Formal Verification of Legal Contracts: A Translation-based Approach

Reiner Hähnle[1 [0000−0001−8000−7613]], Cosimo Laneve[2 [0000−0002−0052−4061]], and Adele Veschetti[1 [0000−0002−0403−1889]]

[1] Department of Computer Science, TU Darmstadt, Germany
{reiner.haehnle,adele.veschetti}@tu-darmstadt.de
[2] Department of Computer Science, University of Bologna, Italy
cosimo.laneve@unibo.it

Abstract. *Stipula* is a domain-specific programming language designed to model legal contracts with enforceable properties, especially those involving asset transfers and obligations. This paper presents a methodology to formally verify the correctness of Stipula contracts through translation into Java code annotated with Java Modeling Language specifications. As a verification backend, the deductive verification tool KeY is used. Both, the translation and the verification of partial and total correctness for a large subset of *Stipula* contracts, those with disjoint cycles, is fully automatic. Our work demonstrates that a general-purpose deductive verification tool can be used successfully in a translation approach.

Keywords: Formal Verification · Stipula Language · Translation-based Verification · Deductive Verification.

1 Introduction

As the legal domain continues its digital transformation, the demand for precise, machine-verifiable representations of legal contracts becomes increasingly important. Traditional legal texts, written in natural language, are inherently ambiguous and prone to misinterpretation, making them challenging to process, analyze, or verify by automated means. To address this issue, several projects are being developed for defining programming languages to write legal contracts, *e.g.* [16–18, 20]. Although these projects introduce a precise syntax for legal contracts and provide graphical tools to associate normative elements with code, they pay limited attention to the verification of correctness. Yet, this aspect is crucial in legal contexts, where ambiguities or unintended behavior during execution can lead to significant legal and financial repercussions. Despite advances in the formal verification of legal contracts, formal reasoning is rare in practice due to tool complexity.

To overcome this lack of automatic verification techniques, in 2021, we designed *Stipula* [7, 8, 15], a new domain-specific language with few concise and intelligible primitives that have a precise correspondence with the distinctive elements of legal contracts. The language has a formal operational semantics, so

that the behavior is fully specified and amenable to automatic verification. Its current tool chain [9] contains a runtime environment, a type checker, a graphical IDE, and an analyzer verifying the reachability of clauses [10, 14].

In this paper, we advance the automatic verification of *Stipula* contracts by introducing a systematic approach for verifying contracts through translation into a general-purpose deductive verification tool. Specifically, we translate *Stipula* contracts into Java programs annotated with JML (Java Modeling Language) specifications, and we employ the KeY verification system [3, 5] as a backend. We define translation patterns and principles that preserve the semantics of core *Stipula* constructs — such as asset transfers, functions, and events — within the program logic supported by KeY. Our approach admits both manual and automated translation, and it accommodates a variety of contract types, including those with cyclic behavior and partial asset transfers.

It is by no means evident that a translation-based approach to verification will succeed. For deductive verification tools such as KeY to operate in a fully automated mode, the availability of sufficiently precise specification annotations is crucial. In the general case, such specifications need to be provided manually, which renders deductive verification inherently interactive and, consequently, costly. In the case of *Stipula*, every smart contract is associated with a well-defined automaton: the states represent control points of the contract, while the transitions correspond to its clauses, such as functions or events. This automaton-based representation makes the contract's behavior explicit and provides a natural foundation for reasoning about possible execution paths. Moreover, when the automaton of a contract satisfies the structural property of *disjoint cycles* – that is, cycles that do not share any states – it becomes possible to automatically synthesize suitable JML annotations, including guarantees and loop invariants. This synthesis step significantly reduces the annotation burden and paves the way for a high degree of automation in the verification process, thus demonstrating the potential of the translation-based approach in practice.

Nonetheless, there are limitations that affect full generality. Time-dependent behavior must be scheduled symbolically and evaluated statically. Our technique handles this by using symbolic boolean variables representing the time constraint. The event is then translated into a conditional guarded by the symbolic variable that KeY evaluates by exploring both possibilities: one where the event occurs and one where it does not. More complex forms of time management, such as dynamically registered events during loops or asynchronous behavior, are outside the current scope. Similarly, non-deterministic behavior, such as contracts where multiple transitions may be enabled concurrently, is not yet supported: our translation assumes that guards deterministically select a unique transition. These aspects pose challenges for extending the approach beyond the verified fragment we currently target.

The paper is structured as follows. Section 2 introduces the *Stipula* language and its execution semantics; an illustrative example highlights how obligations, permissions, and events are expressed in *Stipula*. We also provide a short introduction to KeY. Section 3 formalizes our translation methodology from *Stipula*

to Java, detailing how fields, assets, functions, events, and contract behavior are encoded in Java and specified with JML. Section 4 discusses the implementation of our translator tool and presents case studies that apply our approach to representative *Stipula* contracts, demonstrating the effectiveness of the method and discussing verification outcomes. Section 5 reviews related efforts in contract languages and formal verification. Section 6 concludes the paper and outlines directions for future work.

2 Background

To set the stage for our translation-based verification method, we briefly review the relevant background. Section 2.1 presents the *Stipula* language, emphasizing its constructs for assets, states, and timed events. Section 2.2 introduces the KeY system, which we later employ as the verification backend.

