Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Design for Dependability in Distributed Ledger Systems

Scientific Student Competition **Report**

Author:
Bertalan Zoltán Péter

Supervisor:
Dr. Imre Kocsis

2022-11-01

Abstract

As their primary capability, blockchains and Distributed Ledger Technologies provide very high integrity data storage without a single trusted party. However, as mounting evidence shows, guaranteeing other extra-functional requirements, such as availability and reliability, is a nontrivial engineering challenge, especially when the network is created, designed, and configured for a specific consortium of organizations or for cross-organizational purposes. Smart contracts deployed on blockchains pose a particular challenge; fault- and attack tolerant consensus protocols do not protect against their potential faults, and development time fault avoidance and removal techniques are not mature yet.

On the other hand, permissioned consensus, restricted access, execute-order-validate blockchain platforms, such as Hyperledger Fabric, can be designed with specific extra-functional requirements in mind and offer a range of runtime dependability mechanisms – two key aspects which are neither appreciated nor utilized in their current application practice.

In my work, I give a novel systematic characterization of the internal fault and error modes of the Hyperledger Fabric architecture, its error containment capabilities, and identify known patterns of fault tolerance which can be readily deployed in it. I propose, prototype, and demonstrate the application of modern qualitative Error Propagation Analysis (EPA) for evaluating the expected end-to-end effectiveness of deployed defences.

N-Version Programming (NVP) is a dependability technique which is especially well-suited to Hyperledger Fabric. I propose two architectural patterns for its introduction and investigate the expected effects of the increased software diversity on attack tolerance, availability and integrity.

Kivonat

A blokklánc és elosztott főkönyv alapú rendszerek elsődleges képessége, hogy kiemelkedően magas integritású adattárolást tesznek lehetővé, egy központi megbízható fél szükségessége nélkül. Egyre gyarapodó bizonyítékok azonban azt mutatják, hogy koránt sem triviális mérnöki feladat további extrafunkcionális követelmények garantálása, mint az elérhetőség vagy a megbízhatóság. Ez különösen igaz célzottan egy konzorcium számára tervezett és konfigurált hálózatok esetében és vállalatközi kooperációt támogató rendszereknél. A blokkláncokra telepített okos szerződések pedig különleges kihívást jelentenek: a hibatűrő és támadások ellen védekező konszenzus protokollok nem óvnak ezek potenciális hibái ellen, és a fejlesztési szakaszban alkalmazható hibatűrő és hiba-eltávolító megoldások még nem tekinthetők érettek.

Másrészről a jogosultsághoz kötött konszenzust alkalmazó, korlátozott hozzáférésű, végrehajtás-sorrendezés-validálás alapú blokklánc platformokat, mint például a Hyperledger Fabricot, lehet extra-funkcionális követelmények mentén tervezni és konfigurálni, és számos futásidejű szolgáltatásbiztonsági mechanizmust kínálnak. Az ismert alkalmazások azonban nem használják ki ezeket, nem fektetnek hansúlyt ezen aspektusokra.

Munkámban a Hyperledger Fabric belső hibamódjairól adok újszerű, szisztematikus áttekintést, megvizsgálom annak hibatűrő képességeit és bemutatom azokat az ismert hibatűrési mintákat, melyek alkalmazhatóak benne. A használt védelmek hatékonyságának kiértékelésére a modern kvalitatív hibaterjedés-analízis (EPA) módszerét javaslom, illetve demonstrálom egy prototípus segítségével.

Az n-verziós programozás (NVP) olyan szolgáltatásbiztonsági technika, amely kifejezetten jól alkalmazható Hyperledger Fabricban. Két architekturális mintát javasolok a bevezetésére és megvizsgálom a megnövekedett szoftverdiverzitás várható hatásait a hibatűrésre, elérhetőségre és integritásra.

Contents

1 Introduction **3**
- 1.1 Problem Statement . . . 3
- 1.2 Contributions . . . 3
- 1.3 Related Work . . . 4
- 1.4 Paper Organization . . . 4

2 Background **5**
- 2.1 Distributed Ledger Technology and Dependability . . . 5
  - 2.1.1 DLT and Blockchains . . . 5
  - 2.1.2 Dependability and Fault Tolerance of Blockchain Systems . . . 6
- 2.2 Error Propagation Analysis . . . 8
  - 2.2.1 Answer Set Programming . . . 9
- 2.3 N-Version Programming . . . 13

3 Sensitivity Analysis-Based DLT Design Support **14**
- 3.1 Analysis of *Fabric*'s Components . . . 14
  - 3.1.1 Endorsement . . . 15
  - 3.1.2 Ordering . . . 16
  - 3.1.3 Ledger update . . . 17
  - 3.1.4 Examples of Failure Chains . . . 17
- 3.2 EPA for *Hyperledger Fabric* using ASP . . . 18
  - 3.2.1 *Fabric*'s model in *gringo* . . . 19
  - 3.2.2 Example Applications to Design Support . . . 23

4 Chaincode Fault Tolerance with N-Version Programming **28**
- 4.1 Classic Approach . . . 28
  - 4.1.1 Master Chaincode as Controller . . . 28
  - 4.1.2 Containerized Approach . . . 29
- 4.2 Consensus-Based Approach ('O-Version Programming') . . . 29

5 DLT Consensus as N-Version Voting **32**
- 5.1 Obtaining a formula for the necessary endorsement policy based on diversity, software faults, and malicious organizations . . . 32

6 Conclusion and Future Work **34**

Acronyms **35**

Appendices **39**

A Complete source code listings **40**

List of Figures

2.1 Hyperledger *Fabric*'s logo 6
2.2 The relationship between faults, errors, and failures (Rehman et al., 2019, fig. 2) 7
2.3 Error Propagation (Avizienis et al., 2004) 9
2.4 The process of solving programs with *clingo* (Gebser et al., 2011) 10

3.1 Generic model of a component 19
3.2 Static *Hyperledger Fabric* metamodel 21
3.3 Transaction flow in *Hyperledger Fabric* 22
3.4 Fault taxomy (Gallina and Punnekkat, 2011) 23
3.5 Instance model of the railroad crossing example 24

4.1 Component diagram of the Master Chaincode-based approach for NVP 29
4.2 Component diagram of the containerized approach for NVP 30
4.3 Possible class diagram of the containerized approach for NVP 31
4.4 Overview of consensus-based N-Version Programming (NVP) in *Fabric* 31

5.1 Two chaincode versions distributed among six organizations 32
5.2 3D surface plot of the bestk function 33

List of Tables

2.1 Diagnostic problems addressed by Error Propagation Analysis (EPA) (Kocsis, 2019, tab. 3.1) 8

3.1 Sensitivity analysis of the endorsement service 16
3.2 Sensitivity analysis of the ordering service 17
3.3 Sensitivity analysis of the ledger update phase 17

5.1 Table to help finding a formula for the required endorsement policy based on Figure 5.1 . . 33

1 Introduction

Distributed Ledger Technology (DLT) systems, especially blockchains, are still actively developing and thriving for their high integrity and decentralization-focused characteristics, both in public and private settings. One of the *Hyperledger Foundation's* many successful projects is the popular open-source permissioned, *execute-order-verify* blockchain platform, *Hyperledger Fabric*.

Fabric is not too complex, but it does have several interconnected components. As the scale of a network grows, it is harder and harder to keep its configuration in sync with the requirements, especially extra-functional requirements such as availability and dependability. In a production environment, a misconfigured network might offer a much lower tolerance of faults and attacks than the platform would be capable of.

1.1 Problem Statement

The infamous DAO hack of 2016, in which $60 million worth of ether (the currency of *Ethereum*) was stolen, has shown clearly that smart contact faults on blockchains may have devastating effects. On *Ethereum,* such bugs might result in users suffering financial losses, such as having their tokens stolen. However, smart contracts exist in other applications of blockchains, like closed, permissioned systems, that may even be deployed in a safety-critical system. Their failures may cause much more severe issues, possibly endangering human life.

While vulnerabilities in the open world of *Ethereum* have been surveyed, we can only make assumptions regarding closed and permissioned systems such as *Hyperledger Fabric* (Praitheeshan et al., 2019). It is not too bold to believe, however, that *Fabric* chaincode being smart contract software as well, will also contain software faults. State-of-the-Art methods to protect against these faults focus on development time techniques, but there is no reason not to explore the possibility of applying classic fault tolerance patterns as runtime defences. This is not viable in *Ethereum,* as the smart contracts' size and complexity imply increased costs, but such problems are a non-issue in permissioned systems.

To the best of my knowledge, there is currently no way to model a *Fabric* (or other blockchain) network to enable root cause or impact analysis during design time. Furthermore, it seems that despite their availability, there have been no attempts to integrate existing fault tolerance patterns into *Fabric*, for example, to mitigate the faults of one of its most vulnerable components, smart contracts (called chaincode in *Fabric*).

1.2 Contributions

To address the problems outlined in the previous section, I make the following three contributions in this paper.

Contribution #1

I introduce my reusable Error Propagation Analysis (EPA) model for *Hyperledger Fabric* to facilitate fault sensitivity, failure root cause and design space analysis for Distributed Ledger Technology (DLT) embedded into critical systems. The model is also able to integrate system-level impact analysis of vulnerabilities and the cross-effects of faults and attacks.

Contribution #2

I propose the application of classic N-Version Programming (NVP) to address smart contract (chaincode) faults in *Hyperledger Fabric* at runtime, in contrast to the almost exclusively development time state-of-the-art approaches of smart contract fault removal, avoidance and mitigation. I also define a supporting software architecture which is fully transparent to *Hyperledger Fabric*'s architecture and consensus.

Contribution #3

As a *Fabric*-specific application of the N-Version Programming approach, I propose 'O-Version Programming', where all organizations participating in an *Hyperledger Fabric* permissioned blockchain bring their own smart contract implementation, and *Fabric*'s consensus mechanism serves as the n-version voter mechanism. I analyse the impact of OVP-based smart contract fault tolerance on the attack tolerance of *Hyperledger Fabric*-based distributed ledgers.

1.3 Related Work

The EPA-related aspects of this work build on the results of Földvári et al., who likewise apply modern EPA using ASP, but to a cyber-physical system (a water tank), investigating the impact of IT security breaches. At the moment, my EPA model of *Hyperledger Fabric* is less concentrated on attacks rather than generic faults and failures. However, the model can be easily extended to include various attacks and could then be used to explore the interactions of internal faults and attacks and their system-wide effects.

Several papers, such as Podgorelec et al. (2019); Hao et al. (2018); Pongnumkul et al. (2017); Melo et al. (2022) focus on evaluating *Hyperledger Fabric*'s performance metrics. To my best knowledge, no work focuses specifically on integrating classic runtime fault tolerance patterns into *Fabric*, especially N-Version Programming. On the other hand, some works investigate *Fabric* chaincode faults and development time removal techniques (Yamashita et al., 2019; Beckert et al., 2018).

1.4 Paper Organization

The rest of this paper is organized as follows. The next chapter, Background, offers additional context and domain knowledge regarding the concepts of DLT, EPA, and NVP. Chapter 3 constitutes my first contribution, describing my *Fabric* model and EPA prototype written in Answer Set Programming (ASP). My second contribution is found in Chapter 4, where I offer two radically different approaches to integrating the practice of NVP into *Hyperledger Fabric*. Related to the second approach, in Chapter 5, I explore the effects of O-Version Programming (OVP) on attack tolerance. Finally, I make my conclusions and explain further plans in Chapter 6.

2 Background

In the following sections, I offer an overview of the key concepts required to understand the contributions of my work, including Distributed Ledger Technology, dependability, fault tolerance, and Error Propagation Analysis. I summarize N-Version Programming and its relevance to DLT.

2.1 Distributed Ledger Technology and Dependability

This section aims to introduce the concept of DLT in general, blockchains, which are one type of DLT, and to describe the basics of dependability as an extra-functional requirement of an Information Technology (IT) system. The later chapters of this paper deal with a specific blockchain system called *Hyperledger Fabric* and concepts revolving around the dependability of concrete *Fabric* network setups.

2.1.1 DLT and Blockchains

Offering distributed, high-integrity data storage, partially owing to the proliferation of cryptocurrencies such as *Bitcoin*, Distributed Ledger Technology has become widely known and used technology. While in the case of open, public networks such as *Ethereum*, one can estimate their growth and size using the publically available data, this is not true for closed, permissioned systems such as *Hyperledger Fabric*, which is geared towards consortial, private applications. Access to the database is restricted (hence 'permissioned') and only available to participating organizations. Surveys and literature suggest *Fabric* is used in numerous cases and that academic research regarding *Fabric* and DLT is active and ongoing (Li et al., 2020; Brotsis et al., 2020; Chowdhury et al., 2019; Palma et al., 2021).

DLT has a variety of applications ranging from digital finance and supply chain networks to perhaps even critical systems. The core of DLT is a transaction *ledger* shared and synchronized among several nodes or peers and a *consensus* algorithm that decides what ledger state the peers jointly agree on. Due to its peer-to-peer (P2P) nature, Single Points Of Failure (SPOFs) are eliminated. The best-known form of DLT is the *blockchain*, an append-only sequence of transactions organized into interconnected blocks. Blockchains power cryptocurrencies such as *Bitcoin* and *Ethereum*, as well as permissioned, consortial systems like *Hyperledger Fabric* or *R3 Corda* (Nakamoto, 2009; Wood, 2015; Androulaki et al., 2018). In this paper, I focus on permissioned[1] blockchain platforms, specifically *Hyperledger Fabric*, but with relatively minor alterations my observations and proposals are also applicable to open systems.

Smart Contracts A great advantage of digital assets stored on blockchains is that it is possible to *program* them: *smart contracts* are software that can be installed on blockchain systems and manipulate the assets on them. In the cryptocurrency world, smart contracts can be deployed to the public network by arbitrary individuals and can later be invoked in the same manner as regular transactions ('flow' of tokens from one address to another) – the network does not differentiate between a smart contract address and a 'normal' one. Private, permissioned systems like *Fabric* limit who may install and invoke smart contracts (which *Fabric* calls *chaincode*). For example, a consortium of organizations might use a smart contract to track the last known physical location of some phsyical entity, which has methods such as `regEnt(id, loc)`, `qEntLoc(id)`, `transferEnt(id, loc)`, to register a new entity in the system, query its last known location, and record its

[1]The word *permissioned* in this context refers to the fact that the ledger is not available for the public to access but requires authentication and authorization. Beyond this, *Fabric* (as well as other platforms) offer more granular permission control. Such features are foreign to public systems like *Bitcoin*.

transfer to a new location respectively. Depending on the platform, this may be *JavaScript* code using the platform's Application Programming Interface (API) to write and read the ledger state. For example, *Fabric* supports chaincode in various languages, such as *Go, node.js,* and *Java*. Other platforms, like *Ethereum*, may require different languages, like *Solidity*.

Unfortunately, software is prone to *bugs,* more formally referred to as *software faults,* which is no regarding smart contracts. An erroneous implementation of the entity-tracking example might result in an entity possibly being recorded twice in the database for two locations. In a financial application, software faults might give way to fraud. In a safety-critical system, such faults threaten human life and potentially damage property. Smart contract quality has a significant effect on the overall dependability of a DLT system.

Hyperledger Fabric

Hyperledger Fabric is a fully open-source permissioned DLT platform written in the *Go* programming language, featuring a modular system with pluggable consensus mechanisms and the capability of executing smart contracts (called *chaincode*) in multiple languages (Androulaki et al., 2018). At the time of writing, *Fabric* was one of the widely used platforms for private blockchain networks, its main competitor being R3's Corda (Brown et al., 2016) and *Ethereum* (Wood, 2015). Compared to other platforms, *Fabric* has a rather complex architecture; setting up a production-grade network (but even a development one, if done manually) can be challenging.

