Towards Developing Trusted Smart Contracts in Simulink

Baoluo Meng*, Meng Li, Benjamin Beckmann, Yoshifumi Nishida, John Carbone, Dan Yang and Michael Durling

GE Research, Niskayuna, NY 12309, USA
*baoluo.meng@ge.com

Abstract. Blockchain has emerged as a subject of intense interest in research fields and beyond. One of key enabling technologies is smart contracts. Smart contracts bring transparency, simplicity, and efficiency to blockchain applications. Several languages and tools have been developed for smart contracts over the years but few are easily usable for domain experts in control systems as most of them are Simulink users and learning a new language and tool could be a challenge for them. In this paper, we propose a trusted smart contract development approach in Simulink Stateflow for blockchain applications. The approach introduces a design environment and generates evidences of trust for smart contracts via formal verification, simulation, automated test generation, and test execution. Finally, the formally verified Stateflow models is automatically synthesized into Solidity for deployment on blockchain platforms. This approach will not only bring the assurance of smart contracts to the next level, but also will make the blockchain technology accessible to the control experts, which could inspire broader blockchain applications in industry. We implemented the framework as a toolbox in Simulink, which has been used by GE Research. We will demonstrate the capabilities and effectiveness of the toolbox using a real transactive energy example to show that valid issues are identified along the development cycle.

Keywords: Block Chain, Smart Contracts, Formal Verification, Simulink, Automated Test Generation.

1 Introduction

Blockchain has emerged as a subject of intense interest in research fields and beyond. It has been used to secure data in communication, transaction, computation etc. Blockchain applications store data using sophisticated math and innovative software rules, which are extremely difficult for attackers to compromise. One may build blockchain applications to protect certain high integrity data, core communications, transactions and computations. One of key enabling technologies in blockchain applications is smart contracts. A smart contract is a computer script running on top of a distributed ledger with a set of rules under which participants agree to interact with each other. Smart

contracts define the behavior of the blockchain application and are important to the correctness and security of the blockchain. Moreover, it brings transparency, simplicity, and efficiency to blockchain applications. However, development of smart contracts suffers from the same software development issues such as design errors, implementation errors, security and safety issues, etc. These issues may lead to severe financial loss or even safety concerns if they are not properly mitigated before the blockchain applications are deployed. Several languages and tools have been developed for smart contracts over the years but few are easily usable for domain experts in control systems as most of them are mainly Simulink users. GE has been a long-time leader in the control and optimization industry, and many of control experts at GE are adept at using Simulink and associated tools. To expand the realm of blockchain, we introduce a Simulink framework to facilitate blockchain applications in the control field, which is also the key motivation for this work. We have partnered with various GE businesses to build blockchain applications to protect our essential assets and to establish secure decentralized transaction systems. In this paper, we propose an approach for developing trusted smart contract in Simulink for blockchain applications. The approach introduces a design environment, a formal verification tool, an automated Solidity code generation tool, an automated test generation tool, and a test execution environment. The key objective of the framework is to facilitate blockchain applications development in the control field in industry. The main contributions of the paper are as follows:

- A novel approach in Simulink to develop trusted smart contracts for blockchain applications
- A toolbox that translates a fragment of Stateflow to Solidity
- A demonstration of the development approach on a real industrial application

The paper is organized as follows: Section 2 describes background and related work in blockchain technology. Section 3 describes the development approach in Simulink, which includes modeling smart contracts, performing simulation and formal verification, synthesizing smart contracts from Stateflow models and tests execution. Section 4 demonstrates the development approach on a real transactive energy application used at GE Research. Section 5 concludes the paper and proposes future work.

