applied sciences

Article

Formalism-Driven Development: Concepts, Taxonomy, and Practice [†]

Yepeng Ding * and **Hiroyuki Sato**

Department of Electrical Engineering and Information Systems, The University of Tokyo, Tokyo 113-8654, Japan; schuko@satolab.itc.u-tokyo.ac.jp

* Correspondence: youhoutei@satolab.itc.u-tokyo.ac.jp

† This paper is an extended version of our paper published in Proceedings of the 26th International Conference on Engineering of Complex Computer Systems, Hiroshima, Japan, 26–30 March 2022.

Abstract: Formal methods are crucial in program specification and verification. Instead of building cases to test functionalities, formal methods specify functionalities as properties and mathematically prove them. Nevertheless, the applicability of formal methods is limited in most development processes due to the requirement of mathematical knowledge for developers. To promote the application of formal methods, we formulate formalism-driven development (FDD), which is an iterative and incremental development process that guides developers to adopt proper formal methods throughout the whole development lifespan. In FDD, system graphs, a variant of transition systems optimized for usability, are designed to model system structures and behaviors with representative properties. System graphs are built iteratively and incrementally via refinement. Properties of system graphs are specified in propositional and temporal logics and verified by model-checking techniques with interpretation over transition system. In addition, skeleton programs are generated based on system graphs and expose implementable interfaces for executing external algorithms and emitting observable effects. Furthermore, we present Seniz, a framework that practicalizes and automates FDD. In this paper, we explicate the concepts and taxonomy of FDD and discuss its practice.

Keywords: formal methods; software engineering; development process; program verification; transition system; temporal logic

check for updates

Citation: Ding, Y.; Sato, H. Formalism-Driven Development: Concepts, Taxonomy, and Practice. *Appl. Sci.* **2022**, *12*, 3415. https://doi.org/10.3390/app12073415

Academic Editor: Arcangelo Castiglione

Received: 18 February 2022
Accepted: 25 March 2022
Published: 27 March 2022

1. Introduction

Development processes are essential in software engineering, determining project workflows and approaches. Although most development processes can be generally called agile development nowadays, an architect still needs to customize processes for different projects. For user-centric system development, interface design could drive processes to enhance user experience in all aspects. For security-first system development, threat models could drive processes to satisfy security requirements. Tools vary in development processes. Apparently, prototyping tools are more useful for UI engineers than security engineers to weave interactions during user-centric system development. On the contrary, penetration testing tools are powerful for security engineers to find system vulnerabilities in security-first system development and may not be necessary to UI engineers. However, no matter what types of systems there are, one of the most important goals of the customized development processes is to implement requirements correctly. Conventionally, software testing techniques play a pivotal role in validating implementation correctness. Nevertheless, they are likely incapable of proving correctness with trade-offs for cost reduction, especially in nondeterministic cases.

Formal methods have proven effective in ensuring correctness by specifying and verifying system properties in a mathematically rigorous manner. Formal specification techniques are used to rigorously describe system behaviors and guide implementations with mathematical notations from various theories such as automata theory, mathematical logic, and type theory. Based on these theories, formal verification techniques are used to prove specified properties with respect to specifications. Compared to software testing techniques, formal methods are usually expensive and require more professional expertise and computing power. Nevertheless, many tools have been developed to advocate the use of formal methods in practice, such as model-checking tools [1,2] and theorem-proving tools [3,4]. These tools can be used to verify systems at the implementation (code) level or verify models extracted from implementations. In the first case, implementation verification focuses on the details of the execution, such as runtime bugs (e.g., null pointer, division by 0, buffer overflow), functional correctness bugs (e.g., undefined behaviors, unexpected algorithm output), and concurrency bugs (e.g., deadlock, race condition). It is crucial to verify implementations and even worthwhile analyzing bytecodes. However, the implementation verification is a unilateral strategy and hard to unravel design flaws. In the second case, it is possible to locate design flaws in a system by verifying the specified properties of models. However, extracting a proper model from the complicated implementation is nontrivial. Extracted models might be too simple to have useful properties or too complicated to be verified [5]. It is hard to judge whether a model extracted from an implementation coincides with its design. If there is a departure from the design, properties associated with that model can be untrustworthy and meaningless, which we call a conformity issue. Therefore, a methodology that elaborates the adoption of formal methods tools is imperative in development. Considering the importance of development processes that control the whole development lifespan, an intuitive method of using formal methods throughout development processes could be the key.

Formalism-driven development (FDD) is an iterative and incremental development process, which was originally proposed for developing provably correct decentralized systems [6]. To date, decentralization has evolved into a new stage with the advent of the blockchain technology [7]. The blockchain was first introduced as the underlying technology of a decentralized payment system named Bitcoin [8]. Later, it was extended by the smart contract [9] and generalized to a concept named distributed ledger technology (DLT) [10]. Based on DLT, numerous decentralized systems have been developed to address security and privacy issues in a wide range of fields such as Internet of Things (IoT) [11,12], data persistence [13,14], and security infrastructure [15,16]. However, developing a trustworthy decentralized system is extremely challenging due to threats and vulnerabilities. At the application level, the DAO attack [17] in 2016, one of the most infamous attacks, brought great damage to the cryptocurrency market and successfully transferred about USD 50M worth of Ether into the control of the attacker by exploiting the re-entrancy vulnerability. In 2017, the Parity wallet firstly suffered a breach causing about USD 40M stolen [18] in June, which was followed by a "suicide" attack in November that caused about USD 150M loss [19]. Furthermore, more kinds of smart contract vulnerabilities were reported in [18] such as re-entrancy and overflow. At the infrastructure level, Geth, the most widely used implementation of Ethereum virtual machine (EVM), was recently found to have a vulnerability (https://github.com/ethereum/go-ethereum/security/advisories/GHSA-9856-9gg9-qcmq (accessed on 15 March 2022)) that led to a minority chain split after the London hard fork.

In contexts such as the development of decentralized systems, especially blockchains, complete empirical testing is not practical due to the uncontrollable environment factors (e.g., gas exactimation in Ethereum) and expensive financial expenses (e.g., high transaction fee in current main public blockchains). The cost of sufficient simulation testing for correctness verification may be no different from a solution based on formal methods. FDD aims to tackle the issues hindering the development of trustworthy decentralized systems with reasonable cost from the perspective of a development process that we regard as the root cause. FDD introduces a life cycle dominated by different types of formal methods to produce rigorous designs, mathematically verifiable models, and provably correct implementations. Formal specification techniques are used for producing readable, visualizable, and rigorous designs of which the correctness can be verified syntactically and semantically. Based on designs, formal specification techniques are also used to generate models that guide verification and implementation. In FDD, the design and model basically have the same semantics except that designs are usually at a higher abstraction level than models. Hence, we use design, model, and design model indistinguishably in this paper. Properties are formulated and proved in propositional and temporal logics to describe system functionalities. Furthermore, FDD puts strict constraints on iterations and increments by refinement techniques. Incremented components are formally defined with formulas and can be expanded into fundamental elements for design, verification, and implementation. Instead of functionalities, a set of (weakly) verified properties defines an effective iteration. In this manner, FDD ensures conformity during the system evolution.

In this paper, motivated by generalizing and promoting the adoption of FDD in various types of development projects, we systematically present the foundations of FDD. Starting from the basic concepts in Section 2, we give a taxonomy of FDD in Section 3 that organizes and unifies theories, architectures, and methods to help researchers to improve the underlying theories and engineers to build FDD tools. Furthermore, we formulate criteria of FDD tools and introduce Seniz, the first FDD framework to illustrate FDD in practice in Section 4. Some limitations and misconceptions are discussed in Section 6. We conclude our work with future directions of FDD in Section 7.

2. Preliminaries

In this section, we introduce core concepts and theories related to FDD.

2.1. Basic Structures

Firstly, let us recall the Kripke structure, which is one of the commonly used models for formal specification.

Definition 1 (Kripke structure)**.** *Let P be a set of atomic propositions. A Kripke structure $\mathfrak{K}$ is a quintuple*

$$\mathfrak{K} \triangleq \langle S, R, I, P, \mathcal{L} \rangle$$

where

- *S is a set of states,*
- *$R \subseteq S \times S$ is a transition relation;*
- *$I \subseteq S$ is a set of initial states;*
- *P is a set of atomic propositions; and*
- *$\mathcal{L} : S \mapsto \wp(P)$ is a labeling function.*

Relation R is left-total, i.e., $\forall s \in S(\exists s' \in S : (s, s') \in R)$. An atomic proposition is an indecomposable proposition defined in propositional logic. The labeling function $\mathcal{L}$ relates a set $\mathcal{L}(s) \in \wp(P)$ of atomic propositions to any state s.

Notably, a Kripke structure is an unlabeled transition system. Therefore, it is preferred in state-based approaches for formal specification. For action-based approaches that assume only the executed actions are observable from outside, a labeled transition system plays a pivotal role.

In this paper, we define a variant of labeled transition system as below.

Definition 2 (Labeled Transition System)**.** *A labeled transition system $\mathfrak{T}$ over set Var of typed state variables is a tuple*

$$\mathfrak{T} \triangleq \langle S, A, \rightarrow, I, P, \mathcal{L} \rangle$$

where

- *$S = [\![Var]\!]$ is a set of states;*
- *A is a set of actions;*

- $\rightarrow \subseteq S \times A \times S$ *is a transition relation;*
- $I \subseteq S$ *is a set of initial states;*
- P *is a set of atomic propositions; and*
- $\mathcal{L} : S \mapsto \wp(P)$ *is a labeling function.*

The state space S is determined by $[\![Var]\!]$, which is the set of evaluations of state variables Var. State $s \in S$ is called a terminal state if it does not have any outgoing transitions, i.e., $\bigcup_{a \in A} \{s' \in S \mid s \xrightarrow{a} s'\} = \emptyset$. The notation $s \xrightarrow{a} s'$ is used as shorthand for $(s, a, s') \in \rightarrow$. In this paper, we assume that S, A, and P are finite sets.

In fact, a labeled transition system can be transformed into a Kripke structure and vice versa [20,21]. In this manner, we can formalize a system from either a state-based view or an action-based view according to concrete contexts.

In the remainder of this paper, we abbreviate *labeled transition system* to *transition system*.

Conditional branching is commonly used in modeling systems. By using conditional branching, it is possible to put constraints on actions. An action can only be triggered while the current evaluation of variables satisfies some conditions. We denote a set of Boolean conditions (propositional formulae) over *Var* as $\|Var\|$. In the interest of modeling conditional branching, we introduce conditional transitions.

Definition 3 (Conditional Transition)**.** *A transition system $\mathfrak{T}$ with conditional transitions over set Var of typed state variables is a tuple*

$$\mathfrak{T} \triangleq \langle S, A, \hookrightarrow, I, g_0, P, \mathcal{L} \rangle$$

according to Definition 2 with differences that

- $\hookrightarrow \subseteq S \times \|Var\| \times A \times S$ *is the conditional transition relation; and*
- $g_0 \in \|Var\|$ *is the initial guard (condition).*

For convenience, we use the notation $s \overset{g\downarrow a}{\hookrightarrow} s'$ as shorthand for $(s, g, a, s') \in \hookrightarrow$. If the guard is a tautology, we can omit it, i.e., $s \overset{a}{\hookrightarrow} s'$.

The behavior in state $s \in S$ depends on the current state variable evaluation $\mathcal{V} \in [\![Var]\!]$. The value of state variable $x \in Var$ is accessible through $\mathcal{V}(x)$. For transition $s \overset{g\downarrow a}{\hookrightarrow} s'$, the execution of action a is only triggered when evaluation $\mathcal{V}$ satisfies guard g, i.e., $\mathcal{V} \models g$. The new evaluation can be represented by changed state variables, e.g., $\mathcal{V}' = \mathcal{V}[x : v]$, meaning that state variable x has value v in $\mathcal{V}'$, and all other state variables are unaffected.

$$\mathcal{V}[x : v](x') = \begin{cases} \mathcal{V}(x') & x \neq x' \\ v & x = x'. \end{cases}$$

Given a transition system with conditional transitions, it is natural that it can be transformed into an equivalent transition system without conditional transitions.

Definition 4 (Semantics of Conditional Transitions)**.** *Let $\mathfrak{T} = \langle S, A, \hookrightarrow, I', g_0, P', \mathcal{L} \rangle$ be a transition system with conditional transitions over set Var of typed state variables. The corresponding transition system $\mathfrak{T}'$ without conditional transitions is the tuple $\langle S, A, \rightarrow, I, P, \mathcal{L} \rangle$ where*

- $\rightarrow$ *is defined by the following rule:*

$$\frac{s \overset{g\downarrow a}{\hookrightarrow} s' \wedge \mathcal{V} \models g}{s \xrightarrow{a} s'} \quad ;$$

- $I = \{\langle s, \mathcal{V} \rangle \mid s \in I', \mathcal{V} \models g_0\}$;
- $P = \|Var\| \cup P'$; *and*

- *S, A, and $\mathcal{L}$ remain the same.*

Remark 1 (State Tautology). *If $S = [\![Var]\!]$, the current state s and current state variable evaluation $\mathcal{V}$ are interchangeable. The tuple $\langle s, \mathcal{V}\rangle$ can be reduced to either s or $\mathcal{V}$.*

In this manner, we can define concepts on top of the transition system with conditional transitions on account of clearance.

2.2. Parallelism

Parallel systems can also be modeled by transition systems. In this paper, we introduce two common types of parallelism [22]: asynchronous concurrency (pure interleaving) and synchronous concurrency (variable sharing).

Asynchronous concurrency models a parallel system composed of a set of interleaving subsystems, of which the next global state is nondeterministic.