Figure 2.1: Hyperledger *Fabric*'s logo

Unlike public platforms like Ethereum, *Fabric* has no built-in cryptocurrency or *coin*. Instead, it handles *assets,* which may be token-like entities resembling coins, if desired. External software that can be deployed to the network in order to manipulate assets programmatically is called *chaincode*. Usually, Public Key Infrastructure (PKI) is used to designate identities: enrolled participants receive so-called Membership Service Provider (MSP) information, including private signing keys (usually there are separate MSPs for organizational and communication (ie Transport Layer Security (TLS)) services). *Fabric* has the concept of cooperating *organizations* and isolated *channels,* which all have their separate ledgers, policies, and configurations. At the end of the day, *Fabric* itself boils down to a key-value store governed by a consensus mechanism.

2.1.2 Dependability and Fault Tolerance of Blockchain Systems

In the field of IT fault tolerance, the meanings of *dependability* and related terms have their known definitions, which will be introduced below. Informally speaking, the concept of what a dependable system means is straightforward: one can *depend* on such a system, knowing that it will provide adequate service under the right conditions and knowing that in case there are problems, the system will be able to handle them, at least to some extent. For example, one can depend on a train to transport them from point A to point B safely. In the unfortunate event that the train's brakes develop a fault, there are secondary emergency brakes that can still be used to stop the vehicle and hopefully avoid an accident. This is a basic form of redundancy, a dependability method, to make a train more fault tolerant.

Definition 2.1: Dependability

The ability of a system to deliver services on which the user can rely in a justifiable way (Avizienis et al., 2004).

Three threats, *faults, errors,* and *failures,* are typically distinguished. The following definitions are from the 2004 article *Basic Concepts and Taxonomy of Dependable and Secure Computing* by Avizienis et al.. As Figure 2.2 shows, these three can be understood in a causal, sequential sense.

Definition 2.2: Failure

An event when the delivered service deviates from the correct service.

Definition 2.3: Error

A deviation of at least one external state of the system from the correct state.

Definition 2.4: Fault

The adjudged or hypothetical cause of an error.

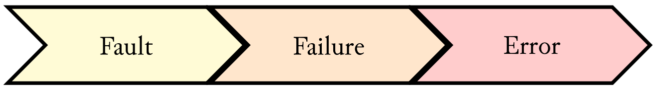


Figure 2.2: The relationship between faults, errors, and failures (Rehman et al., 2019, fig. 2)

The attributes of dependability

Availability	The readiness of the system for correct service.
Reliability	The continuity of correct service.
Integrity	The absence of manipulation of the system.
Maintainability	The possibility of modification or extension of the system and performing repairs.
Security	The ability of the system to preserve the confidentiality of data.
Safety	The absence of potential threats to financial assets or human life by the system.

These six attributes comprise the complex notion of dependability. Finally, dependability also includes specific methods or means intended to make a system more dependable, such as *fault prevention, tolerance,* or *fault removal* (Avizienis et al., 2004).

Application to DLT and Blockchain Systems My work overall is aimed at designing DLT and blockchain systems to be as dependable as the extra-functional requirements prescribe – something which, to my knowledge, is not a well-researched, trivial problem and the methods proposed in this paper have never been used for this purpose. Integrity protection, being the primary promise of blockchains, is not sufficient, as it does not offer defence against attacks and does not ensure the system will also have high enough availability. Furthermore, for some platforms, including *Hyperledger Fabric*, even integrity is not as certain as one might assume: networks may be on a much smaller scale than public networks (such as those hosting cryptocurrencies), and faults of network connections and host problems become much more impactful, possibly harming integrity. In the context of DLT, dependability analysis includes answering questions such as *how many independent host failures can the system tolerate providing service still?* or *what is the minimal-rank failure mode vector that causes a given system failure?*. *Fabric*'s architecture includes several system components, including chaincode (how *Fabric* calls smart contracts), whose faults' potential combinations' effects on the system may be hard to tell by conventional methods. Also, the network configuration, especially *endorsement policies,* plays a defining role in what the system tolerates.

2.2 Error Propagation Analysis

Error Propagation Analysis (referred to as Fault Propagation Analysis or Failure Propagation Analysis in some contexts) is a more generic, automated method compared to classical approaches, such as Fault Tree Analysis (FTA) or Failure Mode and Effects Analysis (FMEA). The main purpose of EPA is to determine the relationship between errors activated by internal failure modes or external faults of system components and their propagation within the system, possible valid and invalid inputs, and the overall system-level effects and failures. Binding two of the three variables (input failure mode, internal failure modes, and output failures) and inferring the third can be used to answer different questions, depending on which variable is unbound. For example, one can consider a component to be internally faulty in some way and then simulate the system's behaviour with some known input to find out what possible effects this combination of input and component fault has on the system's output overall. Conversely, the known output of the system can also be bound. Then one can ask either what input causes the given behaviour assuming specific fault activations or what fault activations would cause the given behaviour when the system gets a known input. Table 2.1 provides an overview of these combinations and what diagnostic problems they tackle (Kocsis, 2019, sec. 3.1). From a system engineering perspective, EPA is intended to take place relatively early in the dependability analysis of the system, constituting the step of *Translation into an Operational Computerized Model* in the modelling process described by Trivedi and Bobbio (2017).

Input	Fault Activation	Output	Diagnostic Problem
✓ known	✓ known, single	?	**fault simulation**
✓ known	✓ known, multiple	?	**parallel fault simulation**
✓ known	?	✓ known, fully	**fault diagnosis**
✓ known	?	✓ known, partially	**partial diagnosis**
✓ known	?	✓ known, fault free	**undetected faults**
?	✓ known	✓ known	**test generation**

Table 2.1: Diagnostic problems addressed by EPA (Kocsis, 2019, tab. 3.1)

Its namesake and a crucial concept in EPA is *error propagation*: the ability of a component to *pass on* errors to other components in the system (on their inputs) or to the output of the entire system. A component may be fault free, but this does not exclude the possibility of producing invalid output as a result of receiving invalid input. For example, a Transmission Control Protocol (TCP) server designed to write all received data to a file will – even under perfectly correct operation – write even unexpected data to the file. Even if the server is supposed to receive Latin letters, without input range checking, it will have no problem forwarding unprintable characters or numbers to the file. This file, in turn, can later be processed by other components, and the unexpected characters can cause a plethora of problems. In this case, the server *propagates* input errors to its output. Figure 2.3 from Avizienis et al. (2004) offers an excellent visualization of this process.

Similarly, an internal fault of a component can be propagated. Considering once again a simple TCP server component, a software fault that causes the server to cease writing to the file when activated will propagate to other components as they will not receive the input they would require. These observations and propagation models are at the heart of EPA.

Modern EPA is also capable of defining reusable error propagation rules. Different components may have similar or the same behaviour regarding error propagation; therefore, their models can be *instantiated* from a common ancestor, possibly with some specialized parameters. The components themselves are also reusable, of course (Kocsis, 2019, sec. 3.1).

As for EPA-related tools, *MARTE* can be used to describe error propagation characteristics in Unified Modelling Language (UML). SysML 'views' can also be used to express such properties. There are funda-

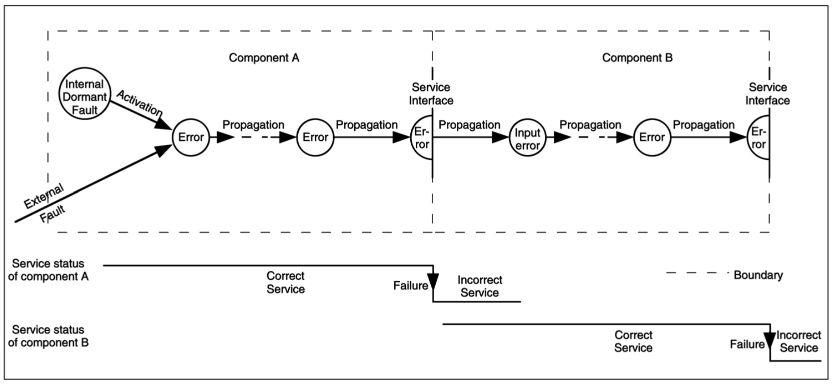


Figure 2.3: Error Propagation (Avizienis et al., 2004)

mentally different analytical approaches to EPA, including model checking of connected automata (Kocsis, 2019; Bernardi et al., 2008; Pataricza, 2002).

EPA problems can be encoded in different ways, but the description usually involves constraints and constraint solving. In his PhD dissertation, Kocsis relies on Constraint Set Programming (CSP) to model error propagation. In a later work, Földvári et al. use ASP with additional temporal logic extensions as their EPA core engine – this latter is also what I use in my work (without its temporal extensions), as it seems to be much more flexible and directly applicable for the purpose.

2.2.1 Answer Set Programming

Answer Set Programming is a form of declarative logic programming, primarily for solving NP-hard search problems, 'based on the stable model (answer set) semantics of logic programming.' The origins of ASP may be traced back to 1997, but the term itself was first used by Marek and Truszczynski in 1998. It can also be thought of as a programming paradigm. LPARSE, a frontend to the answer set solver SMODELS[2], can be considered a traditional tool for ASP, which takes a *Prolog*-like expression of answer set problems as its input (Lifschitz, 2008). In my work, I use *clingo* from the *Potsdam Answer Set Solving Collection* (Gebser et al., 2011), a wrapper for two other programs, `gringo` and `clasp` – the grounder and solver systems, respectively. Under the hood, `gringo` uses LPARSE.

The input language of *clingo* and `gringo` (simply referred to as *gringo* from now on) also resembles *Prolog*, with a few extra directives and, of course, different semantics. Programs are customarily divided into two segments: the problem instance and the problem encoding. The latter can also be divided into parts such as the 'generate' and 'test' parts.

Potassco's user guide[3] comes with great examples to get started. Listing 2.1 contains a full *gringo* program to determine n-colourings of graphs. Graphs are are described by *facts* analogous to *Prolog* facts. A graph's six nodes can be expressed by a range shortcut as `node(1..6)`. The semicolon character can also be used to expand one rule to multiple ones. Constants can be defined, which can also be specified on the command line when solving by the `#const` directive. Rules can be expressed as a set-like notation; in the example code, the generator line describes that every node `X` must have precisely one `colour` where this colour is a number between `1` and `n` (a constant). Finally, the last feature seen in the example is the ability to specify *integrity constraints*, which can be thought of as descriptions of what models should not be considered: the last line specifies that there must be no edge incident on two nodes of the same colour. Given this input, `clingo`

[2]http://www.tcs.hut.fi/Software/smodels/
[3]`https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf`

outputs possible answer *sets* (of atoms), including potential values for the `colour` predicate, such as `color(1,2)` `color(2,1)` `color(3,1)` `color(4,3)` `color(5,2)` `color(6,3)` for the example graph instance. *Clingo* can also deal with optimization problems; another example included in the user guide is an encoding of the *Travelling Salesman* problem in *gringo*.

Listing 2.1: Graph colouring example from the Potassco user guide

```
%% Problem Instance
% Nodes
node(1..6).
% Edges
edge(1, (2;3;4)).   edge(2, (4;5;6)).   edge(3, (1;4;5)).
edge(4, (1;2)).     edge(5, (3;4;6)).   edge(6, (2;3;5)).

%% Problem Encoding
% Default
#const n = 3.
% Generate
{ colour(X, 1..n) } = 1 :- node(X).
% Test
:- edge(X, Y), colour(X, C), colour(Y, C).
```

The under-the-hood operation of *clingo* is out of scope for this paper, but Figure 2.4 shows the basic process of solving *gringo* programs. The logic program (*gringo* code) is first run through the grounder `gringo`, which transforms it to *stable models*, and subsequently fed into `clasp`, which finally generates the answer sets for the program.

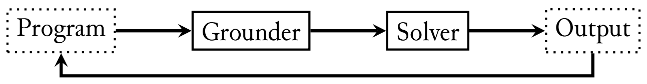


Figure 2.4: The process of solving programs with *clingo* (Gebser et al., 2011)

Temporal Logic Extensions *telingo* is an extension of the *clingo* ASP system that adds elements to describe finite linear time temporal logic to the *gringo* language. Used in the preceding work by Földvári et al., this extension is quite helpful to ease describing the dynamic behaviour of modelled systems. However, in my ASP model, I have decided not to use this extension, but 'plain' *gringo*, as my current model has no temporal dimension.

Introductory ASP Example

To illustrate answering EPA-related questions with ASP, I show its application to a tiny problem: consider a tank filled with a substance that violently decomposes (explodes) at high temperatures. The system is equipped with a temperature sensor and two actuators: one heating the contents of the tank (this might be the sun shining on the container) and another cooling them. When a threshold temperature is reached, the tank explodes.

The question is, how do the actuators' behaviours (their heating and cooling rate) and the initial temperature of the tank affect the substance's fate? Ideally, the temperature is low, and the heating and cooling actuators cancel out one another – tank temperature is constant, and the system is safe. However, if the heating actuator is *stronger* than the cooling one, eventually, the temperature will reach critical. Of course, the actuators' behaviours may change over time.

There are several ways one might approach this problem. With such physical systems, it is relatively straightforward to *simulate* them in a logic program. To this end, one must decide the level of abstraction of components, time, behaviour, etc.

Quantitative Simulation Perhaps the most basic method is to work with numbers – values, that could be *measured* in a real system. Time can be quantified into discrete timesteps, and temperature can be represented by an integer value. The heating and cooling components have a constant rate at which they increase or decrease the temperature at each timestep. Then, the tank's temperature at each timestep can be simulated, and it can be determined whether the actuators succeed in keeping the temperature within the acceptable range in the simulation time.

This method can be encoded in ASP code, as seen in Listing 2.2. In this implementation, component faults must be activated manually by changing the heating or cooling rates accordingly; for example, a failure of the cooler may be activated by binding `cool(0)`. The integrity constraint `bang(_)` ensures that only models where no explosion occurs are considered. Therefore an unsatisfiable model means the given set of bindings is unsafe.

As is, the program could not be satisfied because of the initial temperature of `90` and the heating rate being higher than the cooling rate. In 20 timesteps, the maximal temperature is reached. If the initial temperature were lower, simulation time would run out before the fatal system state could be reached.

Listing 2.2: Using ASP for Quantitative Simulation

```
init_temp(90).  % Initial temperature
steps(20).      % Lenght of simulation
heat(2).        % Heating factor
cool(1).        % Cooling factor
max(100).       % Maximum tolerated temperature

% there is a single temperature value at each timestep
time(1..N) :- steps(N).
{ temp(Temp, Time) } = 1 :-
    temp(TempPrev, Time - 1),
    heat(H), cool(C), Temp == TempPrev + H - C,
    time(Time).

% at timestep 0, the temperature is its initial value
temp(Temp, 0) :- init_temp(Temp).
% the temperature is tolerable if it is below the maximum
ok(Time) :- temp(Temp, Time), max(M), Temp ≤ M, time(Time).
bang(Time) :- not ok(Time), steps(N), time(Time).

% the fatal case must not occur
:- bang(_).

#show temp/2.
```

Qualitative Simulation In the context of IT systems and especially in my work, a higher level of abstraction can be helpful. Neither time nor the simulated variables (the temperature, in this case) have to have numerical values. Instead, one may describe the range of temperatures by categorical values, such as *nominal, too low,* and *too high*. There are also ways to avoid having to deal with explicit timesteps, modelling only cause-effect relationships and not how exactly they happen over time.

For the tank example, the categorical temperature values imply a higher-level definition of how these values change. This can be done by describing the direction of change (ie sign of the second derivative), with values such as *stagnant, decreasing,* or *increasing*.

