ABSTRACT
Cross-organizational, blockchain-based distributed ledger networks in general, and those based on Hyperledger Fabric in particular, have an architecture which can be adapted to specific application requirements. However, network design can be a particularly challenging task, as the connection between architectural and deployment decisions and extra-functional properties can be subtle and the requirements may contradict each other, requiring trade-offs.

In this paper, we propose a model-based distributed ledger architecture design approach which enables expert exploration of design options. We capture key requirements and define architecture fragments using partial modelling. We enumerate qualitatively different architectural candidates by graph generation. We evaluate and rank order candidates in logic solver tooling. As a result, our approach provides generative architectures for distributed ledger networks by enabling efficient exploration of design alternatives.

CCS CONCEPTS
• Computer systems organization → Distributed architectures; Dependable and fault-tolerant systems and networks; • Software and its engineering → Search-based software engineering; Architecture description languages.

KEYWORDS
Model Generation, Partial Modelling, Blockchain, HyperLedger Fabric, Design-space Exploration, Generative Architecture

1 INTRODUCTION
Distributed ledger technology (DLT) [27], predominantly still implemented over the "blockchain" principles pioneered by Bitcoin, facilitates trustworthy collaboration between distant parties over a ledger-like transaction database, shared and maintained by a network of independent parties through some Byzantine fault (and attack) tolerant consensus mechanism. While the use and ongoing development of large, cryptocurrency-accounting distributed ledger networks remains robust despite the ebbs and flows in the valuation of the cryptoassets they handle, DLT is also being increasingly used in a very wide range of use cases that do not involve cryptocurrencies – from industrial and healthcare data handling through digital identity management to replacing trusted third parties with a distributed ledger in various financial settings [42].

Consensus being open or closed for the participation of the general public and ledger services being open or authorization-bound are two key aspects of a distributed ledger. (Consensus) unpermissioned, open access networks – "public blockchains" – are many times ill-suited for non-crypto applications for a host of reasons, ranging from unpredictably fluctuating transaction costs (paid in cryptocurrencies) and transaction delays to the simple fact of the shared ledger being accessible to the general public. Permissioned consensus and permissioned access blockchains – often called "consortial", or even "private" blockchains –, on the other hand, can provide a dedicated distributed ledger for cross-organizational collaborations.

In stark contrast to public blockchains, a consortial distributed ledger can be "bespoke" in the sense that, as much as the DLT platform permits, it can be engineered against a set of extra-functional requirements. Although the leading DLTs used for creating consortial distributed ledgers allow addressing specific requirement sets differently, all aim to provide the necessary facilities. Solution engineering for (i) Hyperledger Fabric [5, 33] has a strong network architecture design aspect, as we later discuss; (ii) in R3 Corda [56], "applications" are designed as complex multi-party message flows across network nodes; and even (iii) the "enterprise" variants of Ethereum [29] (e.g., Quorum [23]) increasingly provide multiple architectural building blocks. Additionally, as cross-DLT interoperability is maturing, (iv) responding to requirements with multi-chain designs [13] is becoming a viable engineering option.
At the same time, design methodologies for consortial distributed ledgers are in their infancy, and it remains a challenging task to create an appropriate design for an extra-functional requirement set, especially so that requirements are frequently contradictory. Due to the complex interaction patterns in these peer-to-peer networks, even experts can find the impacts of individual design decision hard to gauge. This is a serious concern, as distributed ledgers tend to implement at least business-critical functions, and the need to re-engineer them after a proper extra-functional analysis – or worse, after a failed system-level validation – is a considerable risk [60].

Model-driven engineering techniques have been successfully applied for the implementation of transaction logic in blockchains [10, 26, 59], but to our best knowledge, no such approach exists for distributed ledger architectures. In this paper, we aim to leverage model-driven architecture synthesis [17, 52] and design-space exploration [3, 16, 32, 39, 47] techniques to this end.

In particular, (1) we propose a domain-specific language for capturing architectures and requirements of distributed ledgers. (2) We propose a novel workflow for distributed ledger architecture generation relying on recent advances in partial graph modeling and diverse graph generation. We (3) score candidates according to extra-functional criteria with Answer Set Programming. We (4) demonstrate our approach on a simplified, but highly representative view of the requirement-based design of Hyperledger Fabric networks.

1.1 Collaborations and distributed ledgers

The core value proposition of all DLTs is avoiding the need for trusted intermediaries in electronic record-keeping settings; instead, users of the ledger have to place trust in the sufficient majority of the parties maintaining a distributed ledger remaining honest [74].

Design processes for realizing novel business value with distributed ledgers are still evolving, but we can already say with certainty that even the “migration” of cross-organizational collaboration models can carry significant business benefits; even with established business model patterns and cross-organizational data exchange relationships.

In this paper, we start with the assumption that a cross-organizational collaboration model has been established and mapped. Our main concern here is architecture design under such given requirements. We do note, however, that our approach has significant potential for the earlier design phases and their associated decision making, too.

1.2 Hyperledger Fabric: Network architecture

In this paper, we work with the Hyperledger Fabric (HLF) design language depicted in Figure 1. A Fabric network is jointly operated by a set of organizations. The network maintains a set of blockchain-backed distributed ledgers, called channels, in Fabric. Each channel is associated with a set of participating organizations; each organization participating in a channel provides network nodes for the operation of the channel on its organizational host machines. Nodes either participate in ordering the transactions of the channel, or computing their side effects (“endorsing”).

Out of the box, Fabric provides only a simple versioned key-value ledger (channel) abstraction, without any native (built-in) asset or transaction type. Chaincode in Fabric (smart contracts) define the transaction types and implement them in general-purpose programming languages, relying on a key get/set style API.