2 Background

2.1 Simulink

Simulink is a versatile graphical programming environment in Matlab, which is widely used in industry for model-based design. It provides an environment for modeling, simulating and analyzing dynamic and embedded systems. Stateflow is a toolbox of Simulink, which is to model state machines and flow charts. Simulink Design Verifier [8] provides the verification and validation capability to identify common design errors such as integer overflow, division by zero and dead logic, check if the models satisfy requirements and also generate test cases. In addition, Simulink also allows third-party

plugins. In our approach, we will mainly use Simulink Stateflow to model smart contracts and use Simulink Design Verifier to perform verification on the models. We will develop a toolbox plugin to Simulink to synthesize Solidity smart contracts. We will briefly explain some major terminologies in Stateflow that are used in our approach including state, transition, action and junction. State describes the internal status of a system at any given time. Entry, during and exit actions are associated with states and is executed to update variables upon entering, staying, and exiting a state respectively. One may transition from a state to another if some events happen or certain conditions are met. Junction is a way to encode conditional branches for transitions. Condition actions may be executed after the condition is met.

2.2 Related Work

After a few cyberattacks on smart contracts, such as the DAO Attack [2], the security of smart contract has increasingly drawn the attentions from the blockchain community. Researchers have been investing efforts to ensure the trustworthiness of blockchain smart contracts. Bigi et al. [3] described the validation of smart contracts using a combination of game theory and formal methods. Bhargavan et al. [4] translated Solidity smart contract program into F* to verify the runtime safety and functional correctness of the smart contract. Mavridou and Laszka [5] presented FSolidM tool to apply finite state machine to model smart contracts and automatically translated finite state machine to Solidity code. They also developed some plugins to help implement security features for preventing common vulnerabilities and common design patterns to facilitate correct contracts. Sergey and Hobor [6] proposed contracts-as-concurrent-objects analogy to enable formal verification especially on concurrency-related properties. Smart contracts as computer programs may have bugs and security flaws. Smart contract failures may lead to severe financial losses or even safety concerns. For example, the DAO attack causes $70 million equivalent of Ethereum being stolen within a few hours. Current manual review is error-prone and time-consuming. Since deployed smart contracts are practically immutable and can be inspected by the public which eliminates security from obscurity, it is preferred to build trust into development, as soon as the errors or flaws are introduced.

3 Towards Trusted Smart Contract Development in Simulink

A blockchain application typically includes blockchain platforms, smart contracts and interactive edge devices transacting using smart contracts. One of key factors for protecting the security of blockchain applications is having trusted smart contracts. In this section, we describe a novel approach for building trusted smart contracts in Simulink Stateflow. The approach stages in four phases: design phase, verification and simulation phase, synthesis and test case generation phase, and test execution phase. The workflow of the process is illustrated in Fig. 1.

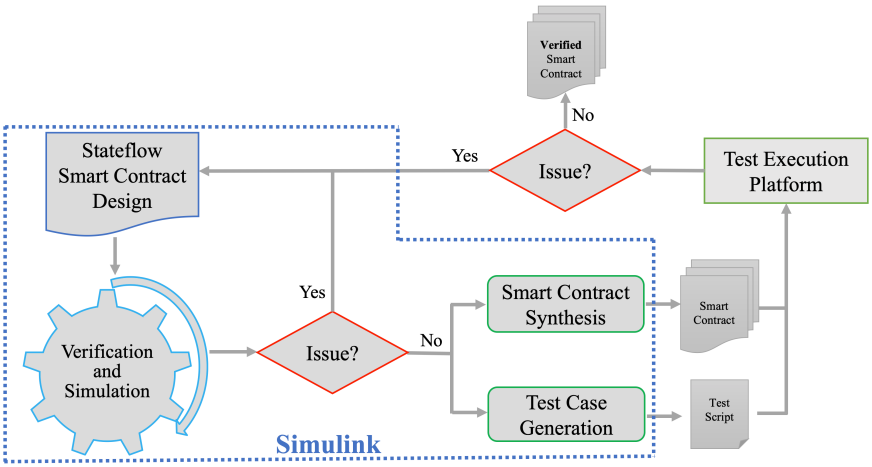

Fig. 1. The workflow of the proposed approach

The workflow starts with a design phase in Simulink Stateflow for modeling the contract model. The contract model is then fed to Simulink Design Verifier for verification and simulation. If any issues are found at this stage, the feedback from the verification and simulation engine will be leveraged to improve the contract design. Otherwise, the contract model will be synthesized into Solidity smart contract, and test case will be generated based on user-defined testing objectives in Simulink. Test cases will be executed against the synthesized smart contracts on popular platforms such as Remix [9] and Truffle [10]. After passing all the tests, the smart contracts will be ready for deployment. Failing tests will be investigated further to enhance the contract model in Simulink. We will use a simple banking example to illustrate each phase in the following sections. Consider a simple banking smart contract that deals with the interactions between a user and a bank. The user can activate or deactivate an account in the bank. Only after the account is activated, the user can deposit into or withdraw from the account. When a user performs the deposit or withdraw action, the bank must validate the action before the action can be executed, and subsequently the effect of the action will be reflected in the account's balance.

