signature, the signing function $\mathbf{sign}(\cdot,\cdot)$ maps a message m and a secret key sk to the signature $\mathbf{sign}(m, sk)$. The atom $verSig(\sigma, m, pk)$ is true when $\sigma = \mathbf{sign}(m, sk)$ is the signature of m and sk with public key $pk = \mathbf{pk}(sk)$.
- The hash function is represent by a collision-resistant function $Hash$.

Note that we use both an unconditionally secure signature and a post-quantum public key signature (e.g., lattice-based signature). They serve different purposes. The unconditionally secure signature is used to prove that a message from node A to other nodes is indeed sent from node A. Since the computer A may have more than one users, the post-quantum public key signature is used to prove the identity of the users of a computer.

This new format of transaction enables us to design various interesting smart contracts, especially contracts related to knowledge representation, automated planning, constraint solving and other areas in which ASP has proven to be applicable. Now, we use several examples for the demonstration.

Example 5 (authorized payment). *Bob receives a coin from Alice if he provides an appropriate signature from* Bob_1 *(see Figure 5). Here, Bob is a (classical or quantum) computer, which is a node of the underlying quantum-secured permissioned blockchain.* Bob_1 *is a user of Bob.* PK_{Bob_1} *and* SK_{Bob_1} *are, respectively, the public and secret key of* Bob_1*, and* $\mathbf{sign}(\text{“}agree\text{”}, SK_{Bob_1})$ *represents the signature of the message “agree” generated by secret key* SK_{Bob_1}*. Similarly, if Alice changes* $verSig(x, \text{“}agree\text{”}, PK_{Bob_1})$ *to* $(verSig(x, \text{“}agree\text{”}, PK_{Bob_1}) \vee verSig(y, \text{“}agree\text{”}, PK_{Bob_2})) \wedge verSig(z, \text{“}agree\text{”}, PK_{Bob_3})$*, then Bob has to provide a signature of* Bob_3 *and at least one of the signatures of* Bob_1 *and* Bob_2*.*

T_0	
send:	Alice
rece:	$\{(\text{Bob}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Bob}, (\emptyset, verSig(x, \text{“}agree\text{”}, PK_{Bob_1}), \emptyset))\}$.

T_1	
send:	Bob
rece:	...
sour:	T_0
cert:	$\{(T_0, V(x) = \mathbf{sign}(\text{“}agree\text{”}, SK_{Bob_1}))\}$
prot:	...

Figure 5. Authorized payment.

Example 6 (conditional payment with Sudoku). *Bob receives 1 coin from Alice on the condition that Bob solves a Sudoku puzzle (see Figure 6). Here, Sudo is a Sudoku puzzle described in logical programs. A detailed formalization of Sudo can be found in Hölldobler and Schweizer [29]. Ans is an answer set for Sudo.*

T_0	
send:	Alice
rece:	$\{(\text{Bob}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Bob}, (Sudo, \top, \emptyset))\}$

T_1	
send:	Bob
rece:	...
sour:	T_0
cert:	$\{(T_0, Ans)\}$
prot:	...

Figure 6. Conditional payment.

Obviously, we can replace *Sudo* by any logical program. This means that we can realize various interesting conditional payments, as long as the condition can be expressed by logic programs.

Example 7 (competition for solving problems)**.** *Alice declares a difficult problem Prob to Bob and Charlie. The one who first solves it before 20211231 will get the reward. (See Figure 7)*

T_0	
send:	Alice
rece:	$\{(\text{Bob}, 1), (\text{Charlie}, 1), (\text{Alice}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Bob}, (Prob, \top, \emptyset)), (\text{Charlie}, (Prob, \top, \emptyset)), (\text{Alice}, (\emptyset, \top, After(20211231)))\}$

Figure 7. Competition for solving problems.

