Requirement-Driven Generation of Distributed Ledger Architectures

Noor Mohammed Sabr Al-Gburi
András Földvári
Kristóf Marussy
noor.algburi@edu.bme.hu
foldvari@mit.bme.hu
marussy@mit.bme.hu
Budapest University of Technology and Economics
Department of Artificial Intelligence and Systems Engineering
Budapest, Hungary

Oszkár Semeráth
Imre Kocsis
semerath@mit.bme.hu
ikocsis@mit.bme.hu
Budapest University of Technology and Economics
Department of Artificial Intelligence and Systems Engineering
Budapest, Hungary

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

ACM Reference Format:
Noor Mohammed Sabr Al-Gburi, András Földvári, Kristóf Marussy, Oszkár Semeráth, and Imre Kocsis. 2024. Requirement-Driven Generation of Distributed Ledger Architectures. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640310.3674097

MODELS '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0504-5/24/09
https://doi.org/10.1145/3640310.3674097

1 INTRODUCTION

Distributed ledger technology (DLT) [27], predominantly still implemented over the "blockchain" principles pioneered by Bitcoin, facilitates trustworthy collaboration between distrusting 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 cryptocurrency) 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 cooperations.

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.

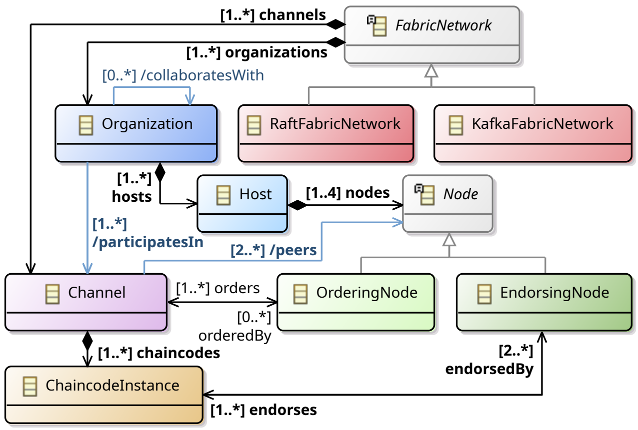


Figure 1: Hyperledger Fabric metamodel

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.

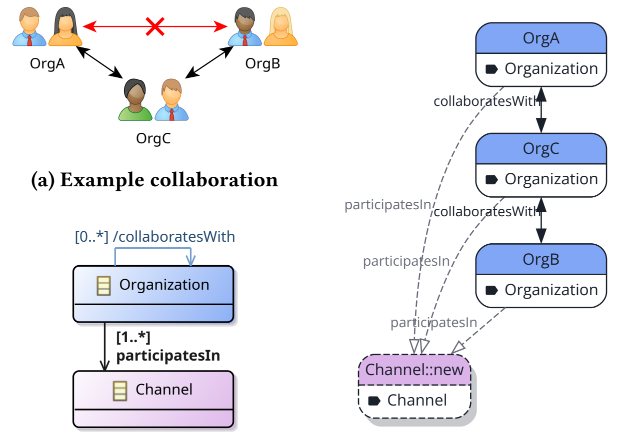
Chaincode is *instantiated on a channel* by deployment to the endorsing nodes of a channel. Subsequently, transaction processing follows an Execute-Order-Validate pattern. First, an organizational client requests the execution of a chaincode method from a number of endorsing nodes over their current (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 withing 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



(a) Example collaboration

(b) Fragment of the metamodel **(c) Corresponding partial model**

Figure 2: Secure data sharing running example

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 $\langle \Sigma, \alpha \rangle$, where the set of *symbols* Σ include unary *class* C_i and existence ε symbols and binary *reference* R_j, *derived reference* D_k, and *equivalence* $\sim$ symbols, while $\alpha\colon \sigma \to \mathbb{N}$ is the *arity* function $\alpha(\mathsf{C}_i) = \alpha(\varepsilon) = 1$, $\alpha(\mathsf{R}_j) = \alpha(\mathsf{D}_k) = \alpha(\sim) = 2$.

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 $\langle \Sigma, \alpha \rangle$ contains the classes $\mathsf{Organization}, \mathsf{Channel} \in \Sigma$, the reference $\mathsf{participatesIn} \in \Sigma$, and the derived reference $\mathsf{collaboratesWith} \in \Sigma$. Moreover, we have $\alpha(\mathsf{Organization}) = \alpha(\mathsf{Channel}) = \alpha(\varepsilon) = 1$, $\alpha(\mathsf{participatesIn}) = \alpha(\mathsf{collaboratesWith}) = \alpha(\sim) = 2$.

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

Definition 2.3. A theory $\mathcal{T}$ over the signature Σ, α is a pair $\langle d, \mathcal{E} \rangle$, where d maps each derived feature D_k to a FOL formula $d(\mathsf{D}_k)$ with *free variables* v_1, v_2, and the set of *error predicates* $\mathcal{E} = \{\psi_1, \dots, \psi_n\}$ is a finite set of FOL formulas.