A possible resulting program can be seen in Listing 2.3. In this implementation, timesteps are still used, but temperature and its change are encoded in categorical terms. Two possible models that fit (and can be obtained by *clingo*) are shown in Listing 2.4. In the first one, the temperature becomes high, but stops increasing at the second timestep, and then starts to decrease. In the second one, it also becomes high at first but then stops changing. In either case, there is no fatal outcome as the failure case is defined to occur when the temperature is `high` and still `increasing`.

Listing 2.3: Using ASP for Qualitative Simulation

```
init(ok, inc).          % Initial temperature and change
steps(3).               % Length of simulation
dir(inc;none;dec).      % Possible change directions
temp(low;ok;high).      % Possible temperature values

% there is a single temperature value at each timestep
time(1..N) :- steps(N).
{
    temp(Temp, Dir, Time) : temp(TempPrev, DirPrev, Time - 1),
                            change(TempPrev, Temp, DirPrev),
                            dir(Dir), time(Time)
} = N :- steps(N).

% describe the possible state changes
change(T, T, none) :- temp(T).
change(low, low, dec).  change(high, ok, dec).  change(ok, low, dec).
change(low, ok, inc).   change(ok, high, inc).  change(high, high, inc).
% discard models with duplicated steps
:- temp(TempA, _, Time), temp(TempB, _, Time), TempA != TempB.
:- temp(_, DirA, Time), temp(_, DirB, Time), DirA != DirB.
% at timestep 0, the temperature and the derivative are their initial
% values
temp(Temp, Dir, 0) :- init(Temp, Dir).

% the fatal case must not occur
pass :- not temp(high, inc, _).
fail :- not pass.
:- fail.

#show temp/3.
#show pass/0.   #show fail/0.
```

Listing 2.4: Two possible models of the ASP program on Listing 2.3

```
Answer: 1
temp(ok,inc,0) pass temp(high,none,1) temp(high,dec,3) temp(high,none,2)

Answer: 2
temp(ok,inc,0) pass temp(high,none,1) temp(high,none,3) temp(high,none,2)
```

Parallel Simulation As shown by Kocsis (2019, sec. 3.1) as well as Pataricza (2006, sec. 3.1), it can be helpful in EPA to allow system components to have faulty 'mutations', defined as aberrations from their 'reference' state. This method can be applied to the tank problem by defining separate rulesets for the reference and the faulty model and introducing a new rule that encodes the difference between the faulty and the reference model at each timestep. Then, the resulting answer sets are simulations of the model, and the diverging behaviour of the faulty version can be observed.

2.3 N-Version Programming

N-Version Programming or *multi-version programming* is a well-known software engineering technique that is capable of increasing *software diversity* by independently implementing the same specification multiple times, favourably by different teams and in different programming languages. The idea is that the same possible faults will likely not be introduced to all versions; therefore, executing all versions and comparing the results statistically decreases the probability of failure (Avizienis, 1986, 1995).

Various levels of NVP are imaginable. The simplest form involves executing the n implementations (possibly concurrently), collecting the results, and performing majority voting on them, but other voting algorithms may also be desirable (Gersting et al., 1991). For example, given three implementations of the specification, an essential voting component decides if at least two out of the three results match. In that case, this outcome is considered the overall result. For n versions, $\lceil \frac{n}{2} \rceil$ results must match or be otherwise acceptable. Naturally, the voting component is a trivial failure point of the system, but voting logic is not difficult.

In more sophisticated situations, special NVP mechanisms are injected into the source codes of the individual versions that allow a designated execution environment (called N-Version eXecution environment (NVX)) to take control, synchronize them, or enable runtime verification of their state.

Remarks about independent faults NVP is built on the concept that the potential introductions of software faults in the multitude of versions are independent of one another. However, in 1987, Knight and Leveson conducted a research where 27 versions of a specification were implemented independently by two universities and then subjected to a million tests. Analysis of their results shows a correlation between the faults, implying that the assumption of fault independence may be misguided or at least that care must be taken to consider dependent faults. Nevertheless, NVP *is* good valid fault tolerance and redundancy technique if done correctly.

NVP in DLT systems As a common fault tolerance pattern, NVP is easily among the first to consider when attempting to increase the fault tolerance of *smart contacts* in DLT systems. However, some of its methods cannot be applied or at least are not trivial to apply, such as the runtime government of the NVX. Smart contracts are usually invoked by transactions from clients, and platforms do not offer any way to interrupt their execution for the sake of synchronization or verification. Of course, it would be possible to extend the platforms themselves to include such capabilities, but this would likely require large-scale, fundamental changes to the software. That said, smart contracts are software like any other operating on some input and returning some output. Executing multiple versions of them and comparing the results is trivial if cross-smart contract invocations are allowed. I propose two radically different approaches to introduce NVP to a concrete DLT platform, *Hyperledger Fabric*, in Chapter 4.

The only other application of NVP in the context of DLT systems today is *Hydra,* which uses a variant of NVP called N-of-N-Version Programming (NNVP), and focuses on error *detection* and safe termination rather than fault *tolerance* (the goal of classic NVP) featuring a bounty system rewarding finders of critical software faults. Technically, *Hydra* is designed for *Ethereum* (Breidenbach et al., 2018). In contrast, my proposals *do* target the classic NVP objective of fault tolerance and focus on consortial DLT platforms, such as *Fabric*.

The reappearance of NVP in AI Due to the concerns mentioned a few paragraphs ago, the popularity of NVP has somewhat dropped in recent years. This clearly shows in the volume of related publications and the fact that new literature speaks of NVP as something old, with phrases such as *New Wine in an Old Bottle,* or *revisiting* this technology. On the other hand, it seems that NVP might have a renaissance in the world of Artificial Intelligence (AI), as new publications suggest using it to improve reliability and resilience in Machine Learning (ML) models. The idea is to overcome the difficulty of reliable ML models by generating n versions of an ML component and then executing these diverse replicas, which costs only more computations, but optimally results in significantly higher reliability (Gujarati et al., 2020; Xu et al., 2019; Machida, 2019; Wu et al., 2018).

3 Sensitivity Analysis-Based DLT Design Support

DLTs, especially blockchains offer features such as append-only, high-integrity data storage, which may prove useful even in critical systems. At the very least, even non-critical usage often requires high dependability. In such situations, it is crucial that the system is *designed* to be dependable with regard to its architecture and configuration.

However, as far as my research of current literature shows, not much has been done to allow designing DLT systems with dependability in mind, while aspects such as

- the number of nodes maintaining the shared ledger,
- their distribution among cooperating organizations,
- the network connections between system components,
- input transaction data validity,
- various software faults,
- and endorsement policy configurations

play a defining role in the matter. For example, an erroneously configured endorsement policy might allow a single organization of many to maliciously inject an illegal transaction into the ledger, essentially defeating the decentralized trust. Smart contracts especially affect dependability, as software faults within them are likely to propagate into the ledger as faulty transactions. In critical systems, faulty smart contracts may pose an unacceptable risk.

In this chapter, I show an analysis of a widespread DLT platform, *Hyperledger Fabric*, examining its components used for transaction processing, followed by a proposal to use Answer Set Programming as an aid for dependable system design.

3.1 Analysis of *Fabric*'s Components

Fabric's operation can be divided into three stages or services:

1. Endorsement (including chaincode execution)
2. Ordering
3. Ledger update (including validation and MultiVersion Concurrency Control (MVCC) handling)

In reality, these stages rely on a number of underlying components. During endorsement, the client sends transaction proposals to some peers, who subsequently return their endorsements of the proposed transaction. During this phase, the peers execute the transaction chaincode invocations[1]. The result of the chaincode invocation is a read/write set over the key-value store that is the current ledger (world) state. Exactly which peers are chosen by the client and how it behaves when some of them are unreachable depends on client-side logic and is not analyzed in the scope of this paper. The network (more accurately, the channel, but for now let us consider single-channel networks only) has an *endorsement policy* configuration, which is basically k-out-of$n$ voting logic over the read/write sets reported by the peers.

Then, the client may submit the proposal to an orderer, which, receiving several other transactions, decides on their order and broadcasts the generated blocks to peers. When the peers are notified of a new block, they

[1]For simplicity, I only consider invocations. In reality, query transactions need not invoke chaincode.

once again *validate* it by executing the transactions within. MultiVersion Concurrency Control (MVCC) is used to ensure that the read/write sets of the transactions are not in conflict. If a transaction *does* cause a conflict or is otherwise invalid, it will not have an effect on the world state of the ledger (but will still be appended as part of the block). It is up to the client to retry the transaction in this case. The fact that the client executes transactions twice is what makes *Fabric* an *execute-order-verify* blockchain platform.

Each of these services has a number of potential internal faults, which result either in degraded system-level performance or system failure. For example, if the ordering service is completely unavailable (because none of the orderer nodes is reachable), there is no way for any transaction to get into a block in a legal way. If the used orderer node *is* available but is under high load, it may take longer to process the transaction, resulting in it being written to the ledger later than usual. In this sense, the ordering service is able to propagate its failure to the next component or service.

In the rest of this section, I analyze the aforementioned component sequence responsible for transaction processing: for each service, I consider a combination of an *external fault,* an *internal fault,*, and the resulting *failure mode*. Failure modes are classified according to the taxonomy introduced in the paragraph titled Failure model in Section 3.2. '$*$' symbolizes a wildcard: depending on the context, any failure mode, external fault, or internal fault can substitute it. Lowercase italic characters are variables.

At this point, I do not consider software faults in *Fabric*'s code: I assume peers and orderers work according to their specifications. However, I do consider the host machines where peer and orderer software is installed; see the next section for more details.

3.1.1 Endorsement

Endorsement is a crucial step since this is when peers first execute chaincode that might contain software faults (bugs). Table 3.1 shows the considered fault propagation characteristics of endorsement and peer nodes. A short summary of the possible internal fault modes and their potential root causes:

Endorsement Internal Fault Modes

not enough organizations are available The endorsement phase entails a number of organizations agreeing to a certain transaction. This number is defined by the configured endorsement policy for the channel. If either the peer hosts of certain organizations or the network links between the client and these hosts are down, the client will not be able to get the necessary endorsements in time.

too many organizations compromised Endorsement policies only protect against some number of malicious organizations at the cost of lower availability. If more organizations are malicious (either due to an intentional fraud by this group of organizations or due to an external attack), basically anything can happen: transactions might fail, invalid transactions might be appended to the ledger, etc.

policy misconfiguration The definition of the endorsement policy is essential in the operation of the network. Poor choices (most probably due to human error) may lead to lower availability, transactions committed with delays, or not at all. An overly permissive policy may allow invalid transactions to be committed to the ledger.

chaincode subtle/coarse data error This failure mode represents a canonical case of a software fault in the chaincode. It could be caused by an erroneously specified loop range or a reversed relational operator, for instance. Subtle errors are considered to be undetectable, as opposed to coarse ones (such as a temperature sensor reporting a value lower than $-273.15\,°\mathrm{C}$ (absolute zero) – a physical impossibility).

An interesting aspect of software faults is that they may or may not be activated at runtime. Furthermore, if the input data is already faulty, there is no way to tell if the outcome will be *subtlely* or *coarsely* faulty.

chaincode slow queries It is a known problem that *CouchDB* range scan queries are currently inefficient because they unfold into several key queries for each key in the range[2] Chaincode relying on such constructs may take a considerable time to execute, resulting in a delayed transaction.

External Fault	Internal Fault	Failure Mode
(p, t, v)	not enough organizations available	$(p, \texttt{LAT}, v)$
$*$	too many organizations compromised	$*$
(p, t, v)	policy misconfiguration	$(p, \texttt{LAT}, v)$/ $(\texttt{OMI}, t, v)$/ $(\texttt{COM}, \texttt{OK}/\texttt{LAT}, v)$
$(p, t, \texttt{OK})$	chaincode subtle data error	$(p, t, \texttt{SUB})$
$(p, t, \texttt{OK})$	chaincode coarse data error	$(p, t, \texttt{COA})$
$(p, t, \texttt{SUB}/\texttt{COA})$	chaincode subtle/coarse data error	$(p, t, \texttt{SUB}/\texttt{COA})$
(p, t, v)	chaincode slow queries	$(p, \texttt{LAT}, v)$

Table 3.1: Sensitivity analysis of the endorsement service

There are some additional failure modes and external faults that I would like to consider in the future but did not model so far. For example, a transaction with an invalid signature causes an *omission* provision failure mode. Potential private data leaks in chaincode imply a completely new dimension of failure modes regarding confidentiality, which I could not categorize into the current taxonomy. If anything, it would belong to the *value* failure category, besides *subtle* and *coarse* failures.

3.1.2 Ordering

Once the client has collected sufficient endorsements from participating organizations, it may submit its endorsed transaction proposal to an orderer node. The orderer is a member of the network's *ordering service*, which may have internal failure modes, some of which are examined below. The propagation characteristics table can be seen in Table 3.2. A functional ordering service eventually publishes a block containing the next batch of transactions for the peers to append to their local ledgers.

Ordering Internal Fault Modes

too many orderers unreachable Depending on the choice of the ordering service (*Raft*, *Kafka*, or *solo*), once a number of orderers are not reachable (due to host or network failures), ordering cannot take place. The worst case is, of course, the *solo* mode, when the chosen orderer becomes a SPOF. In the other modes, the majority of the ordering nodes must be available.

too many orderers compromised Like peers, orderers may be compromised, which renders the ordering service unpredictable.

ordering takes long There are a few reasons that might delay a transaction getting into a block. An 'innocent' possibility is that there is simply such a high load of transactions to process that the ordering service did not get to it yet. Otherwise, it is absolutely possible for an orderer to maliciously delay transactions or favour the transactions of select organizations over others.

As orderers do not perform much validation (they only look at block headers), the ordering service will, under normal operation, propagate all input faults to the next component.

[2]Evidence of this issue can be seen on `https://jira.hyperledger.org/browse/FAB-18507` (accessed on 2022-10-31).

External Fault	Internal Fault	Failure Mode
(p, t, v)	too many orderers unreachable	$(p, \texttt{LAT}, v)$
$*$	too many orderers compromised	$*$
(p, t, v)	ordering takes long	$(p, \texttt{LAT}, v)$

Table 3.2: Sensitivity analysis of the ordering service

3.1.3 Ledger update

As a final phase, once ordering has taken place and the peers have been notified about the new block, they once again validate the contents of the block and append it to their ledgers. In real life, not all peers take part in this process and only receive information about new blocks via *peer gossip*. For simplicity, I assume all peers receive new block publications and ignore gossip.

There is still one pitfall here that could cause the failure of a transaction: if an MVCC conflict is detected by the peer. This is considered the only failure mode in this phase, shown in Table 3.3.

Ledger Update Internal Fault Modes

unexpected MVCC conflict *Hyperledger Fabric* does not utilize locks for concurrency but rather aborts transactions in case of conflicts such as a dirty read or write. It is up to clients to retry such transactions. MVCC conflicts are not so much failures as unexpected, unfortunate events that still might cause transactions to fail completely or at least be delayed.

External Fault	Internal Fault	Failure Mode
(p, t, v)	unexpected MVCC conflict	$(p, \texttt{LAT}, v)$

Table 3.3: Sensitivity analysis of the ledger update phase

3.1.4 Examples of Failure Chains

It is easier to understand how errors (or failures) can propagate within a DLT system by looking at some exemplary *chains* of causes and effects and the final system-level result. This section contains some basic ideas for such chains with descriptions regarding in what scenario they may occur. These scenarios are not necessarily possible to model and analyze using the prototype ASP implementation introduced in Section 3.2 but serve merely as examples showing how certain faults can have system-wide effects.

A cloud computing provider's servers are down and transactions are late

Failure chain

1. Several organizations in a network host their peers on the same cloud computing service provider, which experiences downtime, making all these peers unavailable for some time.

2. Submitted transactions (proposals) take longer to endorse than usual since the client(s) must wait for the unavailable peers to come back online.

3. Assuming otherwise correct operation, the transaction eventually gets committed to a block, but this happens much later than expected.