2.3.1. Multi-Party Lottery

Lottery is a part of the gambling industry with a turnover of billions of dollars [34]. Traditionally, a lottery game is organized by a trustworthy authority. In order to enter a lottery game, players buy tickets. Then, the authority organizing the game initiates a random process that determines the winning tickets. The revenue, in many cases, is large enough to induce temptation to cheat. To ensure fairness of a lottery game and trust of the players, several requirements on a lottery protocol were formulated (c.f. [16,35–39]):

1. Randomness. All tickets are equally likely to win.
2. Unpredictability. No player can predict the winning ticket.
3. Unforgeability. Tickets cannot be forged. In particular, it is impossible to create a winning ticket after the outcome of the random process is known.
4. Verifiability. The number and the revenue of winning tickets are publicly verifiable.
5. Decentralization. The random process does not rely on a single authority.

Note that decentralization allows to organize a lottery without the need for an authority that all players trust, replacing it by alternative mechanisms, such as Blockchain.

In [8], a two-party lottery protocol satisfying the above requirements is defined. We extend it to a multi-party lottery protocol.

Example 8 (multi-party lottery)**.** *The lottery protocol consists of several steps.*

1. *Alice commits a secret to Bob by making a deposit. Bob commits a secret to Charlie by making a deposit. Charlie commits a secret to Alice by making a deposit. (See Figure 8).*
2. *Alice sends a conditional transfer to Alice, Bob and Charlie. Bob sends a conditional transfer to Alice, Bob and Charlie. Charlie sends a conditional transfer to Alice, Bob and Charlie. (See Figure 9).*
   Here, AliceWin is specified by $(Hash^{-1}(x) = 111) \wedge (Hash^{-1}(y) = 222) \wedge (Hash^{-1}(z) = 333) \wedge x + y + z \equiv_3 0$*, where* $\equiv_3$ *means equivalence modulo 3. BobWin is specified by* $(Hash^{-1}(x) = 111) \wedge (Hash^{-1}(y) = 222) \wedge (Hash^{-1}(z) = 333) \wedge x + y + z \equiv_3 1$*. CharlieWin is specified by* $(Hash^{-1}(x) = 111) \wedge (Hash^{-1}(y) = 222) \wedge (Hash^{-1}(z) = 333) \wedge x + y + z \equiv_3 2$*. Alice can also redeem her conditional transfer after 20220130. This condition ensures that Alice can get her money back in case any participant aborts the lottery game before the results can be determined. This is similarly the case for Bob and Charlie.*

3. *Alice reveals her secret and gets her deposit back. Bob reveals his secret and gets his deposit back. Charlie reveals his secret and gets his deposit back. (See Figure 10).*
4. *Now, Alice, Bob and Charlie's secrets are public and the winner can be determined. The winner redeems the loser's conditional transfer and his/her own conditional transfer. If Alice is the winner, then she redeems T_3, T_4 and T_5. This is similarly true for Bob/Charlie when they are the winner. (See Figure 11).*

T_0	
send:	Alice
rece:	$\{(\text{Alice}, 1), (\text{Bob}, 1), \}$
sour:	...
cert:	...
prot:	$\{(\text{Alice}, (\emptyset, Hash(x) = 111, \emptyset)), (\text{Bob}, (\emptyset, \emptyset, After(20211230)))\}$

T_1	
send:	Bob
rece:	$\{(\text{Bob}, 1), (\text{Charlie}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Bob}, (\emptyset, Hash(x) = 222, \emptyset)), (\text{Charlie}, (\emptyset, \emptyset, After(20211230)))\}$

T_2	
send:	Charlie
rece:	$\{(\text{Charlie}, 1), (\text{Alice}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Charlie}, (\emptyset, Hash(x) = 333, \emptyset)), (\text{Alice}, (\emptyset, \emptyset, After(20211230)))\}$

Figure 8. Lottery: deposit.

T_3	
send:	Alice
rece:	$\{(\text{Alice}, 1)\}, \{(\text{Bob}, 1), \{(\text{Charlie}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Alice}, (\emptyset, AliceWin \vee (After(t) \wedge t > 20220130), \emptyset)), (\text{Bob}, (\emptyset, BobWin, \emptyset)), (\text{Charlie}, (\emptyset, CharlieWin, \emptyset))\}$.