Chaincode is instantiated on a channel by deployment to the endorsing nodes of a channel. Subsequently, transaction processing follows an Execute-Order-Validate pattern. First, an organizational client requests the execution of a chaincode method from a number of endorsing nodes over their current (channel) ledger view, but without actually making ledger modifications. If the client can collect enough matching replies on the “simulated” write effects, it submits those “endorsements” to the ordering service.

The ordering service orders endorsed transaction proposals, forms the next block of the blockchain and distributes it to the endorsing nodes. (a) In Kafka-based ordering, nodes of a dedicated “ordering organization”, running an Apache Kafka [6] cluster, perform the ordering. (b) In Raft-based ordering, the nodes of the participating organizations realize the ordering service through an implementation of the Raft [57] peer-to-peer consensus protocol.

1.3 Architecture design challenges

It is easy to see that the collaborative design of a Fabric network by the participating organizations is an exercise in engineering trade-offs. For a single channel, an $n$-out-of-$n$ organizations endorsement policy certainly maximizes integrity; at the same time, the inability of a single organization to endorse transactions translates to unavailability. In contrast, for a $k$-out-of-$n$ organizations endorsement policy ($k < n$), availability should be only sensitive to multiple, simultaneous, independent faults – at the expense of ledger integrity. In summary, architecture designers need to make a complex design decisions while considering the reliability, performance, and cost of the architecture while satisfying different organization requirements.

Example 1.1. In our running example, we consider secure data sharing within a multi-organizational distributed ledger network. Our scenario, depicted in Figure 2a involves a HLF network with three organizations: OrgA, OrgB, and OrgC. The network architecture ensures that OrgA and OrgB operate independently and do not collaborate directly. However, OrgA and OrgC, as well as
OrgB and OrgC, can securely share confidential data via established communication channels.

The goal of the architecture synthesis process is to generate design alternatives that satisfy these information flow requirements while also ensuring network reliability. In particular, OrgA and OrgB should not participate in any secure channels together to prevent unauthorized communication, while OrgA and OrgC, as well as OrgB and OrgC should share at least one channel, respectively.

1.4 Generative architectures for blockchains

Previous modeling techniques (Section 6) in DLT design are limited to smart contract implementations and do not address architectural concerns. The aim of our work is to provide (1) a systematic way to gather and formalize requirements during the initial steps of consortial DLT design and (2) a tool to automatically derive candidate architectures (i.e., suggested designs) for engineers to iterate on.

For DLT-related problems, the high-level data sharing and communication requirements themselves have a graph structure. In our approach, we create a partial model based on these requirements that serves as an initial model for graph generation (Section 3.2), leaving platform-specific concepts to be filled in by the tool (Section 3.3). Since both the requirements and the generated architectures are graphs, this problem is especially amenable to be tackled as a graph generation problem.

To our best knowledge, (i) architecture generation has not been used previously for DLT, and (ii) capturing requirements and architecture fragments for architecture generation in a partial model is novel for other domains as well.

In general, our approach is applicable in other domains where (a) the requirements imposed on the architecture can be evaluated as necessary matches of model queries or derived features [34, 52, 73] and (b) result in a graph generation problem (i.e., models have complex graph structures with interconnected elements). Similar techniques have been applied to, e.g., avionics [47], cyber-physical systems [2], and satellite constellations [52].

2 PRELIMINARIES

Now we recall some concepts related to partial models, which we will use to describe functional requirements and proposed architectures of distributed ledger with mathematical precision. We also overview Answer Set Programming (ASP), which will later be used to analyze the extra-functional properties of architectures.

2.1 Metamodeling with First-Order Logic

In this paper, we will use domain-specific modeling languages to capture platform-independent requirements of distributed ledger networks, as well as platform-specific information about architecture proposals. We will capture functional requirements of distributed ledger architectures in the derived features and well-formedness constraints of models. Moreover, we will use partial modeling to precisely describe the available information and the design decisions yet to be made at each step of the design process.

Similarly to [34, 52], we rely on First-Order Logic (FOL) as a semantic basis for domain-specific models to formalize (a) structural constrains arising from metamodels, (b) functional requirements, and (c) other design rules. FOL is highly expressive can formalize other constraints languages widely used in model-driven engineering, such as OCL [49, 71] and graph patterns [73].

Definition 2.1. A metamodel is a FOL signature $⟨\Sigma, α⟩$, where the set of symbols $\Sigma$ include unary class $C_i$ and existence $ε$ symbols and binary reference $R_j$, derived reference $D_k$, and equivalence $∼$ symbols, while $α : α → N$ is the arity function $α(C_i) = α(ε) = 1$, $α(R_j) = α(D_k) = α(∼) = 2$.

In the Eclipse Modeling Framework (EMF) [70], each EClass corresponds to a class symbol $C_i$. EReferences correspond either to reference symbols $R_j$ or derived reference symbols $D_k$ depending on whether they have the derived flag set. This denotes that a value of a given reference is not standalone but is to be computed from the values other classes $C_i$ and references $R_j$ already in the model.

Example 2.2. Figure 2b shows a fragment of the metamodel in Figure 1 selected for illustration. The associated signature $⟨\Sigma, α⟩$ contains the classes Organization, Channel $∈ \Sigma$, the reference participatesIn $∈ \Sigma$, and the derived reference collaboratesWith $∈ \Sigma$. Moreover, we have $α(Organization) = α(Channel) = α(ε) = 1$, $α(participatesIn) = α(collaboratesWith) = α(∼) = 2$.

We collect the definitions of derived references and other well-formedness constraints into a first-order theory.

Definition 2.3. A theory $T$ over the signature $\Sigma$, $α$ is a pair $(d, E)$, where $d$ maps each derived feature $D_k$ to a FOL formula $d(D_k)$ with free variables $v_1, v_2$, and the set of error predicates $E = \{ψ_1, . . . , ψ_n\}$ is a finite set of FOL formulas.