2.1 *Stipula*

A *Stipula* contract consists of a set of parties, states, assets, fields, and a set of functions and events, generically called *clauses*. The declaration of a contract is defined in Fig. 1, where $\mathtt{C}$ is the name of the contract, $\overline{\mathtt{h}}$ and $\overline{\mathtt{x}}$ are the *assets* and *fields*, respectively, $\overline{\mathtt{A}}$ are the *parties*. The **agreement** construct declares the

$$
\begin{array}{l}
\textbf{stipula}\ \mathtt{C}\ \{ \\
\quad \mathtt{asset}\ \overline{\mathtt{h}} \\
\quad \mathtt{field}\ \overline{\mathtt{x}} \\
\quad \textbf{agreement}(\overline{\mathtt{A}})\ \{ \\
\qquad \overline{\mathtt{A}_1}\ :\ \overline{\mathtt{x}_1} \\
\qquad \cdots \quad // \ \bigcup_{i\in 1..n} \overline{\mathtt{A}_i} \subseteq \overline{\mathtt{A}}\ , \quad \bigcup_{i\in 1..n} \overline{\mathtt{x}_i} \subseteq \overline{\mathtt{x}}\ , \quad \bigcap_{i\in 1..n} \overline{\mathtt{x}_i} = \varnothing \\
\qquad \overline{\mathtt{A}_n}\ :\ \overline{\mathtt{x}_n} \\
\quad \}\Rightarrow \mathtt{@Q} \\
\quad F \\
\}
\end{array}
$$

$$
\begin{array}{llll}
\textit{Functions} & F & ::= & \_ \mid \mathtt{@Q\ A}: \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\,(E)\{\,S\,;\,W\,\} \Rightarrow \mathtt{@Q}'\ F \\
\textit{Prefixes} & P & ::= & E \to \mathtt{x} \mid E \to \mathtt{A} \mid E \multimap \mathtt{h}, \mathtt{h}' \mid E \multimap \mathtt{h}, \mathtt{A} \\
\textit{Statements} & S & ::= & \_ \mid P\ S \mid \textbf{if}\,(E)\,\{\,S\,\}\,\textbf{else}\,\{\,S\,\}\ S \\
\textit{Events} & W & ::= & \_ \mid \textbf{now} + k \gg \mathtt{@Q}\,\{\,S\,\} \Rightarrow \mathtt{@Q}'\ W \\
\textit{Expressions} & E & ::= & v \mid X \mid E\,\mathsf{op}\,E \mid \mathsf{uop}\,E \\
\textit{Values} & v & ::= & n \mid \textbf{false} \mid \textbf{true} \mid s
\end{array}
$$

Fig. 1. Syntax of *Stipula*

parties that set the initial value of the fields and the initial state of the contract. For example, if the agreement has three parties $\mathtt{A}_1, \mathtt{A}_2, \mathtt{A}_3$ and the contract has

two fields $\mathtt{x}_1, \mathtt{x}_2$, if it declares $\mathtt{A}_1 : \mathtt{x}_1$ and $\mathtt{A}_2, \mathtt{A}_3 : \mathtt{x}_2$ then $\mathtt{x}_1$ will be set by $\mathtt{A}_1$ and $\mathtt{x}_2$ will be set upon agreement on the value between $\mathtt{A}_2$ and $\mathtt{A}_3$. When the agreement is concluded, the parties may invoke a function in F.

A *function* $\mathtt{@Q\ A}: \ \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\,(E)\{\,S;\ W\,\} \Rightarrow \mathtt{@Q'}$ can be invoked by a party $\mathtt{A}$ if the contract is in state $\mathtt{Q}$ and the guard E is *true*. The names $\overline{\mathtt{y}}$ and $\overline{\mathtt{k}}$ are the formal parameters of $\mathtt{f}$; they are kept separate because $\overline{\mathtt{y}}$ are field values while $\overline{\mathtt{k}}$ are asset quantities.

Function bodies are *statements* followed by *events*. The former include value transfers, asset movements, conditional logic, and field assignments. *Stipula* distinguishes between different transfer operations: field and message updates use the symbol $\rightarrow$ with the usual semantics of assignment, while asset transfers use the linear implication operator $\multimap$ to emphasize the conservation semantics. For instance, an expression like "$1 \multimap \mathtt{wallet}, \mathtt{Seller}$" denotes exclusive transfer of a unit in `wallet` to the `Seller` and, *at the same time*, the `wallet` is decreased by 1. In contrast "$\mathtt{code} \rightarrow \mathtt{Licensee}$" models non-exclusive passing of information. The operation "$\mathtt{wallet} \multimap \mathtt{wallet}, \mathtt{Seller}$" is always shortened to "$\mathtt{wallet} \multimap \mathtt{Seller}$".

Events $\mathtt{now} + k \gg \mathtt{@Q}\,\{\,S\,\} \Rightarrow \mathtt{@Q'}$, where k is either a natural number or a field name, define a statement S to be executed if, *after* k time units from the current execution, the contract is in the state $\mathtt{Q}$. If the event is executed, the contract will transition to the state $\mathtt{Q'}$.

We refer to article [8] for background on the design of *Stipula*, as well as its formal semantics. A comment about the model of time in *Stipula* may be useful for what follows. The model has a multiset of events to be executed; every event has a time value that is a natural number representing minutes. This number is computed when the event is created by replacing `now` with 0 in the expression $\mathtt{now} + k$ (recall that k is either a natural number or a field name). Time advances when the contract has no statements to execute and no events can be triggered. In such cases, a "tick" occurs, decrementing the time values in the multiset of events. Events with negative time values are discarded. Subsequently, any event whose initial state matches the current contract state and whose time value reaches zero may be scheduled for execution.[3]

In this paper we use the state transition models of *Stipula* contracts, called the *underlying automata*. The states of these automata are those of the *Stipula* contract. The transitions correspond to clauses and are labelled either with the function name (the party name is always omitted, for simplicity's sake we assume that function names are pairwise different) or with the event line number (*e.g.*, $\mathtt{ev}_{10}$ is the event at code line 10). Fig. 2 shows an underlying automaton.

We illustrate *Stipula* through two representative examples that allow us to highlight crucial *Stipula* features:

***normative permissions*:** functions are enabled only for specific parties in specific states;

***asset safety*:** assets are never duplicated or lost; transfers are explicitly encoded and conditional;

[3] The syntax of *Stipula* in [8] admits absolute time expressions like `"2022/1/1:00:15"`; these expressions are rewritten into terms $\mathtt{now} + k$ when the event is created.

***timed obligations*:** the `event` construct encodes deadlines and enforces compliance without external intervention;

***stateful logic*:** contract progress is encoded through explicit state transitions, supporting both branching and linear workflows.