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

$$d(\mathsf{collaboratesWith})(v_1, v_2) \coloneqq v_1 \neq v_2 \land \exists c\colon (\mathsf{participatesIn}(v_1, c) \land \mathsf{participatesIn}(v_2, c)).$$

We may add an error pattern $\psi \in \mathcal{E}$ to capture the lower multiplicity constraint [1..*] on participatesIn:

$$\psi(v_1) \coloneqq \text{Organization}(v_1) \land \neg\exists c\colon \text{participatesIn}(v_1, c),$$

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* $1/2$ 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 $\langle \Sigma, \alpha \rangle$ is a pair $\langle O_P, \mathcal{I}_P \rangle$, where O_P is a finite set of *objects* and $\mathcal{I}_P$ provides a 3-valued *interpretation* $\mathcal{I}_P(\sigma)\colon O_P^{\alpha(\sigma)} \to \{1, 0, 1/2\}$ 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 $\mathcal{I}_P(\mathrm{C}_i)(o) = 1, 0, 1/2$ if it is true, false, or unknown, respectively, whether the object $o \in O_P$ is of type C_i. We may similarly set $\mathcal{I}_P(\mathrm{R}_j)(o_1, o_2)$ and $\mathcal{I}_P(\mathrm{D}_k)(o_1, o_2)$ 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 $\mathcal{I}_P(\varepsilon)(o) = 1/2$ denotes *uncertain existence*, i.e., models that may be removed from the model. Objects with $\mathcal{I}_P(\varepsilon)(o) = 0$ can be removed outright. Objects with $\mathcal{I}_P(\sim)(o, o) = 1/2$ represent *multi-objects* that can be *split* to represent multiple concrete model elements. For simplicity, we will require $\mathcal{I}_P(\sim)(o_1, o_2) = 0$ 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 $O_P = \{$OrgA, OrgB, OrgC, Channel::new$\}$. Types are written inside the boxes corresponding to objects, e.g., we have $\mathcal{I}_P(\text{Organization})(\text{OrgA}) = 1$.

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

$$\mathcal{I}_P(\text{collaboratesWith})(\text{OrgA}, \text{OrgC}) = 1,$$

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

$$\mathcal{I}_P(\text{participatesIn})(\text{OrgA}, \text{Channel::new}) = 1/2.$$

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

$$\mathcal{I}_P(\text{collaboratesWith})(\text{OrgA}, \text{OrgB}) = 1.$$

The object Channel::new is a multi-object representing all Channel instances to be added, i.e., $\mathcal{I}_P(\varepsilon)(\text{Channel::new}) = 1/2$ (denoted with dashed border) and $\mathcal{I}_P(\sim)(\text{Channel::new}, \text{Channel::new}) = 1/2$ (denoted with a shadow).

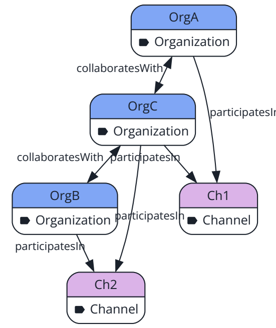
2.3 Semantics and consistency

Partial models allow us to evaluate FOL formulas φ 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 $[\![\varphi]\!]^P_Z$ of a FOL formula φ with *free variables* $v_1, \dots, v_n$ and *variable binding* $Z\colon \{v_1, \dots, v_n\} \to O_P$ on the partial model $P = \langle O_P, \mathcal{I}_P \rangle$ is given in Figure 3a [53].

$$
\begin{aligned}
[\![\mathrm{C}_i(v)]\!]^P_Z &= \mathcal{I}_P(\mathrm{C}_i)(v),\\
[\![\mathrm{R}_j(v_1, v_2)]\!]^P_Z &= \mathcal{I}_P(\mathrm{R}_j)(v_1, v_2),\\
[\![\mathrm{D}_k(v_1, v_2)]\!]^P_Z &= \mathcal{I}_P(\mathrm{D}_k)(v_1, v_2),\\
[\![v_1 = v_2]\!]^P_Z &= \mathcal{I}_P(\sim)(v_1, v_2),\\
[\![\neg\varphi]\!]^P_Z &= 1 - [\![\varphi]\!]^P_Z,\\
[\![\varphi_1 \lor \varphi_2]\!]^P_Z &= \max\{[\![\varphi_1]\!]^P_Z, [\![\varphi_2]\!]^P_Z\},\\
[\![\varphi_1 \land \varphi_2]\!]^P_Z &= \min\{[\![\varphi_1]\!]^P_Z, [\![\varphi_2]\!]^P_Z\},\\
[\![\exists v\colon \varphi]\!]^P_Z &= \\
\max\nolimits_{o \in O_P} \min\{&\mathcal{I}_P(\varepsilon)(o), [\![\varphi]\!]^P_{Z, v \mapsto o}\},\\
[\![\forall v\colon \varphi]\!]^P_Z &= [\![\neg\exists v\colon \neg\varphi]\!]^P_Z
\end{aligned}
$$

(a) 3-valued semantics



(b) Example concrete model

Figure 3: Partial model semantics and refinement

Definition 2.8. A partial model $P = \langle O_P, \mathcal{I}_P \rangle$ over the signature $\langle \Sigma, \alpha \rangle$ is *consistent* with the theory $\mathcal{T} = \langle d, \mathcal{E} \rangle$, written as $P \models \mathcal{T}$, if

- $[\![d(\mathrm{D}_k)]\!]^P_{v_1 \mapsto o_1, v_k \mapsto o_2} \succcurlyeq \mathcal{I}_P(\mathrm{D}_k)(o_1, o_2)$ for all derived references $\mathrm{D}_k \in \Sigma$ and objects $o_1, o_2 \in O_P$, and
- $[\![\psi]\!]^P_Z \succcurlyeq 0$ for all error predicates $\psi \in \mathcal{E}$ and bindings Z of the free variables of ψ,

where $1/2 \succcurlyeq 1/2$, $1/2 \succcurlyeq 1$, $1/2 \succcurlyeq 0$, $1 \succcurlyeq 1$, $0 \succcurlyeq 0$, is the *information order* on $\{1, 0, 1/2\}$ [53]. If P is concrete, we may replace $\succcurlyeq$ with $=$.