3.1 Design Phase

In the design phase, the smart contracts are captured in Simulink Stateflow. We choose Stateflow because (i) it is a popular and user-friendly tool that has a large user base in the control area; (ii) it is supported by Simulink Design Verifier that provides formal verification and automated test generation capabilities; and (iii) it is a toolbox of Simulink that provides visualized simulation.

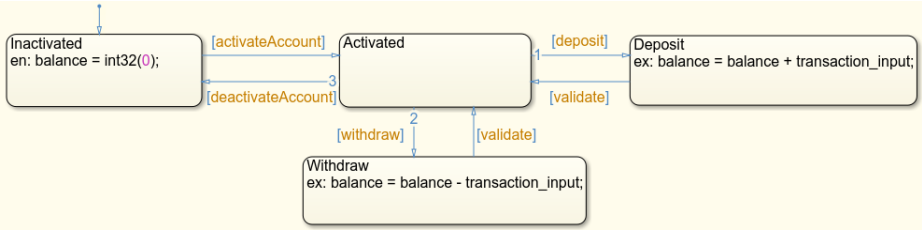

Fig. 2. A smart contract model for the banking example in Stateflow

The Stateflow representation of the simple banking example is shown in Fig. 5. The contract model consists of four states: *Inactivated*, *Activated*, *Deposit*, and *Withdraw*. The initial state of the smart contract model is the *Inactivated* state. As shown in Fig. 5, the state of the smart contract model can transition from *Inactivated* to *Activated* through the *activateAccount* event. The smart contract model can go back to the *Inactivated* state when *deactivateAccount* event is triggered from the *Activated* state. In addition, when in the *Activated* state, the smart contract model can transition to *Deposit* or *Withdraw* state with *deposit* or *withdraw* event being triggered. However, the model can only transition back to the *Activated* state when the validate event is issued and the account's balance is updated when exiting the *Deposit* or *Withdraw* state.

In general, smart contracts consist of a method set with defined rules. In our Simulink Stateflow modeling environment, we use an event to model a function-call in smart contracts, and non-event-triggered transitions combined with junctions to model rules within a call. An advantage of this design method over traditional hand-coded design is that it enables users to intuitively define states and visualize transitions among states without concerns for case statements or control logic.

3.2 Simulation and Verification Phase

To simulate Stateflow smart contract, the contract model is encapsulated in two parallel states called "*mainChart*" and "*events*", where "events" is an event-triggering state at the top-level to broadcast to "*mainChart*" to trigger events inside the contract model.

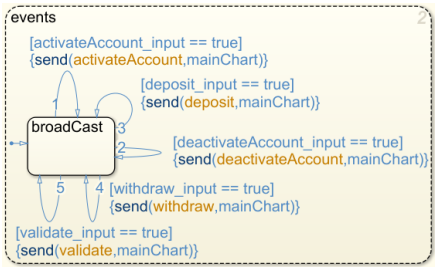

Fig. 3. Event-triggering Stateflow contract model for Simulation

Fig. 3 shows the event-triggering Stateflow smart contract model for the simulation. Each event used in the smart contract will have an associated external Boolean input

signal to trigger the event from the Simulink simulation environment. When the Boolean input signal is set to true, the corresponding event will be broadcasted to the smart contract model; otherwise, the event stays inactivated. Additional constraints are set outside of the Stateflow chart to ensure that at most one of the Boolean input signals for event-triggering state transition is set to true at any time. Simulation helps designers identify the functional flaws of the smart contract model by exercising different use cases. It can also localize issues when counterexamples from the formal verification are leveraged to analyze the smart contract model.

