Transition-Oriented Programming: Developing Verifiable Systems

Yepeng Ding and Hiroyuki Sato

The University of Tokyo, Tokyo, Japan
{youhoutei,schuko}@satolab.itc.u-tokyo.ac.jp

Abstract. It is extremely challenging to develop verifiable systems that are regulated by formal specifications and checked by formal verification techniques in practice. Although formal verification has made significant progress over the past decades, the issue caused by the gulf between the system implementation and formal verification still has a huge cost. To fundamentally solve the issue, we propose transition-oriented programming (TOP), a novel programming paradigm, to instruct developers to develop verifiable systems by thinking in a formal way. TOP introduces the theories of the transition system as the joint of the implementation and formal verification to promote formal thinking during development. Furthermore, we propose a novel programming language named Seni to support the TOP features. We argue that TOP is useful and usable to develop verifiable systems in a wide range of fields.

Keywords: Programming paradigm · Software engineering · Verifiable system · Formal verification · Transition system.

1 Introduction

Formal verification is a powerful method of ensuring the satisfaction of the formal specifications, which supports in designing and implementing verifiable and provably correct systems and protocols. Typical methods of formal verification include model checking and theorem proving. There have been lots of useful tools that implement these methods such as SPIN [10], NuSMV [3], JKIND [8], and AdamMC [7] for model checking, or Coq [18], Isabelle [14] for theorem proving.

However, current formal verification tools usually have their own domain-specific languages and best practices for modeling and verification, which leads to the huge gulf between verification and implementation. For instance, the verification enforced by the SPIN model checker depends on the formal modeling of the system with Promela [11], while the system is implemented by some general-purpose languages such as Java or C++. The gulf highly increases the cost of verification during the development, which is reflected in three aspects.

– Design cost. It lacks unitarity between the implementation and verification, which increases the complexity of the system design due to the necessity of the compromise.
– Maintenance expense. It requires at least two sets of codes describing the same system merely from different perspectives: implementation and verification.
– Labor cost. The use of formal verification requires domain knowledge, which usually needs additional specialists to collaborate with developers.

Our motivation is to bridge the gulf between the formal verification and implementation to facilitate developing verifiable and provably correct systems with less cost. To remedy the situation, we introduce transition that is a concept originated from the transition system as the joint of the implementation and verification. The concept of the transition system has been well studied in theoretical computer science. It is of fundamental importance for the abstraction of both the sequential and concurrent systems, which makes it capable of guiding the implementation of systems in a rigorous manner. Besides, the transition system plays an imperative role in formal verification, especially in model-checking techniques. It is trivial to check the correctness and properties of the abstraction compared to the implementation.

In this paper, we propose a novel programming paradigm called transition-oriented programming (TOP) that instructs developers to develop verifiable systems by thinking in the transition system. With the theories of the transition system, TOP promotes formal thinking during the development by regarding real-world systems as transition systems. The system behavior is rigorously controlled by the formal specification. Besides, TOP supports both vertical modularity (refinement) and horizontal modularity (modularization) that are imperative in modern software engineering. Furthermore, the parallelism is also well supported in TOP to design, implement, and verify complex systems. Additionally, we propose a general-purpose programming language named Seni, the first language to support TOP.

We summarize the main advantages of TOP as follows:
1. TOP enables formal thinking during the development to ensure the rigorous implementation and facilitate formal verification.
2. By using TOP, programmers can develop verifiable systems to ensure the correctness and properties in a natural way.
3. TOP reduces the cost of formal verification by circumventing the issues caused by the gulf between the implementation and verification.

The remainder of this paper is organized as follows. Section 2 briefly reviews related work. Section 3 presents the core concepts and features of TOP, followed by the introduction of Seni to support TOP in Section 4. We discuss the vision of TOP in the application in Section 5. Section 6 concludes the paper.

2 Related Work

Our work is inspired by the transition system theory that plays an important role in formal methods [4,1,19,2,5,16]. With the support of the transition system theory, many tools of formal methods have been developed such as model checkers [10,20,3] and SMT solvers [6,12].
To improve the usability of these tools and understandability of the application of formal methods, many front-end frameworks such as translators and wrappers have been developed. Java PathFinder [9] is a translator from Java to Promela, which allows programmers to annotate the Java program with assertions and verify them with the SPIN model checker after the translation to Promela. PlusCal [13] provides a pseudocode-like interface for the TLA+ specification language, which makes the mathematical specification easier for programmers to understand. Wrappers also facilitate the use of some tools such as various bindings of Z3 for various programming languages.