2.4 Refinement and model generation

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

Definition 2.9. The function $\mathrm{bwd}\colon O_Q \to O_P$ is a *refinement function* from the partial model P to Q, written as $P \succcurlyeq_{\mathrm{bwd}} Q$, if

- for all $\sigma \in \Sigma$ and $q_1, \dots, q_{\alpha(\sigma)}$, we have $\mathcal{I}_P(\sigma)(\mathrm{bwd}(q_1), \dots, \mathrm{bwd}(q_{\alpha(\sigma)})) \succcurlyeq \mathcal{I}_Q(\sigma)(q_1, \dots, q_{\alpha(\sigma)})$; and
- surely existing objects do not disappear, i.e., there is some $q \in O_Q$ with $\mathrm{bwd}(q) = p$ for all $p \in O_P$ with $\mathcal{I}_P(\varepsilon)(p) = 1$.

Definition 2.10. The *model generation problem* [53] $\langle \Sigma, \alpha, \mathcal{T}, P_0 \rangle$ consist of a *metamodel signature* $\langle \Sigma, \alpha \rangle$, a *theory* $\mathcal{T}$, 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., $P_0 \succcurlyeq M$) and is consistent with $\mathcal{T}$ (i.e., $M \models \mathcal{T}$).

Automated model generators [52, 53] derive such solutions M along a *chain of refinements* $P_0 \succcurlyeq P_1 \succcurlyeq \cdots P_n \succcurlyeq M$, exploiting the (i) *associativity* of partial model refinement $\succcurlyeq$ 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 $P_0 \succcurlyeq_{\mathrm{bwd}} M$ of the partial model P_0 shown in Figure 2c with

$$\mathrm{bwd}(\text{OrgA}) = \text{OrgA}, \mathrm{bwd}(\text{OrgB}) = \text{OrgB}, \mathrm{bwd}(\text{OrgC}) = \text{OrgC},$$
$$\mathrm{bwd}(\text{Ch1}) = \mathrm{bwd}(\text{Ch2}) = \text{Channel::new}.$$

Since it is consistent ($M \models \mathcal{T}$) with the theory $\mathcal{T}$ from Example 2.4, M is a solution of the model generation problem for $\mathcal{T}$ 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 programs consist of *atoms* of the form $\sigma(x_1, \dots, x_n)$, where $\sigma \in \Sigma$ is a logical symbol and $x_1, \dots, x_n$ are variables or objects constants, as well as *literals*, and *rules* of the form

$$\texttt{r :- } a_1, \dots a_n, \textbf{not } b_1, \dots \textbf{not } b_m.$$

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, *not* b_i, 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.
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.

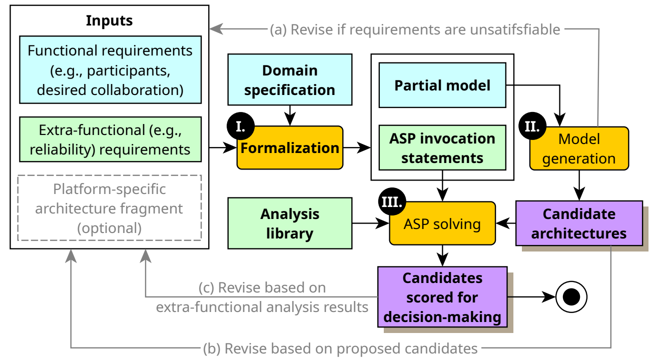


Figure 4: Proposed workflow for requirement-based distributed ledger architecture generation

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.

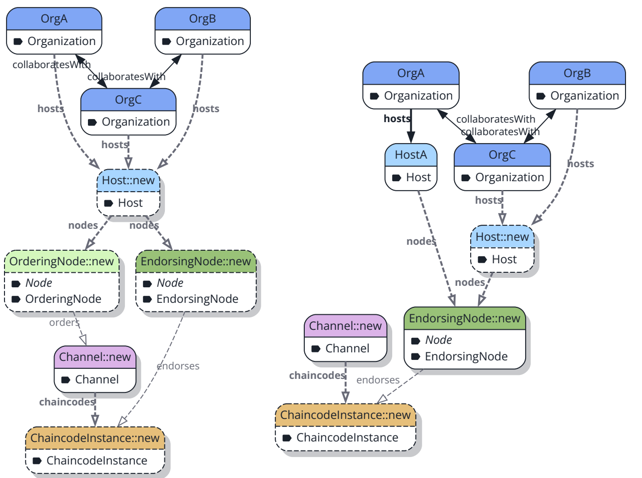
The output of the proposed workflow is a set of **design candidates scored for decision-making** according the specified extra-functional requirements. For a single requirement, this allows rank-ordering candidates, while for multiple requirements, we may sample the *Pareto frontier* of the possible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