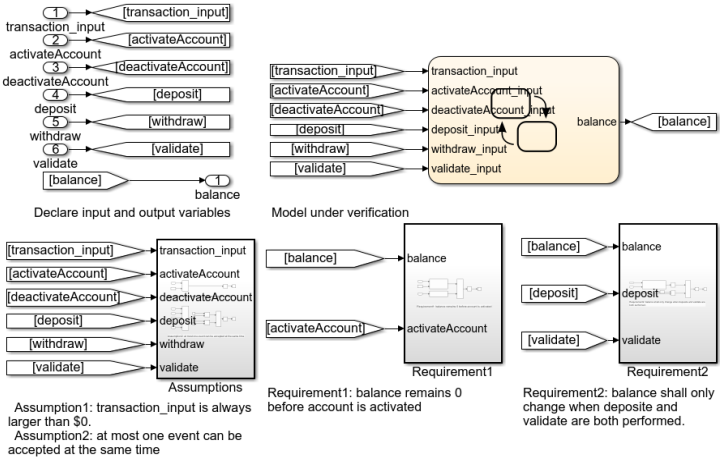

Fig. 4. Simulink Stateflow model to perform formal verification for the contract

Formal verification of the Stateflow smart contract model can be performed against critical functional, security, and safety properties. In this phase, safety properties and associated assumptions are captured in the Simulink environment. Properties and assumptions for the banking contract model are provided in Fig. 4. The properties state that 1) balance remains 0 before account is activated, and 2) balance shall only change when deposit and validate are both performed. Two assumptions are made to ensure that at most one event is triggered and the deposit or withdraw transaction value is always a positive number.

3.3 Smart Contract Synthesis and Test Case Generation Phase

The Stateflow smart contract will be translated into Solidity contracts after the simulation and verification phase. Solidity is a contract-oriented language for implementing smart contracts. It is a statically typed high-level programming language highly influenced by object-oriented languages like C++ and Python. Solidity has been widely used by blockchain platforms such as Ethereum, Sawtooth, Tendermint, ErisDB, Counterparty, etc. A contract consists of states and functions, where states are described by variables, and functions are reusable code that may modify variables. Conceptually, some smart contracts can be viewed as event-driven finite state machines, which is motivation of our translation. In this section, we discuss the translation from a fragment

of Stateflow model to Solidity code per Stateflow features. The translation mappings are summarized in Table 1.

Table 1. Translation mapping of state transitions between Stateflow and Solidity

Simulink Stateflow	Solidity
State: S_1, S_2, …, S_n	**Enumerated Type**: StateEnum { S_1, S_2, …, S_n } State : StateEnum;
Junctions: J_1, J_2, …, J_n	**Enumerated Type**: JunctionEnum { J_1, J_2, …, J_n } Junction : JunctionEnum;
Initial State: S_{init} **Entry Action**: $action_{init}$	**Constructor**: Constructor () public {$action_{init}$; State = S_{init};}

Event-Driven State Transitions:
S_{src1} [$event_1$] {$action_1$} S_{dest1};
S_{src2} [$event_2$] {$action_2$} S_{dest2};
…
Source State Actions:
en_action_{src1}, du_action_{src1}, ex_action_{src1}
en_action_{src2}, du_action_{src2}, ex_action_{src2}
…
Destination State Actions:
en_action_{dest1}, du_action_{dest1}, ex_action_{dest1}
en_action_{dest2}, du_action_{dest2}, ex_action_{dest2}
…
	Public function:
	function executeEvent () **public** return () {
	if (State == S_{src1}) {
	executeStateExitAction(); State = S_{dest1}; $action_1$;
	executeStateEntryAction();
	} else if (State == S_{src2}) {
	executeStateExitAction(); State = S_{dest2}; $action_2$;
	executeStateEntryAction();
	} … {
	} else {executeStateDuringAction();}}
	function executeStateEntryAction() **internal** {
	if (State == S_{src1}) {$en\_action_{src1;}$}
	else if (State == S_{src2}) {$en\_action_{src2;}$}
	…}
	function executeStateExitAction() **internal** {
	if (State == S_{src1}) {$ex\_action_{src1;}$}
	else if (State == S_{src2}) {$ex\_action_{src2;}$}
	…}
	function executeStateDuringAction() **internal** {
	if (State == S_{src1}) {$du\_action_{src1;}$}
	else if (State == S_{src2}) {$du\_action_{src2;}$}
	…}