However, these front-end frameworks focus on addressing issues of the usability and understandability of the verification. Perhaps other works such as state-oriented programming [17] have similar interfaces as TOP, but they illustrate completely different ideas. To the best of our knowledge, there has not been a work focusing on bridging the gulf of the implementation and formal verification from the perspective of the programming paradigm.

3 Transition-Oriented Programming

This section focuses on the main concepts and features of TOP. The conceptual model of TOP is visualized in Figure 1. The TOP paradigm supported with a transition-oriented programming language aims to implement verifiable systems with the integration of built-in formal methods such as formal specification, formal verification, and formal testing.
As a programming paradigm, TOP highlights formal thinking during the
development. It regards every real-world system as a transition system. Based
on the fundamental theories of the transition system, TOP presents many useful
features and makes the formal verification more usable in practice.

3.1 Transition System

The term transition connects the static state with the dynamic change within
a system. In TOP, a minimal program unit is a transition system, which is also
called an abstraction.

Definition 1 (Transition system). In TOP, a transition system $\mathcal{S}$ over set $V$ of typed state variables and set $F$ of pure functions is defined as follows:

\[
\mathcal{S} \triangleq \langle S, A, \rightarrow, I, P, L \rangle,
\]

where

- $S$ is a set of states that are determined by $[V]$, the set of evaluations of state variables,
- $A$ is a set of actions that determines the modifications of state variables with pure functions,
- $\rightarrow \subseteq S \times A \times S$ is a transition relation,
- $I \subseteq S$ is the initial state that is determined by the set of initial evaluations of state variables,
- $P$ is a set of propositions,
- $L \triangleq S \mapsto 2^P$ is a labelling function.

It is adequate to implement computable systems with the transition system
structured in TOP by abstracting the observable states and actions. Furthermore, states are bond with propositions by the labeling function, which is the key to the connection between the implementation and verification in TOP.

It is noteworthy that the concept of the state variable is different from the ordinary variable. State variables are used to describe states in the transition system while ordinary variables are used to implement certain functions. Besides, states variables can only be modified by actions while ordinary variables can be modified in their scopes including actions and pure functions.

The pure function is used to assist the implementation of actions to improve
the reusability, which cannot modify the state variable because it has no side
effects. Hence, a set of pure functions are usually embedded in an action body
or defined globally to perform tasks. Different from the pure function, the action
is used to specify the system behaviors by modifying state variables. It does not
implement concrete tasks but declares the way of the state transition.

3.2 Specification-Driven Control Flow

For simplicity, we will use the term specification to refer to the formal specifica-
tion. In TOP, a specification is composed of a set of actions and sub-systems,
which is formulated in the temporal logic. It forces the developers to construct
unambiguous, complete, and minimal specifications to describe the system behavior rigorously.

For instance, the behavior of a twinkling LED light can be specified as $S = \Box(SwitchOn, SwitchOff)$ where $SwitchOn, SwitchOff \in A$. Here, $\Box$ is a temporal modality that can be pronounced as always, which means that the actions in this scope will be executed now and forever in the future. The comma is used to define the ordering relation between actions.

A system must contain at least one specification to represent the main control flow. One specification can be decomposed into multiple smaller ones to specify different types of system behaviors, which is supported by the logical connective.

It is noteworthy that all actions in TOP can only be triggered in specifications, which means that all operations associated with the modification of state variables are specified in TOP specifications. In this manner, the system behavior is rigorously controlled.

### 3.3 Modularity

TOP puts a high value on modularity to improve flexibility and reusability. We divide the modularity of TOP into two types: vertical modularity and horizontal modularity. Vertical modularity refers to the support of the refinement, which allows programmers to describe the transition system at different abstraction levels while preserving the equivalence or preorder relations. Horizontal modularity allows the programmers to modularize the system by decomposing a large transition system into modules to improve reusability. Generally, a complete transition system contains a set of modules with multiple abstractions, which is shown in Figure 2.