(a) Initial partial model P_0

(b) Initial partial model with platform-specific architecture fragment P_0'

Figure 5: Formalizing requirements with partial models

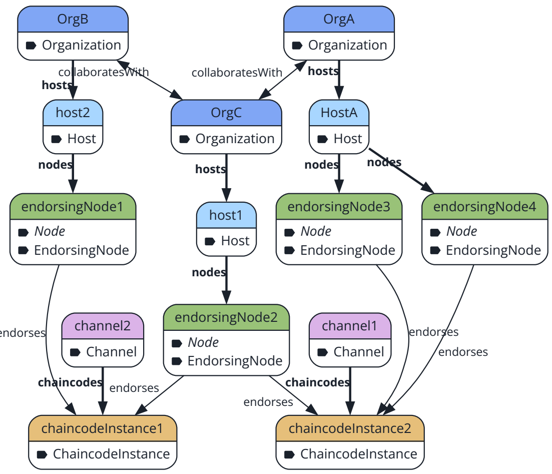


Figure 6: Generated architecture model M

specification corresponding to the selected distributed ledger technology consist of a (1) *metamodel* of both platform-independent and -dependent concepts; (2) *derived reference definitions*, where each platform-independent concept is mapped to platform-dependent concepts implementing it; and (3) *well-formedness rules* expressing design rules and realizable architectures.

In line with the semantic basis presented in section 2, we capture (1) as a signature $\langle \Sigma, \alpha \rangle$, and (2) and (3) as the derived references d and error patterns $\mathcal{E}$ of a theory $\mathcal{T} = \langle d, \mathcal{E} \rangle$, respectively. We partition symbols as $\Sigma = \Sigma_{\text{PI}} \uplus \Sigma_{\text{PD}}$ into platform-independent and -dependent symbols. Formulas $d(\text{D}_k)$ express derived references $\text{D}_k \in \Sigma_{\text{PI}}$ using symbols from Σ_{PD}.

Example 3.1. Figure 5a shows a partial model P_0 capturing the collaboration requirements from Figure 2a. In addition to the platform-independent concepts $\{\text{Organization}, \text{collaboratesWith}\} \subseteq \Sigma_{\text{PI}}$, platform-specific concepts (e.g., Host, OrderingNode, EndorsingNode) for Hyperledger Fabric have been added by the domain library. The interpretation $\mathcal{I}_{P_0}(\sigma)$ of platform-specific symbols $\sigma \in \Sigma_{\text{PD}}$ is all $1/2$, since the implementation remains unconstrained.

P_0' in Figure 5b also incorporates preliminary decisions about the architecture. As *Raft* was selected as the consensus protocol, which does not rely on any dedicated ordering nodes, we have $\mathcal{I}_{P_0'}(\varepsilon)(\text{OrderingNode::new}) = \mathtt{0}$. We also decided that OrgA has only a single Host, HostA, which is represented as

$$\mathcal{I}_{P_0'}(\text{hosts})(\text{OrgA}, \text{HostA}) = 1, \quad \mathcal{I}_{P_0'}(\text{hosts})(\text{OrgA}, \text{Host::new}) = \mathtt{0}.$$

The architecture fragment *refined* the network design as $P_0 \succcurlyeq P_0'$.

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 Σ_{PD} from the domain specification.

3.3 Model generation

During **II. Model generation,** we rely on a *partial modeling based model generator* to the refine the *initial partial model* P_0 into concrete models $P_0 \succcurlyeq M$ that represent a *diverse* set of **candidate architectures.** Consistency with the theory $M \models \mathcal{T}$ 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 $\langle \Sigma, \alpha, \mathcal{T}, P_0' \rangle$, where $\langle \Sigma, \alpha \rangle$ and $\mathcal{T}$ 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 $\mathcal{I}_{P_0'}(\text{collaboratesWith})(o_1, o_2) \succcurlyeq \mathcal{I}_M(\text{collaboratesWith})(o_1, o_2) = [\![d(\text{collaboratesWith})]\!]^M_{o_1,o_2}$ for all organizations $o_1, o_2 \in \mathcal{O}_M$, 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.

```
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..*] endorses opposite endorsedBy }
class OrderingNode extends Node {
	Channel[1..*] orders opposite orderedBy }
class Channel { contains ChaincodeInstance[1..*] chaincodes
		OrderingNode[0..*] orderedBy 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 $\mathcal{I}_M(\sigma)(o_1, o_2) = 1$ for some $\sigma \in \Sigma$ and $o_1, o_2 \in \mathcal{O}_M$, 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 solving ASP. Thus, the analysis library and invocation statements are expressed using Clingo's textual notation.

3.5 Feedback and iterative development

The proposed approach enables iteration and revision of requirements 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 solutions 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 diverse 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.