Example 2.4. We may formalize the theory $T = (d, E)$ associated with the running example Figure 2 as follows. The derived reference collaboratesWith should connect Organization instances that participate in the same Channel. Formally,

$$d(\text{collaboratesWith})(\ell_1, \ell_2) ::= \ell_1 \neq \ell_2 \land \exists c : (\text{participatesIn}(\ell_1, c) \land \text{participatesIn}(\ell_2, c))$$.
We may add an error pattern \( \psi \in E \) to capture the lower multiplicity constraint \([1^*] \) on participates:

\[
\psi(v_1) := \text{Organization}(v_1) \land \neg \exists c: \text{participatesIn}(v_1, c),
\]

i.e., it is an error to have an Organization \( v_1 \) with no Channel \( c \).

### 2.2 Partial models and concrete models

**Partial models** [30, 53, 61] capture possible incomplete information using 3-valued logic [48], which adds an extra unknown \( \frac{1}{2} \) truth value to the usual true \( 1 \) and false \( 0 \) truth values. This allows us to interpret partial models as 3-valued FOL logic structures:

**Definition 2.5.** A partial model \( P \) over a signature \( (\Sigma, \alpha) \) is a pair \((O_P, I_P)\), where \( O_P \) is a finite set of objects and \( I_P \) provides a 3-valued interpretation \( I_P(\sigma): O_P^\alpha(\sigma) \rightarrow \{1, 0, 1/2\} \) for each symbol \( \sigma \in \Sigma \). We call a partial model with no \( 1/2 \) values, i.e., with all aspects fully known, a concrete model.

We set \( I_P(C_i)(o) = 1, 0, 1/2 \) if it is true, false, or unknown, respectively, whether the object \( o \in O_P \) is of type \( C_i \). We may similarly set \( I_P(R_j)(o_1, o_2) \) and \( I_P(D_k)(o_1, o_2) \) to denote whether the relationship \( R_j \) or derived relationship \( D_k \) is present between the objects \( o_1, o_2 \in O_P \).

The value \( I_P(\epsilon)(o) = 1/2 \) denotes uncertain existence, i.e., models that may be removed from the model. Objects with \( I_P(\epsilon)(o) = 0 \) can be removed outright. Objects with \( I_P(\neg)(o, o) = 1/2 \) represent multi-objects that can be split to represent multiple concrete model elements. For simplicity, we will require \( I_P(\neg)(o_1, o_2) = 0 \) if \( o_1 \neq o_2 \), i.e., distinct objects can never be equal.

**Example 2.6.** Figure 2c shows an example partial model \( P \) corresponding to the collaboration in Figure 2a over the signature associated with the metamodel Figure 2b. We have \( O_P = \{ \text{OrgA}, \text{OrgB}, \text{OrgC}, \text{Channel\_new} \} \). Types are written inside the boxes corresponding to objects, e.g., we have \( I_P(\text{Organization})(\text{OrgA}) = 1 \).

Solid lines correspond to certain links, e.g.,

\[
I_P(\text{collaboratesWith})(\text{OrgA, OrgC}) = 1,
\]

dashed lines correspond to uncertain links, e.g.,

\[
I_P(\text{participatesIn})(\text{OrgA, Channel\_new}) = 1/2,
\]

Links not depicted are always false, e.g.,

\[
I_P(\text{collaboratesWith})(\text{OrgA, OrgB}) = 1.
\]

The object Channel\_new is a multi-object representing all Channel instances to be added, i.e., \( I_P(\epsilon)(\text{Channel\_new}) = 1/2 \) (denoted with dashed border) and \( I_P(\neg)(\text{Channel\_new, Channel\_new}) = 1/2 \) (denoted with a shadow).

### 2.3 Semantics and consistency

Partial models allow us to evaluate FOL formulas \( \varphi \) according to 3-valued logic semantics. On a concrete model, evaluation always results in either \( 1 \) or \( 0 \) [53]. On uncertain models, evaluation may yield \( 1/2 \), which signifies that there is not enough information in the partial model to provide a definite result.

**Definition 2.7.** The 3-valued semantics \([\varphi]_{3}^{P} \) of a FOL formula \( \varphi \) with free variables \( v_1, \ldots, v_n \) and variable binding \( Z = \{ v_1, \ldots, v_n \} \) \( \rightarrow O_P \) on the partial model \( P = \langle O_P, I_P \rangle \) is given in Figure 3a [53].

\[
\begin{align*}
[\text{Organization}(v_1)]_{3}^{P} & = I_P(\text{Organization}(v_1)), \\
[\text{participatesIn}(v_1, c)]_{3}^{P} & = I_P(\text{participatesIn}(v_1, c)).
\end{align*}
\]

**Figure 3:** Partial model semantics and refinement

**Definition 2.8.** A partial model \( P = \langle O_P, I_P \rangle \) over the signature \( (\Sigma, \alpha) \) is consistent with the theory \( T = (d, E) \), written as \( P \models T \), if

- For all error predicates \( \psi \) and bindings \( Z \) of the free variables of \( \psi \), where \( 1/2 \models 1/2 \), \( 1/2 \models 0 \), \( 1 \models 0 \), \( 0 \models 0 \), is the information order on \( \{1, 0, 1/2\} \) [53]. If \( P \) is concrete, we may replace \( \models \) with \( = \).

**2.4 Refinement and model generation**

Partial model refinement gradually incorporates information into partial models while obeying the information ordering relation \( \models \), i.e., not contradicting previously established facts.