![Fig. 2. The illustration of the modularity within a transition system.](image)
With the vertical modularity, it is reasonable to start from the abstraction and land on the concrete implementation by a set of refining iterations, which is supported by the simulation theory of the transition system. For instance, if a more detailed transition system $S$ refines $S'$, then the simulation order $S \succeq S'$ holds, which means that transition system $S$ is simulated by $S$. Due to the reflexivity and transitivity of $\succeq$, there can be a set of refinements at different abstraction levels. Additionally, if $S \succeq S'$, then these two transition systems are simulation-equivalent.

In the meanwhile, TOP supports the import of encapsulated sub-systems. The pure functions globally defined in the imported sub-systems can be reused in the actions while the imported sub-systems can be reused in the specifications.

### 3.4 Parallelism

In TOP, parallelism is well supported by the operational models of the transition system including interleaving, concurrency, synchronous, and asynchronous message passing. It is flexible to describe a wide range of parallel real-world systems, especially distributed systems.

The specification in a transition system can specify the behavior of the parallel composition of a set of transition systems, which is supported by the horizontal modularity of TOP. For instance, an effective specification for a parallel system can be represented as $S_1 \parallel S_2 \parallel \cdots \parallel S_n$.

### 3.5 Intrinsic Verification

The transition system makes the joint of the implementation and verification. TOP has native support for defining propositions, properties, and formulae. All defined propositions are attached to the state by the labeling function $L$ automatically by observing the state variables. In this manner, some techniques of formal verification can be integrated naturally.

For instance, TOP allows developers to check system properties statically with a built-in model checker during the compile-time. The properties can be formulated into temporal logics such as linear-time, branching-time, and real-time logics. Take the twinkling LED light as an example, we can define two propositions $isOn$ and $isOff$ for the twinkling LED light as follows:

$$isOn \triangleq \llbracket isEnable \rrbracket = T,$$
$$isOff \triangleq \llbracket isEnable \rrbracket = F.$$

In this manner, the property $S \implies \Box (isOn \lor isOff)$ can be verified during the compile-time.

Furthermore, runtime verification can be also supported by TOP to avoid the complexity of the model checking by interacting with the executing transition system.

Another instance is the support of the SMT solver. In TOP, developers can check the satisfiability of a certain formula composed of a set of propositions in
first-order logic. For example, if we declare three Boolean state variables \( r \), \( g \), \( b \) to represent the status of red, green, and blue light respectively, the satisfiability of formula \([r] \land [g] \rightarrow [b] = \top\) can be determined with satisfying models.

4 Seni Language Ecosystem

To make TOP usable in practice, we propose Seni, the first transition-oriented programming language, together with its ecosystem.

4.1 The Seni Programming Language

The design of Seni integrates many features to facilitate the development of verifiable systems under the TOP paradigm. Seni is defined and implemented rigorously with formal semantics [15]. Besides, Seni introduces the concept of the pure function in functional programming to implement the action and the concept of the record to define the object structure. Furthermore, Seni aims to provide a user-centric interface for the understandability and maintainability with a set of useful syntactic sugars and reserve words for specification, implementation, and verification.

The main features of Seni are summarized as follows.

Transition-oriented. In Seni, it fully supports the TOP paradigm by organizing the program as the transition system with declarations of state variables, actions, initialized action, specifications, propositions, formulae, and properties. The main control flow is driven by the specification formulated in temporal logics. For modularity, Seni reserves the word refine to refine a system at a more detailed abstraction level and the word import to import functions or sub-systems.

General-purpose. Seni is designed to be capable of developing verifiable systems in a wild variety of application domains.

Type-safe. Seni is a strongly statically typed language supported by a type system with type inference. It enforces statically type checking to prevent type errors and contributes to program correctness.

Verification-integration. In Seni, it has a built-in verification mechanism supported by techniques of formal methods to verify properties formulated by programmers.

Parallelism-support. Seni allows programmers to define several types of parallel systems including interleaving systems, concurrent systems, and message-passing systems, which is well supported by the parallelism theory of the transition system. Furthermore, the properties of parallel systems are also verifiable.
4.2 Ecosystem

Seni is a compiled language that requires the compiler to generate the intermediate code running on a virtual machine. The model checker and SMT solver are also integrated into the compiler toolchain for verification.