Definition 5 (Asynchronous Concurrency of Transition System). *Given two transition systems with conditional transitions $\mathfrak{T}_1 = \langle S_1, A_1, \hookrightarrow_1, I_1, g_{0,1}, P_1, L_1\rangle$ over Var_1 and $\mathfrak{T}_2 = \langle S_2, A_2, \hookrightarrow_2, I_2, g_{0,2}, P_2, L_2\rangle$ over Var_2, the asynchronous concurrency of them $\mathfrak{T}_1 \mid\mid\mid \mathfrak{T}_2$ is defined by:*

$$\mathfrak{T}_1 \mid\mid\mid \mathfrak{T}_2 \triangleq \langle S, A_1 \uplus A_2, \hookrightarrow, I_1 \times I_2, g_{0,1} \wedge g_{0,2}, P, \mathcal{L}\rangle$$

where

- *$S = [\![Var_1 \setminus \widehat{Var}]\!] \times [\![Var_2 \setminus \widehat{Var}]\!]$ where $\widehat{Var} = Var_1 \cap Var_2$;*
- *$\hookrightarrow$ is defined by the rules:*

$$\frac{s_1 \xhookrightarrow{g\downarrow a}_1 s_1'}{\langle s_1, s_2\rangle \xhookrightarrow{g\downarrow\langle a,*\rangle} \langle s_1', s_2\rangle} \qquad \frac{s_2 \xhookrightarrow{g\downarrow a}_2 s_2'}{\langle s_1, s_2\rangle \xhookrightarrow{g\downarrow\langle a,*\rangle} \langle s_1, s_2'\rangle}$$

- *$P \supseteq P_1 \uplus P_2$; and*
- *$\mathcal{L} : S \mapsto \wp(P)$.*

Synchronous concurrency models a parallel system whose subsystems share a global clock, where all subsystems make either a transition or an idle step on each clock pulse.

Definition 6 (Synchronous Concurrency of Transition System). *Given two transition systems with conditional transitions $\mathfrak{T}_1 = \langle S_1, A_1, \hookrightarrow_1, I_1, g_{0,1}, P_1, L_1\rangle$ over Var_1 and $\mathfrak{T}_2 = \langle S_2, A_2, \hookrightarrow_2, I_2, g_{0,2}, P_2, L_2\rangle$ over Var_2, the synchronous concurrency of them $\mathfrak{T}_1 \mid\mid \mathfrak{T}_2$ is defined by:*

$$\mathfrak{T}_1 \mid\mid \mathfrak{T}_2 \triangleq \langle S_1 \times S_2, A_1 \times A_2, \hookrightarrow, I_1 \times I_2, g_{0,1} \wedge g_{0,2}, P, \mathcal{L}\rangle$$

where

- *$\hookrightarrow$ is defined by the rule:*

$$\frac{s_1 \xhookrightarrow{g_1\downarrow a_1} s_1' \wedge s_2 \xhookrightarrow{g_2\downarrow a_2} s_2'}{\langle s_1, s_2\rangle \xhookrightarrow{g_1\wedge g_2\downarrow\langle a_1, a_2\rangle} \langle s_1', s_2'\rangle};$$

  and
- *P and $\mathcal{L}$ are defined as Definition 5.*

Remark 2 (Variable Evaluation Merging and Demerging). *Given two variable evaluations $\mathcal{V}_1, \mathcal{V}_2$ with $Dom(\mathcal{V}_1) \cap Dom(\mathcal{V}_2) = \emptyset$, merging operator $\oplus$ merges them into one variable evaluation $\mathcal{V}$ such that*

- *$Dom(\mathcal{V}) = Dom(\mathcal{V}_1) \cup Dom(\mathcal{V}_2)$; and*
- *$\forall x_i \in Dom(\mathcal{V}_i) : \mathcal{V}(x_i) = \mathcal{V}_i(x_i), i \in [1, 2]$.*

Given two variable evaluations $\mathcal{V}_1, \mathcal{V}_2$ with $Dom(\mathcal{V}_1) \supseteq Dom(\mathcal{V}_2)$, Demerging operator $\ominus$ demerges a variable evaluation from another. $\mathcal{V}_1 \ominus \mathcal{V}_2$ produces a variable evaluation $\mathcal{V}'$ such that

- $Dom(\mathcal{V}') = Dom(\mathcal{V}_1) \setminus Dom(\mathcal{V}_2)$; *and*
- $\forall x \in Dom(\mathcal{V}') : \mathcal{V}'(x) = \mathcal{V}_1(x)$.

Remark 3 (State Rewriting). *If $s = \langle \mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_n \rangle$ where $s \in S$ and $\bigcap_{i=1}^{n} Dom(\mathcal{V}_i) = \emptyset$, s can be rewritten as a merged variable evaluation $\mathcal{V} = \bigoplus_{i=1}^{n} \mathcal{V}_i$. Here, $\bigoplus$ notation is used to indicate repeated $\oplus$.*

2.3. Communication

To model distributed systems, a communication model is indispensable. In this paper, we model the communication by channels. A channel is a buffer based on a queue where messages are stored and held to be processed later.

Given channel c, we define a set of functions to access the properties of c. c has a finite capacity *Cap(c)* $\in \mathbb{N}$ and a domain $Dom(c)$. The current number of messages in c is fetched by *Len(c)*. We can manipulate contents of c by a set of operations. $Enq(c, m)$ puts message m at the rear of the buffer, whereas $Deq(c)$ pops an element from the front of the buffer.

We introduce two actions for sending and receiving messages based on the operations of c.

- $c!m$: send the message m along channel c, i.e., $Enq(c, m)$;
- $c?x$: receive a message via channel c and variable x has value of the message, i.e., $x : Deq(c)$.

With two message-passing actions, we define the set of communication actions *Com* as: $Com = \{c!m, c?x \mid c \in Chan, m \in Dom(c), x \in Var \text{ with } Dom(x) \supseteq Dom(c)\}$, where *Chan* is a set of channels with typical element c and *Var* is a set of variables as in Definition 2.

Definition 7 (Channel System). *A transition system with conditional transitions $\mathfrak{T}$ over $(Var, Chan)$ is a tuple*

$$\mathfrak{T} \triangleq \langle S, A, \hookrightarrow, I, g_0, P, \mathcal{L} \rangle$$

according to Definitions 2 and 3 with the only difference that $\hookrightarrow \subseteq S \times \|Var\| \times (A \cup Com) \times S$.

A channel system $\mathfrak{C}$ over $(Var, Chan), Var = \bigcup_{i=1}^{n} Var_i$ with $\bigcap_{i=1}^{n} Var_i = \emptyset$, consisting of transition systems $\mathfrak{T}_i$ over $(Var_i, Chan), i \in [1, n]$ is defined as

$$\mathfrak{C} \triangleq [\mathfrak{T}_1 \mid \dots \mid \mathfrak{T}_n].$$

Remark 4 (State Structure of a Channel System). *Let $\mathfrak{C} = [\mathfrak{T}_1 \mid \dots \mid \mathfrak{T}_n]$ be a channel system over $(Var, Chan)$. The global states S of $\mathfrak{C}$ are tuples of the form $\langle s_1, \dots, s_n, \mathcal{V}, \mathcal{C} \rangle$, where*

- *$s_i \in S_i$ is the current state (variable evaluation) of subsystem $\mathfrak{T}_i$, i.e., $s_i = \mathcal{V}_i \in [\![Var_i]\!]$;*
- *$\mathcal{V} = \bigoplus_{i=1}^{n} \mathcal{V}_i \in [\![Var]\!]$ is the current variable evaluation (state) of $\mathfrak{C}$, i.e., $\mathcal{V} = s \in S$; and*
- *$\mathcal{C} \in [\![Chan]\!]$ is the current channel evaluation.*

$\mathcal{C}$ is a mapping from channel $c \in Chan$ onto a sequence $\mathcal{C}(c) \in Dom(c)^*$ such that $Len(\mathcal{C}(c)) \leqslant Cap(c)$, e.g., $\mathcal{C}(c) = [v_1 v_2 \dots v_k]$ with $Cap(c) \geqslant k$. If $Cap(c) = 0$, c is a synchronous channel. Otherwise, c is an asynchronous channel.

Notably, a channel system can have a nested structure; i.e., a subsystem can also be a channel system or parallel system. If a channel system only contains one transition system with conditional transitions, it is merely a transition system with conditional transitions and channel extension. Therefore, a channel system is capable of describing the complex structures and behaviors of a distributed system.

Furthermore, given a channel system, there exists an equivalent transition system

Definition 8 (Transition System Semantics of a Channel System). *Let* $\mathfrak{C} = [\mathfrak{T}_i \mid \ldots \mid \mathfrak{T}_i]$ *be a channel system over* $(Var, Chan)$ *with*

$$\mathfrak{T}_i \triangleq \langle S, A, \hookrightarrow, I, g_0, P, \mathcal{L}\rangle, i \in [1, n].$$

The transition system $\mathfrak{T}(\mathfrak{C})$ *of channel system* $\mathfrak{C}$ *is the tuple*

$$\mathfrak{T}(\mathfrak{C}) \triangleq \langle S, A, \rightarrow, I, P, \mathcal{L}\rangle$$

where

- $S = (S_1 \times \cdots \times S_n) \times [\![Var]\!] \times [\![Chan]\!]$;
- $A = \biguplus_{i=1}^{n} A_i \uplus \{\tau\}$ *where* τ *is a distinguished symbol to represent all communication actions;*
- $I = \{\langle s_1, \ldots, s_n, \mathcal{V}, \mathcal{C}_0\rangle \mid \forall i \in [1, n] : s_i \in I_i \wedge \mathcal{V} \models g_{0,i}\}$;
- $P \supseteq \biguplus_{i=1}^{n} P_i s$;
- $\mathcal{L} : S \mapsto \wp(P)$; *and*
- $\rightarrow$ *is defined according to different cases below.*

Case 1 (Interleaving). *Given action* $a \in A_i$,

$$\frac{s_i \overset{g\downarrow a}{\hookrightarrow} s_i' \wedge \mathcal{V} \models g}{\langle s_1, \ldots, s_i, \ldots, s_n, \mathcal{V}, \mathcal{C}\rangle \xrightarrow{a} \langle s_1, \ldots, s_i', \ldots, s_n, \mathcal{V}', \mathcal{C}\rangle}$$

where $\mathcal{V}' = (\bigoplus_{j \in [1,n], j \neq i} \mathcal{V}_j) \oplus \mathcal{V}_i'$

Case 2 (Asynchronous Message Passing). *Let* $c \in Chan, Cap(c) > 0$ *be a channel.*

- *For* $c?x$:

$$\frac{s_i \overset{g\downarrow c?x}{\hookrightarrow} s_i' \wedge \mathcal{V} \models g \wedge Len(\mathcal{C}(c)) > 0 \wedge \mathcal{C}(c) = v_1 \ldots v_k}{\langle s_1, \ldots, s_i, \ldots, s_n, \mathcal{V}, \mathcal{C}\rangle \xrightarrow{\tau} \langle s_1, \ldots, s_i', \ldots, s_n, \mathcal{V}', \mathcal{C}'\rangle}$$

  where $\mathcal{V}' = \mathcal{V}[x : v_1]$ *and* $\mathcal{C}' = \mathcal{C}[c : v_2 \ldots v_k]$.
- *For* $c!v, v \in Dom(c)$:

$$\frac{s_i \overset{g\downarrow c!v}{\hookrightarrow} s_i' \wedge \mathcal{V} \models g \wedge Len(\mathcal{C}(c)) < Cap(c) \wedge \mathcal{C}(c) = v_1 \ldots v_k}{\langle s_1, \ldots, s_i, \ldots, s_n, \mathcal{V}, \mathcal{C}\rangle \xrightarrow{\tau} \langle s_1, \ldots, s_i', \ldots, s_n, \mathcal{V}, \mathcal{C}'\rangle}$$

  where $\mathcal{C}' = \mathcal{C}[c : v_1 v_2 \ldots v_k v]$.

Case 3 (Synchronous Message Passing). *Given channel* $c \in Chan, Cap(c) = 0$,

$$\frac{s_i \overset{g_1\downarrow c?x}{\hookrightarrow} s_i' \wedge \mathcal{V} \models g_1 \wedge s_j \overset{g_2\downarrow c!v}{\hookrightarrow} s_j' \wedge \mathcal{V} \models g_2 \wedge i \neq j}{\langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n, \mathcal{V}, \mathcal{C}\rangle \xrightarrow{\tau} \langle s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n, \mathcal{V}', \mathcal{C}\rangle}$$

where $\mathcal{V}' = \mathcal{V}[x : v]$.

The interpretation from a channel system to a transition system can be automated according to the transition system semantics of a channel system, which permits us to model a system on top of a channel system without considering the details of its underlying transition system. It is also flexible to use different models or their combinations according to concrete contexts.

2.4. Properties

One important reason to mathematically model a system is to facilitate specifying and studying its properties in a rigorous way. In our current work, we use temporal logic, a formalism par excellence for mathematically expressing properties about system behaviors. More concretely, we use propositional temporal logic, including linear-time and branching-time logic.

Let us recall syntaxes of linear temporal logic (LTL) and computation tree logic (CTL).

Definition 9 (Syntax of Linear Temporal Logic)**.** *The syntax of LTL formulae over a countable set P of atomic propositions is defined as follows:*

$$\varphi ::= \top \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \sqcup \varphi_2$$

where $p \in P$. $\bigcirc$ and $\sqcup$ are two basic temporal modalities denoting next and until, respectively. $\Diamond$ and $\Box$, denoting eventually and always, are derived from $\sqcup$ as follows:

$$\Diamond\varphi \triangleq \top \sqcup \varphi, \Box\varphi \triangleq \neg\Diamond\neg\varphi.$$

Definition 10 (Computation Tree Logic)**.** *The syntax of CTL state formulae over a countable set P of atomic propositions is defined as follows:*

$$\phi ::= \top \mid p \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \exists\varphi \mid \forall\varphi_2$$

where $p \in P$ and φ denotes a path formula. φ is formed according to the syntax below.

$$\varphi ::= \bigcirc\phi \mid \phi_1 \sqcup \phi_2$$

where ϕ, ϕ_1, and ϕ_2 are state formulae.

Notably, the expressiveness of LTL and CTL are mathematically incomparable. They need to be used according to concrete contexts. When specifying properties of a complex system, both are indispensable most of the time.