**Definition 2.9.** The function \( \text{bwd}: O_Q \rightarrow O_P \) is a refinement function from the partial model \( P \) to \( Q \), written as \( P \models_{\text{bwd}} Q \), if

- For all symbols \( \sigma \) and \( q_1, \ldots, q_n(\sigma) \), we have \( I_P(\sigma)(\text{bwd}(q_1), \ldots, \text{bwd}(q_n(\sigma))) \models I_Q(\sigma)(q_1, \ldots, q_n(\sigma)); \)

- Surely existing objects do not disappear, i.e., there is some \( q \in O_Q \) with \( \text{bwd}(q) = p \) for all \( p \in O_P \) with \( I_P(\sigma)(p) = 1 \).

**Definition 2.10.** The model generation problem [53] \((\Sigma, \alpha, T, P_0)\) consists of a metamodel signature \((\Sigma, \alpha)\), a theory \( T \), and an initial partial model \( P_0 \). A solution of the model generation problem is a concrete model \( M \) that is a refinement of \( P_0 \) (i.e., \( P_0 \models M \)) and is consistent with \( T \) (i.e., \( M \models T \)).

Automated model generators [52, 53] derive such solutions \( M \) along a chain of refinements \( P_0 \models P_1 \models \cdots \models P_n \models M \), exploiting the (i) associativity of partial model refinement \( \models \) and the (ii) monotonicity of the consistency relation \( \models \) w.r.t. refinements.

**Example 2.11.** The concrete model \( M \) in Figure 3b is a refinement \( P_0 \models_{\text{bwd}} M \) of the partial model \( P_0 \) shown in Figure 2c with \( \text{bwd}(\text{OrgA}) = \text{OrgA}, \text{bwd}(\text{OrgB}) = \text{OrgB}, \text{bwd}(\text{OrgC}) = \text{OrgC}, \)

\[
\text{bwd}(\text{Channel\_1}) = \text{bwd}(\text{Channel\_2}) = \text{Channel\_new}.
\]

Since it is consistent \((M \models T)\) with the theory \( T \) from Example 2.4, \( M \) is a solution of the model generation problem for \( T \) and \( P_0 \). In other words, it satisfies the requirements set forth in Section 1.3 without violating any design rules.
2.5 Answer Set Programming

Answer Set Programming (ASP) [21, 50] is a declarative formalization approach to model and solve search and optimization problems. ASP is based on the stable model semantics of logic programming [36]. The concept of a stable model is used to define declarative semantics for logic programs with negation as failure.

An ASP program consists of atoms of the form \( \sigma(x_1, \ldots, x_n) \), where \( \sigma \in \Sigma \) is a logical symbol and \( x_1, \ldots, x_n \) are variables or objects constants, as well as literals, and rules of the form

\[ r \leftarrow a_1 \ldots a_m, \text{not } b_1 \ldots \text{not } b_m. \]

The rule above is derived, meaning that the head (r) is true, when all the atoms and literals in the body \((a_i, b_i)\) are true in the following sense: a non-negated literal \( a \) is true if the atom \( a \) has a derivation (from another rule or as a fact). A negated literal, \( \text{not } b \), is true according to negation as failure semantics if the atom \( b \) does not have derivation. Facts are rules an empty body. The derivation of facts is always true.

The results of ASP are answer sets (stable models) that satisfy the program, i.e., constitute a consistent assignment of truth values to the atoms in the program according to the prescribed rules.

ASP tools often enhance the core semantics with additional statements. An aggregate statement (e.g., \#sum, \#count) applies to an ASP atom set and returns a number. Special optimization statements are used to specify optimization criteria to be maximized (\#maximize) or minimized (\#minimize) when finding solutions to a problem. These criteria are expressed as integer values associated with the answer sets, and ASP solvers aim to find solutions that optimize these values according to the specified criteria.

The \#show statement specifies the projection of logical symbols to display in answer sets. It allows users to request specific information about the solution(s) produced by the solver, making it easier to analyze and interpret the results.

3 REQUIREMENT-BASED DISTRIBUTED LEDGER ARCHITECTURE GENERATION

In this paper, we propose a requirement-based approach for the automated generation of design candidates for distributed ledger networks. Figure 4 depicts the proposed workflow. Contributions specifically introduced in this paper are highlighted in bold.

In the following, we overview the inputs, outputs, and major steps of the workflow. In section 4, we instantiate this generic workflow for the generation of Hyperledger Fabric architectures.

3.1 Inputs and output

The inputs of the proposed workflow are as follows:

1. The functional requirements of the architecture can be expressed in a platform-independent manner, i.e., without reference to specific distributed ledger implementation concepts like Channel or Node instances.

2. In our case study, we consider collaboration requirements: the distributed ledger should either allow secure collaboration between two organizations, or disallow direct collaboration without involving a trusted third party.

The source of such requirements may be higher-level organizational requirements allocated to the distributed ledger, including existing organization collaboration patterns [24] to be transformed into a distributed ledger. Additionally, data sharing and privacy regulations [28] may require or constrain inter-organizational information flows.

2. The extra-functional requirements may include cost, reliability or performability requirements [34, 69, 72]. The simultaneous satisfaction of these requirements may require complex trade-offs and constrain the possible distributed ledger architectures. Similarly to functional requirements, our approach enables capturing extra-functional requirements independently of the implementation details of the selected distributed ledger.

3. Optionally, an architecture fragment (with platform-specific elements) may be added, which will be incorporated into any candidate designs. This allows both fine-tuning the generation to include or exclude specific designs and proposing extensions to existing distributed ledgers.

The output of the proposed workflow is a set of design candidates scored for decision-making according the specified extra-functional requirements. For a single requirement, this allows rank-ordering candidates, while for multiple requirements, we may sample the Pareto frontier of the possible trade-offs.

3.2 Formalization of requirements

In the I. Formalization step, the input requirements are translated into a partial model and Answer Set Programming (ASP) queries as an intermediate formal representation.