Example 1 (The License Contract). Listing 1 defines a *license contract* that regulates a licensing transaction between a `Licensor` and a `Licensee`, with time-bound trial periods and the possibility to purchase or decline the license. In particular, the `Licensee` may request a trial and then decide whether to buy the license. If the `Licensee` does not purchase the license before the trial period expires, the contract terminates and the cost is automatically returned to the `Licensee`.

The contract begins with an **`agreement`** clause in Line 4, where both parties define the trial start time (`t_start`), its duration (`t_limit`), and the license `cost`. This mirrors the legal principle of mutual consent ("meeting of the minds"): no contract behavior is enabled until consensus is reached. Upon agreement, the first control state is `@Init` (Line 6).

```
1  stipula License {
2      asset balance, token
3      field t_start, t_limit, cost, code
4      agreement (Licensor, Licensee)(t_start, t_limit, cost) {
5          Licensor, Licensee : t_start, t_limit, cost
6      } => @Init
7      @Init Licensor: offer(x)[n] {
8          n —o token
9          x → code
10         now + t_start ≫ @Prop { token —o Licensor } => @End
11     } => @Prop
12     @Prop Licensee: activate()[b] (b == cost) {
13         b —o balance
14         code → Licensee ;
15         now + t_limit ≫ @Trial {
16                 balance —o Licensee
17                 token —o Licensor
18                 } => @End
19     } => @Trial
20     @Trial Licensee: buy()[] {
21         balance —o Licensor
22         token —o Licensee
23     } => @End
24 }
```

Listing 1. The License contract in *Stipula*

The *underlying automaton* of `License` is shown in Fig. 2. In the `Init` state, the `Licensor` may invoke `offer`, transferring a `token` (representing the license) into escrow and generating a license `code`. A scheduled `event` is simultaneously registered at `t_start` time units in the future to reclaim the token if the `Licensee` fails to act within the trial start window. The contract then moves to `Prop`. If

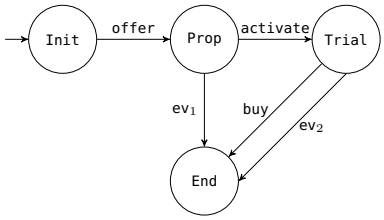

Fig. 2. Underlying automaton of the `License` contract

no function is called in this state, then time is advanced by one unit, eventually triggering the timeout at `t_start`. These event transitions are dynamically added to the state machine during function execution, *e.g.*, the two events in Fig. 2 are not initially present. This dynamic creation increases the expressiveness of *Stipula* [10].

In `Prop`, the `Licensee` can activate the trial by paying the `cost` (observe that `b` is an asset of the `Licensee`), which is transferred to the contract's `balance`: the contract acts as a notary for assets that are not finally disposed. Now the license code is revealed to the `Licensee` and another `event` is scheduled to handle the expiration of the trial period: if the license is not purchased before `t_limit` time units, the balance is refunded and the token is returned to the `Licensor`.

In the `Trial` state, the `Licensee` has the option to purchase the license via `buy`, which finalizes the transaction by transferring the `balance` to the `Licensor` and assigning the `token` permanently to the `Licensee`. In either case, the contract terminates in the `End` state, without retaining any asset.

Example 2 (The Deposit Contract). Listing 2 presents a *deposit contract* that models the interaction between a `Farm` and a `Client`. The `Farm` deposits flour, while the `Client` purchases and withdraws the corresponding amount at an agreed price. The contractual terms are enforced over a validity period of 365 days. The clause `send` allows the `Farm` to deposit flour into the contract's stock. In `send()[h]`, the assignment `h→Client` is only an informational message to the `Client` about the deposited amount (`h` is not emptied), while the asset transfer `h⊸flour` increases the contract's internal balance of flour (and, at the same time, empties the asset `h`). Observe that the converse ordering of the instructions (`h⊸flour  h→Client`) sends 0 to the `Client` as informational message. Also, no flour is transferred to the `Client` by send. Actual delivery occurs only with `buy()[w]`, where `(w/cost_flour)⊸flour, Client` transfers flour from the contract's stock to the `Client` (hence `w/cost_flour` must not be greater than `flour`) and `w⊸Farm` represents the payment to the `Farm`. In this way, send supplies the stock, whereas buy withdraws from it under payment.

```
1  stipula Deposit {
2      asset flour
3      field cost_flour
4      agreement (Client, Farm)(cost_flour) {
5          Client, Farm : cost_flour
```

```
6       } ⇒ @Start
7       @Start Farm : begin()[h]{
8           h ⟶ Client
9           h ⊸ flour;
10          now + 365 ≫ @RunF { flour ⊸ Farm } ⇒ @End
11          now + 365 ≫ @RunC { flour ⊸ Farm } ⇒ @End
12      } ⇒ @RunC
13      @RunF Farm : send()[h]{
14          h ⟶ Client
15          h ⊸ flour
16      } ⇒ @RunC
17      @RunC Client : buy()[w](w/cost_flour <= flour){
18          (w/cost_flour) ⊸ flour, Client
19          w ⊸ Farm
20      } ⇒ @RunF
21  }
```

Listing 2. The `Deposit` contract in *Stipula*

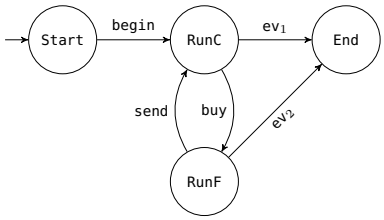

Fig. 3. Underlying automaton of the `Deposit` contract

Unlike the `License` contract, the `Deposit` contract has cyclic interactions: the `Farm` and `Client` can repeatedly invoke `send` and `buy` as long as the contract remains valid, as shown in the automaton in Fig. 3. This is modeled by the function `send` transitioning from `RunF` to `RunC` and the function `buy` in the reverse direction, enabling iteration until the validity period expires.