Scenario For example, this could happen in a supply chain blockchain network. A shipped item could have already arrived at its destination, but due to a large delay in transaction processing, the ledger state may still reflect its status as being in a temporary depot. It is also possible to flood the network in a Denial Of Service (DOS) attack (which may also be involuntary, eg in the case of a faulty client application).

Smart contract contains bug which erroneously changes ledger state

Failure chain

1. A smart contract contains a software fault, due to which the resulting read/write sets might contain invalid values that are still within range (a subtle data fault).
2. A client invokes the smart contract with input data that trigger this fault, leading to an undesired ledger state update.
3. The ledger state is now technically valid, but in reality, it is wrong (a subtle data fault).

Scenario For example, the chaincode may allow users to keep a record of how many *items* (which may be anything in reality) they possess. Let us say that each item has a unique secret code, knowledge of which must be proven in order to record the ownership of the item. Due to a bug in the chaincode, a specially crafted invalid secret code causes the ledger to be updated in such a way that the user is recorded to possess an item, even when it does not really exist. This way, a user may have unlimited items even if they have none.

All orderers or peers are down, and the network is rendered unusable

Failure chain

1. Due to an external problem or an adversarial attack, either every Ordering node or every peer (or both) is taken out of operation.
2. Clients have no way of accessing the ledger contents, submitting transactions, or really doing anything.
3. The service provided by the network is down.

Scenario For example, if the network is responsible for facilitating payments (eg a Central Bank Digital Currency (CBDC) implementation), such downtime can cause a total halt of all dependent financial processes. There are a number of possible causes ranging from simple power outages to software faults which crash the orderer or peer nodes.

3.2 EPA for *Hyperledger Fabric* using ASP

I approach EPA on *Fabric* by concentrating on the outcome of a single *reference transaction* being processed by the system. First, I establish a model of a *Fabric* network, including its physical and logical components, the static structure of the network built from these components and the connections between them. Then, I add the behavioural models of the individual parts. Finally, I show examples of how after defining some binding constraints, ASP is able to infer the rest of the model and how this can be applied to system design.

$$\begin{tikzcd}
\text{External Fault} \arrow[r] & \boxed{\text{Component}} \arrow[r] & \text{Output Failure Mode} \\
 & \text{Internal Fault Mode} \arrow[u] &
\end{tikzcd}$$

Figure 3.1: Generic model of a component

3.2.1 *Fabric*'s model in *gringo*

Each component is modelled as a black box having a certain `IFM` internal fault mode that receives some `EF` external fault and produces some `OFM` output failure mode. This simple component model is visualized in Figure 3.1. The (`IFM`, `EF`, `OFM`) triples describe the dynamic behaviour of the component.

The implementation of the ASP program is split among several source files, structured in the following way:

```
(project root)
├── model/
│   ├── behaviour.lp ........................................ dynamic model
│   ├── failure.lp ............................................ failure model
│   └── structure.lp ......................................... static model
├── constants.lp ........................... bounding values for model generation
├── bindings.lp ................................... fixed (bound) model elements
└── top.lp ......................................... only include statements
```

Note that for the sake of completeness, I have included the contents of all of these files in the appendix of this document.

Static Model

Figure 3.2 offers an overview of the entire structural metamodel encoded in *gringo*, but I briefly go over the details below. The program is written in such a way that whatever model elements are not bound can be generated by the solver instead. For example, to define how many organizations there are in total, the fact `orgs(N)` may be used with the desired number in place of `N`. If no such fact is defined by the user (in the `bindings.lp` file), then the following statement near the top of `structure.lp` will generate it:

```
{ orgs(1..max_orgs) } = 1.
```

Where `max_orgs` is actually a constant, limiting up to how many organizations should be generated if any. Then, to define an organization called `org1`, one would establish `org(org1)`. Defining this single organization does not imply there may not be more; by default, the program will consider `N` organizations where `N` is bound by the fact `orgs(N)`.

Physical hosts belong to organizations; for example, `host(org1, org1h1)` defines `org1h1` to be a host at the `org1` organization. As with the organizations themselves, an `org_hosts(N)` fact defines how many organizations each host has. As system components, hosts have internal fault modes defined separately. For example, to establish that host `org1h1` is down: `host_ifm(org1h1, down)`. Other possible fault modes are `ok` (nominal state) and `compromised`. Hosts are also connected (linked) as `host_link(A, B, State)` where `A` and `B` are host identifiers such as `org1h1` and `State` is either `up` or `down`, depending on whether the connection is broken. To simplify, hosts that have no physical link between them are also modelled as having a link with the `down` state. Additionally, the following simple rule takes care of the fact that links are always bidirectional:

```
host_link(A, B, State) :- host_link(B, A, State).
```

Other components – the singular *client, peers, chaincode executors,* and *orderers* – are allocated to physical hosts with facts such as `orderer_alloc(org1o1, org1h1)`. Some components are only logical and have no host allocation, such as the ordering *service* of an organization. However, all components have internal fault modes.

There are some extra facts and rules, such as the client being 'subscribed' to one of the peers – this is important because, after a successful transaction, the client only finds out about the transaction making it into a ledger by finding it in a block on a peer it queries. The `ordering(Type)` fact chooses one of the built-in *Fabric* ordering services (the now deprecated `solo` and `kafka` or the currently recommended `raft`). It is worth mentioning that even *raft* is only Crash Fault Tolerant (CFT) and a Byzantine Fault Tolerant (BFT) consensus implementation is underway for some time. The model could already be easily extended to understand BFT consensus.

Endorsement policies can also be defined by facts such as

`endorsement_policy(Node, Operator, A, B)`

where `Node` is an identifier of a node in the syntax tree composed of the logical $\wedge$ (`AND`) and $\vee$ (`OR`) operators, `Operator` is one of these operators (encoded as `and` and `or`) and `A` and `B` are the operands to the `Operator`. A special `top` node name marks the root of the tree. For example, to encode the simple policy of `AND(OR(Org1, Org2), Org3)`, one would do

```
endorsement_policy(org1_or_org2, or, org1, org2).
endorsement_policy(top, and, org1_or_org2, org3).
```

As a side note regarding fact generation: it is not trivial to generate facts such as `orderer(orgN, orgNoM)` for several `N` and `M` values because this implies generating atom names. *gringo* understands `fact(1..N)` but not `fact(foo1..N)`. Nevertheless, it is useful to have such identifiers, because simple integers would convey less information and the generated models would be much harder to read. As a workaround, I took advantage of the scripting capabilities of *clingo:* one can insert *Lua* or *Python* code blocks such as the one seen on Listing 3.1 to define functions which can then be used from *gringo* code.

Listing 3.1: Small *Lua* script to generate orderer name atoms

```lua
clingo = require('clingo')
F = clingo.Function

function orgorderer(org_, i_)
    org = org_.name
    i = i_.number
    return F(org .. 'o' .. i)
end
```

Dynamic Model

The dynamic model is found in a separate file, `behaviour.lp`. Here is where the error propagation characteristics of the components are defined. My behavioural model follows a slightly modified version of the transaction flow diagram provided in *Fabric*'s documentation[3] and is included in this paper as Figure 3.3. Purple-coloured actors indicate that the component is not present in the system in any observable form, but I logically considered it as a separate component for modelling purposes. For example, there is normally no such thing as an *endorsement service* in an organization (even though one is imaginable). Clients send their endorsement requests to the peers they choose themselves. My model assumes that an organization follows the same endorsement strategy in a given channel for a given chaincode.

[3]`https://hyperledger-fabric.readthedocs.io/en/release-2.4/txflow.html`, accessed on 2022-10-27

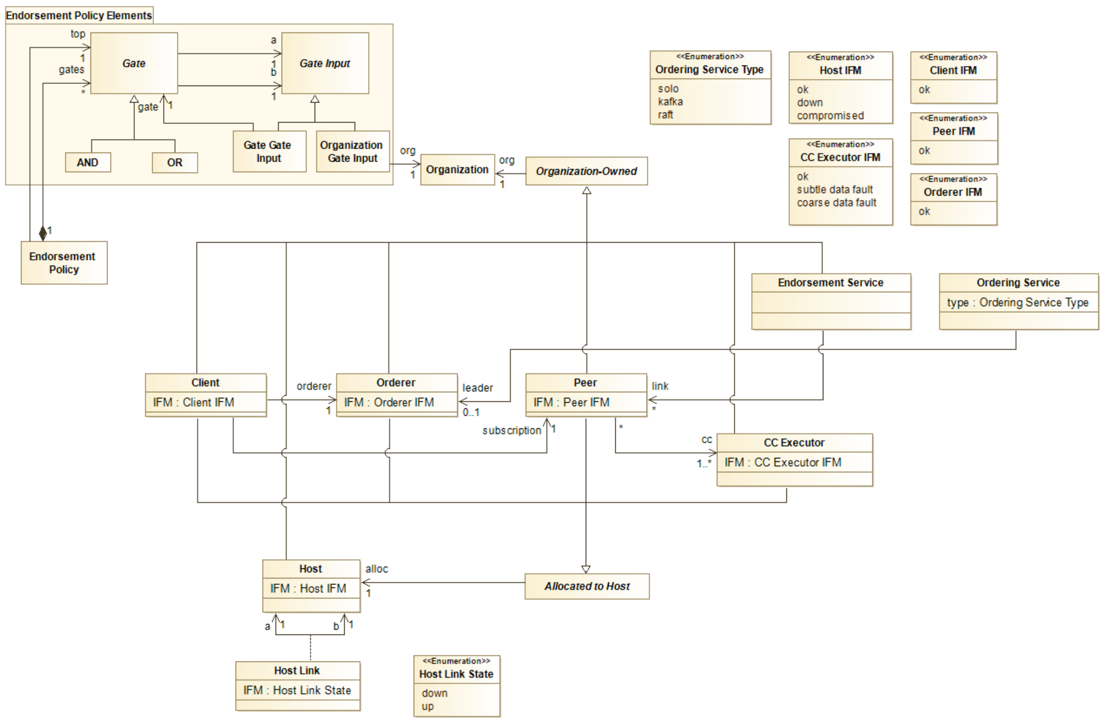


Figure 3.2: Static *Hyperledger Fabric* metamodel

I took care to separate the generic fault propagation behaviour of the components from the fault propagation of the component *instances* in a given model (at least where it made sense). For example, the chaincode executor component propagates faults in the following way:

```
ccexec_fp_g(ok, F, F) :- failure_mode(F).
ccexec_fp_g(subtle_fault, (P, T, V), (P, T, (V; subtle))) :-
    failure_mode((P, T, V)).
ccexec_fp_g(coarse_fault, (P, T, V), (P, T, (V; coarse))) :-
    failure_mode((P, T, V)).
```

Meaning that in an `ok` state, the output failure mode matches the received external fault, and in the `subtle_fault` and `coarse_fault` internal modes, the internal fault mode *may* be propagated to the output. This is to simulate the *activation* of the chaincode software fault. The rules above generate several facts, such as `ccexec_fp_g(ok, (ok, ok, ok), (ok, ok, ok))`, `ccexec_fp_g(ok, (ok, late, ok), (ok, late, ok))`, and so on, but this is not the error propagation of a specific chaincode executor instance in the model. These are merely the way all chaincode executors behave. A separate, instance-specific propagation rule is what defines the behaviour of a concrete instance:

```
{
    ccexec_fp_i(ID, EF, OFM)
    :   ccexec_alloc(ID, Host), host_ifm(Host, ok),
        ccexec_ifm(ID, IFM),
        ccexec_fp_g(IFM, EF, OFM),
        client_fp_i(_, EF)
    ;
    ccexec_fp_i(ID, (P, T, V), (omission, T, V))
    :   ccexec_alloc(ID, Host), host_ifm(Host, down),
```

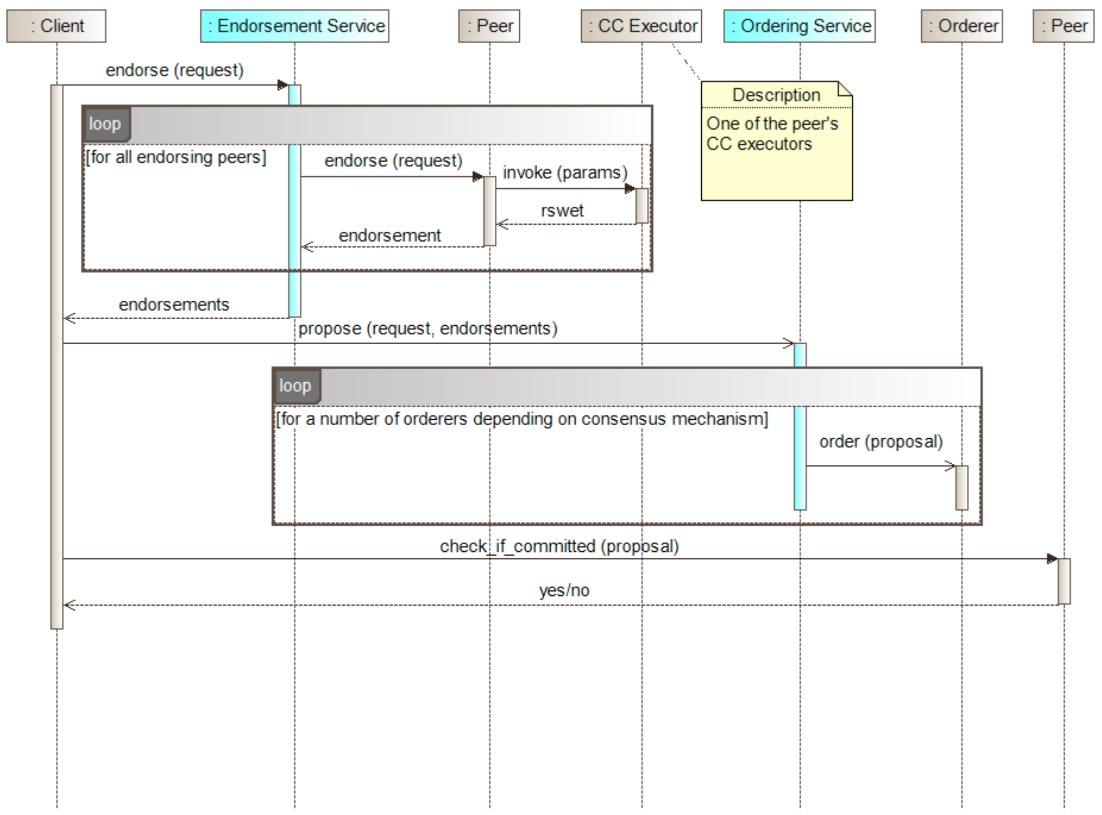


Figure 3.3: Transaction flow in *Hyperledger Fabric*

```
        client_fp_i(_, (P, T, V))
} = 1 :-
    ccexec(_, ID).
```

What this means is that for every single chaincode executor with name `ID`, a single `ccexec_fp_i(ID, EF, OFM)` fact must exist – the failure propagation of that executor. The rule is further divided into two cases: if the host the chaincode executor is allocated to is up, then the component behaves as it is supposed to behave according to the generic rules shown earlier. Otherwise, if the underlying hosts happen to be down, then the generic fault propagation rules do not matter since we can be sure that the component will be unable to respond, resulting in propagating an *omission* provision failure mode to its output.

Failure model As Kocsis and previously Gallina and Punnekkat, I modelled failures in three 'dimensions': *provision, timing,* and *value*. Provision refers to whether an expected action occurs, usually a ledger state change in *Fabric*'s case. We can talk about either *omission* failures or *commission* failures, when something is done which should not have been done, ie a transaction has been erroneously recorded on the ledger. Timing failure modes include being too *early* and too *late*, and value failure modes are divided into *subtle* and *coarse* failures. Subtlety refers to whether the failure is *detectable,* which is quite subjective. Figure 3.4 visualizes this simple taxonomy of failures.