```
% Error patterns
error orderingNodeInRaftNetwork(n) <->
	RaftFabricNetwork(hlf), organizations(hlf, o),
	hosts(o, h), nodes(h, n), OrderingNode(n).
error channelWithoutOrderingNodeInKafkaNetwork(n) <->
	KafkaFabricNetwork(hlf), channels(hlf, c), !orderedBy(c, _).
% Derived features
pred peers(ch, n) <-> endorsedBy(ch, n)
     ; chaincodes(ch, ci), orderedBy(ci, n).
pred participatesIn(org, ch) <->
	hosts(org, h), nodes(n, h), peers(ch, n).
pred collaboratesWith(o1, o2) <->
	o1 != o2, participatesIn(o1, ch), participatesIn(o2, ch).
```

Figure 8: Error patterns and derived references (excerpt)

P_0

```
import hyperledger_fabric.
collaboratesWith(OrgA, OrgC).
collaboratesWith(OrgB, OrgC).
!collaboratesWith(OrgA, OrgB).
```

$P_0 \succcurlyeq P'_0$

```
RaftFabricNetwork(Hlf).
hosts(OrgA, HostA).
!hosts(OrgA, Host::new).
!exists(OrderingNode::new).
```

Figure 9: Initial partial models P_0 and P'_0

4.1 Domain definition

Our domain definition for HLF is based on the metamodel in Figure 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 implementation 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 $\mathcal{T}$ 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 OrderingNode instances to order transactions sent to Channels. In contrast, in *Raft* [57] all nodes are assumed to participate in transaction ordering implicitly.

EndorsingNodes execute *chaincodes* (*smart contracts*) which implement 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 impelementations* deployed to EndorsingNodes directly; instead, a ChaincodeInstance is allocated to the Channel. Implementation for any chaincodes endorsedBy 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* `[2..*]` on endorsedBy ensures that each ChaincodeInstance has at least 2 EndorsingNodes to maintain concensus. We cannot use the same constraint for orderedBy, because RaftFabricNetworks have no explicit OrderingNodes.
- 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 OrderingNodes are mandatory in *Kafka* networks but forbidden in *Raft* networks.

Crucially, we use derived references for checking the satisfaction of functional requirements. A Node is considered a peer of a Channel and hence having access to all transaction data if it is (i) an ordering node for the channel in a *Kafka* network, or (ii) is endorsing transactions for some ChaincodeInstance of the channel. Organizations participateIn a Channel if they host some peer of it. Organizations collaborateWith each other if they participateIn some common Channel.

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 ($!\sigma(o_1, o_2)$). 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 **import**ing 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 Hosts. 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.

```
% Network links
networkUse(N1, N2) :- nodes(H1, N1), nodes(H2, N2), H1 != H2,
     peers(N1, Channel), peers(N2, Channel).
% Cost function
cost(COST) :- A = #count{Node : node(Node)},
    B = #count{N1, N2 : networkUse(N1, N2)}, COST = 10 * A + B / 2.
% Invocation statement for cost calculation
#show cost/1.
```

Figure 10: Cost calculation