The alternation between `RunF` and `RunC` in Example 2 is by design: it ensures that each iteration of the cycle is well-formed and contributes to disjointness of cycles in the underlying automaton. Allowing the `Client` to buy repeatedly without intervening deposits would introduce overlapping cycles, which are outside the scope of the current automatic verification approach. Cycles increase the complexity of formal analysis [14]. In particular, this feature, combined with deadlines (the events) and permission-dependent transitions enlarge the state space and introduce subtle cross dependencies, making automated verification challenging.

Definition 1 (Cyclic Contracts and Disjoint Cycles). *A Stipula contract is* cyclic *if its underlying automaton has a sequence of pairwise different states*

$\mathtt{Q}_1, \ldots, \mathtt{Q}_n$ *with transitions* $\mathtt{Q}_i \xrightarrow{\mu_i} \mathtt{Q}_{i+1}$ *and* $\mathtt{Q}_n \xrightarrow{\mu_n} \mathtt{Q}_1$ *where* μ_i *are either function names or events. The contract has* disjoint cycles *if different cycles have no state in common.*

We focus on contracts whose automata contain only disjoint cycles — a structural property that simplifies path generation and enables modular, fully automatic verification. For example, the `Deposit` contract includes a single cycle, while `License` is entirely acyclic. Disjoint cycles ensure determinism: If a state has multiple outgoing transitions, at most one of them belongs to any given cycle. This allows our tool to synthesize all execution paths statically, without ambiguity or runtime checks.

While this restriction enhances tractability, it limits generality. Contracts with overlapping cycles are currently out of scope, though we discuss possible extensions in the conclusion. The current implementation does not yet enforce that cycles are disjoint; an algorithm for checking it is presented in the extended version of the paper [**?**] and will be integrated in a future version.

2.2 Deductive Verification with the KeY System

The KeY system [3, 5] is a deductive verification framework for Java that combines symbolic execution, invariant reasoning, and method contracts[4] on top of a calculus for an expressive program logic. The main use case of KeY is formal verification of Java programs annotated with specifications written in JML. It provides an interactive user environment, where one can construct correctness proofs of Java methods against their JML contracts. These contracts typically include preconditions (**`requires`**), postconditions (**`ensures`**), frame conditions (**`assignable`**), class invariants, and auxiliary annotations such as loop invariants and assertions. Given a JML-annotated Java class, KeY translates the specifications into logical proof obligations and attempts to discharge them using its symbolic execution engine. The details of the verification process are irrelevant for the purpose of this paper: KeY is used as a *black box*, the input is a JML-annotated Java file that results from translation of a *Stipula* contract. In general, KeY is used in interactive or auto-active mode, however, for all case studies discussed below, the verification is *fully automatic*.

3 Translation Approach

To enable formal verification of *Stipula* contracts, we define a systematic translation into Java code annotated with JML specifications. Java and JML are versatile enough to express *Stipula* in a natural manner: functions are modeled by stateful Java methods, the specific semantics of assets are modeled by suitable JML method contracts, etc.; the details are given below. Our translation

[4] Not to be confused with *Stipula* contracts. In this paper, both kinds of contract are featured, but it should be obvious from the context when we mean *Stipula* contracts or Java/JML method contracts.

preserves the semantics of *Stipula*'s normative constructs, including asset ownership, state transitions, permissions, and events, by representing them as verifiable Java/JML proof obligations.

3.1 Assets, Fields and States

We distinguish three categories of contract data: *fields*, *assets*, and the *control state*. Each is translated into corresponding elements in the target Java class, along with accompanying JML specifications to capture correctness properties.

Fields. *Stipula* contracts define mutable fields that store contract-specific state information, such as numeric counters, time stamps, or configuration parameters. These fields evolve during execution and influence both transitions and obligations. We must ensure that field updates are explicitly modeled and tracked in the translated Java code.

Definition 2 (Field Mapping). *Let $\mathcal{F}$ be the set of mutable fields declared in a* Stipula *contract. Each field* `f` $\in \mathcal{F}$ *is translated to a* `static` *Java field, which holds the contract state data. Changes to these fields are tracked and constrained using JML* `assignable` *clauses.*

This representation allows JML to reason about field updates and supports the specification of frame conditions through `assignable` clauses.

Assets. In *Stipula*, assets are classified automatically as *divisible* or *indivisible* to enable explicit ownership modeling and prevent asset duplication or loss. The translator infers this heuristically from usage: assets transferred with a variable quantity ("`v—∘ h, h'`") are treated as divisible and represented as an `int`, while those transferred directly ("`h—∘ h'`") are considered indivisible and represented as a `boolean`. To enforce exclusivity for indivisible assets, safety preconditions like "`requires !h.token`" are generated. Based on this classification, we now formally define how asset ownership is represented.

Definition 3 (Asset Ownership Representation). *Let $\mathcal{A}$ be the set of declared assets, and $\mathcal{P}$ the set of participants in a* Stipula *contract* `C`*. Each asset* `a` $\in \mathcal{A}$ *is represented by a set of Java fields:*

$$\{\texttt{P.a} \mid \texttt{P} \in \mathcal{P}\} \cup \{\texttt{C.a}\},$$

where each field `P.a` *is either of a numeric type (*`int`*) indicating the quantity of asset* `a` *held by party* `P` *or a boolean variable indicating that the party* `P` owns *asset* `a`*. The field* `C.a` *tracks the quantity owned by the contract itself.*

To ensure that each indivisible asset is held by exactly one owner at all times, we define an invariant that formalizes the exclusivity constraint.

Property 1 (Indivisible Asset Exclusivity). Each asset is owned by exactly one party in $\mathcal{P}$ or by the contract `C` at any given time. Formally, the following invariant must hold for all $\mathtt{a} \in \mathcal{A}$: $\bigvee_{X \in \mathcal{P} \cup \{\mathtt{C}\}} \left(X.\mathtt{a} \wedge \bigwedge_{Y \neq X} \neg Y.\mathtt{a} \right)$.

This invariant becomes part of the translated Java class as a static JML `invariant`, ensuring asset linearity throughout execution.