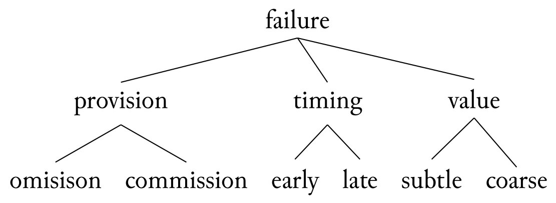


Figure 3.4: Fault taxomy (Gallina and Punnekkat, 2011)

Security considerations

A dimension of possible faults in *Fabric* networks, specifically security vulnerabilities are not included in my model yet, or at most in an oversimplified way: hosts where the network components are deployed to have a *compromised* fault mode, in which case the components allocated to them may behave any possible way.

In the future, I plan to extend the model with known *Fabric* vulnerabilities that can be found in literature (Andola et al., 2019; Yamashita et al., 2019; Dabholkar and Saraswat, 2019). The model is actually already prepared for this; one simply needs to add the vulnerability dictionary to the internal fault modes of the components and the corresponding failure propagation behaviours. In his master's thesis, Hambuch also collected several vulnerabilities in a 'Chaincode Weakness Classification Registry,' but mostly for chaincodes. For instance, `CWC-100 No input validation` would be fairly easy to integrate into my model: it would constitute an additional `no_input_validation` internal fault mode of the chaincode executor component and a new rule describing that executors in this mode always propagate external subtle or coarse data failures. Correctly operating chaincode executors would then be defined to be able to detect such external faults and ensure they are not propagated (by failing the transaction).

3.2.2 Example Applications to Design Support

To illustrate the usefulness of my contributions, I gathered two applications where it can be used to obtain nontrivial results. The first is related to critical systems, where the dependable design of the blockchain used by a smart railroad system is paramount to ensure that accidents do not happen. The second application is aimed at a generic consortial network where an optimal endorsement policy is sought that ensures tolerance against specific faults. The latter also demonstrates how the optimization features of *clingo* can be used together with my *Fabric* model to give answers to questions that are otherwise hard to answer because of the enormous state space.

Railroad crossing

This application setup comes straight from Kocsis (2018). A self-driving car arrives at a railroad crossing. Following the positive control principle, the car is allowed to only cross once it has received a grant to do so for a given time window. Communication between the car and train takes place on a blockchain in order to ensure decentralized, high-integrity storage of the records allowing entities to pass.

In this situation, omission failures are not too problematic; they merely cause the car to wait long at the crossing. A *late* timing failure has the same effect. However, commission and value type failures can have catastrophic outcomes. Even a subtle data fault in the governing chaincode (eg somewhere, a zero is flipped to a one) might cause the car to start passing at the wrong moment and drive right in front of the coming train.

Let us consider the following model. There are two organizations involved: one for the railroad and one for the car. The endorsement policy requires transactions to be endorsed by both organizations. Furthermore, both organizations have two hosts. The railway organization has a peer node and a chaincodechaincode executor installed on one of its hosts and an orderer node on its other host. The other organization has two peers, one on each node, each along with one chaincode executor. All four hosts are linked and can reach one another over the network. Unfortunately, the chaincode software contains a *subtle value fault*; otherwise, components are healthy. The instance model described in this paragraph is much easier to understand by looking at Figure 3.5. The figure also clearly shows the fault propagation described below. Please note that some elements have been simplified or left out from the diagram to ease understanding. For example, the four hosts form a complete graph via *Host Link* relations, but this is not shown in the diagram. Also, only faulty components' internal fault modes are visible; technically, all four hosts, the three peers, and the orderer all have their internal fault modes set to `ok`. Finally, the *car* object is not really modelled, it has been added to the diagram to show how the fault eventually ends up potentially causing an accident. Strictly speaking, system-level failure already occurs at `org2p2`.

The subtle data fault in the chaincode might activate in one of the chaincode executor components. In the fault propagation of the model shown in the figure, the fault activates in the chaincode installed on the railway organization's peer `org1p1`. The orange colouring of model elements shows that they take part in the fault propagation. The peer propagates the fault by returning a faulty read/write set to the client (through `org2`'s *endorsement service*, which is not actually a real *Fabric* component). The client then proceeds to send its transaction proposal with the faulty read/write set to the only orderer in the system, which eventually broadcasts it as an element of a block, which the `org2p2` peer receives (as well as other peers), validates, and writes to its ledger. The end result of the transaction is a subtle data failure, which is considered undetectable. However, it is possible that such a fault is capable of causing an accident in this critical system, for example, if it means a timestamp is not accurate and the record instructs the car to go earlier than safe.

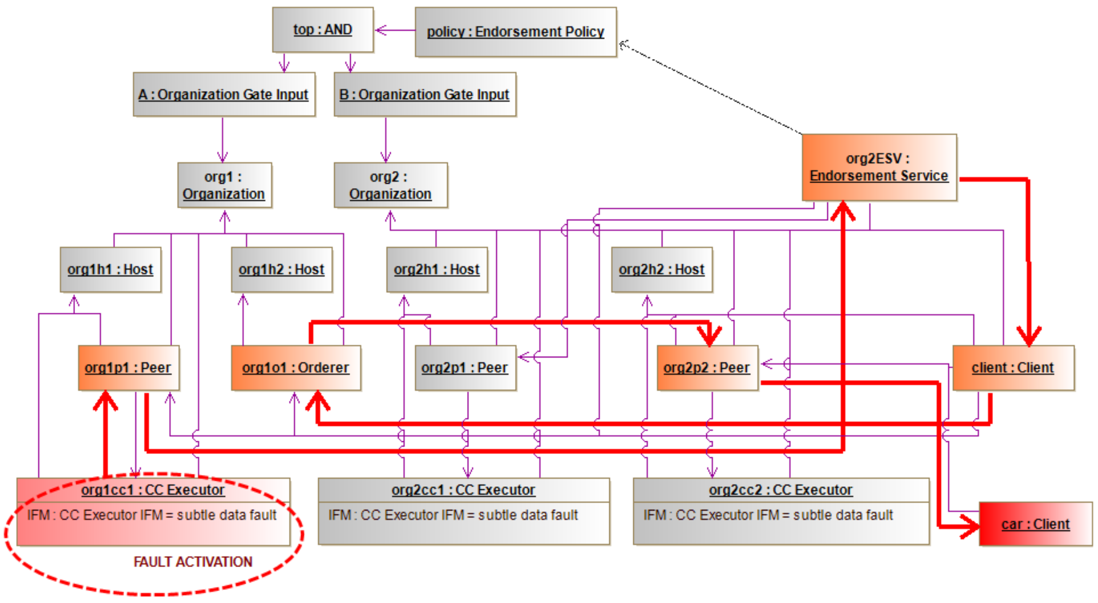


Figure 3.5: Instance model of the railroad crossing example

The ASP encoding of the model can be seen Listing 3.2. Notice how one can take advantage of ASP rules when defining the desired bindings of the objects, expressing more in less code. For example, instead of specifying the internal fault mode of each host as a seprate fact, such as `host_ifm(org1h1, ok)`, `host_ifm(org1h2, ok)`, and so on, one can instead intuitively say: the internal fault mode of any host is `ok`. In *gringo*: `host_ifm(H, ok) :- host(_, H)` – meaning a `host_ifm(H, ok)` fact is generated for every `host(_, H)` fact. The latter are not found anywhere among the bindings, because they are generated by other parts of the program (that do not need to be modified), based on the `org_hosts(Org, N)` fact for each organization.

The final line of the bindings file, `:- ref((ok, ok, ok))` ensures that only models where the reference transaction's result is faulty are considered. This is necessary to force *clingo* to show models that illustrate the possible fault propagation chain on Figure 3.5, because in several models, the chaincode executor components will not transform their input external fault injecting a *subtle data failure*, simulating cases when the software fault does not activate.

The result of running *clingo* on this input and filtering the otherwise incredibly verbose output to only show some relevant lines are found on Listing 3.3.

Listing 3.2: *gringo* bindings file of the railroad crossing model

```
input_failure_mode((ok, ok, ok)).

orgs(2).

org_hosts(O, 2) :- org(O).
host_ifm(H, ok) :- host(_, H).
host_link(A, B, up) :- host(_, A), host(_, B).

org_ccexecs(org1, 1). ccexec_alloc(org1cc1, org1h1).
org_ccexecs(org2, 2).
ccexec_alloc(org2cc1, org2h1). ccexec_alloc(org2cc2, org2h2).
ccexec_ifm(CC, subtle_fault) :- ccexec(_, CC).

org_peers(org1, 1). peer_alloc(org1p1, org1h1).

org_peers(org2, 2).
peer_ccexecs(P, 1) :- peer(_, P).
peer_alloc(org2p1, org1h1). peer_ccexec(org2p1, org2cc1).
peer_alloc(org2p2, org1h2). peer_ccexec(org2p2, org2cc2).

endorsement_policy(top, and, org1, org2).
endorsement_link(org2ESV, (org1p1; org1p2)).

org_orderers(org1, 1). orderer_alloc(org1o1, org1h2).
org_orderers(org2, 0).
ordering(solo).

blockvalidation_ifm(ok).

:- ref((ok, ok, ok)).
```

Listing 3.3: Output of the railroad crossing ASP program

```
$ clingo top.lp --out-ifs='\n' | grep -E
↪  'ref|input_failure_mode|endorsement_result|peer_endorsement|(ccexec|peer|orderer)_fp_i'
input_failure_mode((ok,ok,ok))
ccexec_fp_i(org1cc1,(ok,ok,ok),(ok,ok,subtle))
ccexec_fp_i(org2cc1,(ok,ok,ok),(ok,ok,ok))
ccexec_fp_i(org2cc2,(ok,ok,ok),(ok,ok,subtle))
peer_fp_i(org1p1,(ok,ok,subtle),(ok,ok,subtle))
peer_fp_i(org2p1,(ok,ok,subtle),(ok,ok,subtle))
peer_fp_i(org2p2,(ok,ok,subtle),(ok,ok,subtle))
endorsement_result((ok,ok,subtle))
orderer_fp_i(org1o1,(ok,ok,subtle),(ok,ok,subtle))
orderer_fp_i(org2o1,(ok,ok,subtle),(ok,ok,subtle))
```

```
peer_endorsement(org1p1,(ok,ok,subtle))
peer_endorsement(org2p1,(ok,ok,subtle))
peer_endorsement(org2p2,(ok,ok,subtle))
ref((ok,ok,subtle))
```

Finding the best endorsement policy

In this example, I only assert that there are a given number of organizations (five) and each organization has two hosts, one peer and one orderer, along with some simplifications, such as all hosts being linked, and some constraints that ensure only 'interesting' models will be generated.

The most important lines are the following three:

```
down_hosts(N) :- #count{ H : host_ifm(H, down) } = N.
#maximize{ N : down_hosts(N) }.
#maximize{ N : endorsement_policy_nodes(N) }.
```

What these mean, is that we wish to maximize the number of unavailable hosts, while also maxizing the nodes present in the endorsement policy. This way, we can obtain a policy that will still make it possible for transactions to succeed, even if several hosts are down.

This kind of computation (optimization) is more complex, so *clingo* takes quite a bit longer to generate the answer sets. For the bindings defined in Listing 3.4, the optimal model returned by *clingo* is what is in Listing 3.5. Clearly, the endorsement policy is not optimal in the sense that it could be simplified (ie converted to conjunctive or disjunctive normal form), which could technically also be possible to accomplish by extending the program. However, even in this form, this serves as a demonstration that the model is not only capable of *impact analysis,* ie showing what might happen given input failures and internal failure modes (in other words, reasoning 'forwards'), but also other directions of reasoning. In this case, I only bind the transaction outcome and optimize parameters that were bound in the previous model.

Listing 3.4: *gringo* bindings file for finding an optimal endorsement policy

```
#const orgN         = 5.
#const hostsPerOrg  = 2.

input_failure_mode((ok, ok, ok)).

orgs(orgN).
org_hosts(O, hostsPerOrg) :- org(O).

% Hosts form a complete mesh
host_link(HostA, HostB, up) :- host(_, HostA), host(_, HostB).

% The client's hosts are up
client_org(org1).
client_alloc(H) :- host_ifm(H, ok).
client_sub(H) :- host_ifm(H, ok).

{ org_peers(O, 1) } = 1 :- org(O).
peer_ifm(P, ok) :- peer(_, P).

org_ccexecs(O, N) :- org_peers(O, N).
ccexec_ifm(CC, ok) :- ccexec(_, CC).

down_hosts(N) :- #count{ H : host_ifm(H, down) } = N.
#maximize{ N : down_hosts(N) }.
#maximize{ N : endorsement_policy_nodes(N) }.
```

```

{ org_orderers(O, 1) } = 1 :- org(O).
orderer_ifm(O, ok) :- orderer(_, O).

ordering(raft).

blockvalidation_ifm(ok).

:- not ref((ok, ok, ok)).
```

Listing 3.5: Output of the optimization problem for finding an optimal endorsement policy

```
Answer: 7
host_ifm(org1h2,ok)
host_ifm(org2h1,ok)
host_ifm(org4h1,ok)
endorsement_policy(endorsement_node_0,and,endorsement_node_7,endorsement_node_6)
endorsement_policy(endorsement_node_1,and,endorsement_node_3,endorsement_node_0)
endorsement_policy(endorsement_node_2,and,endorsement_node_3,org5)
endorsement_policy(endorsement_node_3,and,endorsement_node_10,org2)
endorsement_policy(endorsement_node_4,or,endorsement_node_10,endorsement_node_8)
endorsement_policy(endorsement_node_5,and,org3,endorsement_node_4)
endorsement_policy(endorsement_node_6,and,endorsement_node_9,endorsement_node_10)
endorsement_policy(endorsement_node_7,and,org4,org2)
endorsement_policy(endorsement_node_8,and,endorsement_node_9,endorsement_node_5)
endorsement_policy(endorsement_node_9,and,org4,org2)
endorsement_policy(endorsement_node_10,and,endorsement_node_9,org1)
ref((ok,ok,ok))
host_ifm(org1h1,down)
host_ifm(org2h2,down)
host_ifm(org3h1,down)
host_ifm(org3h2,down)
host_ifm(org4h2,down)
host_ifm(org5h1,down)
host_ifm(org5h2,down)
```

4 Chaincode Fault Tolerance with N-Version Programming

In this chapter, I propose two ways to introduce N-Version Programming (NVP) to *Fabric* chaincode, increasing software diversity in an effort to be more fault tolerant. The first, 'classic' approach is not specific to *Fabric* and does not rely on its features. It is transparent to *Fabric*, built on top of its existing architecture and operation. I show a possible software architecture and implementation design for this setup.

The second approach is rather different, for it puts *Fabric*'s consensus mechanism to work to achieve n-version voting, meaning peers have their own, potentially private implementations of the chaincode specification and the DLT system itself takes the role of the n-version voter component, deciding what goes into the ledger in the end. This is very much in the spirit of DLT, but does require modifications to *Fabric*'s consensus mechanism.

Although my current *Fabric* model shown in Section 3.2 does not include them, I plan to add support for analyzing the usage of both of these approaches at design time. Then, it will be possible to use it to answer questions such as *how many indenepdent implementations do we need to ensure that n software faults are tolerated by the system?*

4.1 Classic Approach

The straightforward method to adopt NVP for *Fabric* chaincode is to simply install not one, but n implementations of the same chaincode specification everywhere (ie on every peer where the single chaincode would normally be installed) and then ensure that each version is executed, the results are compared and some business logic decides the end result. This is the bare minimum, but more useful features can be added, such as an additional layer of input and output validation before and after the versions are executed.