```
% Combinatorial variables for Node and Host failures
{hostFailure(Host) : host(Host)}. {nodeFailure(Node) : node(Node)}.
inoperative(Node) :- nodeFailure(Node).
inoperative(Node) :- nodes(Host, Node), hostFailure(Host).
% Violation if there are less than two active Orgs per Channel
activeOrgs(Channel, Org) :- peers(Channel, Node), nodes(Host, Node),
     hosts(Org, Host), not inoperative(Node).
violation :- channel(Channel),
     #count{Org : activeOrgs(Channel, Org)} < 2.
% Checking the cases where active faults violated the constraint
:- not violation.
% Score function
resilienceScore(SCORE) :- A = #count{Host : hostFailure(Host)},
    B = #count{Node : nodeFailure(Node)}, SCORE = -2 * A - 1 * B.
% Invocation statement for optimization
#maximize(SCORE : resilienceScore(SCORE)).
```

Figure 11: Resilience score calculation

4.2.2 Resilience score. We define a general *resilience score* (Figure 11) to quantify the resiliece 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 analysing 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 instructs the ASP solver to only consider answer sets where violation is true.

In the resilience score, hostFailure has a weight of −2 and nodeFailure has a weight of −1. *Lower* resilience scores are *better,* i.e., indicate that more failures are needed to compromise ledger integrity. In the invocation statement, we #**maximize** 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?

Problem ID	size	Orgs	Nodes	Constraints	Channels
raft-size1	15..25	3	3..15	3	3..*
raft-size2	15..25	4	3..15	6	4..*
raft-size3	15..35	6	4..20	9	4..*
raft-size4	15..40	9	4..20	14	4..*
raft-size5	15..50	12	4..20	20	5..*
raft-size6	15..70	15	4..60	26	5..*

Problem ID	size	Orgs	Nodes	Ordering and Endorsing	Constraints	Channels
kafka-size1	15..60	3	8..30	4..15	3	3..*
kafka-size2	15..60	4	8..30	4..15	6	4..*
kafka-size3	20..70	6	8..50	4..25	9	4..*
kafka-size4	20..90	9	8..50	4..25	14	4..*
kafka-size5	20..120	12	10..60	5..30	20	5..*
kafka-size6	20..140	15	10..80	5..40	26	5..*

Table 1: Raft and Kafka generation configurations

5.1 Measurement setup

For Refinery model generation, architecture transformation and ASP solving, the following computational configuration was used[1]. 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 Hosts 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

[1]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-21.0.2.13.1, Heap Space: 16GB

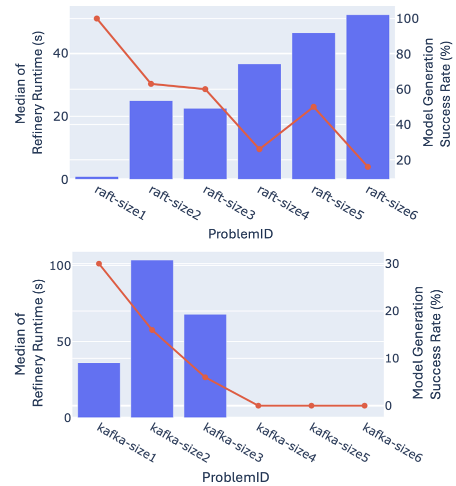


Figure 12: Median generation runtime and success rates

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.

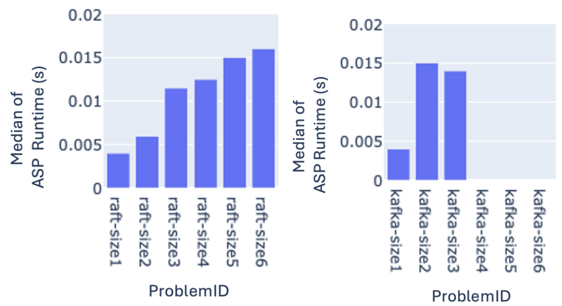
In case of Raft, it can be observed that larger models take longer time, and it is increasingly difficult to generalize the architecture. In contrast to the five-minute timeout, the median time to successfully generate the largest model is under 53 seconds. However, in the case of the largest model,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



(a) Raft configurations (b) Kafka configurations

Figure 13: Median of ASP solving times

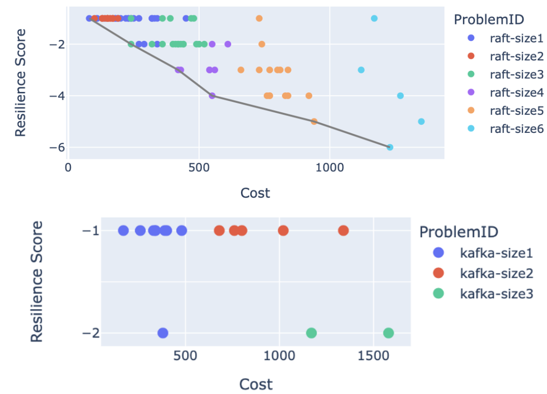


Figure 14: Resilience Score over Cost

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].

MOMoT [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. Burdusel et al. [15] proposed the automated generation of transformation rules that preserve a limited class of hard constraints (multiplicity constraints).

Viatra-DSE [2, 39] is a rule-based DSE tool that relies on the Viatra [73] language. SHEPhERd [25] and EASIER [7] aim to derive sequences software architecture refactorings according to extra-functional criteria.

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 Evacon 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.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel architecture generation framework for Hyperledger Fabric networks, which can enumerate possible valid designs while satisfying organization-level requirements 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 examle 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 (SME4DD). The work of I. K. was also partially funded under DEP grant agreement no. 101123471 (EDGE-Skills). The work of A. F. and I. K. was partially performed in the DigitalTech EDIH project (jointly funded by EU DEP grant no. 101083965 and national support agreement KTM-KÁF/70/2/2024 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 ÚNKP-23-4-II-BME-219 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

REFERENCES

[1] 2020. *Xcore.* https://wiki.eclipse.org/Xcore.

[2] Hani Abdeen, Dániel Varró, Houari Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. 2014. Multi-objective optimization in rule-based design space exploration. In *ASE*. ACM, 289–300.

[3] Aditya Agrawal, Tihamer Levendovszky, Jon Sprinkle, Feng Shi, and Gabor Karsai. 2002. Generative Programming via Graph Transformations in the Model-Driven Architecture. In *Workshop on Generative Techniques in the Context of Model Driven Architecture, OOPSLA*.

[4] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2010. On challenges of model transformation from UML to Alloy. *Softw. Syst. Model.* 9, 1 (2010), 69–86.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Srinivasan Muralidharan, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Chet Murthy, Christopher Ferris, Gennady Laventman, Yacov Manevich, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains.

[6] Apache Foundation. 2023. *Kafka.* https://kafka.apache.org/.