In our current encoding, exclusivity of indivisible assets is often enforced via explicit postconditions on methods. Alternatively, these properties can be stated as JML class invariants, which would provide a more uniform and modular treatment; we plan to explore this refinement in future versions of the tool.

Example 3. The asset `token` in the License contract in Section 2 is modeled by three Java fields: `License.token`, `Licensee.token`, and `Licensor.token`. Property 1 is ensured in JML as:

```
static invariant (License.token∧¬Licensee.token∧¬Licensor.token)
                ∨(Licensee.token∧¬License.token∧¬Licensor.token)
                ∨(Licensor.token∧¬License.token∧¬Licensee.token)
```

For divisible assets, rather than enforcing exclusivity, we ensure that the total quantity of the asset remains constant throughout the contract execution. This principle is captured by the following property, which generalizes the invariants used in examples such as currency transfers and resource tracking.

Property 2 (Divisible Asset Conservation). For each divisible asset $\mathtt{a} \in \mathcal{A}$, declared with a fixed total quantity $\kappa_{\mathtt{a}}$, the system ensures that the total amount held by all participants in $\mathcal{P}$ and the contract `C` remains constant:

$$\sum_{X \in \mathcal{P} \cup \{\mathtt{C}\}} X.\mathtt{a} = \kappa_{\mathtt{a}}$$

Example 4. The `Deposit` contract provides an illustration of Property 2. In this contract, the farmer transfers an amount `h` of the divisible asset `flour` to the contract by calling the `send(h)` function. Importantly, this transfer does not create assets: the amount `h` is subtracted from the farmer's `Farm.flour` field and added to the contract's `Deposit.flour` field. This is reflected in the generated JML postcondition:

```
/*@ public normal_behavior
   ...
   @ ensures     Deposit.flour == \old(Deposit.flour) + h
   @             && Farm.flour == \old(Farm.flour) - h
   @             && Client.flour == \old(Client.flour);
   ...
   @*/
   public final static void send(int h) {
       Deposit.flour = Deposit.flour + h;
       Farm.flour = Farm.flour - h;
   }
```

Thus, the total quantity of the asset `flour` remains constant across the system, as required by the conservation property. The complete translation of the `Deposit` contract, including this method, is shown in [**?**].

Functions. In *Stipula*, contract behavior is defined by transitions that correspond to functions and events. Each transition is enabled under specific conditions, such as the current control state, the invoking party, and the availability of required assets. And it may update fields, transfer assets, or trigger further state changes. To reflect this logic in Java, each *Stipula* function is translated into a static Java method with a formal specification in terms of a JML contract:

- the **`requires`** clause specifies the initial control state, party permissions, and preconditions over asset ownership and fields;
- the **`ensures`** clause describes the resulting state transition, updates to fields, and asset transfers;
- the **`assignable`** clause enumerates the state variables that may be modified.

Preconditions in the generated JML specifications are derived directly from *Stipula*'s guards, permissions, and asset requirements. In particular, they capture the enabling conditions for invoking a function, rather than being reconstructed backwards from the post-state.

Example 5. We illustrate the encoding of functions and states in JML with the `buy()` function in Listing 1. Observe that the invariant for asset exclusivity is implicitly present in the pre- and postcondition. Fully automatic verification of the contract in KeY, including invariant preservation, takes fractions of a second.

```
/*@ public normal_behavior
  @ requires   License.balance && License.token;
  @ ensures    Licensor.balance && !License.balance &&
  @            Licensee.token && !License.token;
  @ assignable Licensor.balance, License.balance, Licensee.token, License.token;
  @*/
  public final static void buy() {
     Licensor.balance = true;
     License.balance = false;
     Licensee.token = true;
     License.token = false;
  }
```

Definition 4 (Indivisible Asset Transfer Semantics). *Let* $a \in \mathcal{A}$ *be an indivisible asset of contract* C*, and* $P, Q \in \mathcal{P}$*. A transfer of* a *from* P *to* Q *is modeled by the following postcondition:*

$$\neg P.a \wedge Q.a \wedge \bigwedge_{R \in \mathcal{P} \cup \{C\} \setminus \{P,Q\}} (R.a = old(R.a))$$

where $old(R.a)$ *refers to the value of* $R.a$ *before the transfer is executed.*

This pattern guarantees that only the ownership of `P` and `Q` changes, preserving the asset exclusivity invariant of *Stipula*. In addition, all changes are confined to the locations occurring in the **`assignable`** clause. If we assume Property 1 to be ensured by the exclusivity invariant, then `Q.a` alone is sufficient as postcondition. Conditional statements and field updates in a *Stipula* function body are translated directly into Java code.

3.2 Time and Events

Timed clauses in *Stipula* define transitions that become enabled only once logical time reaches a specified value. Such a transition is written as

$$\texttt{now} + \texttt{k} \gg \texttt{@Q}\,\{\,\texttt{S}\,\}\,\Rightarrow\,\texttt{@Q}'$$

which expresses that, when time reaches **`now`** + `k` and the current state is `Q`, the contract may execute `S` and move to state `Q′`. In the Java translation, these time-dependent transitions are not realized by manipulating a global clock. Instead, each timed event is guarded by a symbolic boolean variable representing the time constraint. For example, a transition scheduled at a future time is translated into code of the form:

```
if (ev_event1) { event1(); return; }
```

Here, `ev_event1` is a symbolic guard introduced by the translator. Verification tools such as KeY handle this guard symbolically, thereby exploring both possibilities: one where the event occurs and one where it does not. Each event is mapped to a dedicated Java method, named systematically (`event1()`, `event2()`, ...), that encapsulates the corresponding body `S`.[5] A key point is that the symbolic guard `ev_event1` does not encode the source state `Q` explicitly. Instead, state constraints are enforced structurally: our translator automatically generates so-called *scenario methods* (see Section 3.4), which represent feasible execution paths. An event method is reachable only from the correct source state along such a path. In other words, the interplay between scenario construction and method preconditions guarantees that events can fire only in the intended states.