An arguably more elegant alteration of this method is to package the entire NVP architecture into a single chaincode container, so in the perspective of *Fabric*, only a single chaincode is installed and it can be invoked as usual. Behind the scenes, this chaincode is a facade hiding several chaincode versions and the validation and voting logic.

4.1.1 Master Chaincode as Controller

If one wishes to follow the first, elementary solution, the following steps must be taken:

1. Create n independent implementations of the chaincode specification, preferably by separate teams, possibly in different programming languages.
2. Create a *master* chaincode as en entry point: it may perform input parameter checking, then invoke all n versions (passing on the input parameters), collect the results, compare them and decide which (if any) result should be considered correct, and finally return that to the peer.
3. Install all n versions as well as the master chaincode on all peers desired to be able to execute the chaincode.
4. Ensure that clients not allowed to directly interact with the n versions, but only the master, controller chaincode.

The correct implementation of the master chaincode is essential, as it is a single point of failure in the system. Thankfully input/output validation and voting logic is not expected to be overly complex to do right. *Fabric*'s permissions can be used to ensure who may invoke what chaincode, so it is possible to forbid the invocation of the individual versions by any client.

One downside of this method is of course that $n + 1$ chaincode versions must be installed and maintained separately and *Fabric* has no way of knowing they are related in any way. On the other hand, this kind of complete separation of the implementations makes it possible to mix different programming languages, further increasing software diversity – something which is not supported by the containerized approach introduced in the next subsection.

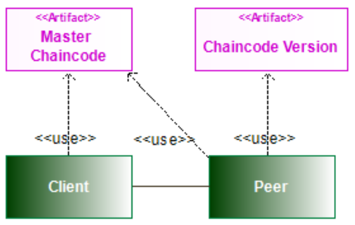


Figure 4.1: Component diagram of the Master Chaincode-based approach for NVP

4.1.2 Containerized Approach

Since *Fabric* chaincode runs in *Docker* containers, it is possible to develop more complex architectures, including the parallel execution of code by threading. The idea is to package the entire $n + 1$ chaincodes into one self-contained unit that can be installed on a *Fabric* peer just like any regular chaincode.

Instead of exposing the peer's interface to the chaincode implementations so that they can read the ledger contents, I propose providing a *proxy* that is able to cache ledger reads (since a single read of a key-value pair is always sufficient, *Fabric* cannot 'read-your-write'). The final read/write set must be built by the NVP Controller (NVC) component, merging the reads intercepted by the proxy and the writes suggested by the chaincode versions.

One possible way to adopt this approach in software is using *Java threads* and the *active object* pattern. Implementations of the same chaincode specification interface are known by the controller class. After ensuring the validity of the input, the implementations are executed, in parallel, by multithreading.

4.2 Consensus-Based Approach ('O-Version Programming')

Instead of actually installing multiple chaincode versions on the peers either directly or using the architecture outlined in Subsection 4.1.2, it is possible to rely on the consensus mechanism of *Fabric*. In this case, organizations and/or peers may have their own chaincode version installed independently (hence the name 'O-Version Programming': the 'O' stands for 'Organization'). In some contexts, this might even be a requirement; for example, consider a weather forecast service where several clients attempt to submit their sensor data to the ledger. The supporting chaincode can be the organization's own, as long as it implements the same specification as all others.

Contrary to the other approaches, in this method, there is no specialized voter component. Deciding which versions' results are correct is deferred to the consensus protocol of the platform: that is, in the end, the configured endorsement policy determines the outcome. For example, for maximum fault tolerance (but least availability), given n peers hosting their own versions, an $n : n$ endorsement policy ensures either every

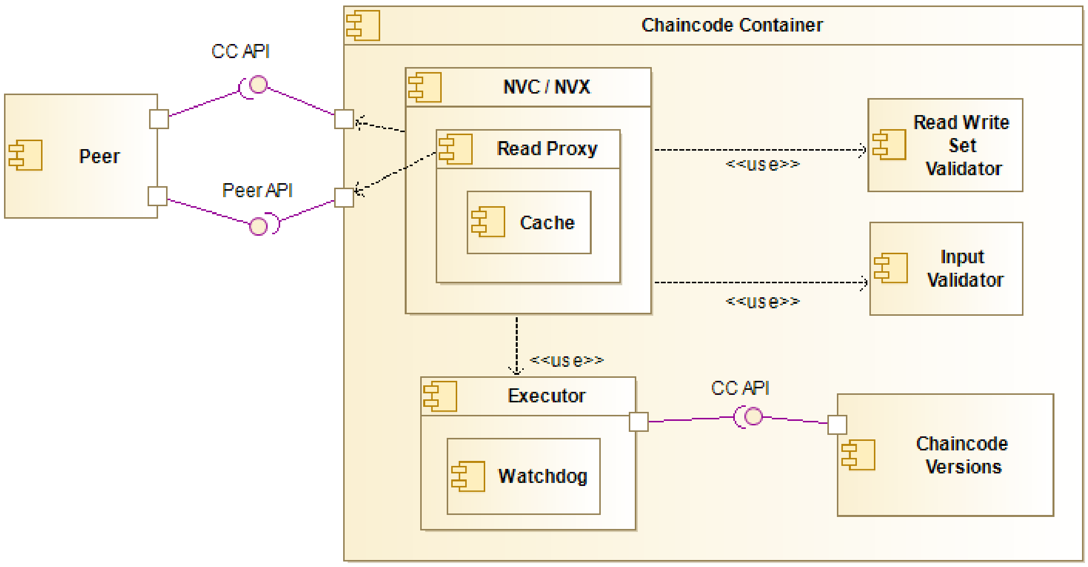


Figure 4.2: Component diagram of the containerized approach for NVP

single implementation has the same active software fault (quite unlikely), or the correct results are appended to the ledger, otherwise the transaction is rejected. Figure 4.4 contains an overview figure of this strategy.

An interesting issue with this approach is that it clearly interferes with the ledger integrity preserving role of endorsement. Normally, all peers have the same one implementation of the chaincode and endorsement ensures that no malicious organization or peer is able to alter ledger contents to their advantage. If there are six peers, than a four-out-of-six endorsement policy can tolerate up to three peers going rogue and endorsing an invalid or otherwise undesired or unacceptable transaction. However, if we also take potentially faulty implementations into consideration, this tolerance metric changes. This aspect is further explored in the next chapter, DLT Consensus as N-Version Voting.

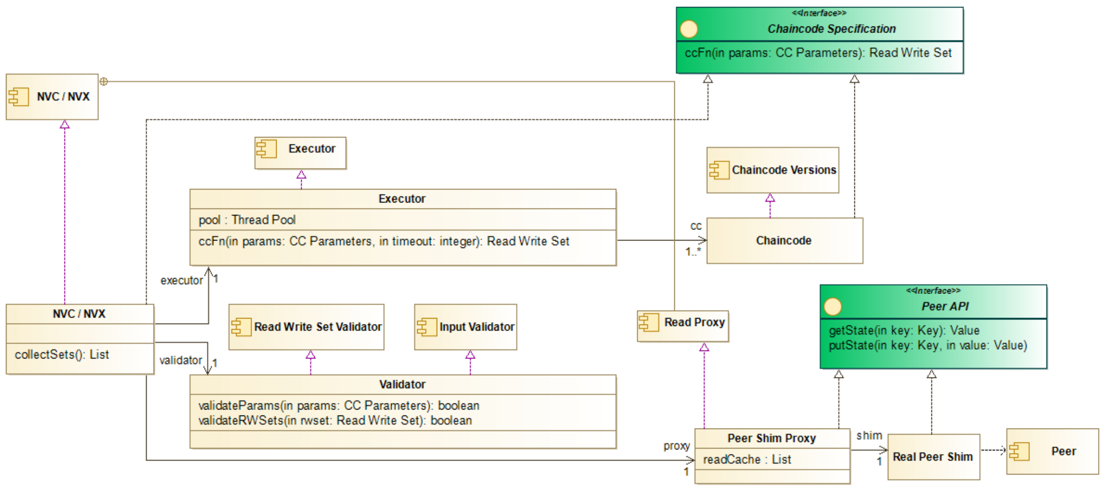


Figure 4.3: Possible class diagram of the containerized approach for NVP

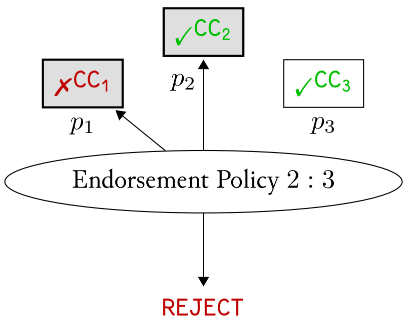
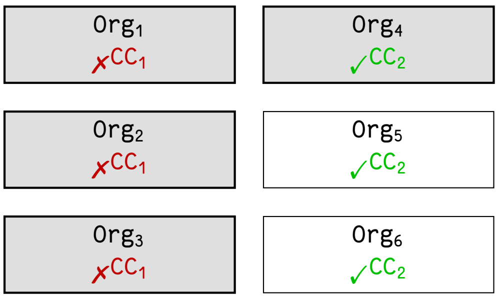


Figure 4.4: Overview of consensus-based NVP in *Fabric*

5 DLT Consensus as N-Version Voting

This chapter continues investigating the options of leveraging the consensus mechanism of DLT systems for NVP purposes, introduced in Section 4.2. Since consensus is already used for integrity protection, it is important to consider how slightly abusing it affects its existing capabilities. For example, consider a network of six organizations: three are running version A of a chaincode specification and the other three are running version B. If we consider one of the two versions faulty, it follows that at least four organizations' endorsement should be required to ensure integrity. This way, even if all three organizations who have the faulty chaincode version propose to append the same faulty transaction to the ledger, the addtionally required fourth organization, who certainly has a fault-free implementation, will prevent the undesired ledger update. Figure 5.1 offers a visualization of this example for better understanding.



If at least four organizations are required to endorse any transaction (thickened, grey boxes), it can be ensured that read/write sets resulting from the three erroneous executions of the chaincode do not end up on the ledger. However, provided one of the organizations is malicious, even one more endorser is required for integrity protection, since in the worst case, the malicious party might be among those who have the correct chaincode version installed.

Figure 5.1: Two chaincode versions distributed among six organizations

5.1 Obtaining a formula for the necessary endorsement policy based on diversity, software faults, and malicious organizations

For simplicity, let us assume for the remainder of this chapter, that every organization only maintains a single peer node. The question the answer to which we seek is the following: *given n peers and v chaincode versions, what $k : n$ endorsement policy is required to tolerate $m = 1, 2, \ldots, n$ malicious peers, if $f = 1, 2, \ldots, v$ of the versions are faulty?* We may combine the n peers and v chaincode versions into a single property, the *diversity ratio* or *ratio of diversity*, $r = \frac{v}{n}$.

Following the configuration of Figure 5.1, we can fill a table, trying to find a relationship: Table 5.1. It is not hard to notice the following patterns:

- Any number of malicious parties increases the necessary k value by one, no matter the number of software faults.
- The effect of a software fault on k depends on the diversity ratio. If there are v versions for n clients, that means $\frac{v}{n}$ peers have the version. Every software fault implies that $\frac{v}{n}$ more endorsers would be needed to eliminate the possibility of committing the faulty result to the ledger.
- Logically, at least one endorser is necessary.

And thus, the formula Equation 5.1 can be obtained. Figure 5.2 shows a 3D surface plot of how this function looks for different input values. The plot has been made with a fixed zero value for number of

n	v	r	f	m	$k:n$
6	1	6	0	0	1
6	1	6	1	$[0,n]$	$\varnothing$
6	1	6	0	1	2
6	1	6	0	2	3
6	1	6	0	3	4
6	1	6	0	4	5
6	1	6	0	5	6
6	1	6	0	6	$\varnothing$

n	v	r	f	m	$k:n$
6	2	3	0	0	1
6	2	3	1	0	4
6	2	3	2	$[0,n]$	$\varnothing$
6	2	3	0	1	2
6	2	3	0	2	3
6	2	3	0	3	4
6	2	3	0	4	5
6	2	3	0	5	6
6	2	3	0	6	$\varnothing$
6	2	3	1	1	5
6	2	3	1	2	6
6	2	3	1	$[3,n]$	$\varnothing$

Table 5.1: Table to help finding a formula for the required endorsement policy based on Figure 5.1

malicious organizations ($m = 0$), but from the formula it is clear that various values of m would simply shift the plot along the z (vertical) axis.

The result shows that for low diversity ratios and a high amount of software faults, very intolerant endorsement policies are necessary, as expected. Software faults have a much higher impact on the necessary endorsement policy than the number of malicious organizations to tolerate – this makes sense, as several organizations might be running the same version.

$$\text{bestk}(r, f, m) = \frac{f}{r} + m + 1 \tag{5.1}$$

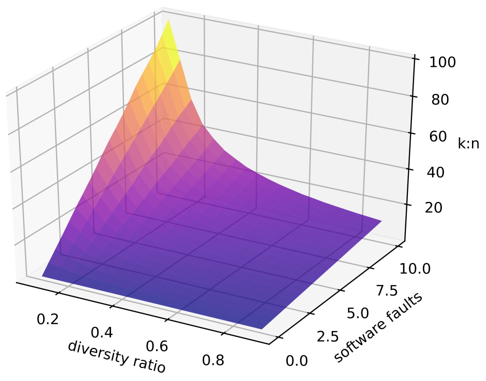


Figure 5.2: 3D surface plot of the bestk function

6 Conclusion and Future Work

In my work, I have successfully used classic Error Propagation Analysis to implement a model of a specific DLT platform, *Hyperledger Fabric*. It includes a number of internal failure modes and propagation characteristics based on my analysis and understanding of the system. I demonstrate using this model to show certain properties of hypothetical model instances, including revealing how a safety-critical application may fail, potentially causing an accident. The model is completely reusable and in the future, I plan on extending it with numerous additional component failure modes (including vulnerabilities, which I have not yet included in the model) and behaviours. Furthermore, I intend to make it possible to enable various fault tolerance mechanisms in the model to analyize their effects.

To address the impact of software faults in chaincode, I have proposed revisiting classic N-Version Programming, which is capable of increasing fault tolerance by means of highering software diversity. I have presented two rather different ways of its integration: a classic approach, and 'O-Version Programming.' As the latter builds on the same mechanism of the network which ensures its high integrity, consensus, I briefly analyzed its impact by observing how the number of chaincode versions and the number of software faults interplay with the endorsement policy and number of malicious participants in the network. I conclude that when conensus is used for NVP, the choice of endorsement policy is mostly dependent on the number of the chaincode versions and the number of peers. I have offered an architectural design for the 'classic' style (instead of relying on consensus, each executor executes the same n versions), which I would like to prototype as future work to demonstrate its viability.

As an additional future improvement, I am considering the development of a simple application that visualizes the models generated by the ASP program that facilitates sensitivity analysis, as its current output is rather hard to read. Optimally, this piece of software should also be able to draw fault and event trees, which would normally have to be done by hand.