[7] Davide Arcelli, Vittorio Cortellessa, Mattia D'Emidio, and Daniele Di Pompeo. 2018. EASIER: An Evolutionary Approach for Multi-objective Software ArchItecturE Refactoring. In *ISCA*. IEEE, 105–114.

[8] Aren A Babikian, Oszkár Semeráth, and Dániel Varró. 2020. Automated Generation of Consistent Graph Models with First-Order Logic Theorem Provers. In *FSE*. Springer, 441–461.

[9] Kacper Bak, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. Clafer: unifying class and feature modeling. *Softw. Syst. Model.* (2013), 1–35.

[10] Luciano Baresi, Giovanni Quattrocchi, Damian Andrew Tamburri, and Luca Terracciano. 2022. A Declarative Modelling Framework for the Deployment and Management of Blockchain Applications. In *MODELS*. ACM.

[11] Artem Barger, Yacov Manevich, Hagar Meir, and Yoav Tock. 2021. A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric. *International Conference on Blockchain* (2021).

[12] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. 2002. Translating the Object Constraint Language into First-order Predicate Logic. In *VERIFY@FLoC*.

[13] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. 2022. A Survey on Blockchain Interoperability: Past, Present, and Future Trends. *ACM Comput. Surv.* 54, 8 (2022), 168:1–168:41.

[14] Peter Bodorik, Christian Gang Liu, and Dawn Jutla. 2023. TABS: Transforming automatically BPMN models into blockchain smart contracts. *Blockchain: Research and Applications* 4, 1 (2023), 100115.

[15] Alexandru Burdusel, Steffen Zschaler, and Stefan John. 2021. Automatic generation of atomic multiplicity-preserving search operators for search-based model engineering. *Softw. Syst. Model.* 20, 6 (2021), 1857–1887.

[16] Alexandru Burdusel, Steffen Zschaler, and Daniel Strüber. 2018. MDEoptimiser: A Search Based Model Engineering Tool. In *MODELS*. ACM, 12–16.

[17] Jasper H Bussemaker, Pier Davide Ciampa, and Bjoern Nagel. 2020. System architecture design space exploration: An approach to modeling and optimization. In *AIAA Aviation 2020 Forum*. 3172.

[18] Fabian Büttner and Jordi Cabot. 2012. Lightweight String Reasoning for OCL. In *ECMFA (LNCS, Vol. 7349)*, Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos (Eds.). Springer, 244–258.

[19] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2007. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE 2017*. ACM, 547–548.

[20] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2014. On the Verification of UML/OCL Class Diagrams using Constraint Programming. *J. Syst. Softw.* (2014).

[21] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. 2019. ASP-Core-2 Input Language Format. *Theor. Practice Logic* 20, 2 (2019), 294–309.

[22] Jeeta Ann Chacko, Ruben Mayer, and Hans-Arno Jacobsen. 2023. How To Optimize My Blockchain? A Multi-Level Recommendation Approach. *Proc. ACM Manag. Data* 1, 1 (2023), 24:1–24:27.

[23] ConsenSys. 2024. Quorum. https://github.com/ConsenSys/quorum

[24] James O. Coplien and Neil B. Harrison. 2004. *Organizational Patterns of Agile Software Development*. Prentice-Hall.

[25] Vittorio Cortellessa, Raffaela Mirandola, and Pasqualina Potena. 2015. Managing the evolution of a software architecture at minimal cost under performance and reliability constraints. *Sci. Comput. Program.* 98 (2015), 439–463.

[26] Simon Curty, Felix Härer, and Hans-Georg Fill. 2023. Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: a structured literature review. *Softw. Syst. Model.* 22 (2023), 1857–1895.

[27] Larry A. DiMatteo, Michel Cannarsa, and Cristina Poncibò (Eds.). 2019. *Part I - General Framework*. Cambridge Univ. Press, 1–58.

[28] Larry A. DiMatteo, Michel Cannarsa, and Cristina Poncibò (Eds.). 2019. *Part IV - Privacy, Security and Data Protection*. Cambridge Univ. Press, 221–268.

[29] Ethereum Foundation. [n. d.]. Ethereum documentation. https://ethereum.org/en/learn/

[30] Michalis Famelis, Rick Salay, and Marsha Chechik. 2012. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE*. IEEE, 573–583.

[31] Michalis Famelis, Rick Salay, Alessio Di Sandro, and Marsha Chechik. 2013. Transformation of models containing uncertainty. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 673–689.

[32] Martin Fleck, Javier Troya, and Manuel Wimmer. 2016. Search-Based Model Transformations with MOMoT. In *ICMT (LNCS, Vol. 9765)*. Springer, 79–87.

[33] Linux Foundation. [n. d.]. How Fabric networks are structured — Hyperledger Fabric Docs main documentation. https://hyperledger-fabric.readthedocs.io/en/latest/network/network.html

[34] Máté Földiák, Kristóf Marussy, Dániel Varró, and István Majzik. 2022. System architecture synthesis for performability by logic solvers. In *MODELS*. ACM, 43–54.

[35] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-shot ASP solving with clingo. *Theor. Practice Logic Program.* 19, 1 (2019), 27–82.

[36] Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming.. In *ICLP/SLP*, Vol. 88. 1070–1080.

[37] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. 2012. EMFtoCSP: a tool for the lightweight verification of EMF models. In *FormSERA*. 44–50.

[38] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. 2009. System Model-Based Definition of Modeling Language Semantics. In *FORTE (LNCS, Vol. 5522)*. Springer, 152–166.

[39] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. 2015. A model-driven framework for guided design space exploration. *Autom. Softw. Eng.* 22, 3 (2015), 399–436.

[40] Sebastian J. I. Herzig, Sanda Mandutianu, Hongman Kim, Sonia Hernandez, and Travis Imken. 2017. Model-transformation-based computational design synthesis for mission architecture optimization. In *IEEE Aerospace Conference*. IEEE.