Event-Driven State Transitions via Junctions:
S_{src1} [event] Junc S_{dest1};
	Public function:
	function executeEvent () internal return () {
	if (State = S_{src1}) {
	ex_action_{src1}; en_action_{dest1};
	Junction = Junc; executeJunctions();}}

Condition-Driven Junctions:
Junction J_1 branch: [C_1]{A_1} S_1
Junction J_2 branch: [C_2]{A_2} S_2
…
Junction J_n branch: S_n
	Internal function:
	function executeJunctions() public return () {
	bool stateReached = false;
	while (stateReached) {
	if (Junction = "J_1" && C_1)
	{A_1; State = S_1; stateReached = true;}}
	else if (Junction = "J_2" && C_2)
	{A_2; State = S_2; stateReached = true;}
	…
	else {State = S_n; stateReached = true;}}

Data Types: Boolean, String, (u)int(8, 16, 32, 64)	**Data Types**: bool, string, (u)int(8, 16, 32, 64)
Output Data: $output_1$, $output_2$, …, $output_n$	**State Variable**: $output_1$, $output_2$, …, $output_n$
Local Data: loc_1, loc2, …, loc_n	**Local Variable**: loc_1, loc_1, …, loc_n
Boolean Operators: !, &&, ||, =	**Boolean Operators**: !, &&, ||, =
Comparison Operators:<, <=, =, !=, >=, >	**Integer Operators**: <, <=, =, !=, >=, >
Bit Operators: &, |, ^, ~	**Bit Operators**: &, |, ^, ~
Arithmetic Operators: +, –, *, /, %, <<, >>	**Arithmetic Operators**: +, –, *, /, %, <<, >>

In Table 1, we use S to denote a state in Stateflow and use en_action, du_action and ex_action to denote entry, during and exit actions of a state respectively. We use event to denote an event, J or Junc to denote a junction, C to denote a condition on a junction branch, and A or action to denote a condition action.

The translation focuses on a fragment of Stateflow language to model the control logic of smart contracts, which is well-aligned with the semantics of Solidity. The fragment mainly encompasses event-triggered state transitions and condition-driven junctions. These restrictions on Stateflow language intend to simplify the engineers' modeling efforts as well as retain the semantics of smart contracts. For state transitions, all outgoing transitions of a state must be event-driven without conditions. One may also specify entry, during and exit actions within states. Outgoing branches of junctions must be complete and condition-driven without events. Branches may have condition and condition actions. Note that the semantics of conditions of state transition can be equivalently encoded using condition of junctions. Loops are also allowed on junctions. However, just as loop is the Achilles' heel in program verification, Stateflow models with loops also suffer from the same curse, but can still be simulated in Simulink.

```
pragma solidity ^0.4.0;
contract bankAccount {
    enum StateEnum {Inactivated, Activated,
                    Deposit, Withdraw}
    StateEnum State;
    int32 transaction_input;
    int32 balance;
    constructor() public {
        State = StateEnum.Inactivated;
        executeStateEntryAction();
    }
    function activateAccount() public {
        if (State == StateEnum.Inactivated) {
            executeStateExitAction();
            State = StateEnum.Activated;
            executeStateEntryAction();
        } else {
            executeStateDuringAction();
        }
    }
    function deposit() public {
        if (State == StateEnum.Activated) {
            executeStateExitAction();
            State = StateEnum.Deposit;
            executeStateEntryAction();
        } else {
            executeStateDuringAction();
        }
    }
    function executeStateEntryAction() internal {
        if (State == StateEnum.Inactivated) {
            balance = int32(0);
        }
    }
    function executeStateDuringAction() internal {}
    function executeActivatedEntryAction() internal
    {
        if (State == StateEnum.Inactivated) {
            balance = int32(0);
            State = Activated;
        }
    }
    function executeStateExitAction() internal {
        if (State == StateEnum.Deposit) {
            balance = balance + transaction_input;
        } else if (State == StateEnum.Withdraw) {
            balance = balance - transaction_input;
        }
    }
    function displayData() constant
        returns (int32, int32) {
        return (transaction_input, balance);
    }
}
```