Acronyms

AI	Artificial Intelligence 13
API	Application Programming Interface 6
ASP	Answer Set Programming 1, 4, 9, 10, 11, 12, 14, 17, 18, 19, 24, 25, 34
BFT	Byzantine Fault Tolerant 20
CBDC	Central Bank Digital Currency 18
CFT	Crash Fault Tolerant 20
CSP	Constraint Set Programming 9
DAO	Decentralized Autonomous Organization 3
DLT	Distributed Ledger Technology 1, 3, 4, 5, 6, 7, 13, 14, 17, 28, 30, 32, 34
DOS	Denial Of Service 18
EPA	Error Propagation Analysis 1, 2, 3, 4, 5, 8, 9, 10, 12, 18, 34
FMEA	Failure Mode and Effects Analysis 8
FPA	
	Fault Propagation Analysis 8
	Failure Propagation Analysis 8
FTA	Fault Tree Analysis 8
IT	Information Technology 4, 5, 6, 11
ML	Machine Learning 13
MSP	Membership Service Provider 6
MVCC	MultiVersion Concurrency Control 14, 15, 17
NNVP	N-of-N-Version Programming 13
NP	Non-deterministic Polynomial time 9
NVC	NVP Controller 29
NVP	N-Version Programming 1, 2, 4, 5, 13, 28, 29, 30, 31, 32, 34, 35
NVX	N-Version eXecution environment 13
OVP	O-Version Programming 1, 4, 29, 34
P2P	peer-to-peer 5
PKI	Public Key Infrastructure 6
SPOF	Single Point Of Failure 5, 16
TCP	Transmission Control Protocol 8
TLS	Transport Layer Security 6
UML	Unified Modelling Language 8

Bibliography

Andola, N., Raghav, Gogoi, M., Venkatesan, S., and Verma, S. (2019). Vulnerabilities on hyperledger fabric. *Pervasive and Mobile Computing*, 59.

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W., and Yellick, J. (2018). Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA. Association for Computing Machinery.

Avizienis, A. (1986). The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11:1491–1501.

Avizienis, A. (1995). *The Methodology of N-Version Programming*, volume 3, page 23–.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, page 11–33.

Beckert, B., Herda, M., Kirsten, M., and Schiffl, J. (2018). Formal specification and verification of hyperledger fabric chaincode.

Bernardi, S., Merseguer, J., and Petriu, D. C. (2008). Adding dependability analysis capabilities to the MARTE profile. In Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., and Völter, M., editors, *Model Driven Engineering Languages and Systems*, page 736–750, Berlin, Heidelberg. Springer Berlin Heidelberg.

Breidenbach, L., Daian, P., Tramèr, F., and Juels, A. (2018). Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, page 1335–1352, Baltimore, MD. USENIX Association.

Brotsis, S., Kolokotronis, N., Limniotis, K., Bendiab, G., and Shiaeles, S. (2020). On the security and privacy of hyperledger fabric: Challenges and open issues. In *2020 IEEE World Congress on Services (SERVICES)*, page 197–204.

Brown, R., Carlyle, J., Grigg, I., and Hearn, M. (2016). Corda: An introduction.

Chowdhury, M. J. M., Ferdous, M. S., Biswas, K., Chowdhury, N., Kayes, A. S. M., Alazab, M., and Watters, P. (2019). A comparative analysis of distributed ledger technology platforms. *IEEE Access*, 7:167930–167943.

Dabholkar, A. and Saraswat, V. (2019). Ripping the fabric: Attacks and mitigations on hyperledger fabric. In Shankar Sriram, V. S., Subramaniyaswamy, V., Sasikaladevi, N., Zhang, L., Batten, L., and Li, G., editors, *Applications and Techniques in Information Security*, page 300–311, Singapore. Springer Singapore.

Dimopoulos, Y., Nebel, B., and Koehler, J. (1997). Encoding planning problems in nonmonotonic logic programs.

Földvári, A., Biczók, G., Kocsis, I., Gönczy, L., and Pataricza, A. (2021). Impact assessment of IT security breaches in cyber-physical systems: Short paper. In *2021 10th Latin-American Symposium on Dependable Computing (LADC)*.

Gallina, B. and Punnekkat, S. (2011). FI4FA: A formalism for incompletion, inconsistency, interference and impermanence failures' analysis. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, page 493–500.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. (2011). Potassco: The Potsdam answer set solving collection. *AI Commun.*, 24:107–124.

Gersting, J., Nist, R., Roberts, D., and Van Valkenburg, R. (1991). A comparison of voting algorithms for n-version programming. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume ii, page 253–262.

Gujarati, A., Gopalakrishnan, S., and Pattabiraman, K. (2020). New wine in an old bottle: N-version programming for machine learning components. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, page 283–286.

Hambuch, K. (2022). Vállalati okosszerződések statikus analízise. Master's thesis, Budapest University of Technology and Economics.

Hao, Y., Li, Y., Dong, X., Fang, L., and Chen, P. (2018). Performance analysis of consensus algorithm in private blockchain. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, page 280–285.

Knight, J. C. and Leveson, N. G. (1987). An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109.

Kocsis, I. (2018). Design for dependability through error propagation space exploration. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, page 172–178.

Kocsis, I. (2019). *Qualitative Models in Resilience Assurance*. PhD thesis, Budapest University of Technology and Economics.

Li, D., Wong, W. E., and Guo, J. (2020). A survey on blockchain for enterprise using hyperledger fabric and composer. In *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, page 71–80.

Lifschitz, V. (2008). What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, page 1594–1597. AAAI Press.

Machida, F. (2019). N-version machine learning models for safety critical systems. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, page 48–51.

Marek, V. W. and Truszczynski, M. (1998). Stable models and an alternative logic programming paradigm.

Melo, C., Oliveira, F., Dantas, J., Araujo, J., Pereira, P., Maciel, R., and Maciel, P. (2022). Performance and availability evaluation of the blockchain platform hyperledger fabric. *J. Supercomput.*, 78(10):12505–12527.

Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*.

Palma, S. D., Pareschi, R., and Zappone, F. (2021). What is your distributed (hyper)ledger? In *2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, page 27–33.

Pataricza, A. (2002). From the general resource model to a general fault modeling paradigm?

Pataricza, A. (2006). Model based dependability analysis. Hungarian Academy of Sciences. DSc thesis.

Podgorelec, B., Keršič, V., and Turkanović, M. (2019). Analysis of fault tolerance in permissioned blockchain networks. In *2019 XXVII International Conference on Information, Communication and Automation Technologies (ICAT)*, page 1–6.

Pongnumkul, S., Siripanpornchana, C., and Thajchayapong, S. (2017). Performance analysis of private blockchain platforms in varying workloads. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, page 1–6.

Praitheeshan, P., Pan, L., Yu, J., Liu, J. K., and Doss, R. (2019). Security analysis methods on ethereum smart contract vulnerabilities: A survey. *CoRR*, abs/1908.08605.

Rehman, A. U., Aguiar, R., and Barraca, J. (2019). Fault-tolerance in the scope of software-defined networking (SDN). *IEEE Access*.

Trivedi, K. S. and Bobbio, A. (2017). *Reliability and Availability Engineering: Modeling, Analysis, and Applications*. Cambridge University Press.

Wood, G. (2015). Ethereum: A secure decentralised generalised transaction ledger.

Wu, A., Rubaiyat, A. H. M., Anton, C., and Alemzadeh, H. (2018). Model fusion: Weighted n-version programming for resilient autonomous vehicle steering control. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, page 144–145.

Xu, H., Chen, Z., Wu, W., Jin, Z., Kuo, S.-y., and Lyu, M. (2019). NV-DNN: Towards fault-tolerant DNN systems with n-version programming. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, page 44–47.

Yamashita, K., Nomura, Y., Zhou, E., Pi, B., and Jun, S. (2019). Potential risks of hyperledger fabric smart contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, page 1–10.

Appendices

A Complete source code listings

See Subsection 3.2.1 for the structuring and roles of the files whose contents are included below.

Listing A.1: Contents of `model/structure.lp`

```
%%%%%%%%%%%%%%%%%%%%%%%%%
%   INPUT               %
%%%%%%%%%%%%%%%%%%%%%%%%%

{ input_failure_mode(F) : failure_mode(F) } = 1.


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ORGANIZATIONS       %
%%%%%%%%%%%%%%%%%%%%%%%%%

#script (lua)
clingo = require('clingo')
F = clingo.Function

function orgname(i_)
    i = i_.number
    return F('org' .. i)
end
#end.

{ orgs(1..max_orgs) } = 1.
{ org(@orgname(1..N)) } = N :- orgs(N).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   HOSTS               %
%%%%%%%%%%%%%%%%%%%%%%%%%

#script (lua)
clingo = require('clingo')
F = clingo.Function

function orghost(org_, i_)
    org = org_.name
    i = i_.number
    return F(org .. 'h' .. i)
end
#end.

{ org_hosts(O, 1..max_org_hosts) } = 1 :- org(O).
{ host(O, @orghost(O, 1..N)) : org(O) } = N :- org_hosts(O, N).

{ host_link(HostA, HostB, (up; down)) } 1 :-
    host(_, HostA), host(_, HostB),
    HostA != HostB, not host_link(HostB, HostA, _).
host_link(HostA, HostB, FM) :- host_link(HostB, HostA, FM).
host_link(Host, Host, up) :- host(_, Host).
host_link(HostA, HostB, down) :-
    host(_, HostA), host(_, HostB), not host_link(HostA, HostB, up).
```

```

{ host_ifm(Host, (ok; down; compromised)) } = 1 :- host(_, Host).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   CLIENT              %
%%%%%%%%%%%%%%%%%%%%%%%%%

{ client_org(Org) : org(Org) } = 1.
{ client_alloc(Host) : host(Org, Host), client_org(Org) } = 1.
{ client_orderer(Ord) : orderer(Org, Ord), client_org(Org) } = 1.
{ client_pub(Peer) : peer(Org, Peer), client_org(Org) } = 1.
{ client_sub(Peer) : peer(Org, Peer), client_org(Org) } = 1.


%%%%%%%%%%%%%%%%%%%%%%%%%
%   PEERS               %
%%%%%%%%%%%%%%%%%%%%%%%%%

#script (lua)
clingo = require('clingo')
F = clingo.Function

function orgpeer(org_, i_)
    org = org_.name
    i = i_.number
    return F(org .. 'p' .. i)
end
#end.

{ org_peers(O, 1..max_org_peers) } = 1 :- org(O).
{ peer(O, @orgpeer(O, 1..N)) : org(O) } = N :- org_peers(O, N).


{ peer_alloc(Peer, Host) : host(Org, Host) } = 1 :- peer(Org, Peer).

{ peer_ifm(Peer, ok) } = 1 :- peer(_, Peer).

{ peer_ccexecs(Peer, 1..N) } = 1 :-
    peer(Org, Peer), org_ccexecs(Org, N).
{ peer_ccexec(Peer, CC) : ccexec(Org, CC) } = N :-
    peer(Org, Peer), peer_ccexecs(Peer, N).

%%%%%%%%%%%%%%%%%%%%%%%%%
%   CC EXECUTORS        %
%%%%%%%%%%%%%%%%%%%%%%%%%

#script (lua)
clingo = require('clingo')
F = clingo.Function

function orgcc(org_, i_)
    org = org_.name
    i = i_.number
    return F(org .. 'cc' .. i)
end
#end.

{ org_ccexecs(O, 1..max_org_ccexecs) } = 1 :- org(O).
{ ccexec(O, @orgcc(O, 1..N)) : org(O) } = N :- org_ccexecs(O, N).
{ ccexec_alloc(CC, Host) : host(Org, Host) } = 1 :- ccexec(Org, CC).

{ ccexec_ifm(CC, (ok; subtle_fault; coarse_fault)) } = 1 :-
    ccexec(_, CC).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ENDORSEMENT POLICY  %
%%%%%%%%%%%%%%%%%%%%%%%%%

#script (lua)
clingo = require('clingo')
F = clingo.Function

function node(i_)
    i = i_.number
    return F('endorsement_node_' .. i)
end
#end.

{ endorsement_policy_nodes(0..max_endorsement_policy_nodes) } = 1.
{ endorsement_policy_node(@node(1..N)) } = N :-
    endorsement_policy_nodes(N).
endorsement_policy_node(endorsement_node_0).

{
    endorsement_policy(Node, (and; or), A, B)
    :   org(A), org(B), A != B
    ;
    endorsement_policy(Node, (and; or), A, B)
    :   org(A), endorsement_policy_node(B), Node != B
    ;
    endorsement_policy(Node, (and; or), A, B)
    :   endorsement_policy_node(A), org(B), Node != A
    ;
    endorsement_policy(Node, (and; or), A, B)
    :   endorsement_policy_node(A), endorsement_policy_node(B),
        Node != A, Node != B, A != B
} = 1 :- endorsement_policy_node(Node).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ENDORSEMENT SVCS    %
%%%%%%%%%%%%%%%%%%%%%%%%%

#script (lua)
clingo = require('clingo')
F = clingo.Function

function orgesv(org_)
    org = org_.name
    return F(org .. 'ESV')
end
#end.

{ endorsement(O, @orgesv(O)) } = 1 :- org(O).

1 {
    endorsement_link(OrgESV, Peer)
    :   endorsement(Org, OrgESV),
        peer(OtherOrg, Peer)
} :- org(Org), org(OtherOrg), OtherOrg != Org.


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ORDERERS            %
%%%%%%%%%%%%%%%%%%%%%%%%%

#script (lua)
clingo = require('clingo')
F = clingo.Function

function orgorderer(org_, i_)
    org = org_.name
    i = i_.number
    return F(org .. 'o' .. i)
end
#end.

{ org_orderers(O, 1..max_org_orderers) } = 1 :- org(O).
{ orderer(O, @orgorderer(O, 1..N)) : org(O) } = N :- org_orderers(O, N).

{ orderer_alloc(Orderer, Host) : host(Org, Host) } = 1 :-
    orderer(Org, Orderer).

{ orderer_ifm(Orderer, ok) } = 1 :- orderer(_, Orderer).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ORDERING SVC        %
%%%%%%%%%%%%%%%%%%%%%%%%%

{ ordering(solo; kafka; raft) } = 1.
{ ordering_leader(Orderer) : orderer(_, Orderer), ordering(solo) } 1.


%%%%%%%%%%%%%%%%%%%%%%%%%
%   BLOCK VALID SCVS    %
%%%%%%%%%%%%%%%%%%%%%%%%%

{ blockvalidation_ifm(ok; conflict) } = 1.
```