[41] Frank Hilken, Martin Gogolla, Loli Burgueño, and Antonio Vallecillo. 2018. Testing models and model transformations using classifying terms. *Soft. Syst. Model.* 17, 3 (2018), 885–912.

[42] Hyperledger Foundation. 2024. Use Case Tracker. https://www.hyperledger.org/learn/use-case-tracker

[43] Kobi Inkumsah and Tao Xie. 2008. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *ASE*. 297–306.

[44] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

[45] Ethan K Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. 2011. Reasoning about metamodeling with formal specifications and automatic proofs. In *Model Driven Engineering Languages and Systems*. Springer, 653–667.

[46] Stefan John, Alexandru Burdusel, Robert Bill, Daniel Strüber, Gabriele Taentzer, Steffen Zschaler, and Manuel Wimmer. 2019. Searching for Optimal Models: Comparing Two Encoding Approaches. *J. Object Technol.* 18, 3 (2019), 6:1–22.

[47] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. 2011. An approach for effective design space exploration. In *16th Monterey Workshop*. Springer, 33–54.

[48] Stephen Cole Kleene, NG De Bruijn, J de Groot, and Adriaan Cornelis Zaanen. 1952. *Introduction to metamathematics*. van Nostrand New York.

[49] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. 2011. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *TOOLS (LNCS, Vol. 6705)*. 290–306.

[50] Vladimir Lifschitz. 2019. *Answer set programming*. Springer Berlin.

[51] Kristóf Marussy, Attila Ficsor, Oszkár Semeráth, and Dániel Varró. 2024. Refinery: Graph Solver as a Service.

[52] Kristof Marussy, Oszkar Semerath, and Daniel Varro. 2022. Automated Generation of Consistent Graph Models With Multiplicity Reasoning. *IEEE Tran. Softw. Eng.* 48 (5 2022), 1610–1629. Issue 5.

[53] Kristóf Marussy, Oszkár Semeráth, Aren A. Babikian, and Dániel Varró. 2020. A specification language for consistent model generation based on partial models. *J. Obj. Technol.* 19 (2020). Issue 3.

[54] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2017. Relational Constraint Solving in SMT. In *CADE (LNCS, Vol. 10395)*. Springer, 148–165.

[55] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In *ICSE*. 609–619.

[56] Debajani Mohanty. 2019. *R3 Corda for Architects and Developers: With Case Studies in Finance, Insurance, Healthcare, Travel, Telecom, and Agriculture*. Apress.

[57] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX*, Garth Gibson and Nickolai Zeldovich (Eds.). 305–319.

[58] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. 2012. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.* 73 (2012), 1–22.

[59] Aidin Rasti, Daniel Amyot, Alireza Parvizimosaed, Marco Roveri, Luigi Logrippo, Amal Ahmed Anda, and John Mylopoulos. 2022. Symboleo2SC: from legal contract specifications to smart contracts. In *MODELS*, Eugene Syriani, Houari A. Sahraoui, Nelly Bencomo, and Manuel Wimmer (Eds.). ACM, 300–310.

[60] Michel Rauchs, Andrew Glidden, Brian Gordon, Gina C. Pieters, Martino Recanatini, François Rostand, Kathryn Vagneur, and Bryan Zheng Zhan. 2019. Distributed Ledger Technology Systems: A Conceptual Framework. *SSRN* (2019).

[61] Thomas W Reps, Mooly Sagiv, and Reinhard Wilhelm. 2004. Static program analysis via 3-valued logic. In *CAV*. 15–30.

[62] Rick Salay and Marsha Chechik. 2015. A Generalized Formal Framework for Partial Modeling. In *FASE*, Alexander Egyed and Ina Schaefer (Eds.). LNCS, Vol. 9033. Springer, 133–148.

[63] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. 2015. A Methodology for Verifying Refinements of Partial Models. *J. Obj. Technol.* 14, 3 (2015), 3:1–31.

[64] Rick Salay, Michalis Famelis, and Marsha Chechik. 2012. Language Independent Refinement Using Partial Modeling. In *FASE*. Springer, 224–239.

[65] Sven Schneider, Leen Lambers, and Fernando Orejas. 2018. Automated reasoning for attributed graph properties. *STTT* 20, 6 (2018), 705–737.

[66] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. 2009. From UML to Alloy and back again. In *MoDeVVa* (Denver, Colorado). ACM, 1–10.

[67] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. 2020. Practical Constraint Solving for Generating System Test Data. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 11 (2020), 48 pages.

[68] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. 2017. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In *ICGT@STAF (LNCS, Vol. 10373)*. Springer, 196–208.

[69] Harish Sukhwani, Nan Wang, Kishor S. Trivedi, and Andy J. Rindos. 2018. Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network). In *NCA*. IEEE, 1–8. https://doi.org/10.1109/NCA.2018.8548070

[70] The Eclipse Project 2019. *Eclipse Modeling Framework*. The Eclipse Project. http://www.eclipse.org/emf.

[71] The Object Management Group 2014. *Object Constraint Language, v2.4*. The Object Management Group.

[72] Kishor S. Trivedi, Gianfranco Ciardo, Manish Malhotra, and Robin A. Sahner. 1993. Dependability and Performability Analysis. In *SIGMETRICS (LNCS, Vol. 729)*. Springer, 587–612.

[73] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98, 1 (2015), 80–99.

[74] World Economic Forum and Accenture. 2019. *Building Value with Blockchain Technology: How to Evaluate Blockchain's Benefits*. Technical Report. World Economic Forum.

[75] Hao Wu, Rosemary Monahan, and James F. Power. 2013. Exploiting Attributed Type Graphs to Generate Metamodel Instances Using an SMT Solver. In *TASE*. 175–182.