3. Formalism-Driven Development

Formalism-driven development (FDD) is an iterative and incremental development process promoting formal methods throughout the lifespan. It is devised to take advantage of formal methods to eliminate design ambiguity, prove model properties, verify and test implementation correctness, and ensure conformity among design, model, and implementation. The core idea is to elaborate transition system theory to bond all concepts from formal specification, verification, and testing.

In fact, the philosophy of iterative and incremental development process has been widely practiced in agile development [23]. Nevertheless, both iteration and increment are not formally defined in agile processes. Generally, iteration means enhancing systems progressively, while increment means delivering the system by pieces. However, it is hard to give a well-defined explanation about what an iteration or increment produces and relations between two iterations and relations between an iteration and an increment. In FDD, iteration and increment are defined based on a formalism with theory support, including modeling, refinement, and verification [24]. With these well-defined theories, iterations and increments can be rigorously managed and used to produce verifiable deliveries.

In FDD, an iteration formulates a design model, proves model properties, implements the model, verifies the model implementation, and integrates or delivers the milestone. An increment organizes subsystems together as a higher-level system. Concretely, an iteration contains four stages: abstraction, verification, implementation, and integration, which is shown in Figure 1. The *Abstraction Stage* produces system graphs as design models. In the *Verification Stage*, system graphs are verified by formal verification. The *Implementation Stage* only accepts verified models to generate skeleton programs and implement concrete functionalities. In the *Integration Stage*, system graphs are integrated into higher-level systems or delivered. Naturally, an increment comes from the *Integration Stage* and can also launch a set of new iterations.

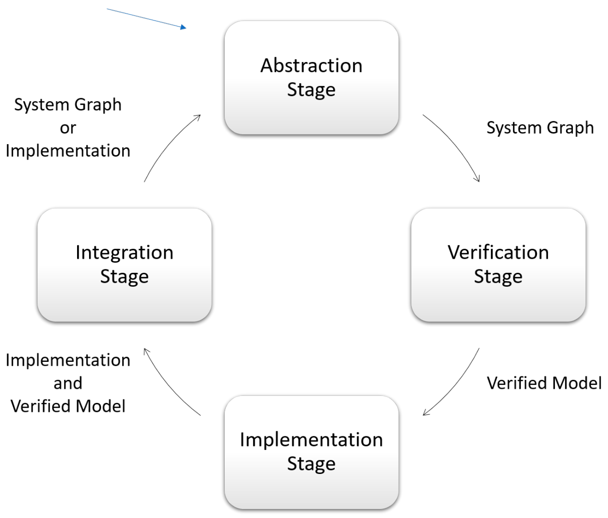


Figure 1. Stage transition graph of an iteration.

3.1. Abstraction Stage

In the *Abstraction Stage*, the goal is to produce a rigorous design model (system graph) for a system. If it is the first iteration of a new system, a model is built from the ground up, which is called the *Origin Stage*. Otherwise, we call it the *Refinement Stage* where a model from the last iteration is refined.

3.1.1. Origin Stage

The *Origin Stage* creates a system graph as a design model. A system graph is built on top of a channel system defined in Definition 7. Although they have equivalent expressiveness, a system graph cuts down the details that describe individual states by using a naming function to describe a group of states. Individual states are inferred from concrete contexts. This keeps a system graph succinct to model complex systems such as decentralized systems.

Definition 11 (System Graph). *A system graph $\mathfrak{S}$ over $(Var, Chan)$ is a tuple*

$$\mathfrak{S} \triangleq \langle D, \mathcal{N}, A, \hookrightarrow, i, g_0, F, P, \mathcal{L} \rangle$$

where

- $D = N \times [\![\widehat{Var}]\!], \widehat{Var} \subseteq Var$ *is a set of state declarators with names in* N;
- $\mathcal{N} : D \mapsto \wp([\![Var]\!])$ *is a naming function;*
- $A \supseteq Com$ *is a set of actions;*
- $\hookrightarrow \subseteq D \times \|Var\| \times A \times D$ *is the conditional transition relation;*
- $i \in D$ *is the initial state declarator;*
- $g_0 \in \|Var\|$ *is the initial guard;*
- $F \subseteq D$ *is a set of terminal state declarators;*
- $P \supseteq \|Var\|$ *is a set of propositions; and*
- $\mathcal{L} : [\![Var]\!] \mapsto \wp(P)$ *is a labeling function.*

Notably, a system graph uses state declarators to describe state sets and infer individual states instead of identifying each state explicitly. A state declarator $d \in D$ *introduces a kind of state with a given name into a system by identifying interesting state variables that are essential to show features of this kind of state. The name of a state declarator is unique, i.e.,*

$$\forall \langle n, \widehat{\mathcal{V}} \rangle \in D (\nexists \langle n', \widehat{\mathcal{V}}' \rangle \in D : \widehat{\mathcal{V}} = \widehat{\mathcal{V}}' \wedge n \neq n').$$

The naming function $\mathcal{N}$ relates a set $\mathcal{N}(d) \in \wp(\llbracket Var \rrbracket)$ of variable evaluations, i.e., states, to any state declarator $d = \langle n, \widehat{\mathcal{V}} \rangle$ such that

- $\forall \mathcal{V} \in \llbracket Var \rrbracket (\exists ! \widehat{\mathcal{V}} \in \llbracket \widehat{Var} \rrbracket : \mathcal{V} \in \mathcal{N}(d))$*; and*
- $\forall x \in Dom(\widehat{\mathcal{V}}) : \widehat{\mathcal{V}}(x) = \mathcal{V}(x), \mathcal{V} \in \llbracket Var \rrbracket, Dom(\widehat{\mathcal{V}}) \subseteq Dom(\mathcal{V})$.

The conditional transition relation is on top of state declarators. Only one initial state declarator exists in a system graph. A system graph is nonterminal if $F = \emptyset$. Propositions are well-formed propositional formulae in propositional logic and constructed from atomic propositions by logical connectives. A set of propositions are related to any variable evaluation, i.e., state, by the labeling function $\mathcal{L}$.

By using the state declarator, it enables describing a system in a succinct form. We are only concerned about the most critical features of states identified by interesting state variables. The evaluation of other state variables is inferred from the preceding state.

Example 1 (Transaction Client)**.** *We use a simplified transaction client in our developed demonstration as an example to illustrate core concepts in this paper. The complete demonstration fully developed by FDD is a prototype of Ethereum including the client-side and server-side systems.*

A visualized system graph $\mathfrak{S}_{tx}$ of the simplified transaction client is shown in Figure 2.

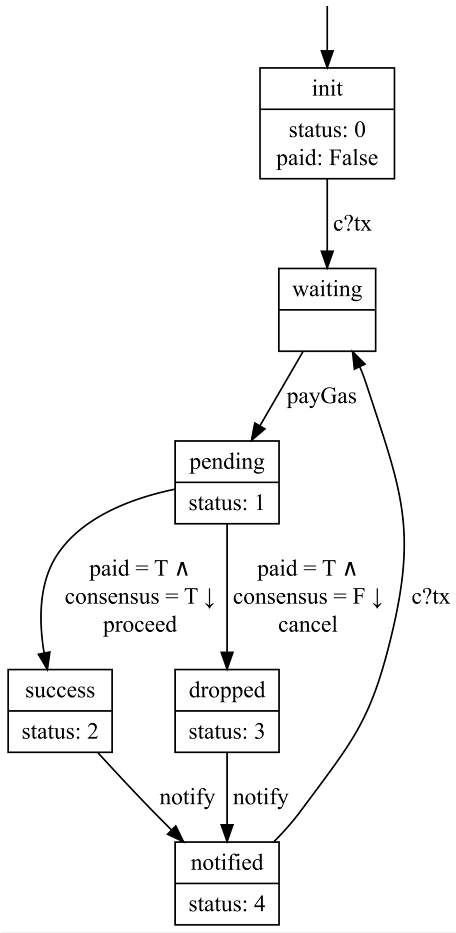


Figure 2. Visualized system graph of the transaction client in Example 1.

Each box is a state declarator consisting of two parts. The big box is also an increment (illustrated in Section 3.4). The above part is a name, while the below part is a set of state variable evaluations. State variable status has type Integer and paid has type Boolean. A one-way arrow pointing from a state declarator to another is a transition relation with a guard and an action. The *initial state declarator is pointed by an arrow without the starting node. We omit the representation of the guard if it is a tautology, e.g., the initial guard.*

Definition 12 (State Inference). *Let $\mathfrak{S}$ be a system graph over $(Var, Chan)$. The state space S of $\mathfrak{S}$ is determined by $[\![\widehat{Var}]\!] \times [\![Var \setminus \widehat{Var}]\!]$.*

Let $\mathcal{V} \in [\![Var]\!]$ be the current variable evaluation. According to Remark 3 and Definition 3, the succeeding state $s' \in S$ named by a state declarator $d' = \langle n', \widehat{\mathcal{V}}' \rangle \in D$ is represented as a variable evaluation $\mathcal{V}' = \langle \widehat{\mathcal{V}}', \mathcal{V} \ominus \widehat{\mathcal{V}}' \rangle = \mathcal{V}[\widehat{\mathcal{V}}']$ such that

$$\forall x \in Var : \mathcal{V}[\widehat{\mathcal{V}}'](x) = \begin{cases} \widehat{\mathcal{V}}'(x) & x \in Dom(\widehat{\mathcal{V}}') \\ \mathcal{V}(x) & x \notin Dom(\widehat{\mathcal{V}}'). \end{cases}$$

The initial state s_0 named by the initial state declarator i is represented as a variable evaluation $\mathcal{V}_0 = \mathcal{V}[i]$ such that

$$\forall x \in Var : \mathcal{V}[i](x) = \begin{cases} i(x) & x \in Dom(i) \\ \epsilon & x \notin Dom(i) \end{cases}$$

where ϵ denotes the default value.

Example 2 (State Inference in Transaction Client). *In Example 1, the state declarator named waiting does not identify any interesting state variables. According to Definition 12, the evaluation of state variables status and paid for the state declarator waiting remains the same with init; i.e., status has a value of* 0 *and paid has a value of False. However, waiting is distinguished from init by a hidden state variable tx. The state declarator waiting implies that the state variable tx has the value of the first element in channel c.*

Remark 5 (Transition Interpretation). *In Definition 12, states of $\mathfrak{S}$ are inferred from contexts. Correspondingly, the declarator-based conditional transition relation $\hookrightarrow$ of $\mathfrak{S}$ is interpreted to a state-based conditional transition relation by the following rule:*

$$\frac{d \overset{g\downarrow a}{\hookrightarrow} d' \wedge \mathcal{V} \models g}{\langle \widehat{\mathcal{V}}, \mathcal{V} \ominus \widehat{\mathcal{V}}' \rangle \overset{g\downarrow a}{\hookrightarrow} \langle \widehat{\mathcal{V}}', \mathcal{V}' \ominus \widehat{\mathcal{V}}' \rangle.}$$

Remark 6 (Transition System Semantics of a System Graph). *Let S over $(Var, Chan)$ be a system graph $\langle D, \mathcal{N}, A, \hookrightarrow, i, g_0, F, P, \mathcal{L} \rangle$. According to Definitions 11 and 12, and Remark 5, $\mathfrak{S}$ can be transformed into a channel system $\mathfrak{C}$ with the requirement that*

- *$\forall p \in P : Atom(p)$ are contained in the atomic proposition set of $\mathfrak{C}$;*
- *$\forall s \in [\![Var]\!] : \forall p \in \mathcal{L}(s) : Atom(p)$ are related to s through the labeling function of $\mathfrak{C}$;*

where $Atom(p)$ denotes all atomic propositions contained in the conjunctive normal form of p.

By the transition system semantics of a channel system defined in Definition 8, $\mathfrak{C}$ can be interpreted over a transition system.

Notably, the termination of a system graph does not imply the termination of its underlying transition system and vice versa. F is omitted during the interpretation.

By interpreting a system graph over a transition system, we can use the high-level design model, system graph, to model systems while safely using transition system theory to support prominent features of FDD in later stages and iterations.

3.1.2. Refinement Stage

The *Refinement Stage* accepts a system graph from the last iteration and produces a more detailed system graph while preserving and extending properties. According to Remark 6, a system graph can be interpreted onto a transition system. In FDD, we use both bisimulation and simulation theory [25,26] to support the refining process. The original purposes of these techniques are generally to optimize the verification process and improve verification efficiency by compacting a model while preserving its properties. However, the *Refinement Stage* inverses the original purpose to extend a small model into a big one while preserving its properties.

A refined system graph (refinement) $\mathfrak{S}'$ of $\mathfrak{S}$ is a more detailed design model that has either a strong relation $\sim$ or a weak relation $\preceq$ to $\mathfrak{S}$. Relation $\sim$ is an equivalence relation that identifies $\mathfrak{S}$ and its refinement with the same branching structure by bisimulation. Relation $\preceq$ is a preorder. $\mathfrak{S}' \preceq \mathfrak{S}$ holds if the refinement $\mathfrak{S}'$ can be simulated by $\mathfrak{S}$.

Definition 13 (Bisimulation). *Let $\mathfrak{S}_1$ and $\mathfrak{S}_2$ be two system graphs. With regard to their transformed transition systems $\mathfrak{T}_1 = \langle S_1, A_1, \rightarrow_1, I_1, P_1, \mathcal{L}_1 \rangle$ over Var_1 and $\mathfrak{T}_2 = \langle S_2, A_2, \rightarrow_2, I_2, P_2, \mathcal{L}_2 \rangle$ over Var_2, a bisimulation for $(\mathfrak{T}_1, \mathfrak{T}_2)$ is a binary relation $R \subseteq S_1 \times S_2$ such that*

- $\forall s_i \in I_i (\exists s_j \in I_j : (s_i, s_j) \in R), i, j \in [1,2], i \neq j,$
- $\forall (s_1, s_2) \in R :$
  - $\mathcal{L}_1(s_1) = \mathcal{L}_2(s_2);$
  - $s_i' \in Succ(s_i) \implies \exists s_j' \in Succ(s_j), i, j \in [1,2], i \neq j, (s_i', s_j') \in R;$

where $Succ(s) = \{s' \in S \mid s \xrightarrow{a} s', a \in A\}$.