T_4	
send:	Bob
rece:	$\{(\text{Alice}, 1)\}, \{(\text{Bob}, 1), \{(\text{Charlie}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Alice}, (\emptyset, AliceWin, \emptyset)), (\text{Bob}, (\emptyset, BobWin \vee (After(t) \wedge t > 20220130), \emptyset)), (\text{Charlie}, (\emptyset, CharlieWin, \emptyset))\}$.

T_5	
send:	Charlie
rece:	$\{(\text{Alice}, 1)\}, \{(\text{Bob}, 1), \{(\text{Charlie}, 1)\}$
sour:	...
cert:	...
prot:	$\{(\text{Alice}, (\emptyset, AliceWin, \emptyset)), (\text{Bob}, (\emptyset, BobWin, \emptyset)), (\text{Charlie}, (\emptyset, CharlieWin \vee (After(t) \wedge t > 20220130), \emptyset))\}$.

Figure 9. Lottery: conditional transfer.

T_6	
send:	Alice
rece:	...
sour:	T_0
cert:	$\{(T_0, V(x) = Hash^{-1}(111))\}$
prot:	...

T_7	
send:	Bob
rece:	...
sour:	T_1
cert:	$\{(T_1, V(x) = Hash^{-1}(222))\}$
prot:	...

T_8	
send:	Charlie
rece:	...
sour:	T_2
cert:	$\{(T_2, V(x) = Hash^{-1}(333))\}$
prot:	...

Figure 10. Lottery: revealing the secrets.

T_9	
send:	Alice
rece:	...
sour:	T_3, T_4, T_5
cert:	$V(x) = Hash^{-1}(111), V(y) = Hash^{-1}(222),, V(z) = Hash^{-1}(333)$
prot:	...

Figure 11. Lottery: determining the winner.

2.3.2. Legal Service

As in the last example, we show that our logical contracts can be used to provide a legal consultation service on blockchain. This is possible because ASP is able to express legal/normative rules, thanks to the close connection between ASP and default logic, which, in turn, can be used to express legal/normative rules. The connection between ASP and default logic is the following. Let P be a program such that the head of every rule of P is a single literal, as follows:

$$L_1 \leftarrow L_2 \wedge \ldots \wedge L_m \wedge \sim L_{m+1} \wedge \ldots \wedge \sim L_n. \tag{1}$$

We can transform P into a default theory $\mathrm{D}(P)$ in the sense of [40] by turning each rule (1) into the default as follows:

$$\frac{L_2 \wedge \ldots \wedge L_m \wedge : \neg L_{m+1}, \ldots, \neg L_n}{L_1}.$$

The correspondence between P and $\mathrm{D}(P)$ is the following: if X is an answer set for P, then the deductive closure of X is a consistent extension for $\mathrm{D}(P)$; conversely, every consistent extension for $\mathrm{D}(P)$ is the deductive closure of an answer set for P.

In deontic default logic [41], a conditional obligation $\phi \rightarrow O\psi$, meaning that if ϕ, then it is obliged to be ψ, and is expressed by a default as follows:

$$\frac{\phi : \psi}{\psi}.$$

Therefore, a conditional obligation $L_1 \rightarrow OL_2$ can be expressed as the following rule in ASP:

$$L_2 \leftarrow L_1 \wedge \sim \neg L_2.$$

Moreover, since a conditional prohibition $\phi \rightarrow F\psi$, meaning that if ϕ, then it is forbidden to be ψ, is defined by $\phi \rightarrow O\neg\psi$, we can express a conditional prohibition $L_1 \rightarrow FL_2$ as the following rule in ASP:

$$\neg L_2 \leftarrow L_1 \wedge \sim L_2.$$

Now, if the protection of a transaction T includes a logical program P, which consists of some facts, conditional obligations and prohibitions expressed in ASP, then a transaction T' redeems T only if it provides an answer set of P. We can view P as a description of a legal situation or a normative system together with some facts. An answer set of P is then a suggestion of actions to be taken. In this sense, legal consultation can be carried out by transactions of blockchain.

3. Conclusions and Future Work

This paper applies answer set programming, enriched with post-quantum cryptographic primitives, to the design and specification of smart contracts on quantum-secured blockchains. The enrichment of post-quantum cryptographic primitives overcomes the limitation of unconditionally secure signatures in existing quantum-secured blockchains, which do not provide mechanisms for user authentication.