We use a partial model to capture platform-independent functional requirements. Concepts in the platform-independent vocabulary include the Organization instances and the desired or forbidden collaboratesWith links between them as shown in Figure 1. Other concepts for functional requirements, such as a data model [22] or models of the business processes to be executed by the distributed ledger [26, 59] could be incorporated by extending platform-independent part of the metamodel.

Platform-specific concepts, such as Channel and participatesIn in Example 2 describe distributed ledger architecture. The domain
We partition symbols as \( \Sigma \in \mathcal{P}_D \) using platform-dependent terms (e.g., projection, optimization) to compute invocation statements. We also decided that OrgA has only a single Host, which is represented as

\[
I_{P_0}^M(\text{hosts})(\text{OrgA, HostA}) = 1, \quad I_{P_0}(\text{hosts})(\text{OrgA, Host: new}) = 0.
\]

The architecture fragment refined the network design as \( P_0 \Rightarrow P_0' \).

Likewise, extra-functional requirements are translated into ASP invocation statements (e.g., projection, optimization) to compute extra-functional metrics, such as system-level fault probability, or total infrastructure costs. For the selected distributed ledger platform (e.g., Hyperledger Fabric), the analysis library defines the metrics using platform-dependent terms \( \Sigma_{PD} \) from the domain specification.

---

**Figure 5: Formalizing requirements with partial models**

**3.3 Model generation**

During the model generation step, we rely on a partial modeling based model generator to refine the initial partial model \( P_0 \) into concrete models \( P_M \Rightarrow M \) that represent a diverse set of candidate architectures. Consistency with the theory \( M \Rightarrow T \) ensures that the candidates satisfy functional requirements and design constraints.

We control the size of the concrete models with scope constraints [44, 52], which set the desired number of model elements and avoid unreasonably small (lacking proper redundancy) or large (excessively costly) design candidates. Multiple models can be created by either repeating the generation process (either sequentially or in parallel).

**Example 3.2.** Figure 6 shows a solution \( M \) of the model generation problem \( (\Sigma, \alpha, T, P_0') \), where \( (\Sigma, \alpha) \) and \( T \) represent the domain specification for HLF architectures (see section 4), and \( P_0' \) is the initial partial model with an architecture fragment from Figure 5b. By Definition 2.8, we have \( I_{P_0'}(\text{collaboratesWith})(0_1, 0_2) = I_M(\text{collaboratesWith})(0_1, 0_2) = \{d(\text{collaboratesWith})\}^{M}_{0_1, 0_2} \) for all organizations \( 0_1, 0_2 \in \mathcal{O}_M \), i.e., \( M \) satisfies the communication requirements prescribed in \( P_0' \) according to the platform-dependent definitions in the domain specification.

As technical implementation, we selected Refinery [51] as the partial modeling and model generation tool in our approach. Thus, the formalized partial models can be expressed by the user with the textual partial modeling notation of Refinery [53] and the domain specification can be linked with Refinery’s import mechanism.

**3.4 Extra-functional analysis**

The III. ASP solving step combines the ASP invocation statements selected according to the extra-functional requirements, the analysis library, and the candidate architectures into Answer Set Programs. By executing the program for each candidate, we calculate scores (i.e., the values of metrics) to support decision-making and enable the selection of a final architecture.
The proposed approach enables iteration and revision of require-
verse population of models [41, 47].

4 FORMALIZATION FOR HLF NETWORKS

3.5 Feedback and iterative development

The proposed approach enables iteration and revision of require-
ments according to feedback from architecture generation steps:

(a) If the model generation problem is found to be unsatisfiable,
inconsistencies in the initial partial model $P_0$ can be highlighted
to point out contradictions in the functional requirements or
the provided architecture fragment.

(b) By inspecting the generated design candidates, requirements
and the architecture fragment may be revised to exclude sol-
tions deemed infeasible, or further refine preferred solutions.

(c) By inspecting candidate scores, trade-offs between extra-
functional requirements may be identified. To generate further
variations of a specific design candidate, parts of it can be
incorporated as an architecture fragment into the input.

The user of the framework can impose further control over the
structure of the generated models (e.g., introduce constraints or
change the size of the models), while repeated generations with
solver-based model generators tend to produce a structurally di-
verse population of models [41, 47].

4 FORMALIZATION FOR HLF NETWORKS

In this section, we present our formalization for Hyperledger
Fabric architectures as Refinery [51] domain definition to capture
platform-independent and -dependent concepts and as a Clingo [53]
analysis library for extra-functional analysis.

abstract class FabricNetwork {
    contains Organization[1..*] organizations
    contains Channel[1..*] channels
} class RaftFabricNetwork extends FabricNetwork.
class KafkaFabricNetwork extends FabricNetwork.
class Organization { contains Host[1..*] hosts }
class Host { contains Node[1..4] nodes }
abstract class Node.
class EndorsingNode extends Node {
    ChaincodeInstance[1..*] chaincodes opposite endorsedBy
} class OrderingNode extends Node {
    Channel[1..*] orders opposite orderBy
} class Channel { contains ChaincodeInstance[1..*] chaincodes
    OrderingNode[0..*] orderBy opposite orders
} class ChaincodeInstance {
    EndorsingNode[2..*] endorsedBy opposite endorses
}

Figure 7: Refinery metamodel for HLF architectures

Positive information about the candidate $M$ is copied into the
program: if we have $I_M(\sigma)(o_1, o_2) = 1$ for some $\sigma \in \Sigma$ and $o_1, o_2 \in \Theta_M$, we add the fact $\sigma(o_1, o_2)$. The 0 values from $M$ do not need to
be copied, as the negation as failure semantics of ASP will interpret
the lack of a matching fact as falsehood by default. Therefore,
the facts database of the ASP matches the interpretation of $M$. This also keeps the program small (linear in the number of 1 values in $M$).