If there exists a bisimulation R for $(\mathfrak{T}_1, \mathfrak{T}_2)$, then $\mathfrak{S}_1 \sim \mathfrak{S}_2 \iff \mathfrak{T}_1 \sim \mathfrak{T}_2$.

Definition 14 (Simulation). *Let $\mathfrak{S}_1$ and $\mathfrak{S}_2$ be two system graphs. With regard to their transformed transition systems $\mathfrak{T}_1 = \langle S_1, A_1, \rightarrow_1, I_1, P_1, \mathcal{L}_1 \rangle$ over Var_1 and $\mathfrak{T}_2 = \langle S_2, A_2, \rightarrow_2, I_2, P_2, \mathcal{L}_2 \rangle$ over Var_2, a simulation for $(\mathfrak{T}_1, \mathfrak{T}_2)$ is a binary relation $R \subseteq S_1 \times S_2$ such that*

- $\forall s_i \in I_i (\exists s_j \in I_j : (s_i, s_j) \in R), i, j \in [1,2], i \neq j;$
- $\forall (s_1, s_2) \in R :$
  - $\mathcal{L}_1(s_1) = \mathcal{L}_2(s_2),$
  - $s_1' \in Succ(s_1) \implies \exists s_2' \in Succ(s_2), (s_1', s_2') \in R.$

If there exists a simulation R for $(\mathfrak{T}_1, \mathfrak{T}_2)$, then $\mathfrak{S}_1 \preceq \mathfrak{S}_2 \iff \mathfrak{T}_1 \preceq \mathfrak{T}_2$.

Example 3 (Refinement in Transaction Client). *First, we assume that the system graph in Example 1 from the last iteration is the input of the current iteration. According to the gas mechanism of Ethereum, it may consume significant time for a network to reach consensus. Hence, it is helpful to provide accelerating service for the transaction client. Notably, the transaction client needs to cancel the original transaction firstly and resend a new transaction with more gas due to the immutability of the blockchain. Consequently, the current transaction is still dropped by the network.*

We can get a refined system graph $\mathfrak{S}_{tx}'$ visualized in Figure 3.

In fact, $\mathfrak{S}_{tx}$ and $\mathfrak{S}_{tx}'$ are bisimulation-equivalent, which can be proved automatically by bisimulation.

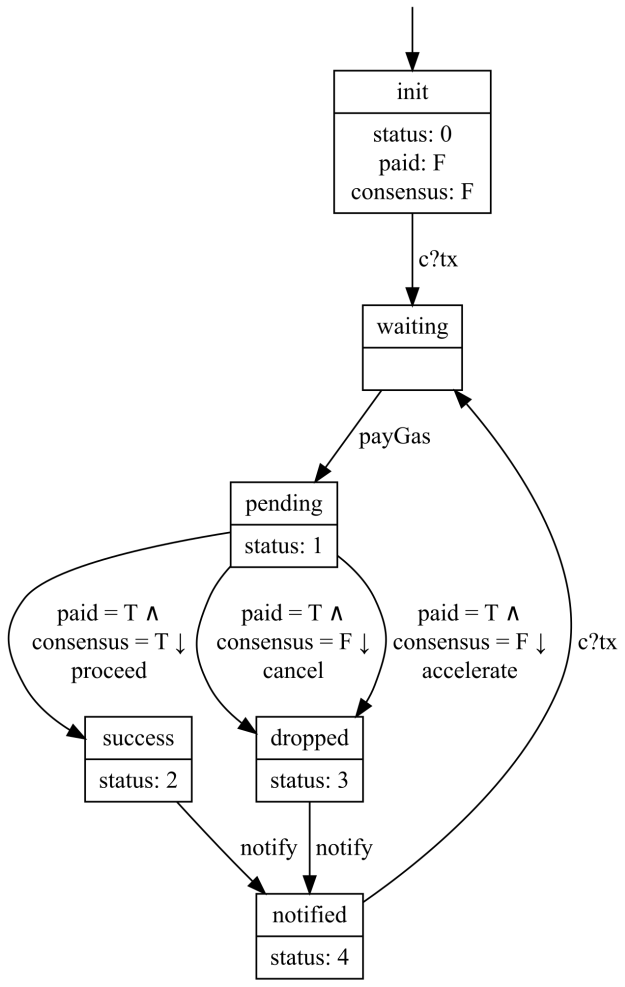


Figure 3. Visualized system graph of a refined transaction client in Example 3.

Theorem 1 (Property Preservation)**.** *Let $\mathfrak{S}_1$ and $\mathfrak{S}_2$ be two system graphs. If $\mathfrak{S}_1 \sim \mathfrak{S}_2$, then*

- $\forall \phi \in$ *LTL formulae* : $\mathfrak{S}_1 \models \phi \iff \mathfrak{S}_2 \models \phi$*; and*
- $\forall \phi \in$ *CTL formulae* : $\mathfrak{S}_1 \models \phi \iff \mathfrak{S}_2 \models \phi$.

  If $\mathfrak{S}_1 \preceq \mathfrak{S}_2$, then

- $\forall \phi \in$ *LTL formulae* : $\mathfrak{S}_2 \models \phi \implies \mathfrak{S}_1 \models \phi$*; and*
- $\forall \phi \in$ *CTL formulae* : $\mathfrak{S}_2 \models \phi \implies \mathfrak{S}_1 \models \phi$.

Example 4 (Property Preservation of Refined Transaction Client)**.** *According to Theorem 1, the refined system graph $\mathfrak{S}'_{tx}$ in Example 3 preserves all properties of $\mathfrak{S}_{tx}$ in Example 1 specified in LTL and CTL.*

For instance, the linear-time property of $\mathfrak{S}_{tx}$ in Example 5 also holds for $\mathfrak{S}'_{tx}$.

By using bisimulation and simulation techniques, the properties of an original system graph can be well preserved in the next iteration if the refined system graph passes either bisimilarity or similarity verification. While encountering a violation, it allows flexible handling methods. If the properties of the original system graph are finalized, then the refinement needs to be modified until passing the verification. It is also a solution to delegate the violation to the verification stage and resolve it by optimizing old properties.

3.2. Verification Stage

The *Verification Stage* produces a verifiable model based on the input system graph by specifying the admissible behaviors of the system graph as properties. In addition, it verifies properties associated with the verified model by formal verification.

3.2.1. Specification

According to Definition 11, a system graph has a set of propositions. Based on these propositions, we can specify essential system behaviors as a set of properties with temporal logic such as linear temporal logic and computation tree logic.

Example 5 (Linear-Time Property in Transaction Client)**.** *To verify whether the transaction client in Example 1 eventually gets notified whenever the gas fee is paid, we firstly define propositions* $Notified \triangleq status = 4$ *and* $PaidGas \triangleq status = 1$ *with respect to system graph* $\mathfrak{S}_{tx}$.

Then, we can formally specify the property as $\Box(PaidGas \rightarrow \Diamond Notified)$. *By the labeling function* $\mathcal{L}_{tx}$ *of* $\mathfrak{S}_{tx}$, *the states are automatically labeled with corresponding propositions. In this manner, the satisfiability of the property can be verified on the transition system under* $\mathfrak{S}_{tx}$.

3.2.2. Enforcement

The enforcement of verification processes depends on the verification mode of FDD. Either a model checker or a theorem prover can be used to prove properties formulated in a formal logic.

Checker Mode

By structuring a system graph and interpreting it over a transition system, it is trivial to enforce model checking to verify the properties.

Prover Mode

The properties of a system graph are verified by a theorem prover that mechanizes the logic used to specify these properties.

3.3. Implementation Stage

In the *Implementation Stage*, a skeleton program is generated from the verifiable model. Based on the skeleton program, a real-world program is implemented with full functionality. In addition, a formal verification and testing process is enforced to ensure implementation correctness.

A skeleton program generated from the verifiable model has strict constraints to ensure conformity between the model and implementation at best efforts. The smallest skeleton program includes

1. Predefined and immutable state variables;
2. Predefined and immutable deterministic control flow; and
3. Predefined and overridable action effects.

The term **predefined** means that the modificand is generated before the manual implementation. Something that can only be accessed but cannot be changed by implementors is **immutable**. An **action effect** of action a is the actual functionality produced every time executing a in the control flow. By overriding an action effect with an effect function, implementors can implement specific functionalities such as executing an algorithm, interacting with an I/O stream, etc.

Example 6 (Action Effect)**.** *In Example 1, all actions produce corresponding action effects such as* $c?tx$, *payGas, proceed, and notify. Each of them is overridable. For instance, a logging function can be called in each of them to write the current timestamp and action name into a local file system.*

Depending on the programming paradigm, generated skeleton programs are different on the code level. In this paper, we illustrate possible generation methods with respect to two mainstream programming paradigms and key points.

3.3.1. General Skeleton

Object-Oriented Programming

The typical features of a system graph are extracted and formed as an abstract class $\mathbf{A}_{sys}$ that defines protected methods associated with the control flow and exposes an entry point to execute the system.

For a system graph $\mathfrak{S} = \langle D, \mathcal{N}, A, \hookrightarrow, i, g_0, P, \mathcal{L} \rangle$ over $(Var, Chan)$, it contains all the information to create an abstract class $\mathbf{A}_{sg}$ that is capable of fully describing $\mathfrak{S}$. $\mathbf{A}_{sg}$ inherits $\mathbf{A}_{sys}$ and implements at least interface $\mathbf{I}_{act}$ that contains a set of method signatures extracted from all manually labeled actions in A. $\mathbf{A}_{sg}$ also overrides the control flow according to $D, \hookrightarrow, i, g_0$. $\mathbf{A}_{sg}$ together with all its associated classes forms the smallest class set (skeleton program) to describe $\mathfrak{S}$. The skeleton program is encapsulated into a package as a software development kit (SDK).

With such an SDK, implementors can create a concrete class $\mathbf{C}$ that inherits $\mathbf{A}_{sg}$. The implementors can override the effect methods (methods declared in $\mathbf{A}_{sg}$) to implement the functionality. Notably, implementors can neither modify state variables nor change the deterministic control flow. In this manner, the verified properties in *Verification Stage* are preserved in the executable system.

Functional Programming

In fact, it is straightforward to construct a system graph in functional programming. Related definitions can be easily formulated with customized data types. The impure action effects are isolated by the monad. All components are packaged into a module as a library that exposes a set of functions taking effect functions as their parameters and the entry point. The implementors can implement functionalities by passing the implementation of effect functions into the exposed functions.

3.3.2. Termination

A system graph is terminated if $F \neq \emptyset$. The execution naturally terminates while reaching a terminal state declarator or a terminal state of its underlying transition system defined in Definition 2. Without considering exception handling, the execution generated from a nonterminal system graph interpreted over a nonterminal transition system will never naturally terminate such as the transaction client in Example 1.

3.3.3. Parallelism

In Section 2.2, we present two types of parallelism: pure interleaving and variable sharing. Both present nondeterminism during the actual execution. The skeleton program handles them by multithreading techniques. Each system graph is encapsulated into a thread. In this manner, the implementation of nondeterminism in a parallel system is delegated to nondeterminism in thread scheduling.

3.3.4. Divergence and Confluence

A system graph may contain nondeterministic transitions after being interpreted over a transition system with conditional transitions. For a state s with a set $\hookrightarrow^{s}_{out}$ of outgoing transitions, if there exist at least two transitions $s \xrightarrow{g \downarrow a} s_1, s \xrightarrow{g' \downarrow a'} s_2 \in \hookrightarrow^{s}_{out}$ where $g \implies g' \wedge a \neq a' \wedge s_1 \neq s_2$, then it is a nondeterministic choice, which is called a divergence.

Example 7 (Divergence in Transaction Client)**.** *In our refined transaction client (Example 3), two transition relations from pending to dropped form a divergence because the truth values of their guards are the same.*

To resolve a divergence, an **interactive event** is emitted to wait for a signal that determines a choice to resume the execution in that branch. An interactive event can be user input via I/O stream, in-memory or on-disk interaction with another program, communication through a network protocol, etc. The skeleton program exposes all divergences in the form of interfaces that need to be implemented as interactive events by implementors. While encountering a divergence, the execution pauses until getting a signal from the interactive event to proceed.

Example 8 (Divergence Resolution for Transaction Client)**.** *To resolve the divergence in Example 7, we can use a keyboard event as the interactive event by implementing a keyboard listener in a local environment for testing. For instance, the effect of action cancel is triggered while getting an input sequence* $c\backslash r\backslash n$.

In our demonstration, the transaction client is developed as a mobile application. Action effects are triggered by touching corresponding buttons in the UI.

Confluence is usually not an interesting problem because the state inference in Definition 12 eliminates the nondeterminism of implicit state variables during the execution. One exception is for a set of systems to be confluent in a parallel system that contains nondeterminism in implementation. If a parallel system has terminal states, then we say this parallel system is naturally confluent. Each terminal state is a confluence where nondeterminism is eliminated. For a parallel system without terminal states, it allows implementors to customize the confluence where all threads join by manually identifying the evaluation of state variables in that confluence.

3.3.5. Channel

According to Remark 4, a channel can be either synchronous or asynchronous. A synchronous channel usually serves synchronization purposes instead of data transfer within a system modeled by a system graph. Its data structure at least contains the metadata. For an asynchronous channel, it contains at least the metadata, a buffer, and a set of operations associated with the buffer.

Regarding the implementation, a channel has two types: internal channel and external channel. An **internal channel** only receives messages within the system while an **external channel** can also receive messages from the outside of the system. An internal channel is naturally embedded into the control flow, while an external channel requires interaction with processes outside of the system. An outside process can be a program that sends messages to channels (by in-memory or on-disk interactions), a user who can input messages to channels (by I/O stream), a network protocol that passes messages to channels (by port), etc. A skeleton program integrates built-in modules to support the implementation of external channels according to concrete requirements.

Notably, execution needs to take care of waiting for a channel. An internal channel c gets into **waiting** if c is synchronous and the sending system is not in the state right before sending a message or c is asynchronous and $Cap(c) < 1$. If c is an external channel, it gets into waiting if no outside process sends a message to c.