All event methods are statically defined during translation and never generated dynamically at runtime. This strategy eliminates the need for explicit clocks or schedulers: time-dependent behavior is captured entirely through symbolic guards. As a result, temporal reasoning can be carried out using standard symbolic verification tools. Furthermore, static analyses can identify and prune unreachable event branches, improving efficiency. For instance, the technique described in [14] can be integrated into our translator, and we plan to pursue this as future work.

[5] Deterministic event names are assigned via an internal counter to guarantee consistency and traceability.

3.3 Cyclic Behavior and Loop Translation

When translating a *Stipula* contract whose underlying automaton exhibits disjoint cycles, each cycle is mapped to a dedicated Java `while` loop. Conceptually, the body of the loop corresponds to a single traversal of the cycle, *i.e.*, one complete execution of the contract operations contained within it. During each iteration, the loop updates both the contract's asset variables and its control-state fields to reflect the effect of the executed operations. Loop execution is not governed by explicit counters hard-coded in the program, but by symbolic scalar variables introduced during translation. These variables represent iteration bounds and loop counters, and they serve two complementary purposes:

1. *operational control*: they determine when and how many times the loop body may be executed;
2. *specification support*: they provide the basis for precise JML annotations, such as loop invariants and postconditions, that capture the intended effect of repeated executions.

This combination ensures that the generated Java code remains faithful to the original contract semantics while enabling deductive verification tools to reason soundly about all possible iterations. By structuring cycles as loops annotated with symbolic constraints, the translation bridges the gap between the automaton view of contracts and the logic-based reasoning frameworks used in verification.

Example 6. Consider the `Deposit` contract in Listing 2. The translator generates scalar parameters `h`, `w`, `h_send`, and `counter` to represent single asset transfer amounts and the number of iterations, respectively. The generated code implements the cyclic behavior as follows: For instance, one part of the invariant states that the amount of `flour` held by the contract evolves according to the sum of incoming deposits minus outgoing uses:

```
/*@ loop_invariant
  @ ... flour == \old(flour) - i * w/cost_flour + i * h_send; ...
  @*/
while (i < counter) {
  buy(w);
  send(h_send);
  i++;
}
```

Other parts of the invariant reflect the evolution of the farm and client asset fields (`Farm.flour`, `Client.flour`, etc.) and include standard loop annotations such as bounds and decreases clauses. The translator automatically generates the full invariant by analyzing how each variable changes during loop execution (see [**?**] for the complete form).

It is important to emphasize a limitation of our current encoding strategy. Parameters such as `w` in Example 6 are treated as symbolic constants throughout the entire execution of a loop. In other words, all iterations are analyzed under

a single symbolic instantiation of the parameter. While the semantics of *Stipula* would, in principle, permit different values of `w` across different iterations, verifying such non-deterministic behavior would require loop invariants that quantify over sequences of iteration-dependent values. At present, verification tools such as KeY cannot discharge such invariants automatically. Supporting this more general setting remains theoretically possible by allowing user-supplied invariants and resorting to interactive proofs, but this goes beyond the scope of our fully automated approach.

We also note that the loop in Example 6 is bounded by the symbolic parameter `counter`, which provides an upper bound on the number of iterations. Our construction associates a decreasing variant, namely `(counter - i)`, which enables KeY to automatically prove termination for contracts whose automata consist of disjoint cycles. For more intricate cyclic structures, where such simple variants are not available, proving termination would require richer annotations and remains an open direction for future work.

Finally, regarding correctness, the loop invariants generated by our translation suffice to establish preservation of the basic asset and state properties in the examples we studied. In general, however, automatically synthesized invariants are intentionally conservative: they guarantee soundness but may be incomplete, in the sense that they do not allow KeY to prove every conceivable postcondition. Our design philosophy prioritizes invariants that are simple, structurally derived, and always generated automatically. This choice ensures robustness and automation for typical *Stipula* contracts, while leaving open the possibility of user-supplied annotations in cases where more complex behaviors demand stronger reasoning power.

3.4 Scenario-Based Reasoning

To verify behavioral properties across complete execution paths within a *Stipula* contract, we declare *scenario methods* that represent legal sequences of contract actions. These methods model complete paths through the underlying automaton of a *Stipula* contract, from initial to final states. Each such method is annotated with a JML contract summarizing its overall effect, including field updates, asset transfers, and control state progression.

Example 7. The following scenario method models the successful completion of the contract in Listing 1. It ensures that the assets are swapped as expected. During verification, the already proven contracts of the called methods are used, the methods bodies need not be inlined.

```
/*@ requires Licensor.token && Licensee.balance ;
  @ ensures  Licensee.token && Licensor.balance ;*/
  public static void success() {
    offer(); activate(); buy();
  }
```

When a contract exhibits disjoint cycles or branching behavior that may lead to structurally distinct executions, our translator automatically generates a dedicated scenario method for each feasible path. Each scenario method corresponds to a linearized execution trace, thereby capturing one possible evolution of the contract. The generation process begins with a static analysis of the contract's control structure. In particular, the translator verifies that the clauses of the contract give rise to disjoint control paths. This is achieved by constructing the set of linear traces induced by the underlying automaton and checking that the cycles explored along each trace do not overlap. Once this structural property has been established, the translator can safely emit one scenario per disjoint trace, with each scenario representing a distinct and non-interfering contract behavior. This design has two key benefits. First, it makes the possible executions of the contract explicit, which facilitates automated reasoning about assets, states, and temporal constraints. Second, by isolating disjoint traces into independent scenario methods, it prevents the combinatorial explosion that would arise from interleaving overlapping cycles.