Fig. 5. A portion of synthesized Solidity code for the banking example

Since operations on global variables of smart contracts are not critical to the correctness of the control logic of contracts, they will be encapsulated in the printing function *disp* in Stateflow, thus are not subject to verification and testing, and will be printed in Solidity directly. States and junctions in Stateflow models are translated as an enumerated type with values being the state and junction names respectively. The initial state will be translated into a constructor populated with entry actions and the initial state assignment. Each Stateflow event will be translated as a public function and the function body is populated according to the rules specified in Table 1. Entry, during and exit actions associated with states are encapsulated in three execution functions respectively. Junctions are translated as an internal function named *executeJunctions*, which will be invoked by the function *executeEvent* when the state transitions are made via junctions. The semantics of junctions are translated by using a *while* loop guard until a destination state is reached. If a junction is followed by nested junctions rather than a state, the *State*

and *stateReach* assignments will be omitted in the translation. Instead, they will be replaced by a corresponding junction assignment. In addition, the *if-then-else* statements in Solidity may encode the execution orders of junctions imposed by Simulink models. Stateflow and Solidity share a common set of data types as shown in Table 1 including String, Boolean and integer of 8, 16, 32 and 64 bits and associated operators. They can be easily mapped from Stateflow and Solidity.

A portion of the synthesized Solidity smart contract is shown in Fig. 5. Due to the space limit, we only show 2 functions corresponding to activation and deposit events. Other functions are similar.

Simulink Design Verifier supports auto-generation of test cases and procedures for the Stateflow smart contract model. The generated tests will be executed on the synthesized Solidity code to verify the conformance of the code against the model. One may specify model coverage criteria and test conditions to autogenerate tests in Simulink.

3.4 Test Execution Phase

Remix [9] and Truffle [10] are two popular tools for testing Solidity smart contracts. Remix is a web browser based integrated development environment that allows deployment and running of the smart contracts. Truffle provides automated smart contract testing with Mocha and Chai so the generated tests from test generation phase in Simulink can be converted to test scripts, which can be executed against the synthesized smart contract.

4 Case Study: A Transactive Energy Application

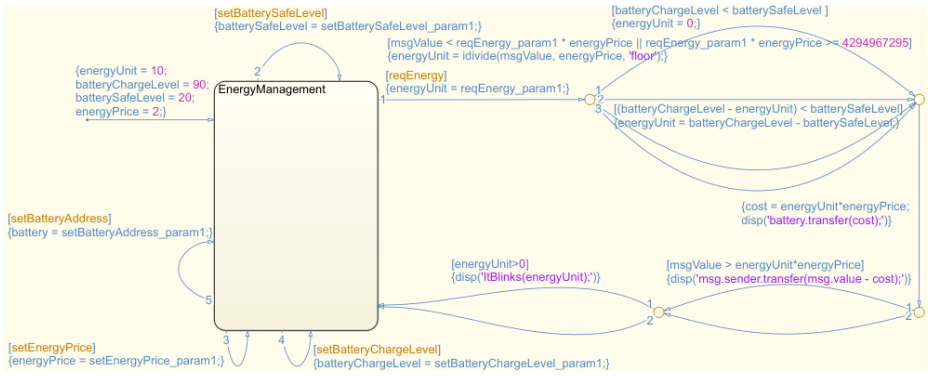

Fig. 6. A transactive energy system model in Stateflow

In this section, we demonstrate the trusted smart contract development approach on a transactive energy application at GE Research. Consider a transactive energy system where a consumer can request energy from a battery. If the battery charge level is less

than a certain safe level, then the request cannot be completed and consumer gets no energy. If the consumer requests an amount of energy that would bring the battery charge level below the safe level, the consumer only gets the amount of energy above the safe level. If the consumer requests more energy than he/she can afford, the consumer only gets the energy that he/she can afford; otherwise, the consumer gets the energy that it requested. Meanwhile, the battery may update its address, energy unit price, the safe level, and charge level.