Example 9 (Channel in Transaction Client)**.** *The transaction client in Example 1 has an anonymous action* $c?tx$ *attached to the transition relation between init and waiting. This communication action will not proceed, i.e., channel c gets into waiting until it consumes a message via channel c. From the perspective of implementation, the execution only resumes when state variable tx has the value received from channel c.*

In our implementation, information about a new transaction is pushed into channel c when that transaction is submitted.

3.3.6. Correctness Verification

After the functionality implementation, a correctness verification process is optionally enforced on fully implemented codes through formal verification and testing techniques. This optional process focuses on general verification items such as type safety and memory safety. Therefore, language-specific verification tools are needed to ensure implementation correctness by leveraging verification cost, which is out of the scope of FDD.

3.4. *Integration Stage*

The *Integration Stage* serves the overall bottom–up approach that embeds or integrates the input system graph into a higher-level system graph, which is an incremental process. This stage also determines the next move to continue the iteration or produce a delivery.

3.4.1. Increment

An increment has two types: horizontal increment and vertical increment. A **horizontal increment** is to embed a system graph into another one. Formally, embedding is defined as follows.

Definition 15 (Embedding)**.** *Let $\mathfrak{S}_i = \langle D_i, \mathcal{N}_i, A_i, \hookrightarrow_i, i_i, g_{0,i}, F_i, P_i, \mathcal{L}_i \rangle, i \in [1,2]$ be two system graphs over $(Var, Chan)$. System graph $\mathfrak{S}$ of embedding $\mathfrak{S}_1$ into $\mathfrak{S}_2$ in the place of state declarator $d_2 \in D_2$ is the tuple*

$$\langle D, \mathcal{N}, A, \hookrightarrow, i, g_0, F, P, \mathcal{L} \rangle$$

where

- $D = D_1 \uplus D_2$, $A = A_1 \uplus A_2$, $P = P_1 \uplus P_2$;
- $\forall d \in D_i : \mathcal{N}(d) = \mathcal{N}_i(d)$;
- $\hookrightarrow = \hookrightarrow_1 \uplus \hookrightarrow_2 \setminus \hookrightarrow_{d_2} \uplus \hookrightarrow'_{d_2}$;
- $i = \begin{cases} i_1 & d_2 = i_2 \\ i_2 & d_2 \neq i_2 \end{cases}, g_0 = \begin{cases} g_{0,1} & d_2 = i_2 \\ g_{0,2} & d_2 \neq i_2 \end{cases};$
- $F = \begin{cases} F_2 \setminus d_2 \uplus F_1 & d_2 \in F_2 \\ F_2 & d_2 \notin F_2 \end{cases};$
- $\forall s \in S_i : \mathcal{L}(s) = \mathcal{L}_i(s)$.

$\hookrightarrow_{d_2} \in \hookrightarrow_2$ is a set of transition relations such that

1. $\forall (d_2, g, a, d'_2) \in \hookrightarrow_2 : d_2 \xhookrightarrow{g\downarrow a} d'_2 \in \hookrightarrow_{d_2}$; *and*
2. $\forall (d'_2, g, a, d_2) \in \hookrightarrow_2 : d'_2 \xhookrightarrow{g\downarrow a} d_2 \in \hookrightarrow_{d_2}$.

$\hookrightarrow'_{d_2}$ is a set of transition relations such that

1. $\forall (d, g, a, d_2) \in \hookrightarrow_2 : d \xhookrightarrow{g_{0,1}\downarrow a} i_1 \in \hookrightarrow'_{d_2}$; *and*
2. $\forall f \in F_1 (\forall (d_2, g, a, d) \in \hookrightarrow_2 : f \xhookrightarrow{g\downarrow a} d)$.

Remark 7 (Module)**.** *In Definition 15, if $\mathfrak{S}_1$ shares state declarators, actions, propositions, naming, and labeling functions with $\mathfrak{S}_2$, i.e., $D_1 \subseteq D_2, A_1 \subseteq A_2, P_1 \subseteq P_2, \forall d \in D_1 : \mathcal{N}_1(d) = \mathcal{N}_2(d), \forall s \in S_1 : \mathcal{L}_1(s) = \mathcal{L}_2(s)$, then $\mathfrak{S}_1$ is a module of $\mathfrak{S}_2$.*

While embedding $\mathfrak{S}_1$ into $\mathfrak{S}_2$, if $\mathfrak{S}_1$ is a module of $\mathfrak{S}_2$, $\uplus$ relation is changed to $\cup$.

A **vertical increment** is an integration of a system graph into another one through parallelisms or communications, i.e., the current system graph is regarded as a subsystem that is parallel with or communicates with other subsystems in a higher-level system.

Example 10 (Increment in Transaction Client)**.** *In fact, system graph $\mathfrak{S}_{tx}$ of the transaction client in Example 1 is a considerably high-level model. Each component encapsulates either a* *horizontal increment or a vertical increment. So does an increment. For a simple but clear instance, pending integrates a system graph depicted in Figure 4 where the rounded dashed box denotes the terminal state declarator.*

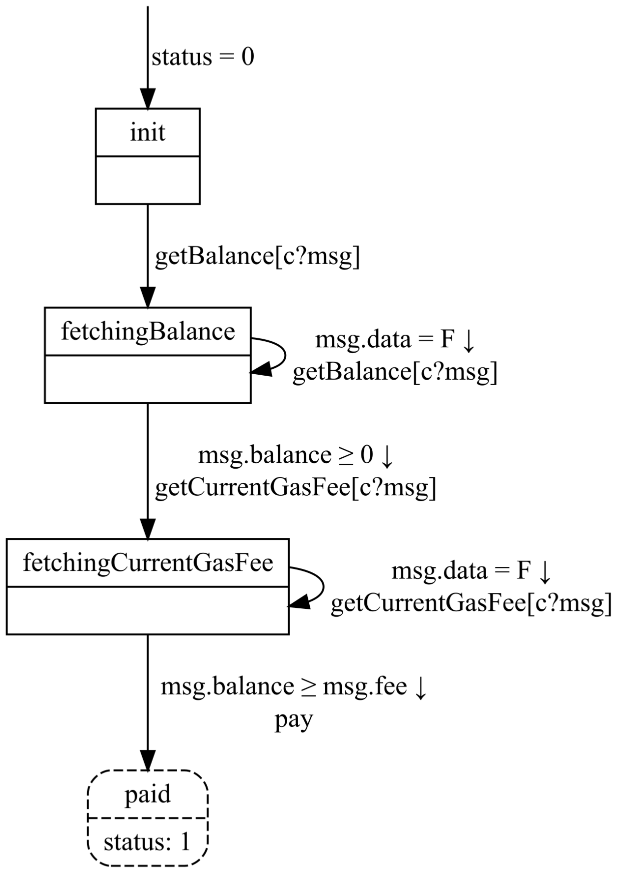


Figure 4. A visualized increment integrated in the system graph of the transaction client in Example 10.

Clearly, this increment is a module of $\mathfrak{S}_{tx}$*. According to Remark 7, we can visualize modularized* $\mathfrak{S}_{tx}$ *in Figure 5.*

3.4.2. Next Move

As the final stage of an iteration, the *Integration Stage* determines the next move according to the current system graph $\mathfrak{S}$. If $\mathfrak{S}$ does not include all details in the design, then it is called **refinable**. Otherwise, it is **unrefinable**. If $\mathfrak{S}$ is not integrated into any other system graph, then it is called **independent**. Otherwise, it is **dependent**.

- If $\mathfrak{S}$ is independent and unrefinable, then terminate its iterative and incremental process and deliver its implementation.
- If $\mathfrak{S}$ is dependent and unrefinable, then integrates $\mathfrak{S}$ into a higher-level system graph $\mathfrak{S}'$, and start an iterative process of $\mathfrak{S}'$.
- If $\mathfrak{S}$ is refinable, always go to the next iteration of $\mathfrak{S}$.

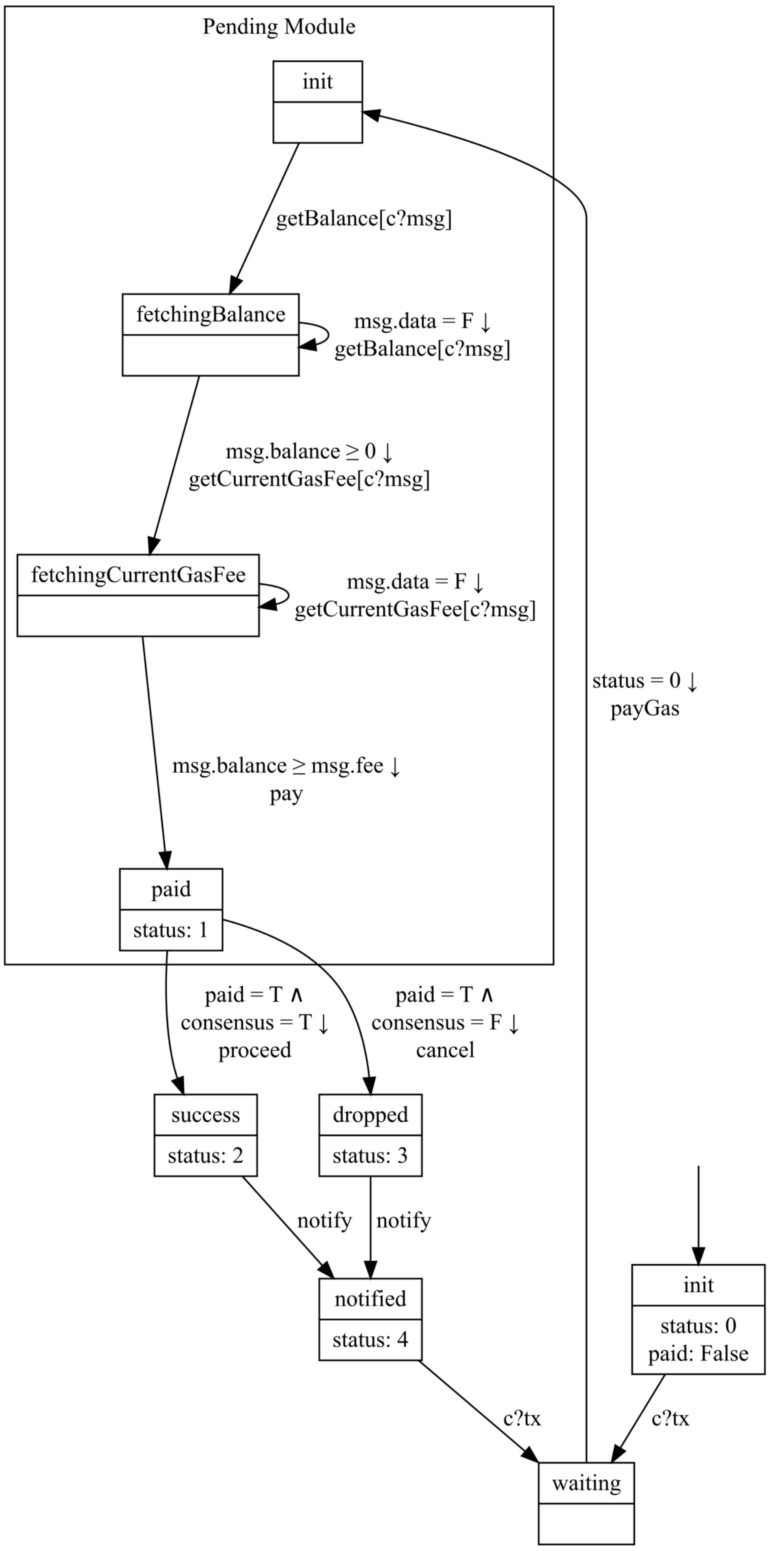

Pending Module
init
getBalance[c?msg]
fetchingBalance
msg.data = F ↓ getBalance[c?msg]
msg.balance ≥ 0 ↓ getCurrentGasFee[c?msg]
fetchingCurrentGasFee
msg.data = F ↓ getCurrentGasFee[c?msg]
msg.balance ≥ msg.fee ↓ pay
paid
status: 1
status = 0 ↓ payGas
paid = T ∧ consensus = T ↓ proceed
paid = T ∧ consensus = F ↓ cancel
success
status: 2
dropped
status: 3
notify
notify
notified
status: 4
init
status: 0
paid: False
c?tx
c?tx
waiting

Figure 5. Visualized system graph of the transaction client with the unfold pending module in Example 10.

4. Formalism-Driven Development in Practice

FDD is designed to be useful and usable to serve practical purposes. Particularly, we intend to minimize the efforts for developers to master FDD and be able to put it in production lines. To achieve this, new tools are required to practicalize FDD. In this section, we discuss the criteria of FDD tools, followed by an introduction of Seniz, a full-fledged framework for FDD. Furthermore, we use case studies to show how a system is developed under FDD with Seniz.

4.1. Criteria

We enumerate a number of functional and quality criteria for FDD tools. Functional criteria define the core functionalities, while quality criteria show important requirements for practicalizing FDD.

4.1.1. Functional Criteria

Structure Representation

In FDD, system graphs are fundamental elements. A language is necessary to represent system graphs and structures derived from them. Although a language in natural form can be used, a formal language is preferred in FDD for automation.

System graphs are designed to be easily represented and mechanized in various programming paradigms. In object-oriented programming, a system graph can be regarded as an object with properties (e.g., state declarators, transition relations, and propositions) and private methods (e.g., naming function and labeling function) executed while being instantiated. In declarative programming paradigms, a system graph is easier to be represented. For instance, a system graph can be defined by data types in functional programming. Notably, domain-specific languages designed and implemented to represent system graphs are likely to be more effective than general programming paradigms.

Stage Mechanization

FDD has a collection of mechanisms in different stages. A tool needs to correctly implement full or partial mechanisms for at least a type of structure representation. For a tool that only implements partial mechanisms, compatibility issues must be addressed to keep consistency among mechanisms.