As a technical implementation, we selected Clingo [35] for solv-
ing ASP. Thus, the analysis library and invocation statements are
expressed using Clingo’s textual notation.

4.1 Domain definition

Our domain definition for HLF is based on the metamodel in Fig-
ure 1. Platform-independent concepts are the Organization class
and the collaboratesWith derived reference, while the rest of the
symbols formalize the HLF-specific distributed ledger implementa-
tion and the satisfaction of functional requirements.

4.1.1 Metamodel. Figure 7 shows the Refinery textual syntax [53]
(similar to Xcore [1] for EMF) for classes and references in the
metamodel. In Refinery, derived references are declared at their
definition site, thus, the code for the metamodel omits them. They
will be discussed along the formalization of the theory $\Gamma$ of derived
references and error patterns.

The root element of the architecture model is an instance of
FabricNetwork. It contains the Organization instances representing
the organization collaborating via HLF, as well as HLF Channels.
Each organization runs physical Hosts, where software components
– represented as Node instances in our metamodel – are deployed.

We formalize two consensus protocols supported by HFL. The
older, Apache Kafka [6] based consensus relies on dedicated Or-
deringNode instances to order transactions sent to Channels. In
contrast, in Raft [57] all nodes are assumed to participate in trans-
action ordering implicitly.

EndorsingNodes execute chaincodes (smart contracts) which im-
plement the transaction validation logic. Such nodes communicate
via Channels by sharing ledger state and transactions associated
with the chaincodes.

We chose not to model the chaincode implementations deployed to
EndorsingNodes directly; instead, a ChaincodeInstance is al-
located to the Channel. Implementation for any chaincodes en-
dorsedBy an EndorsingNode is assumed to be deployed to it. In the
future, our metamodel could be extended with further concepts if
tracking of specific chaincode implementations is required (e.g., to
enforce software redundancy by requiring multiple implementations
for transaction logic in each ChaincodeInstance).
4.1.2 Well-formedness constraints. Our formalization specifies well-formedness rules for valid HLF architectures. Some well-formedness rules are inferred from the metamodel by Refinery:

- **Containment constraints** (denoted with the `contains` keyword) ensure that each object except the root `FabricNetwork` has a single container and the containment hierarchy is acyclic.
- The **multiplicity constraint** ensures that each `Channel` of a common `EndorsingNode` participates in the channel. We cannot use the same constraint for `EndorsingNode`s, because `RaftFabricNetwork` has no explicit `OrderingNode`s.
- We represent the finite capacity of a `Host` to run nodes with an upper bound of `[1..4]`.

Figure 8 shows our additional error patterns (denoted with the `error` keyword) and derived references (denoted with the `pred` keyword). They are expressed in logical programming notation [53], where comma (,) and semicolon (;) correspond to logical or and and, respectively, and variables are existentially quantified.

The error patterns ensure that `OrderingNode`s are mandatory in `Kafka` networks but forbidden in `Raft` networks.

4.1.3 Initial partial model. Initial partial models may also be expressed using Refinery’s logical notation. Such a partial model is a collection of facts, either positive ($\sigma(o_1, o_2)$) or negative ($\neg\sigma(o_1, o_2)$). The interpretation of facts not provided in the partial model is considered `1/2` and multi-objects with the suffix `:new` are added for concrete classes automatically.

**Example 4.1.** The left side of Figure 9 shows the Refinery textual notation for the initial partial model $P_0$ in Figure 5a. After importing our domain specification, positive and negative `collaboratesWith` facts are provided.

To encode $P_0$ from Figure 5b, additional platform-specific facts on the right side of Figure 9 should also be included.

4.2 Analysis library

The analysis library defines the metrics using platform-dependent terms from the domain specification. As a syntactic limitation of Clingo [35], all logical symbols have their name start with a lowercase letter, e.g., we use `node` to refer to the `Node` type in Refinery. Otherwise, the set of ASP facts corresponds to the architecture model generated in the previous step.

4.2.1 Cost calculation. In our example formalization, we use a simple weighted function of the model elements shown in Figure 10 to determine infrastructure operational expenses. The auxiliary `networkUse` rule computes the `pairs` of `Node` instances that are peers of a common `Channel`, yet are located on distinct `Host`s. In the cost function, we assume that the upkeep of a single `Node` is 10 times as large as that of a network link. Note that we divide the number of `Node` pairs (N1, N2) with `networkUse` to account for bidirectional links. The corresponding `#show cost/2 invocation statement` causes the ASP solver to print the computed cost.

4.2.2 Resilience score. We define a general resilience score (Figure 11) to quantify the resilience of the HLF network architecture against independent `Host` or `Node` faults. Note that our score does not take interactions between `ChaincodeInstances` and the corresponding transaction into account. Thus, it is an application-agnostic metric of resilience. If desired, application-specific metrics could be developed in the future by further modeling transaction logic and analyzing the impact chain of failures.

We add auxiliary variables `hostFailure(_)` and `nodeFailure(_)` for faults of `Hosts` and `Nodes`, respectively. The failure of a `Host` also causes `nodes` to become inoperative. We assume a 2-out-of-n `Organizations` channel policy, i.e., an integrity violation is detected if some `Channel` has operative peers from less than 2 `Organizations`. The statement `:- not violation.` maximizes the score, i.e., find the most pessimistic outcome for each design candidate.

5 EVALUATION

We conducted an initial performance and diversity evaluation to answer the following research questions:

**RQ1:** How can model generation scale concerning the size of the generated architectures and a number of constraints?

**RQ2:** How diverse are the generated architectures concerning their cost and resilience?
A time limit of 5 minutes was set for the model generation. While ASP solving, the following computational configuration was used:

- Java VM: OpenJDK 64-Bit Server VM Corretto-21.0.2.13, Heap Space: 16GB
- OS: MacOS 14.4.1, CPU: 2.6 GHz 6-Core Intel i7, Memory: 32 GB 2667 MHz DDR4
- Java VM: OpenJDK 64-Bit Server VM Corretto-21.0.2.13, Heap Space: 16GB

5.1 Measurement setup

For Refinery model generation, architecture transformation and ASP solving, the following computational configuration was used. A time limit of 5 minutes was set for the model generation. While other generation methods like [40] invest an extensive computation resource to provide a few complex solutions, we aim to support model development with quick response.