The application of ASP allows us to design various interesting smart contracts. Compared to procedural languages, the smart contracts written in our logical language are easier to be understood and formally verified.

In the future, we plan to systematically explore the usage of our framework in multi-party secure computation. We are also interested in further extending our logical language to include primitives of quantum computation. Such an extension will create a programming language of smart contracts in quantum logic.

Author Contributions: Conceptualization, X.S., P.K. and M.S.; methodology, X.S. and P.K.; validation, P.K. and M.S.; writing—original draft preparation, X.S.; writing—review and editing, P.K. and M.S.; supervision, P.K.; project administration, P.K.; funding acquisition, P.K. All authors have read and agreed to the published version of the manuscript.

Funding: The project is funded by the Minister of Education and Science within the program under the name "Regional Initiative of Excellence" in 2019–2022, project number: 028/RID/2018/19, to the amount: 11,742,500 PLN.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 23 July 2021).
2. Szabo, N. The Idea of Smart Contracts. Available online: https://nakamotoinstitute.org/the-idea-of-smart-contracts (accessed on 23 July 2021).
3. Bartoletti, M.; Cimoli, T.; Giamberardino, P.D.; Zunino, R. Vicious circles in contracts and in logic. *Sci. Comput. Program.* **2015**, *109*, 61–95. [CrossRef]
4. Governatori, G.; Idelberger, F.; Milosevic, Z.; Riveret, R.; Sartor, G.; Xu, X. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artif. Intell. Law* **2018**, *26*, 377–409. [CrossRef]
5. Atzei, N.; Bartoletti, M.; Cimoli, T.; Lande, S.; Zunino, R. SoK: Unraveling Bitcoin Smart Contracts. *Princ. Secur. Trust. LNCS* **2018**, *10804*, 217–242. [CrossRef]
6. Grishchenko, I.; Maffei, M.; Schneidewind, C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *International Conference on Principles of Security and Trust*; Springer: Cham, Switzerland, 2018; pp. 243–269. [CrossRef]
7. Atzei, N.; Bartoletti, M.; Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Lecture Notes in Computer Science, Proceedings of the Principles of Security and Trust—6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, 22–29 April 2017*; Maffei, M., Ryan, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10204, pp. 164–186. [CrossRef]
8. Sun, X.; Sopek, M.; Wang, Q.; Kulicki, P. Towards Quantum-Secured Permissioned Blockchain: Signature, Consensus, and Logic. *Entropy* **2019**, *21*, 887. [CrossRef]
9. Sun, X.; Wang, Q.; Kulicki, P.; Sopek, M. A Simple Voting Protocol on Quantum Blockchain. *Int. J. Theor. Phys.* **2019**, *58*, 275–281. [CrossRef]
10. Sun, X.; Kulicki, P.; Sopek, M. Lottery and Auction on Quantum Blockchain. *Entropy* **2020**, *22*, 1377. [CrossRef] [PubMed]