In the *Abstraction Stage*, basic mechanisms such as *State Inference* and *Transition System Transformation* are required to interpret semantics of system graphs. In addition, (bi)simulation mechanisms are indispensable for the refinement. Depending on the goal of tools, verification mechanisms in the *Verification Stage* have two types of effective implementations. A tool can either directly verify the properties of a system graph or translate it together with its properties to another dedicated tool for verification purposes. Code generation algorithms play a pivotal role in the *Implementation Stage* to transform system graphs to executable programs with action effects. Furthermore, implementations should contain specific mechanisms to handle technical points such as termination, parallelism, divergence, confluence, and channel. The same with *Structure Representation* criteria, increment mechanisms including both horizontal and vertical increment mechanisms in the *Integration Stage* are independent of languages and paradigms and easy to be implemented. Additionally, a version control mechanism is required to manage the workflow decided in *Next Move*.

Design Visualization

Human-centered design should be a crucial aspect of an FDD tool. A design in FDD tools must be readable, intuitive, and rigorous. As core components, structures and some mechanisms should be visualized to improve readability and understandability. We define two levels of visualization: syntax level and semantics level. At the syntax level, a diagram correctly depicts structures described by system graphs such as figures in running examples: Figures 2–5. Semantic-level diagrams are usually more expressive than syntax-level diagrams. They carry out computations in mechanisms and render results. For example, *State Inference* can be enforced synchronously with the syntax-level rendering process or asynchronously triggered in an interactive manner. Furthermore, complex functionalities can be implemented based on semantic-level diagrams such as optimizing and refactoring system graphs.

4.1.2. Quality Criteria

Simplicity

FDD tools should have concise rules and verbose syntax to use. Concise rules aim to keep the usage as simple as possible to make developers master basic operations with a minimum knowledge of formal methods. Consequently, tools should encapsulate underlying mathematics and promote intuitive operations similar to widely used methods such as UML state machine for design and object-oriented programming for implementation.

A certain verbosity level means that syntactic sugar should be introduced for frequently used syntactic constructs. For instance, to construct a complex system graph, redundant and repeated works should be conveniently omitted or simplified with verbose statements that are automatically rephrased in fundamental syntax. Typical techniques such as parameterization and higher-order function can be used for this purpose. Parameterization can generalize system graphs to manufacture argument-dependent system graphs. Higher-order functions such as *fold* can help assemble complex system graphs and propositional formulas.

Applicability

Compatibility and modularity are crucial for applicability. For a dedicated tool, its applicability is not only determined by how powerful it is but also by how well it collaborates with other tools to contribute to the success of an FDD process. System architects may prefer compatible enhancements of current systems by introducing a new tool. For a comprehensive tool that implements a set of functionalities, modularity is essential, meaning that a tool should separate its functionalities and support enabling a subset of them without impairment. In this manner, system architects can select appropriate tools and build tech stacks to satisfy requirements in a development context.

Interoperability

A system developed with FDD tools should be able to communicate and integrate with systems developed under other development processes. FDD is not a silver bullet for developing all types of systems. Therefore, subsystems within a complex system may be developed in different processes. For those systems developed with FDD tools, high-level APIs play an important role in communicating with external systems without exposing internal states.

Testability

Currently, verification cannot replace testing and vice versa. FDD does not refuse testing techniques. On the contrary, testing is indispensable in FDD. Functionalities implemented by FDD tools need to be testable, especially implementations produced in the *Implementation Stage*. These functionalities require testing techniques to assist code-level verification tools to detect runtime exceptions caused by programming errors such as poorly implemented algorithms and compiler defects. In addition, for unverifiable properties in the *Verification Stage*, testing techniques can be used to verify correctness under certain constraints partially.

Performance

For any FDD tool, its performance consists of two aspects: efficiency and scalability. Efficiency requirements vary in purposes. For design-intensive stages (i.e., *Abstraction Stage* and *Integration Stage*), the visualization process needs to have a short response time to ensure high working efficiency and good user experience. Although efficiency is also an important factor in implementation-intensive stages (i.e., *Verification Stage* and *Implementation Stage*), scalability could be imperative. As an iterative and incremental process, a tool needs to adapt to system scaling with a minimum sacrifice of functionalities and efficiency.

4.2. Seniz

Seniz is an FDD framework that provides a collection of tools to support FDD processes. As shown in Figure 6, Seniz consists of a modeling language, a verification generator, a skeleton generator, and a version controller.

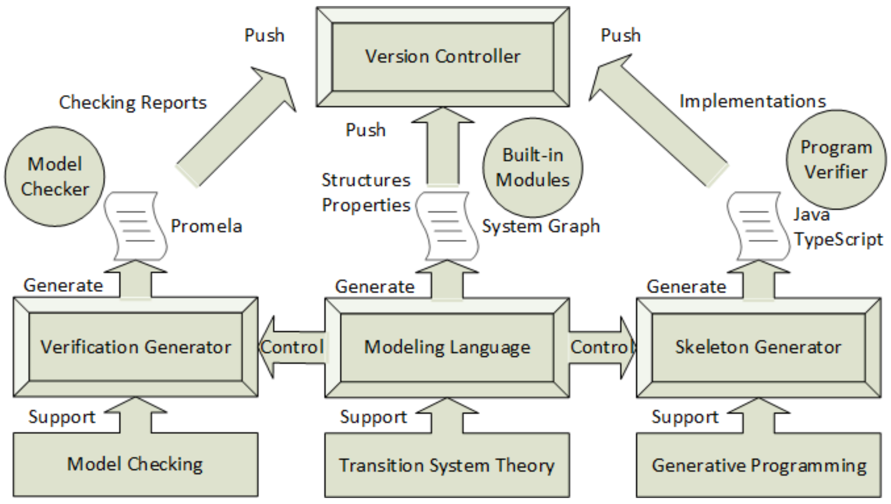


Figure 6. Seniz architecture.

4.2.1. The Seniz Language

The Seniz language is a modeling language designed to abstract system graphs from real-world systems. It allows developers to formulate static structures by state declarators, dynamic changes by actions and transition relations, as well as expected system behaviors by formal propositions and properties. We use two examples to show the main features of the Seniz language and list the core syntax and common operators in Appendix A.

Example 11 (Transaction Client with Seniz)**.** *A Seniz program that codes the system graph in Example 1 is shown in Listing 1.*

Example 12 (Block Writing Problem)**.** *We provide another example that programs the block writing problem widely used in EVM implementations to enable multithread block persistence. For instance, a set of workers separately run in multiple threads to process received transactions, while an EVM should expect that only one thread can write the block and associated states to the database. Hence, a mutual exclusion mechanism is necessary only to allow a worker that has acquired a mutex to proceed with the writing function.*

We use a generalized mechanism based on a semaphore to model this problem. We define a system graph Worker over a local variable set Vars in Listing 2 and a global variable set Lock in Listing 3. We simply use the variable loc to represent the current execution fragment.

To model the interleaving of a set of workers, we program a control system BlockWriting that instantiates two worker systems. We also formulate three LTL properties with respect to the satisfaction of the mutual exclusion and two types of fairness.

Listing 1. Transaction client coded in the Seniz language.

```
main system TransactionClient(id::int) over TXClientVar, TXClientChan {

init { status: 0, paid: false } –> receiveTx(c?tx) waiting –> PayGas() pending
pending [paid=true & consensus = true] –> Proceed() success –> Notify() notified
pending [paymentFailed] –> Cancel() dropped –> Notify() notified
notified –> NotifyTx(c?tx) waiting

waiting = {
status: 0,
paid: false,
consensus: false
}

pending = {
status: 1,
paid: true
}

success = {
status: 2,
paid: true,
consensus: true
}

dropped = {
status: 3,
paid: true,
consensus: false
}

notified = {
status: 4
}

prop paymentFailed {
paid = true and consensus = false
}

prop paidGas {
status = 1
}

prop notified {
status = 4
}

ltl infinitelyOftenNotified {
always (paidGas –> eventually notified)
}
}

varset TXClientVar {
status :: int,
paid :: bool,
consensus :: bool,
tx :: string
}

chanset TXClientChan {
c :: external
}
```

Listing 2. Block writing worker.

```
import Lock

system Worker() over Vars with Lock {

init processing –> readyToWrite
readyToWrite –> readyToWrite [s > 0] –> @decreaseS acquiredLock
acquiredLock –> writeBlock() @increaseS releasedLock
releasedLock –> processing

processing = {
loc: 1
}

readyToWrite = {
loc: 2
}

acquiredLock = {
loc: 3
}

releasedLock = {
loc: 4
}

@increaseS = {
s: s + 1
}

@decreaseS = {
s: s – 1
}

prop hasLock {
loc = 3
}

}

varset Vars {
loc :: int
}
```

Listing 3. Global (shared) variable set *Lock*.

```
varset Lock {
s :: int
}
```

In the program of Example 11, we define a system graph named *TransactionClient* over a state variable set *TXClientVar* and a channel set *TxClientChan*. We declare a set of named state declarators from line 8 to 33. The initial state declarator *{status: 0, paid: false}* is anonymous, which contains two state declarations describing interesting state variables. In fact, it is a form of abductive reasoning supported in the Seniz language. According to Definition 11, each state declarator has a unique name. To distinguish anonymous state declarators, the Seniz compiler applies the hash function to the inferred state variable evaluations and uses hash values as corresponding names.

A set of transition rules are declared from line 3 to 6. The simplest form of a transition rule should contain a source state declarator, a symbol ->, and a destination declarator. In this form, guard is regarded as tautology, action is interpreted as **epsilon action** (i.e., no action effect is emitted), and there is no global variable change. Nevertheless, a transition rule carries an action to emit effects such as line 6 where the transition from *notified* state declarator to *waiting* state declarator emits effects caused by action *NofityTx*. The transition can also be guarded by a propositional formula such as line 4 where the transition from *pending* to *success* requires the condition *paid=true* ∧ *consensus=true* to be satisfied. Guards can be defined by propositions such as line 5 where proposition *paymentFailed* is put into square brackets to become a part of the guard for transition from *pending* to *dropped*, which implements the proposition definition in Definition 11. Additionally, we provide syntactic sugar to simplify sequential transitions such as the codes in line 14 and 15 of Listing 1.

Propositions are declared with keyword *prop* and structured by Boolean expressions and propositional expressions. We define three propositions from line 35 to 45. Proposition *paidGas* and *notified* are used to declare the LTL property in Example 5 from line 47 to 49.

In the Seniz language, we have two modifiers for a system graph: *main* and *control*. A modifier *main* notes the entry point of a program. Hence, any program must contain exactly one **main system**. A **control system** is a high-level system that represents a composition of subsystems. It has exactly one control statement to describe the parallelism of a set of subsystems. Keyword *async* denotes the asynchronous concurrency defined in Definition 5, while *sync* denotes the synchronous concurrency defined in Definition 6. In the *BlockWriting* control system shown in Listing 4, two *Worker* systems are instantiated as two asynchronous systems and share the global variable set *Lock*.

Listing 4. Block writing control system.

```
import Lock
import Worker

main control system BlockWriting() over Lock {

async Worker() as w1, Worker() as w2

ltl mutexHolds {
G (!w1.hasLock and !w2.hasLock)
}

ltl unconditionalFairnessHolds {
G F (w1.hasLock) and G F (w2.hasLock)
}

ltl strongFairnessHolds {
G F (w1.isReady −> G F (w1.hasLock)) and G F(w2.isReady −> G F(w2.hasLock))
}

}
```

Additionally, the Seniz language supports system arguments and multi-file compilation for genericity and modularity. A **system argument** is a global constant carrying information from a higher-level system. System arguments of the main system, also called **environment arguments**, are supplied to a program at the beginning of execution. By using keyword *import*, a Seniz program allows importing system graphs, state variable sets, channel sets, and records defined in local or remote files.

Furthermore, the Seniz language translates a system graph into a DOT program for visualization. The program given in Example 11 is visualized in Figure 7.

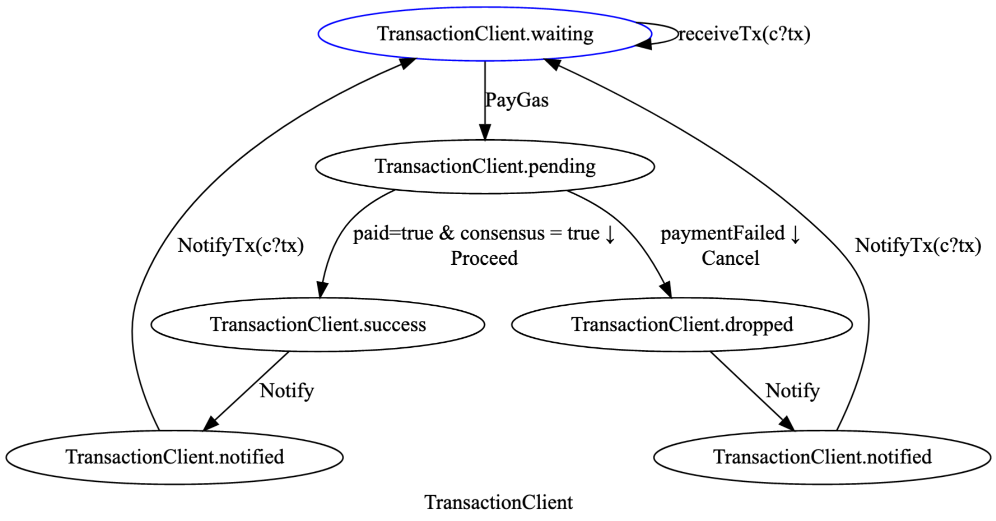

TransactionClient.waiting
receiveTx(c?tx)
PayGas
TransactionClient.pending
paid=true & consensus = true ↓
Proceed
paymentFailed ↓
Cancel
NotifyTx(c?tx)
NotifyTx(c?tx)
TransactionClient.success
TransactionClient.dropped
Notify
Notify
TransactionClient.notified
TransactionClient.notified
TransactionClient

Figure 7. Visualized system graph of the transaction client in Seniz.

We can find that Figure 7 has slight differences from Figure 2 in Example 1. The Seniz compiler merges anonymous state declarators into named state declarators with their supersets of state variable declarations. Therefore, the *init* state declarator is merged into the *waiting* state declarator in Figure 7. In addition, the Seniz compiler processes the system graph with state inference. In Example 11, two different *notified* states are inferred and distinguished, which creates two separate branches in Figure 7.