Listing A.2: Contents of `model/behaviour.lp`

```
%
%   T R A N S A C T I O N    F L O W
%
%  CLIENT ⟷ ENDORSEMENT SVC           ([1] tx proposal)
%              : ⟷ PEER1 ⟷ CC1..N
%              : ⟷ PEER2 ⟷ CC1..N
%  CLIENT ⟶ ORDERING SVC              ([2] ordering)
%              : ⟷ PEER1 ⟷ CC1..N     ([3] validaton)
%              : ⟷ PEER2 ⟷ CC1..N
%  CLIENT ⟷ PEERX                     (client sees block)
%

%
```

```
%   G L O S S A R Y
%
% F:    Fault
% EF:   External Fault
% FP:   Failure/Fault Propagation
% IFM:  Internal Failure Mode
% OFM:  Output Failure Mode
%
% _g postfix means the generic fault propagation behaviour
%   -> in an IFM, when receiving an EF, what is the resulting OFM
%
% _i postfix means instance fault propagation behaviour
%   (fault propagation behaviour bound to an instance of the component)


%%%%%%%%%%%%%%%%%%%%%%%%%
%   CLIENT              %
%%%%%%%%%%%%%%%%%%%%%%%%%

% The client completely propagates the input failure mode.
{
    % Client host is up, propagate input failure
    client_fp_i((P, T, V), (P, T, V))
    :   client_alloc(Host), host_ifm(Host, ok),
        input_failure_mode((P, T, V))
    ;
    % Client host is down, guaranteed omission failure mode
    client_fp_i((P, T, V), (omission, T, V))
    :   client_alloc(Host), host_ifm(Host, down),
        input_failure_mode((P, T, V))
} = 1.


%%%%%%%%%%%%%%%%%%%%%%%%%
%   PEER                %
%%%%%%%%%%%%%%%%%%%%%%%%%

% Peers are always OK.
% External faults are propagated without transformation.
peer_fp_g(ok, F, F) :- failure_mode(F).

% Peer external faults come from their CC executors.
{
    % Peer host is up, link with CC executor host is up
    peer_fp_i(ID, EF, OFM)
    :   peer_alloc(ID, Host), host_ifm(Host, ok),
        peer_ifm(ID, IFM),
        peer_fp_g(IFM, EF, OFM),
        peer_ccexec(ID, CC), ccexec_fp_i(CC, _, EF),
        peer_alloc(ID, PH), ccexec_alloc(CC, CCH),
        host_link(PH, CCH, up)
    ;
    % Peer host is up, link with CC executor host is down -> omission
    peer_fp_i(ID, (P, T, V), (omission, T, V))
    :   peer_alloc(ID, Host), host_ifm(Host, ok),
        peer_ifm(ID, IFM),
        peer_ccexec(ID, CC), ccexec_fp_i(CC, _, (P, T, V)),
        peer_alloc(ID, PH), ccexec_alloc(CC, CCH),
        host_link(PH, CCH, down)
    ;
    % Peer host is down -> omission
    peer_fp_i(ID, (P, T, V), (omission, T, V))
    :   peer_alloc(ID, Host), host_ifm(Host, down),
        peer_ccexec(ID, CC), ccexec_fp_i(CC, _, (P, T, V))
} = 1 :-
    peer(_, ID).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   CC EXECUTORS        %
%%%%%%%%%%%%%%%%%%%%%%%%%

% CC executors are either OK or are SUBTLy or COARSEly faulty.
% External faults MAY be propagated (as faults may activate or not).
ccexec_fp_g(ok, F, F) :- failure_mode(F).
ccexec_fp_g(subtle_fault, (P, T, V), (P, T, (V; subtle))) :-
    failure_mode((P, T, V)).
ccexec_fp_g(coarse_fault, (P, T, V), (P, T, (V; coarse))) :-
    failure_mode((P, T, V)).

% CC executor external faults come from the input data.
{
    % CC executor host up
    ccexec_fp_i(ID, EF, OFM)
    :   ccexec_alloc(ID, Host), host_ifm(Host, ok),
        ccexec_ifm(ID, IFM),
        ccexec_fp_g(IFM, EF, OFM),
        client_fp_i(_, EF)
    ;
    % CC executor host down -> omission
    ccexec_fp_i(ID, (P, T, V), (omission, T, V))
    :   ccexec_alloc(ID, Host), host_ifm(Host, down),
        client_fp_i(_, (P, T, V))
} = 1 :-
    ccexec(_, ID).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ENDORSERMENT        %
%%%%%%%%%%%%%%%%%%%%%%%%%

{ peer_endorsement(Peer, Fault) : peer_fp_i(Peer, _, Fault) } = 1 :-
    peer(_, Peer).

{
    org_endorsement(Org, Fault)
    :   peer_endorsement(Peer, Fault), peer(Org, Peer)
} = 1 :- org(Org).

% Always the `best' endorsement is chosen per organization.  Here we
% eliminiate solutions that would choose a faulty peer endorsement for
% an organization, while an `OK' endorsement exists.
:-  org_endorsement(Org, (P, _, _)), P != ok,
    peer_endorsement(Peer, (ok, _, _)), peer(Org, Peer).
:-  org_endorsement(Org, (_, T, _)), T != ok,
    peer_endorsement(Peer, (_, ok, _)), peer(Org, Peer).
:-  org_endorsement(Org, (_, _, V)), V != ok,
    peer_endorsement(Peer, (_, _, ok)), peer(Org, Peer).

% Organization `nodes' in the endorsement policy tree match with the
% organization's endorsement result
endorsement_sub(Org, org, na, na, Endorsement) :-
    org_endorsement(Org, Endorsement).
% Logical `nodes' in the endorsement policy tree are `delegated'
{
    endorsement_sub(Node, or, A, B, (ok, T, V))
    :   endorsement_policy(Node, or, A, B),
        endorsement_sub(A, _, _, _, (ok, T, V))
    ;
    endorsement_sub(Node, or, A, B, (ok, T, V))
    :   endorsement_policy(Node, or, A, B),
        endorsement_sub(B, _, _, _, (ok, T, V))
    ;
    endorsement_sub(Node, and, A, B, (ok, T, V))
    :   endorsement_policy(Node, and, A, B),
        endorsement_sub(A, _, _, _, (ok, T, V)),
        endorsement_sub(B, _, _, _, (ok, T, V))
    ;
    endorsement_sub(Node, Op, A, B, (omission, ign, ign))
    :   endorsement_policy(Node, Op, A, B),
        not endorsement_sub(A, _, _, _, (ok, _, _)),
        endorsement_sub(B, _, _, _, (ok, _, _))
    ;
    endorsement_sub(Node, Op, A, B, (omission, ign, ign))
    :   endorsement_policy(Node, Op, A, B),
        endorsement_sub(A, _, _, _, (ok, _, _)),
        not endorsement_sub(B, _, _, _, (ok, _, _))
} = 1 :-
    endorsement_policy(Node, _, _, _).
% Final endorsement result is that of the top/root/0th `node'.
endorsement_result(F) :-
    endorsement_sub(endorsement_node_0, _, _, _, F).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ORDERERS            %
%%%%%%%%%%%%%%%%%%%%%%%%%

% Orderers are always OK.
% External faults are propagated without transformation.
orderer_fp_g(ok, F, F) :- failure_mode(F).

% Orderer external faults come from the endorsements (proposals) they
% receive.
{
    % Orderer host is up, link with client host is up
    orderer_fp_i(ID, EF, OFM)
    :   orderer_alloc(ID, Host), host_ifm(Host, ok),
        orderer_ifm(ID, IFM),
        orderer_fp_g(IFM, EF, OFM),
        endorsement_result(EF),
        client_alloc(CH), host_link(Host, CH, up)
    ;
    % Orderer host is up, link with client host is down → omission
    orderer_fp_i(ID, (P, T, V), (omission, T, V))
    :   orderer_alloc(ID, Host), host_ifm(Host, ok),
        endorsement_result((P, T, V)),
        client_alloc(CH), host_link(Host, CH, down)
    ;
    % Orderer host is down → omission
    orderer_fp_i(ID, (P, T, V), (omission, T, V))
    :   orderer_alloc(ID, Host), host_ifm(Host, down),
        endorsement_result((P, T, V))
} = 1 :-
    orderer(_, ID).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ORDERING SVC        %
%%%%%%%%%%%%%%%%%%%%%%%%%

% SOLO ordering: selected (leading) orderer completely propagates
% failure mode
{
    ordering_fp_i(F, F)
    :   orderer_fp_i(Orderer, _, F)
} = 1 :-
    ordering(solo), ordering_leader(Orderer).

% KAFKA or RAFT ordering: ceil(N/2) orderers must be working (ie not
% have an omission provision fault) and the client must have an up link
% to them for ordering to succeed (not result in omission).
%
% Count how many orderers there are in total
orderer_count(N) :- N = #count{ Orderer : orderer(_, Orderer) }.
% Calculate ceil(N/2), the majority orderer count
orderer_majority(K) :- orderer_count(N), K = N / 2 + 1.
% Count orderers that do not propagate an omission provision fault
orderer_non_omission_count(M) :-
    M = #count{
        Orderer
        :   orderer_fp_i(Orderer, _, (P, _, _)), P != omission
    }.
% Ordering service fault propagation rule
{
    % There are enough orderers that are not propagation omission
    ordering_fp_i(EF, (P, T, V))
    :   orderer_majority(K), orderer_non_omission_count(M), M ⩾ K,
        orderer_fp_i(_, EF, (P, T, V)), P != omission
    ;
    % There are not enough orderers that are not propagation omission
    % → omission is inevitable since ordering cannot take place
    ordering_fp_i(EF, (omission, T, V))
    :   orderer_majority(K), orderer_non_omission_count(M), M < K,
        orderer_fp_i(_, EF, (_, T, V))
} = 1 :-
    ordering(kafka; raft).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   BLOCK VALID SVCS    %
%%%%%%%%%%%%%%%%%%%%%%%%%

blockvalidation_fp_g(F, F) :- failure_mode(F).

% Block validation propagates completely
{
    blockvalidation_fp_i(EF, OFM)
    :   blockvalidation_ifm(ok),
        blockvalidation_fp_g(EF, OFM), ordering_fp_i(_, EF)
    ;
    blockvalidation_fp_i((P, T, V), (P, late, V))
    :   blockvalidation_ifm(conflict),
        ordering_fp_i(_, (P, T, V))
} = 1.

%------------------------------------------------------------------------

%%%%%%%%%%%%%%%%%%%%%%%%%
%   REFERENCE TX RESULT %
%%%%%%%%%%%%%%%%%%%%%%%%%

{ ref(F) : blockvalidation_fp_i(_, F) } = 1.
```

Listing A.3: Contents of `model/failure.lp`

```
provision_failure_mode(ok; omission; commission; ign).
timing_failure_mode(ok; early; late; ign).
value_failure_mode(ok; subtle; coarse; ign).

failure_mode((P, T, V)) :-
    provision_failure_mode(P),
    timing_failure_mode(T),
    value_failure_mode(V).
```

Listing A.4: Contents of `constants.lp`

```
% Maximum total number of organizations to generate
#const max_orgs = 5.
% Maximum hosts to generate per organization
#const max_org_hosts = 3.
% Maximum number of peers to generate per organization
#const max_org_peers = 6.
% Maximum number of CC executors to generate per organization
#const max_org_ccexecs = 3.
% Maximum number of ordering nodes to generate per organization
#const max_org_orderers = 2.
% Maximum number of intermediary nodes in endorsement policy tree
#const max_endorsement_policy_nodes = 10.
```

Listing A.5: Contents of `bindings.lp`

```
%%%%%%%%%%%%%%%%%%%%%%%%%
%   INPUT               %
%%%%%%%%%%%%%%%%%%%%%%%%%

input_failure_mode((ok, ok, ok)).

%------------------------------------------------------------------------

%%%%%%%%%%%%%%%%%%%%%%%%%
%   ORGANIZATIONS       %
%%%%%%%%%%%%%%%%%%%%%%%%%
% org(ID)

```

```prolog
orgs(3).
org(org1).
org(org2).
org(org3).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   HOSTS               %
%%%%%%%%%%%%%%%%%%%%%%%%%
% host(Org, ID)
% host_ifm(ID, IFM)
% host_link(HostA, HostB, State)

org_hosts(org1, 2).
%
host(org1, org1h1).
host_ifm(org1h1, ok).
%
host(org1, org1h2).
host_ifm(org1h2, ok).

org_hosts(org2, 2).
%
host(org2, org2h1).
host_ifm(org2h1, ok).
%
host(org2, org2h2).
host_ifm(org2h2, ok).

org_hosts(org3, 1).
%
host(org3, org3h1).
host_ifm(org3h1, down).

% Hosts form a complete mesh
host_link(HostA, HostB, up) :- host(_, HostA), host(_, HostB).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   CLIENT              %
%%%%%%%%%%%%%%%%%%%%%%%%%
% client_org(Org).
% client_alloc(HostID)
% client_orderer(OrdererID)
% client_pub(PeerID)
% client_sub(PeerID)

client_org(org2).
client_alloc(org1h2).
client_sub(org2p1).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   PEERS               %
%%%%%%%%%%%%%%%%%%%%%%%%%
% peer(Org, ID)
% peer_alloc(ID, HostID)
% peer_ifm(ID, IFM) → always OK

org_peers(org1, 2).
```

```
%
peer(org1, org1p1).
peer_alloc(org1p1, org1h1).
peer_ifm(org1p1, ok).
peer_ccexec(org1p1, org1cc1).
peer_ccexec(org1p1, org1cc2).
%
peer(org1, org1p2).
peer_alloc(org1p2, org1h2).
peer_ifm(org1p2, ok).
peer_ccexec(org1p2, org1cc3).
:- peer_ccexec(org1p2, org1cc2);

org_peers(org2, 1).
%
peer(org2, org2p1).
peer_alloc(org2p1, org2h1).
peer_ifm(org2p1, ok).
peer_ccexec(org2p1, org2cc1).

org_peers(org3, 1).
%
peer(org3, org3p1).
peer_alloc(org3p1, org3h1).
peer_ifm(org3p1, ok).
peer_ccexec(org3p1, org3cc1).



%%%%%%%%%%%%%%%%%%%%%%%%%
%   CC EXECUTORS        %
%%%%%%%%%%%%%%%%%%%%%%%%%
% ccexec(Org, ID)
% ccexec_alloc(ID, HostID)
% ccexec_ifm(ID, IFM) -> ok or subtle_fault or coarse_fault

org_ccexecs(org1, 3).
%
ccexec(org1, org1cc1).
ccexec_alloc(org1cc1, org1h1).
ccexec_ifm(org1cc1, ok).
%
ccexec(org1, org1cc2).
ccexec_alloc(org1cc2, org1h2).
ccexec_ifm(org1cc2, ok).
%
ccexec(org1, org1cc3).
ccexec_alloc(org1cc3, org1h1).
ccexec_ifm(org1cc3, ok).

org_ccexecs(org2, 1).
%
ccexec(org2, org2cc1).
ccexec_alloc(org2cc1, org2h1).
ccexec_ifm(org2cc1, ok).

org_ccexecs(org3, 1).
%
ccexec(org3, org3cc1).
ccexec_alloc(org3cc1, org3h1).
```

```
ccexec_ifm(org3cc1, ok).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ENDORSEMENT POLICY  %
%%%%%%%%%%%%%%%%%%%%%%%%%
% endorsement_policy_nodes(N)   ← excluding endorsement_node_0
% endorsement_policy(Node/endorsement_node_0, LogicalOperator, A, B)

% AND(OR(org1, org2), org3)  ⟺  (org1 v org2) ^ org3
endorsement_policy_nodes(1).
% endorsement_policy(endorsement_node_1, or, org1, org2).
% endorsement_policy(endorsement_node_0, and, endorsement_node_1, org3).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ENDORSEMENT SVCS    %
%%%%%%%%%%%%%%%%%%%%%%%%%

endorsement(org1, org1ESV).
endorsement(org2, org2ESV).
endorsement(org3, org3ESV).

% Endorsement links form a complete mesh
endorsement_link(ESV, Peer) :-
    endorsement(Org, ESV), peer(OtherOrg, Peer), Org != OtherOrg.


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ORDERERS            %
%%%%%%%%%%%%%%%%%%%%%%%%%
% orderer(Org, ID)
% orderer_alloc(ID, HostID)
% orderer_ifm(ID, IFM) → always OK

org_orderers(org1, 2).
%
orderer(org1, org1o1).
orderer_alloc(org1o1, org1h1).
orderer_ifm(org1o1, ok).
%
orderer(org1, org1o2).
orderer_alloc(org1o2, org1h2).
orderer_ifm(org1o2, ok).

org_orderers(org2, 1).
%
orderer(org2, org2o1).
orderer_alloc(org2o1, org2h1).
orderer_ifm(org2o1, ok).

org_orderers(org3, 0).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   ORDERING SVC        %
%%%%%%%%%%%%%%%%%%%%%%%%%
% ordering(Type) → solo, kafka, or raft
% [ if solo: ordering_leader(OrdererID) ]

```

```
ordering(raft).
% ordering_leader(org1o1).


%%%%%%%%%%%%%%%%%%%%%%%%%
%   BLOCK VALID SVCS    %
%%%%%%%%%%%%%%%%%%%%%%%%%
% blockvalidation_ifm(IFM) -> ok or conflict

blockvalidation_ifm(conflict).

%-----------------------------------------------------------------------

%%%%%%%%%%%%%%%%%%%%%%%%%
%   REFERENCE TX        %
%%%%%%%%%%%%%%%%%%%%%%%%%

:- ref((ok, ok, ok)).
```

Listing A.6: Contents of `top.lp`

```
#include "bindings.lp".
#include "constants.lp".
#include "model/behaviour.lp".
#include "model/failure.lp".
#include "model/structure.lp".
```