Example 8. Consider a `Loan` contract that regulates a sequence of financial operations between a `Client` and a `Bank`.

```
@Start Bank : give_money()[w](w == amount) {
    w —∘ Client ;
    now + 30 >> @Pay1 {  } ⇒ @Fail
} ⇒ @Pay1

@Start Bank : withdraw()[u](u == amount * interest_rate) {
    u —∘ Client
    "The_Bank_withdraws" → Client;
} ⇒ @Withdraw
```

In particular, the `Client` and the `Bank` agree on the `amount` of the loan and the `interest_rate`. They also agree on a number of installment payments and on a conditional early withdrawal by the `Bank`. If the `Bank` exercises the early withdraw then it has to pay a penalty to the client that is equal to `amount * interest_rate`. (The complete code with three payment installments is available in [**?**].) This contract exhibits two disjoint execution paths: one where the `Client` proceeds through all the payment steps, and another where the `Bank` initiates an early withdrawal. The two execution paths are represented by the following two scenarios produced by the translator:

```
public final static void seq1() {
  give_money(w);
  if (ev_event1) { event1(); return; }
  pay_installment1(h1);
  if (ev_event2) { event2(); return; }
  pay_installment2(h2);
  if (ev_event3) { event3(); return; }
  pay_installment3(h3);
}
```

```
public final static void seq2() {
  withdraw(u);
}
```

4 Implementation and Evaluation

To evaluate our translation and verification method, we developed a prototype translator available in the online repository [13]. The tool, implemented in approximately 2,000 lines of Java, uses ANTLR4 for parsing and a listener-based traversal to extract contract components such as parties, assets, fields, states, and transitions. We assess its effectiveness in this section by analyzing four representative *Stipula* contracts:

Betting: illustrating branching resolution logic based on external outcomes;
Deposit: modeling recurring resource exchanges between client and provider, with timed fallback events and cyclic asset flows;
Loan: encoding installment-based repayment with symbolic arithmetic over loan parameters and time-triggered enforcement mechanisms;
License: involving timed obligations and conditional asset transfers.

These case studies highlight a range of relevant aspects of *Stipula* contracts, including asset transfers, exclusive ownership, timed events, and cyclic interactions. While not exhaustive, this selection demonstrates that our approach can handle several of the main constructs found in practice. Each contract has been automatically translated using our tool chain (see [**?**] for the *Stipula* source and generated Java+JML code), and verified using KeY in automatic mode. No manual proof steps or user-supplied annotations were required. The performance results reported were obtained on a MacBook Pro (2023) equipped with an Apple M2 Pro processor and 16 GB of RAM.

We focus on contract properties that are automatically generated together with the Java+JML code and verified without user interaction. These include functional correctness, loop invariants, total termination, and symbolic handling of time-triggered transitions. Custom properties could be added manually as additional postconditions, but are not required for the scenarios we test. The following verification goals are considered:

P1: Functional Correctness Each scenario method satisfies the expected final state as specified by the automatically generated `ensures` clauses.
P2: Loop Termination All loops are annotated with decreasing variant terms that are sufficient to prove termination.
P3: Loop Invariant Preservation Automatically generated inductive invariants ensure that key state relationships are preserved across iterations.
P4: Time-Guard Soundness Event-triggered clauses are guarded by symbolic boolean variables (e.g., `ev_event1`), ensuring that transitions corresponding to timed clauses occur only under valid scheduling conditions.

Table 1 summarizes the properties successfully verified for each contract. Verification time reflects the duration reported by KeY when proving the main top-level scenario. Verification times for other scenarios tend to be somewhat less and the times to verify individual function and event contracts are negligible.

Contract	P1	P2	P3	P4	Time
`Betting`	✓	–	–	✓	∼2.1s
`Deposit`	✓	✓	✓	–	∼1.3s
`Loan`	✓	–	–	✓	∼1.8s
`License`	✓	–	–	✓	∼0.7s

Table 1. Properties verified in the case studies. A dash ("−") denotes that no property of the corresponding category applies to the given contract.

For the `Betting` contract, the outcome logic is expressed through mutually exclusive branches in its `data()` method. Verification ensures that all valid symbolic inputs result in safe asset redistribution, and that execution halts in a consistent final configuration.

In the `Deposit` contract, cyclic interactions between client and provider are translated into a **`while`** loop with a symbolic iteration bound (`counter`). Loop invariants preserve the consistency of transferred assets across rounds, while a variant term guarantees termination. Timed transitions are abstracted via symbolic events (e.g., `ev_event1`, `ev_event2`), which may interrupt execution early.

The `Loan` contract models installment-based repayment with time constraints. The translated code verifies that each installment is transferred correctly and only when the corresponding symbolic time guard holds. The contract's structure supports properties **P1** and **P4**.

In the `License` contract, verification ensures that all conditional obligations and asset transfers, such as license activation or revocation, occur only under valid scheduling conditions. All possible contract outcomes are covered, based on symbolic inputs like price, deadlines, and initial ownership.

5 Related Work

The formal modeling and verification of digital contracts receives increasing attention, especially at the intersection of legal informatics, programming languages, and formal methods. Our work contributes by bridging a legal domain-specific language (*Stipula*) with a deductive verification framework (KeY) through translation into JML-annotated Java.

Legal modeling frameworks like Ergo [19], OpenLaw [20], Lexon [16], and Accord [18] embed contracts into broader systems, but they lack a precise formal semantics. *Stipula* offers an operational model with explicit permissions, assets, and timed clauses. While Catala [17] formalizes legislative logic and has a runtime environment, it does not address contract verification.

Closer to our work, prior efforts explored translation-based verification: OCL-to-Java with JML [12], and Circus-to-Java for formal reasoning [11]. Our approach applies this paradigm to legal contracts, preserving their normative and temporal semantics.

While the current *Stipula* prototype is a centralized Java application [9], the language is implementation-agnostic and could be compiled to blockchain smart contracts like Solidity [1] or Obsidian [2]. This path is promising, as our verification approach using KeY [3] can build on established work that already applies the system to blockchain platforms like Hyperledger Fabric [6] and Solidity via the SolidiKeY prototype [4]. Using these advances would enable rigorous verification of *Stipula* contracts on decentralized infrastructures.