4.2.2. Verification Generator

Seniz provides a verification generator that translates Seniz programs developed in the abstraction stage into Promela programs for the verification stage.

A state declarator is translated into a *macro* and an *inline*. A *macro* is defined as the conjunction of all state variables identified in the declarator and additional critical variables inferred by the compiler.

Example 13 (State Declarator Translation for Promela)**.** *For the acquiredLock state declarator in Example 12, it is translated to a macro and an inline shown in Listing 5.*

Listing 5. Translated *acquiredLock* state declarator in Promela.

```
#define ACQUIREDLOCK loc[_pid] == 3

inline acquiredLock() {
atomic {
loc[_pid] = 3;
}
}
```

The transition relations, used to construct the control flow, are translated into a *proctype*.

Example 14 (Transition Relation Translation for Promela)**.** *A proctype is generated according to the translation relations programmed in Example 12, which is shown in Listing 6.*

Listing 6. Translated transition relations in Promela.

```
proctype Worker() {
processing();
do
:: PROCESSING –> readyToWrite();
:: READYTOWRITE –>
atomic {
if
:: _s > 0 –> _decreaseS(); acquiredLock();
fi;
}
:: ACQUIREDLOCK –> atomic {_increaseS(); releasedLock();}
:: RELEASEDLOCK –> processing();
od;
}
```

The translation of propositions and temporal properties is trivial on account of the similar syntax between the Seniz language and the Promela language.

The generated Promela program can be directly used for verification by model checkers such as the SPIN model checker [1]. For the Promela program generated from Example 12, the checking report implies property *mutexHolds* is satisfied while both fairness properties are violated.

4.2.3. Skeleton Generator

The current version of Seniz supports generating Java and TypeScript programs from Seniz programs. The core method is illustrated in Section 3.3 and follows the OOP paradigm.

The generator integrates a powerful toolchain to support the generated SDK and allows customizing the tech stack. The FDD developers only need to focus on the implementation of action effects.

Example 15 (Implementation of Action Effects)**.** *For the block writing in Example 12, the generated Java SDK contains an abstract class ActionExecutor that implements an interface Action including a method writeBlock generated from the non-epsilon action set. To implement the effect of action writeBlock, we only need to extend the abstract class ActionExecutor, which is shown in Listing 7.*

The effect is quite simple, which prints the current Worker system identifier and state variable loc to the output stream at the beginning and end. In this example, we should not observe the interleaving outputs of different workers thanks to the verified property mutexHolds.

Listing 7. Implementation of the effect of action *writeBlock* in Java.

```
public class ActionEffect extends ActionExecutor {

@Override
public void writeBlock() {
String entryInfo = "Worker␣%s␣is␣writing␣block␣at␣location␣%s"
.formatted(getArgument(ID, ID.getType()).orElseThrow(),
getVariable(LOC, LOC.getType()).orElseThrow());
System.out.println(entryInfo);

String exitInfo = "Worker␣%s␣finished␣writing.%n"
.formatted(getArgument(ID, ID.getType()).orElseThrow());
System.out.println(exitInfo);
}

}
```

Two *SystemExecutor*s are generated with respect to the *Worker* system and *BlockWriting* control system. The *Worker SystemExecutor* extends an abstract and generic class *SystemExecutorThread* implementing the *Callable* interface. Hence, *WorkerSystemExecutor* can run in thread and managed by the Java *ExecutorService* integrated in the *BlockWritingSystemExecutor*. The *BlockWriting SystemExecutor* also handles setting global variables and system arguments.

Example 16 (Execution)**.** *To execute the system, we merely need to instantiate BlockWriting SystemExecutor and invokes method run. A demonstration is shown in Listing 8.*

Listing 8. Executing the *BlockWriting* control system in Java.

```
@Test
public void testBlockWriting() {
SystemExecutor systemExecutor = new SystemExecutor();
systemExecutor.run();
}
```

4.2.4. Version Controller

Additionally, a rigorous version controller is mechanized in Seniz. Based on the rigorous definition of iterations and increments illustrated in Section 3.4 and cryptographic hash function, the version controller automatically archives iterations and increments and labels them with verified properties. Furthermore, the version controller supports the branching workflow.

1. Create or refine a system graph $\mathfrak{S}$ with the Seniz language.
2. Generate a Promela program from $\mathfrak{S}$.
3. Specify and verify the properties of $\mathfrak{S}$ by a model checker.
4. If all specified properties pass verification, then move to the next step. Otherwise, go back to step 1.
5. Generate a Java or TypeScript skeleton program and encapsulate it into an SDK from the verified $\mathfrak{S}$.
6. Implement exposed interfaces of the skeleton program such as action effects and interactive events with the support of the SDK to satisfy functional requirements.
7. If $\mathfrak{S}$ is independent and unrefinable, terminate the workflow. If $\mathfrak{S}$ is dependent and unrefinable, integrate $\mathfrak{S}$ into a higher-level system graph $\mathfrak{S}'$ to obtain $\mathfrak{S}^*$, start a new parallel iteration in $\mathfrak{S}'$ branch with $\mathfrak{S}^*$, and go into the next iteration with $\mathfrak{S}$. Otherwise, go into the next iteration with $\mathfrak{S}$.

4.2.5. Graphical User Interface

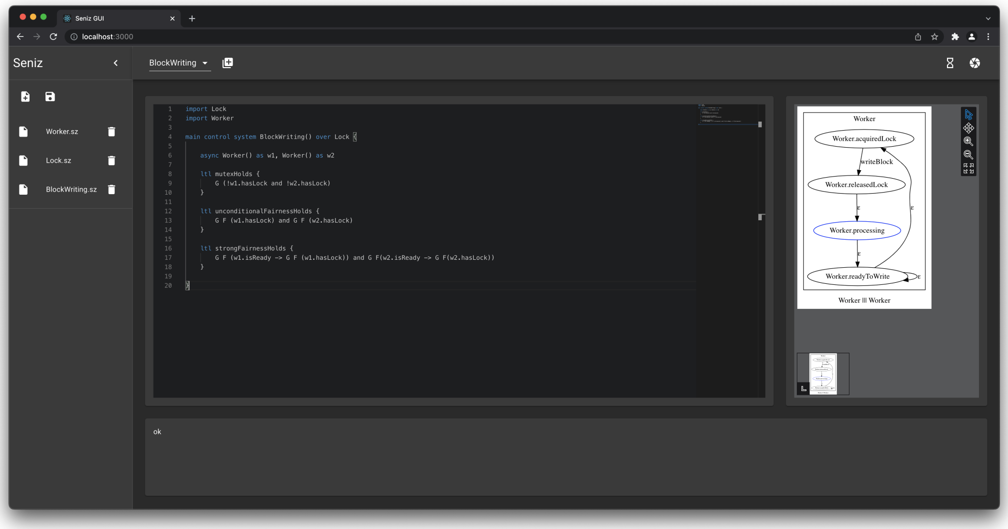
Seniz has a web-based user interface and scaffolds to provide a lightweight IDE (integrated development environment). It facilitates project management, coding, debugging, and version control. An example of the main interface is shown in Figure 8.

4.3. Evaluation of Seniz

Seniz is the first FDD framework, which implements core tools for FDD processes and satisfies all functional criteria. The Seniz language satisfies syntactic completeness, meaning that for each structure in FDD, there exists a corresponding representation in the Seniz language that can be formulated. In addition, system graphs defined in the Seniz language can be visualized and exported as design documents. Based on the Seniz language, the verification generator and skeleton generator help implement mechanisms in different stages.

For quality criteria, Seniz uses high-level APIs to encapsulate the underpinning mathematics of FDD and implements the support for parameterization and higher-order functions for simplicity. The generated skeleton programs can be directly used in Java or TypeScript environments to interact with other systems, which implements a certain level of interoperability.

However, the tools in Seniz are centered around the Seniz language and not usable out of the framework. In addition, the Seniz language is not testable and cannot recognize functional and computational errors. Moreover, performance issues become significant in visualization, compilation, and generation for complex system graphs [6].



```
import Lock
import Worker

main control system BlockWriting() over Lock {

    async Worker() as w1, Worker() as w2

    ltl mutexHolds {
        G (!w1.hasLock and !w2.hasLock)
    }

    ltl unconditionalFairnessHolds {
        G F (w1.hasLock) and G F (w2.hasLock)
    }

    ltl strongFairnessHolds {
        G F (w1.isReady -> G F (w1.hasLock)) and G F(w2.isReady -> G F(w2.hasLock))
    }

}
```

```
ok
```

Figure 8. A snapshot of the Seniz web-based interface.

5. Related Work

5.1. Model-Driven Development

Model-driven development (MDD) [27,28] focuses on formulating a model as an abstraction of a system and derives source codes from the model. In addition, model transformation [29–31] enhances the flexibility and extends the scope of MDD by enabling working with multiple and interrelated models.

MDD has been widely adopted in many fields such as web development [32], mobile app development [33], IoT application development [34] based on the UML, or proposed domain-specific languages.

The benefits of MDD are evaluated in many studies [35–37] including both correctness and efficiency improvement compared to code-centric development. Although some works such as [38] point out that MDD is more suitable in academic settings, a recent study [39] contradicts the claim by conducting real-world development experiments, which shows the great significance of using models in software engineering.

Some research has started using model-driven development for developing blockchain applications. Lorikeet [40] is developed to facilitate the development of blockchain applications by automatically generating well-tested smart contracts from specifications in the business process and data registry models. It allows users to use an extended BPMN to model business processes. Smart contracts are generated by a BPMN translator, registry generator, and blockchain trigger. In [41], a smart contract generator based on the UML state diagram is proposed to coordinate the usage of cyber-physical systems. In their work, a mapping from the state and transition to transactions of the blockchain platform is constructed for the code generation. A method based on Petri Nets is proposed in [42]. It uses Petri Nets to specify workflow and enforcement to generate secure and understandable smart contracts through a translation engine. In addition, the logical errors can be minimized by verifying Petri Nets properties.

5.2. Verification-Driven Engineering

Verification-driven engineering (VDE) [43] integrates the formal methods in MDD and particularly promotes formal verification during the development process. In [43], the principles and requirements for the better use of formal methods in MDD are illustrated. It concludes that it is necessary to switch from MDD to VDE even though it still needs more sophisticated techniques to support it.

Some works also enhance the MDD by introducing verification-driven methods such as [44]. It proposes a verification-driven slicing technique to partition the model into submodels while preserving properties by formal verification. Sphinx [45] is proposed as a VDE toolset for modeling and verifying hybrid systems. It defines semantics for the UML activity diagrams. In [46], a verification-driven framework named FIDDle is presented and evaluated by developing parts of the K9 Mars Rover model.

However, none of them tackle the problem from the perspective of the development process. The relationships between models also lack formalization. FDD formally defines different stages and introduces a formalism to manage the process rigorously.

5.3. Blockchain-Oriented Engineering

Recently, the research on the blockchain-oriented development process has made some progress.

The work [47] studies three approaches to model and implement a taxi dispatcher application on a blockchain, including an extended BPMN approach, using synchronized state-machines, and high-level Petri nets. In [48], a code generation method for smart contracts is proposed based on MDD for collaborative business processes.

In addition to MDD, other methods are also studied to optimize the development process of blockchain applications, such as [49,50]. In [49], it proposes an agile software engineering method to organize the development process with concrete plans and introduces a set of new UML stereotypes to enhance the modeling capability. Some architectural patterns extracted from existing decentralized applications are studied in [50].

Notably, they focus on providing application-level solutions for the development of decentralized applications and barely introduce formal methods, as a critical component in their methodologies.

6. Discussion

FDD is a type of development process. Architects and project managers need to concretize FDD processes based on the development context, such as requirement specification, developer skills, tech stack, project budget, and duration. As an iterative and incremental process, components should be built from small portions that are refined repeatedly through FDD four stages and eventually formed by continuous integration as illustrated in Section 3.4.

FDD is compatible with testing techniques. Testing is still the fastest way to check correctness in each stage, though SMT solvers such as Z3 [51] can be used in daemons to validate some propositional formulas in real time. Furthermore, formal testing techniques such as model-based testing [52] can be naturally integrated into FDD processes.

FDD is not a verification process. Verification is a critical step in FDD to the success of developing trustworthy systems. Nevertheless, design and implementation are also significant in FDD. The former improves development efficiency and system maintainability, while the latter ensures that deliveries satisfy the needs. In a concretized FDDD process, architects need different types of formal methods tools, including specification, verification, and optional testing according to the criteria in Section 4.1 and the development context.

FDD is not a silver bullet. We do not intend to replace the existing development processes and methodologies completely but to provide an alternative and reference to promote the application of formal methods. Its applicability is highly restricted by the limitations of formal methods such as performance and automation while facing state explosion problems and provability issues.

FDD is not suitable for studying algorithms. Although mechanizing functional models of computation such as rewriting systems and Lambda calculus in FDD formalisms could be theoretically interesting, it is not very attractive for developers in practice.

7. Conclusions

This paper presents the concepts, taxonomy, and practice of formalism-driven development, which is an iterative and incremental development process for developing provably correct systems with formal methods. FDD can be regarded as an alternative to the existing development processes and a template for the application of formal methods throughout the development lifespan. We summarize its main advantages as follows.

- FDD produces readable, visualizable, and rigorous designs by formal specification techniques.
- Models are directly derived from designs and mathematically verifiable via formal verification techniques.
- Verification processes are simplified by the translation from models to verification-oriented programs.
- Implementation processes are driven by designs and preserve control-flow properties.
- Continuous integration and delivery are naturally enabled in FDD processes.
- Iterations and increments are well defined to ensure consistency and manageability.

The current main disadvantages are listed as below.

- FDD is expensive due to the cost of verifying complicated properties and the requirement of learning, though not much, additional skills above the code level.
- FDD tools are not sophisticated.
- No community support.

Meanwhile, FDD must continually confront usability issues. Our future directions will be centered around usability, including

1. Optimizing the underlying theories, especially the algebra of transition systems, refinement theory, and concurrency theory;
2. Implementing FDD tools that well satisfy both functional and quality criteria;
3. Extending the scope of FDD for more types of development projects.