Furthermore, the standard libraries are essential for the actual development such as the IO library, network library, Math library, GUI library, etc.

5 Discussion

In this section, we discuss the vision for the application of TOP.

Compiler. In TOP, it is trivial to describe and implement automata and related algorithms that play a major role in compiler construction. Most importantly, TOP satisfies the safety requirement of the compiler implementation by the support of formal verification. Additionally, a TOP language is generally capable of bootstrapping.

Distributed system. With the support of modularity and parallelism, distributed systems with complex architectures can be well designed and implemented in TOP.

Decentralized system. With the advancement of blockchain technology, numerous decentralized systems have been developed that highly require security assurance due to the lack of central supervision. TOP facilitates the development of verifiable decentralized systems to help to solve trust issues.

Artificial intelligence (AI) system. As AI techniques become ubiquitous, safety and explainability have attracted huge attention. TOP provides a rigorous way of developing AI systems, especially knowledge-based systems with the support of formal knowledge representation.

Protocol design. In TOP, the specifications and properties are formulated in temporal logics, which facilitates the design and verification of the protocol. With the support of message-passing in parallelism, TOP can handle the design and implementation of distributed protocols.

Hardware design. It is vital to prevent errors in hardware design due to the high fabrication costs. The transition system can represent many models in hardware design such as circuits. Meanwhile, formal verification has been widely used in hardware design. Therefore, TOP provides a feasible way of making rigorous hardware design with the support of the handy implementation of the transition system and enforcements of formal verification.
6 Conclusion

We have presented TOP, a novel programming paradigm for developing verifiable systems. It bridges the gulf between the implementation and formal verification by instructing developers to think in the transition system during the development, which reduces the cost of applying formal verification in practice. Besides, we have proposed Seni, the first transition-oriented programming language, to practicalize the TOP paradigm. Furthermore, we have discussed the vision of the application of TOP that has the potential to dramatically change the way of developing verifiable and provably correct systems in a wide range of fields.

References

1. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
2. Bradley, A.R.: SAT-based model checking without unrolling. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer (2011)
3. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: International Conference on Computer Aided Verification. pp. 359–364. Springer (2002)
4. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM transactions on Programming Languages and Systems (TOPLAS) 16(5), 1512–1542 (1994), ISBN: 0164-0925 Publisher: ACM New York, NY, USA
5. Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model checking. MIT press (2018)
6. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
7. Finkbeiner, B., Gieseking, M., Hecking-Harbusch, J., Olderog, E.R.: AdamMC: A model checker for Petri nets with transists against Flow-LTL. In: International Conference on Computer Aided Verification. pp. 64–76. Springer (2020)
8. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JK ind model checker. In: International Conference on Computer Aided Verification. pp. 20–27. Springer (2018)
9. Havelund, K.: Java PathFinder A Translator from Java to Promela. In: International SPIN Workshop on Model Checking of Software. pp. 152–152. Springer (1999)
10. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on software engineering 23(5), 279–295 (1997), ISBN: 0098-5589 Publisher: IEEE
11. Holzmann, G.J., Lieberman, W.S.: Design and validation of computer protocols, vol. 512. Prentice hall Englewood Cliffs (1991)
12. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)
13. Lamport, L.: The PlusCal algorithm language. In: International Colloquium on Theoretical Aspects of Computing. pp. 36–60. Springer (2009)
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
15. Rou, G., erbnut, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming 79(6), 397–434 (2010), iSBN: 1567-8326 Publisher: Elsevier
16. Rusu, V., Grimaud, G., Hauspie, M.: Proving partial-correctness and invariance properties of transition-system models. Science of Computer Programming 186, 102342 (2020), iSBN: 0167-6423 Publisher: Elsevier
17. Samek, M., Montgomery, P.: State oriented programming. Embedded Systems Programming 13(8), 22–43 (2000)
18. Welcome! | The Coq Proof Assistant, https://coq.inria.fr/
19. Tretmans, J.: Model based testing with labelled transition systems. In: Formal methods and testing, pp. 1–38. Springer (2008)
20. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Advanced Research Working Conference on Correct Hardware Design and Verification Methods. pp. 54–66. Springer (1999)