11. Gentry, C.; Peikert, C.; Vaikuntanathan, V. Trapdoors for hard lattices and new cryptographic constructions. In Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, BC, Canada, 17–20 May 2008; Dwork, C., Ed.; ACM: New York, NY, USA, 2008; pp. 197–206. [CrossRef]
12. Ducas, L.; Micciancio, D. Improved Short Lattice Signatures in the Standard Model. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2014—34th Annual Cryptology Conference, Santa Barbara, CA, USA, 17–21 August 2014*; Part I; ; Garay, J.A., Gennaro, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8616; pp. 335–352. [CrossRef]
13. Li, C.; Chen, X.; Chen, Y.; Hou, Y.; Li, J. A New Lattice-Based Signature Scheme in Post-Quantum Blockchain Network. *IEEE Access* **2019**, *7*, 2026–2033. [CrossRef]
14. Wang, L.J.; Zhang, K.Y.; Wang, J.Y.; Cheng, J.; Yang, Y.H.; Tang, S.B.; Yan, D.; Tang, Y.L.; Liu, Z.; Yu, Y.; et al. Experimental authentication of quantum key distribution with post-quantum cryptography. *NPJ Quantum Inf.* **2021**, *7*, 67. [CrossRef]
15. Rouhani, S.; Deters, R. Security, Performance, and Applications of Smart Contracts: A Systematic Survey. *IEEE Access* **2019**, *7*, 50759–50779. [CrossRef]
16. Andrychowicz, M.; Dziembowski, S.; Malinowski, D.; Mazurek, L. Modeling Bitcoin Contracts by Timed Automata. In *Lecture Notes in Computer Science, Proceedings of the Formal Modeling and Analysis of Timed Systems—12th International Conference, FORMATS 2014, Florence, Italy, 8–10 September 2014*; Legay, A., Bozga, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8711, pp. 7–22. [CrossRef]
17. Amnell, T.; Behrmann, G.; Bengtsson, J.; D'Argenio, P.R.; David, A.; Fehnker, A.; Hune, T.; Jeannet, B.; Larsen, K.G.; Möller, M.O.; et al. UPPAAL—Now, Next, and Future. In *Lecture Notes in Computer Science, Proceedings of the Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, 19–23 June 2000*; Cassez, F., Jard, C., Rozoy, B., Ryan, M.D., Eds.; Springer: Berlin/Heidelberg, Germany, 2000; Volume 2067, pp. 99–124. [CrossRef]
18. O'Connor, R. Simplicity: A New Language for Blockchains. In Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, 30 October 2017; ACM: New York, NY, USA, 2017; pp. 107–120. [CrossRef]
19. Valliappan, N.; Mirliaz, S.; Vesga, E.L.; Russo, A. Towards Adding Variety to Simplicity. In *International Symposium on Leveraging Applications of Formal Methods*; Springer: Cham, Switzerland, 2018; pp. 414–431. [CrossRef]
20. Bartoletti, M.; Zunino, R. BitML: A Calculus for Bitcoin Smart Contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, 15–19 October 2018; Lie, D., Mannan, M., Backes, M., Wang, X., Eds.; ACM: New York, NY, USA, 2018; pp. 83–100. [CrossRef]
21. Kowalski, R.A.; Sadri, F. Reactive Computing as Model Generation. *New Gener. Comput.* **2015**, *33*, 33–67. [CrossRef]
22. Kowalski, R.A.; Sadri, F. Programming in logic without logic programming. *Theory Pract. Log. Program.* **2016**, *16*, 269–295. [CrossRef]
23. Kowalski, R.A.; Sadri, F.; Calejo, M. How to do it with LPS (Logic-Based Production System). In Proceedings of the Doctoral Consortium, Challenge, Industry Track, Tutorials and Posters @ RuleML+RR 2017 hosted by International Joint Conference on Rules and Reasoning 2017 (RuleML+RR 2017), London, UK, 11–15 July 2017; Bassiliades, N., Bikakis, A., Costantini, S., Franconi, E., Giurca, A., Kontchakov, R., Patkos, T., Sadri, F., Woensel, W.V., Eds.; CEUR-WS.org: Aachen, Germany, 2017; Volume 1875.
24. Azzolini, D.; Riguzzi, F.; Lamma, E.; Bellodi, E.; Zese, R. Modeling Bitcoin Protocols with Probabilistic Logic Programming. In Proceedings of the 5th International Workshop on Probabilistic Logic Programming, PLP 2018, Co-Located with the 28th International Conference on Inductive Logic Programming (ILP 2018), Ferrara, Italy, 1 September 2018; Bellodi, E., Schrijvers, T., Eds.; CEUR-WS.org: Aachen, Germany, 2018; Volume 2219, pp. 49–61.
25. Azzolini, D.; Riguzzi, F.; Lamma, E. Analyzing Transaction Fees with Probabilistic Logic Programming. In *Lecture Notes in Business Information Processing, Proceedings of the Business Information Systems Workshops—BIS 2019 International Workshops, Seville, Spain, 26–28 June 2019*; Revised Papers; Abramowicz, W., Corchuelo, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; Volume 373, pp. 243–254. [CrossRef]
26. Azzolini, D.; Riguzzi, F.; Lamma, E. A semantics for Hybrid Probabilistic Logic programs with function symbols. *Artif. Intell.* **2021**, *294*, 103452. [CrossRef]
27. Bartoletti, M.; Cimoli, T.; Zunino, R. Fun with Bitcoin Smart Contracts. In *International Symposium on Leveraging Applications of Formal Methods*; Springer: Cham, Switzerland, 2018; pp. 432–449. [CrossRef]
28. Lifschitz, V. Answer set programming and plan generation. *Artif. Intell.* **2002**, *138*, 39–54. [CrossRef]
29. Hölldobler, S.; Schweizer, L. Answer Set Programming and CLASP—A Tutorial. In Proceedings of the Young Scientists' International Workshop on Trends in Information Processing (YSIP) Co-Located with the Sixth International Conference on Infocommunicational Technologies in Science, Production and Education (INFOCOM-6), Stavropol, Russia, 22–25 April 2014; Hölldobler, S., Malikov, A., Wernhard, C., Eds.; CEUR-WS.org: Aachen, Germany, 2014; Volume 1145, pp. 77–95.
30. Cabalar, P.; Pearce, D.; Valverde, A. Answer Set Programming from a Logical Point of View. *Künstliche Intell.* **2018**, *32*, 109–118. [CrossRef]
31. Baral, C.; Gelfond, M. Logic Programming and Knowledge Representation. *J. Log. Program.* **1994**, *19/20*, 73–148. [CrossRef]
32. Dantsin, E.; Eiter, T.; Gottlob, G.; Voronkov, A. Complexity and expressive power of logic programming. *ACM Comput. Surv.* **2001**, *33*, 374–425. [CrossRef]
33. Bernstein, D.J. Introduction to post-quantum cryptography. In *Post-Quantum Cryptography*; Bernstein, D.J., Buchmann, J., Dahmen, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 1–14. [CrossRef]