5.2 Compared approaches and metrics

In our evaluation, we automatically generated a set of architectures for an increasing set of organizations and communication requirements. Problem size (Problem ID) can be defined by the type of architecture (Raft/Kafka), the size (i.e., the number of objects in the model), the organizations involved (Orgs), the number of Hyperledger Endorser and Orderer Nodes (Nodes), the communication constraints between the organizations (Constraints), and the number of channels (Channels). For Kafka-type problems, the scope is extended by explicitly stating the range of the number of Ordering and Endorsing nodes (Ordering and Endorsing Nodes) to help the generator create each type of node.

The measurements were carried out with several model sizes (Refinery scope). We defined a range of scenarios with increasing size in the candidate architecture. Table 1 summarizes the scope configurations. For each setting, 30 measurements were run (and 5 extra to mitigate any warm-up effects), and the following data collected were used for the evaluation:

- Problem ID: Identifies the architecture type.
- Refinery Runtime: Time to generate an instance model in seconds.
- ASP Runtime: Total ASP solving time in seconds.
- Cost: Total estimated cost defined in Figure 10.
- Resilience Score: A value derived from the number of Nodes and Orgs that should fail to violate integrity (defined in Figure 11).

5.3 RQ1: Scalability of model generation

To answer RQ1, we conducted a performance evaluation with increasing model sizes. We measured the runtime and success rates of the model generation process for each problem size and network type as proposed in [52]. Figure 12 shows the median of the successful model generation times and success rates (out of 30 runs).

For a single run, the model generator produces a single consistent concrete model, or times out. On the left y-axis (blue bar chart), we show the median runtime of successful generator runs (i.e., producing a consistent concrete model). On the right y-axis (red line chart), we show the percentage of runs below within the timeout limit. For configurations with a success rate above 0%, generator runs can be repeated as needed to produce a consistent model population. Therefore, runtime/success rate represents the expected time for generating a solution without restarts or parallel execution.

In case of Raft, it can be observed that larger models take longer time, and it is increasingly difficult to generalize the architecture. In contrast to the five-minute timeout, the median time to successfully generate the largest model is under 53 seconds. However, in the case of the largest model, the success rate fell below 20%.

Since, in the Kafka architecture, different Nodes are responsible for ordering and endorsement, it is necessary to generalize larger and more complex models. It can be observed that even with the smallest input configuration, the median time was around 40 seconds, while with kafka-size2 and 3 it was about 1 minute. For larger problems, no solution was found within the 5-minute timeout.

The complexity of the problem is also shown by the fact that while in the raft case, all model generation was successful in the case of the smallest configuration, here, this success rate is 30%.

For both architectures generation time increases with the number of Nodes and constraints to be generated. The generation time also depends greatly on the choice of scope that requires domain knowledge about the architecture.

The runtime of the ASP solver was negligible compared to the model generation (Figure 13a and 13b). The median ASP runtime for all successfully generated models was less than 0.02 seconds. It is observed that the ASP solving time increases with model size because of the number of possible failing component combinations.

Table 1: Raft and Kafka generation configurations

<table>
<thead>
<tr>
<th>Problem ID</th>
<th>size</th>
<th>Orgs</th>
<th>Nodes</th>
<th>Constraints</th>
<th>Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td>raft-size1</td>
<td>15.25</td>
<td>3</td>
<td>3.15</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>raft-size2</td>
<td>15.25</td>
<td>4</td>
<td>3.15</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>raft-size3</td>
<td>15.35</td>
<td>6</td>
<td>4.20</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>raft-size4</td>
<td>15.40</td>
<td>9</td>
<td>4.20</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>raft-size5</td>
<td>15.50</td>
<td>12</td>
<td>4.20</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>raft-size6</td>
<td>15.70</td>
<td>15</td>
<td>4.60</td>
<td>26</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 12: Median generation runtime and success rates

![Figure 12: Median generation runtime and success rates](image-url)
RQ1: The generation time scales with the number of nodes and constraints. In the evaluation, architectures with a maximum of 70 nodes were successfully generated. To generate a Kafka network with equivalent capability (in Org number and communication constraint) to the same Raft counterpart, requires about twice as many nodes and additional architectural constraints. However, in all cases, the successful model generations take a manageable amount of time (largest median within 110 seconds), even for large models (maximum potential size of 70 nodes).

5.4 RQ2: Cost-resilience trade-off

We evaluated the generated models using a cost-resilience analysis specific to Hyperledger Fabric networks. Each model estimated cost based on the number of objects (e.g., organizations, channels, hosts, nodes). The ledger resilience was measured through metrics derived from the ASP solver output, focusing on the number of node failures required to disrupt consensus on at least one channel.

To visualize this trade-off, we constructed Pareto fronts that depict the relationship between cost and resilience of the architecture for different problem sizes. These fronts show a curve where models with lower costs (fewer nodes) have lower resilience scores (fewer failures tolerated). As we move along the curve, models with increasing costs (more nodes) should demonstrate improved resilience scores. This visualization will allow users designing Hyperledger Fabric networks to select a model that best suits their specific needs based on the balance between cost and resiliency.

Figure 14 shows the cost-resilience trade-off for the generated architectures. Metrics are computed for each model individually. It is observable for Raft that smaller models have lower resilience, but their associated costs are also low. In the case of larger models, with the increasing number of Organizations, as the costs increase, so does the resilience score. The more organizations in the network, the better the consensus can be secured.