6 Conclusion

We presented a translation-based approach for verifying *Stipula* contracts by translating them into JML-annotated Java and applying deductive verification. This enables reasoning over normative properties — permissions, asset transfers, state transitions, and timed clauses — using existing tools. What we verify in this setting is the correctness of the Java/JML encoding of *Stipula*'s semantics, derived automatically from generation of scenarios, rather than an independent abstract notion of functional correctness of contracts. Allowing users to state additional derived properties directly in *Stipula* would be a natural extension, but it is beyond the scope of this work. Alternative paradigms based on state-based formalisms (e.g., statecharts or timed automata) could in principle support reasoning in temporal logics over possible behaviors. We chose a deductive setting with KeY to leverage its mature automation and direct JML support, while exploring complementary verification approaches remains a promising avenue for the future.

Our translation targets an expressive yet analyzable fragment of *Stipula*, with symbolic time, disjoint loops, loop-free clauses, and limited non-determinism. Within this fragment, verification is fully automatic and requires no manual annotation or interactive proof. Case studies confirm that functional and temporal behavior can be verified compositionally, suggesting that legal contracts can be verifiable by design if execution semantics is preserved.

The current approach synthesizes scenario methods by statically traversing the contract automaton. This relies on a structural restriction: all cycles must be disjoint. Disjointness ensures determinism within each cycle and enables complete path generation without ambiguity or runtime checks, making verification fully automatic and modular. Future work will focus on lifting this restriction to support overlapping cycles and more general control flow. One possible direction is to introduce dynamic scheduling mechanisms, such as a *dispatch* table, that can track and trigger enabled clauses at runtime. While this would increase expressiveness, it also introduces verification challenges that may require interactive proofs or hybrid verification strategies.

Deductive tools like KeY can often produce counterexamples, which, when mapped back to the legal domain, help authors detect inconsistencies and refine their contracts. Another direction is integrating runtime or hybrid verification to support cases where full deductive reasoning is infeasible.

Acknowledgments We thank Maximilian Scheid for his work on the implementation of the translator.

References

1. Solidity Documentation: State Machine Common Pattern. https://docs.soliditylang.org/en/v0.8.0/common-patterns.html#state-machine
2. Obsidian: A safer blockchain programming language. Language Site at http://obsidian-lang.com/ (2018)
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification: The KeY Book, LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
4. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: ISoLA (3). Lecture Notes in Computer Science, vol. 12478, pp. 9–24. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_2
5. Beckert, B., Bubel, R., Drodt, D., Hähnle, R., Lanzinger, F., Pfeifer, W., Ulbrich, M., Weigl, A.: The Java verification tool KeY: A tutorial. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Proc. 26th Intl. Symp. on Formal Methods, Milan, Italy. LNCS, vol. 14934, pp. 597–623. Springer, Cham (Sep 2024). https://doi.org/10.1007/978-3-031-71177-0_32
6. Beckert, B., Herda, M., Kirsten, M., Schiffl, J.: Formal specification and verification of hyperledger fabric chaincode. In: 3rd Symp. on Distributed Ledger Technology (SDLT), Gold Coast, Australia, November 12, 2018. pp. 44–48. Institute for Integrated and Intelligent Systems (2018)
7. Crafa, S., Laneve, C.: Programming legal contracts - A beginners guide to stipula. In: The Logic of Software. A Tasting Menu of Formal Methods. Lecture Notes in Computer Science, vol. 13360, pp. 129–146. Springer (2022). https://doi.org/10.1007/978-3-031-08166-8_7
8. Crafa, S., Laneve, C., Sartor, G., Veschetti, A.: Pacta sunt servanda: Legal contracts in *Stipula*. Science of Computer Programming **225**, 102911 (2023). https://doi.org/10.1016/j.scico.2022.102911
9. Crafa, S., Laneve, C., Veschetti, A.: Stipula Prototype (July 2022), available on github: `https://github.com/stipula-language`
10. Delzanno, G., Laneve, C., Sangnier, A., Zavattaro, G.: Decidability problems for micro-stipula. In: COORDINATION. Lecture Notes in Computer Science, vol. 15731, pp. 133–152. Springer (2025). https://doi.org/10.1007/978-3-031-95589-1_7
11. Freitas, A.F., Cavalcanti, A.: Automatic translation from *Circus* to Java. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) Formal Methods, 14th Intl. Symp. on Formal Methods, Hamilton, Canada. LNCS, vol. 4085, pp. 115–130. Springer (2006). https://doi.org/10.1007/11813040_9
12. Hamie, A.: Translating the Object Constraint Language into the Java modelling language. In: Haddad, H., Omicini, A., Wainwright, R.L., Liebrock, L.M. (eds.) Proc. of the ACM Symp. on Applied Computing (SAC), Nicosia, Cyprus. pp. 1531–1535. ACM (2004). https://doi.org/10.1145/967900.968206
13. HÃd'hnle, R., Laneve, C., Veschetti, A.: Tool Implementation Prototype (September 2025), https://github.com/stipula-language/stipula/tree/master/Stipula-KeY-Tool
14. Laneve, C.: Reachability analysis in Micro-Stipula. In: Proc. 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024. pp. 17:1–17:12. ACM (2024). https://doi.org/10.1145/3678232.3678247

15. Laneve, C., Parenti, A., Sartor, G.: Legal contracts amending with Stipula. In: Jongmans, S., Lopes, A. (eds.) Coordination Models and Languages, 25th IFIP WG 6.1 Intl. Conf., COORDINATION, Lisbon, Portugal. LNCS, vol. 13908, pp. 253–270. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-35361-1_14
16. Lexon Foundation: Lexon Home Page. http://www.lexon.tech (2019)
17. Merigoux, D., Chataing, N., Protzenko, J.: Catala: A programming language for the law. Proc. ACM Program. Lang. **5**(ICFP), 1–29 (Aug 2021). https://doi.org/10.1145/3473582
18. Open Source Contributors: The Accord Project. https://accordproject.org (2018)
19. Roche, N., Hernandez, W., Chen, E., Siméon, J., Selman, D.: Ergo - a programming language for smart legal contracts. CoRR **abs/2112.07064** (2021), https://arxiv.org/abs/2112.07064
20. Wright, A., Roon, D., ConsenSys AG: OpenLaw Web Site. https://www.openlaw.io (2019)