34. Isidore, C. Americans Spend More on the Lottery Than on.... Available online: https://money.cnn.com/2015/02/11/news/companies/lottery-spending/ (accessed on 25 August 2021).
35. Chow, S.S.M.; Hui, L.C.K.; Yiu, S.; Chow, K.P. An e-Lottery Scheme Using Verifiable Random Function. In *Lecture Notes in Computer Science, Proceedings of the Computational Science and Its Applications—ICCSA 2005, International Conference, Singapore, 9–12 May 2005*; Proceedings Part III; Gervasi, O., Gavrilova, M.L., Kumar, V., Laganà, A., Lee, H.P., Mun, Y., Taniar, D., Tan, C.J.K., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3482, pp. 651–660. [CrossRef]
36. Bentov, I.; Kumaresan, R. How to Use Bitcoin to Design Fair Protocols. In *Lecture Notes in Computer Science, Proceedings of the Advances in Cryptology—CRYPTO 2014—34th Annual Cryptology Conference, Santa Barbara, CA, USA, 17–21 August 2014*; Proceedings Part II; Garay, J.A., Gennaro, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8617, pp. 421–439. [CrossRef]
37. Bartoletti, M.; Zunino, R. Constant-Deposit Multiparty Lotteries on Bitcoin. In *Lecture Notes in Computer Science, Proceedings of the Financial Cryptography and Data Security—FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, 7 April 2017*; Revised Selected Papers; Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10323, pp. 231–247. [CrossRef]
38. Grumbach, S.; Riemann, R. Distributed Random Process for a Large-Scale Peer-to-Peer Lottery. In *Distributed Applications and Interoperable Systems*; Chen, L.Y., Reiser, H.P., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 34–48.
39. Miller, A.; Bentov, I. Zero-Collateral Lotteries in Bitcoin and Ethereum. In Proceedings of the 2017 IEEE European Symposium on Security and Privacy Workshops, EuroS & P Workshops 2017, Paris, France, 26–28 April 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 4–13. [CrossRef]
40. Reiter, R. A Logic for Default Reasoning. *Artif. Intell.* **1980**, *13*, 81–132. [CrossRef]
41. Horty, J.F. Defaults with Priorities. *J. Philos. Log.* **2007**, *36*, 367–413. [CrossRef]