Author Contributions: Conceptualization, Y.D. and H.S.; methodology, Y.D.; software, Y.D.; validation, Y.D.; formal analysis, Y.D.; investigation, Y.D.; resources, H.S.; data curation, Y.D.; writing—original draft preparation, Y.D.; writing—review and editing, H.S.; visualization, Y.D.; supervision, H.S.; project administration, H.S.; funding acquisition, H.S. and Y.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by KAKENHI (Grant-in-Aid for JSPS Fellows) 21J21087, KAKENHI (Grants-in-Aid for Scientific Research) (C) 19K11958, and KAKENHI Grant-in-Aid for Scientific Research (B) 19H04098.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: We would like to give special thanks to Leonidas Lampropoulos for his support in the preliminary stage of this research.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Sample Availability: The Seniz framework for educational use is available from the authors.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
BPMN	Business Process Modeling Notation
CTL	Computation Tree Logic
DLT	Distributed Ledger Technology
FDD	Formalism-Driven Development
LTL	Linear Temporal Logic
MDD	Model-Driven Development
SDK	Software Development Kit
SMT	Satisfiability Modulo Theories
UML	Unified Modeling Language
VDE	Verification-Driven Engineering

Appendix A. The Seniz Language

Appendix A.1. Core Syntax

The core syntax of the Seniz language is as follows.

$$\begin{aligned}
\mathit{Unit} &::= \mathit{Import}^{*}\ \mathit{System}?\ \mathit{VarSet}?\ \mathit{ChanSet}?\ \mathit{Record}^{*}\\
\mathit{System} &::= \mathit{Modifier}^{*}\ \textbf{system}\ \mathit{SystemId}\ \mathit{SystemArg}\ \textbf{over}\ \mathit{SystemParam}\ \mathit{SystemBody}\\
\mathit{VarSet} &::= \textbf{varset}\ \mathit{VarSetId}\ \textbf{\{}\ \mathit{VarSetExpr}^{*}\ \textbf{\}}\\
\mathit{ChanSet} &::= \textbf{chanset}\ \mathit{ChanId}\ \textbf{\{}\ \mathit{ChanSetExpr}^{*}\ \textbf{\}}\\
\mathit{SystemBody} &::= \mathit{ControlBody} \mid \mathit{PlainBody}\\
\mathit{ControlBody} &::= \mathit{ControlStmt}\ \mathit{globalStateDecl}?\ \mathit{LogicStmt}^{*}\\
\mathit{PlainBody} &::= \mathit{StateDecl}^{*}\ \mathit{TransitionDecl}^{*}\ \mathit{LogicDecl}^{*}\\
\mathit{StateDecl} &::= \mathit{StateId}\ \textbf{=}\ \textbf{\{}\ \mathit{StateExpr}^{*}\ \textbf{\}}\\
\mathit{TransitionDecl} &::= \textbf{init}?\ \mathit{StateId}\ (\mathit{Guard}?\ \textbf{->}\ \mathit{ActionDecl}?\ \mathit{GlobalStateId}?\ \mathit{StateId})^{*}\\
\mathit{StateExpr} &::= \mathit{VarId}\ \textbf{:}\ \mathit{Expr}\\
\mathit{VarSetExpr} &::= \mathit{VarId}\ \textbf{::}\ \mathit{Type}\\
\mathit{ChanSetExpr} &::= \mathit{ChanId}\ \textbf{::}\ \mathit{ChanType}
\end{aligned}$$

Appendix A.2. Common Operators

We summarize notable common operators in Table A1.

Table A1. Operators in the Seniz language.

Operator	Meaning	Scope
:	has value	state variable
::	has type	state variable, channel
->	from (left) to (right)	transition relation
@	global	state variable, state declarator
!	send	channel
?	receive	channel
=	structural equality	state declarator
=	physical equality	proposition
!	logical not	proposition
and	logical and	proposition
or	logical or	proposition
->	implication	proposition
always, *G*	always	temporal proposition
eventually, *F*	eventually	temporal proposition

References

1. Holzmann, G.J. The model checker SPIN. *IEEE Trans. Softw. Eng.* **1997**, *23*, 279–295. [CrossRef]
2. Cimatti, A.; Clarke, E.; Giunchiglia, E.; Giunchiglia, F.; Pistore, M.; Roveri, M.; Sebastiani, R.; Tacchella, A. Nusmv 2: An open-source tool for symbolic model checking. In *International Conference on Computer Aided Verification*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 359–364.
3. Paulson, L.C. *Isabelle: A Generic Theorem Prover*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 1994; Volume 828.
4. Moura, L.d.; Kong, S.; Avigad, J.; Doorn, F.v.; Raumer, J.v. The Lean Theorem Prover (System Description). In *International Conference on Automated Deduction*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 378–388.
5. Valmari, A. The state explosion problem. In *Advanced Course on Petri Nets*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 429–528.
6. Ding, Y.; Sato, H. Formalism-Driven Development of Decentralized Systems. In Proceedings of the 2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS), Hiroshima, Japan, 26–30 March 2022; pp. 81–90.
7. Zheng, Z.; Xie, S.; Dai, H.N.; Chen, X.; Wang, H. Blockchain challenges and opportunities: A survey. *Int. J. Web Grid Serv.* **2018**, *14*, 352–375. [CrossRef]
8. Nakamoto, S. *Bitcoin: A Peer-to-Peer Electronic Cash System*; Technical Report; Manubot: Online, 2019.
9. Buterin, V. A Next-Generation Smart Contract and Decentralized Application Platform. *White Paper* **2014**, *3*.
10. Sunyaev, A. Distributed ledger technology. In *Internet Computing*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 265–299.
11. Reyna, A.; Martín, C.; Chen, J.; Soler, E.; Díaz, M. On blockchain and its integration with IoT. Challenges and opportunities. *Future Gener. Comput. Syst.* **2018**, *88*, 173–190. [CrossRef]
12. Novo, O. Blockchain meets IoT: An architecture for scalable access management in IoT. *IEEE Internet Things J.* **2018**, *5*, 1184–1195. [CrossRef]
13. Ding, Y.; Sato, H. Dagbase: A decentralized database platform Using DAG-based consensus. In Proceedings of the 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, 13–17 July 2020; pp. 798–807.
14. Ding, Y.; Sato, H. Derepo: A distributed privacy-preserving data repository with decentralized access control for smart health. In Proceedings of the 2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom), New York, NY, USA, 1–3 August 2020; pp. 29–35.
15. Maesa, D.D.F.; Mori, P.; Ricci, L. A blockchain based approach for the definition of auditable access control systems. *Comput. Secur.* **2019**, *84*, 93–119. [CrossRef]
16. Ding, Y.; Sato, H. Bloccess: Towards fine-grained access control using blockchain in a distributed untrustworthy environment. In Proceedings of the 2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), Oxford, UK, 3–6 August 2020; pp. 17–22.
17. Mehar, M.I.; Shier, C.L.; Giambattista, A.; Gong, E.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *J. Cases Inf. Technol.* **2019**, *21*, 19–32. [CrossRef]
18. Sayeed, S.; Marco-Gisbert, H.; Caira, T. Smart contract: Attacks and protections. *IEEE Access* **2020**, *8*, 24416–24427. [CrossRef]
19. Destefanis, G.; Marchesi, M.; Ortu, M.; Tonelli, R.; Bracciali, A.; Hierons, R. Smart contracts vulnerabilities: a call for blockchain software engineering? In Proceedings of the 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), Campobasso, Italy, 20 March 2018; pp. 19–25.
20. De Nicola, R.; Vaandrager, F. Action versus state based logics for transition systems. In *LITP Spring School on Theoretical Computer Science*; Springer: Berlin/Heidelberg, Germany, 1990; pp. 407–419.
21. Reniers, M.A.; Willemse, T.A. Folk theorems on the correspondence between state-based and event-based systems. In *International Conference on Current Trends in Theory and Practice of Computer Science*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 494–505.
22. Nielsen, M.; Winskel, G. Models for Concurrency. In *MFCS*; Oxford University Press: Oxford, UK, 1991; pp. 43–46.
23. Shore, J. *The Art of Agile Development: Pragmatic Guide to Agile Software Development*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2007.
24. Baier, C.; Katoen, J.P. *Principles of Model Checking*; MIT Press: Cambridge, MA, USA, 2008.
25. Milner, R. *An Algebraic Definition of Simulation between Programs*. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1971; pp. 481-489.
26. Groote, J.F.; Vaandrager, F. An efficient algorithm for branching bisimulation and stuttering equivalence. In *International Colloquium on Automata, Languages, and Programming*; Springer: Berlin/Heidelberg, Germany, 1990; pp. 626–638.
27. Atkinson, C.; Kuhne, T. Model-driven development: A metamodeling foundation. *IEEE Softw.* **2003**, *20*, 36–41. [CrossRef]
28. Tolvanen, J.P.; Kelly, S. Model-driven development challenges and solutions: Experiences with domain-specific modelling in industry. In Proceedings of the 2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Rome, Italy, 19–21 February 2016; pp. 711–719.
29. Sendall, S.; Kozaczynski, W. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.* **2003**, *20*, 42–45. [CrossRef]
30. Czarnecki, K.; Helsen, S. Feature-based survey of model transformation approaches. *IBM Syst. J.* **2006**, *45*, 621–645. [CrossRef]
31. Jouault, F.; Allilaire, F.; Bézivin, J.; Kurtev, I. ATL: A model transformation tool. *Sci. Comput. Program.* **2008**, *72*, 31–39. [CrossRef]

32. Ceri, S.; Daniel, F.; Matera, M.; Facca, F.M. Model-driven development of context-aware Web applications. *ACM Trans. Internet Technol. (TOIT)* **2007**, *7*, 2–es. [CrossRef]
33. Vaupel, S.; Taentzer, G.; Gerlach, R.; Guckert, M. Model-driven development of mobile applications for Android and iOS supporting role-based app variability. *Softw. Syst. Model.* **2018**, *17*, 35–63.
34. Sosa-Reyna, C.M.; Tello-Leal, E.; Lara-Alabazares, D. Methodology for the model-driven development of service oriented IoT applications. *J. Syst. Archit.* **2018**, *90*, 15–22. [CrossRef]
35. Krogmann, K.; Becker, S. A case study on model-driven and conventional software development: The palladio editor. In *Software Engineering 2007–Beiträge zu den Workshops–Fachtagung des GI-Fachbereichs Softwaretechnik*; Gesellschaft für Informatik e. V.: Bonn, Germany, 2007.
36. Kapteijns, T.; Jansen, S.; Brinkkemper, S.; Houët, H.; Barendse, R. A comparative case study of model driven development vs traditional development: The tortoise or the hare. In *From Code Centric to Model Centric Software Engineering: Practices, Implications and ROI*; University of Twente: Enschede, The Netherlands, 2009; Volume 22.
37. Navarrete, J.I.P.; Dieste, O.; Marín, B.; España, S.; Vegas, S.; Pastor, O.; Juristo, N. Evaluating model-driven development claims with respect to quality: A family of experiments. *IEEE Trans. Softw. Eng.* **2018**, *47*, 130–145.
38. Panach, J.I.; España, S.; Dieste, O.; Pastor, Ó.; Juristo, N. In search of evidence for model-driven development claims: An experiment on quality, effort, productivity and satisfaction. *Inf. Softw. Technol.* **2015**, *62*, 164–186. [CrossRef]
39. Domingo, Á.; Echeverría, J.; Pastor, Ó.; Cetina, C. Evaluating the Benefits of Model-Driven Development. In *International Conference on Advanced Information Systems Engineering*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 353–367.
40. Lu, Q.; Binh Tran, A.; Weber, I.; O'Connor, H.; Rimba, P.; Xu, X.; Staples, M.; Zhu, L.; Jeffery, R. Integrated model-driven engineering of blockchain applications for business processes and asset management. In *Software: Practice and Experience*; Wiley Online Library: Hoboken, NJ, USA, 2020; ISBN 0038-0644.
41. Garamvölgyi, P.; Kocsis, I.; Gehl, B.; Klenik, A. Towards model-driven engineering of smart contracts for cyber-physical systems. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Luxembourg, 25–28 June 2018; pp. 134–139.
42. Zupan, N.; Kasinathan, P.; Cuellar, J.; Sauer, M. Secure smart contract generation based on petri nets. In *Blockchain Technology for Industry 4.0*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 73–98.
43. Kordon, F.; Hugues, J.; Renault, X. From model driven engineering to verification driven engineering. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 381–393.
44. Shaikh, A.; Clarisó, R.; Wiil, U.K.; Memon, N. Verification-driven slicing of UML/OCL models. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, 20–24 September 2010; pp. 185–194.
45. Mitsch, S.; Passmore, G.O.; Platzer, A. Collaborative verification-driven engineering of hybrid systems. *Math. Comput. Sci.* **2014**, *8*, 71–97. [CrossRef]
46. Menghi, C.; Spoletini, P.; Chechik, M.; Ghezzi, C. A verification-driven framework for iterative design of controllers. *Form. Asp. Comput.* **2019**, *31*, 459–502. [CrossRef]
47. Dittmann, G.; Sorniotti, A.; Völzer, H. Model-Driven Engineering for Multi-party Interactions on a Blockchain–An Example. In *International Conference on Service-Oriented Computing*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 181–194.
48. Xu, X.; Weber, I.; Staples, M. Model-Driven Engineering for Blockchain Applications. In *Architecture for Blockchain Applications*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 149–172.
49. Marchesi, M.; Marchesi, L.; Tonelli, R. An agile software engineering method to design blockchain applications. In Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia, Moscow, Russia, 12–13 October 2018; pp. 1–8.
50. Wessling, F.; Gruhn, V. Engineering software architectures of blockchain-oriented applications. In Proceedings of the 2018 IEEE International Conference on Software Architecture Companion (ICSA-C), Seattle, WA, USA, 30 April–4 May 2018; pp. 45–46.
51. Moura, L.d.; Bjørner, N. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 337–340.
52. Tretmans, J. Model based testing with labelled transition systems. In *Formal Methods and Testing*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 1–38.