As evidenced by Figure 14, the generated structurally diverse models achieved a diversity of metrics and covered a variety of trade-offs along the Pareto front.

For Kafka architectures, it shows that it is difficult to generate diverse and resilient models due to the complexity (in terms of the number of nodes and constraints). Defining the scope for generation is also difficult, and Kafka is a more complex model generation task.

However, Kafka was deprecated in Hyperledger Fabric v2.x and is no longer supported in v3.x.

RQ2: With a larger model size, more diverse results are obtained, so that for large models (e.g., raft-size6), higher resilience (-6) can be achieved. The cost is also higher for the Kafka network due to twice as many peers compared to the Raft architecture (e.g., for size-2, the cost is higher than 500 for Kafka, it is below 250 for the Raft architecture). However, these increased costs are not associated with an increase in resilience.

6 RELATED WORK

Hyperledger Fabric modeling tools. As reflected by the survey of Curty et al. on the use of MDE in model-based applications [26], currently, model-driven approaches are dominantly employed for smart contract design. Specifically for BPMN and Ethereum-based multi-chain deployment, early, architecturally relevant results are appearing [14], but these do not approach “architecture” as a true first-class concept and, to our knowledge, have not been translated to Hyperledger Fabric yet.

Logically, the deployment automation needs of (consortial) blockchains should lead to at least DSLs relevant to architecture description. For Ethereum, [10] introduces KATENA, a framework that simplifies the deployment and management of blockchain applications. For Fabric, the Hyperledger Labs project Fablo uses declarative network descriptions in JSON for deployment. For deployment on Kubernetes, Helm charts are also available. However, to the best of our knowledge, these deployment-modeling DSLs do not have any design support yet.

Uncertain models. Partial models are similar to uncertain models, which offer a rich specification language [30, 62] amenable to analysis. They provide a more intuitive, user-friendly language compared to 3-valued interpretations, which allows the designers to annotate existing models with uncertainty (e.g. weather an object may or may not exists). This allows the developer to reuse existing models as initial designs plans, but it cannot be used to specify requirements. Moreover, uncertain models needs to be formatted as valid models, which disables specific structures (e.g., objects with multiple potential containers in Figure 5a). Additionally, uncertain models does not handle WF constraints natively.
Potential concrete models compliant with an uncertain model can be synthesized by the Alloy Analyzer [64], or refined by graph transformation rules [63]. Approaches like [31] analyze possible matches and executions of model transformation rules on partial models by using a SAT solver (MathSAT4) or by automated graph approximations using graph query engines [53].

**Generative architectures with logic solver approaches.** These approaches translate graphs and WF constraints into a logic formulae and use underlying solvers to generate graphs that satisfy them. Back-end technologies used for this purpose include SMT solver such as Z3 [45, 65, 75], SAT-based model finders (like Alloy [44]) [4, 9, 41, 49, 52, 54, 66], CSP-solvers [18–20, 37], theorem provers [8], first-order logic [12], constructive query containment [58] and higher-order logic [38]. For most of these approaches, scalability is limited to small models/counter-examples.

Some solver-based model generation approaches combine solver calls with other calculations: [55] proposes higher-order solver calls to evaluate more complex properties, and [34] relies on external numerical solvers to calculate and optimise metrics. Our proposed approach can be considered a special combination of those two, which aims to derive a wide range of design alternatives while calculating multiple metrics (including a resilience metric).

**Generative architectures with Design Space Explorers.** Graph-based DSE use graph transformations [3] or refactorings to generate candidate designs as graph models. They either rely on model-based search, where a graph model is mutated, or rule-based search, where solutions constructed as a sequence of graph transformations [46].

MOoToT [32] and MDEOptimiser [16] rely on the Henshin model transformation language [68] for model-based exploration. Consistency constraints pose a challenge for such approaches: they are either handled by relaxing hard constraints into soft constraints or by encoding them in the transformation rules. Burdel et al. [15] proposed the automated generation of transformation rules that preserve a limited class of hard constraints (multiplicity constraints).


**Hybrid approaches.** These approaches divide the model generation task into multiple sub-tasks and use a different underlying techniques to resolve each one. In this paper, we are using the Refinery framework [51] for model generation, which can be considered as a hybrid generation approach. The PLEDGE model generation tool [67] provides such a scalable implementation by combining meta-heuristic search for model structure generation with an SMT-solver based approach for attribute handling. The Evocon tool [43] implements a search-based evolutionary testing approach followed by symbolic execution to generate tests for object-oriented programs. Autograph [65] sequentially combines a tableau-based approach for model structure generation with an SMT-solvers for attributes. specified as partial models. The proposed approach uses the Refinery model generation framework to derive valid architecture candidates and answer set programming to analyze candidates concerning their resilience. We evaluate our framework by generating Apache Kafka and Raft networks.

In future work, we aim to extend our approach to PBFT networks [11], simplifying the network architecture design and thus potentially improving the performance of our approach. Additional materials and example generated models are available at: https://doi.org/10.5281/zenodo.13145716.

**Acknowledgements.** The work of N. M. S. A-G. and I. K. was partially funded by the European Union’s Digital Europe Programme (DEP) under grant agreement no. 101100768 (SM4EDID). The work of I. K. was also partially funded under DEP grant agreement no. 10123471 (EDGE-Skill). The work of A. F. and I. K. was partially performed in the DigitalTech EDHIT project (jointly funded by EU DEP grant no. 101085965 and national support agreement KTM-KAF/70/2/2014 of the Ministry of Regional Development). The work of K. M. and O. S. was partially supported by the Department of Navy award (N629092412063) issued by the Office of Naval Research. The work of K. M. was partially supported by the UNKFP-23-BME-239 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

**REFERENCES**