4.1 Design Phase

Fig. 6 shows the Stateflow smart contract design for the application. Only one state was put in the design since there were no dependencies among function-calls. Each event represented a function-call in the smart contract that led to one or more transitions that process a sequence of rules.

4.2 Simulation and Formal Verification Phase

We have encoded two safety-related properties for the Stateflow models: (i) user shall not obtain energy when battery charge level is lower than safe level, and (ii) user shall not obtain more energy than one can afford. We use Simulink Design Verifier to perform formal verification on the Stateflow models. Two valid issues are identified: (1) divide by zero, and (2) property (ii) is disapproved due to overflow.

The issue (1) was uncovered in the detect-design-error mode of Simulink Design Verifier. When the consumer has a balance 0 on the account and the energy unit price "*energyPrice*" is set to be zero, the second transition after accepting "*reqEnergy*" event will be activated and the division operation on the transition will be performed with denominator "*energyPrice*" set as zero, which triggers the divided-by-zero error. We fixed this error by changing comparison operator from "<=" to "<" in the transition condition to ensure that all requested energy must be strictly less than the allowed amount.

The issue (2) was identified by Simulink Design Verifier. The counterexample shows that when the consumer has 4,294,967,295 balance on the account, it can successfully obtain 70 unit of energy with unit price of 2,431,166,277 which is more than the consumer can afford. By running the simulation with the counterexample, we localized the overflow error at the second junction transition after "*reqEnergy*" event is accepted, where the result of "*reqEnergy_param1 * energyPrice*" overflows and is evaluated to be maximum value of uint32 (unsigned 32-bit integer) in Simulink. The overflow causes "*msgValue*" to less than "*reqEnergy_param1 * energyPrice*" (note that we have changed the "<=" to "<" to fix the divided-by-zero error) and therefore bypasses the rule that a consumer is only granted the amount of energy per consumer balance. We further fix the issue (2) by adding an additional condition "*reqEnergy_param1 * energyPrice* >= 4294967295" to accommodate the overflow case.

4.3 Smart Contract Synthesis and Test Case Generation Phase

We synthesize the Solidity smart contract code leveraging our toolbox in Simulink. In addition, we auto-generate three test cases to cover all reachable transitions of the smart contract model with 32 test objectives.

4.4 Test Execution Phase

The test scripts are executed in Remix against the synthesized contract code with one test failure. The failure further reveals an underflow error in the Solidity contract. This is due to that Stateflow and Solidity handle underflow differently, where Simulink would set the underflow expression to the minimum value of its type, whereas Solidity would perform a unit wrap around for underflow expressions. For the underflow issue in the contract, Solidity evaluates the third junction transition to be not activated, when *batteryChargeLevel* = 90, *energyUnit* = 1073741822, and both variables are uint32, since Solidity evaluated "*batteryChargeLevel – energyUnit*" to be maximum value of uint32 minus the difference between 90 and 1073741822 which is a larger number than "*batterySafeLevel*". While Stateflow evaluates the transition to be activated, since it evaluated "*batteryChargeLevel – energyUnit*" to be zero. By updating the code to be "*batteryChargeLevel* < *batterySafeLevel* + *energyUnit*" instead of "*batteryCharge-Level* - *energyUnit* < *batterySafeLevel*", all tests pass.

4.5 Observations

The modified Solidity smart contract is deployed on a platform with Ethereum blockchain platform. The transactive energy application is able to successfully perform the energy trading from the consumer to the battery and supply the requested energy to the consumer from the battery.

We also compare our development approach against the traditional development approach without the help of the Stateflow design environment. It took our developer one month to design, program, and deploy the same smart contract. However, with the help of the proposed development approach and toolbox, a developer with similar knowledge of smart contract can get through to the deployment within a week. Further and more importantly, with the traditional approach, it is harder to uncover the overflow, underflow or divided-by-zero issues that were identified by the proposed approach.

The proposed development approach and toolbox have also been applied to blockchain application with multiple smart contracts interacting with each other at GE. This kind of application demonstrates even bigger value of the proposed approach, as reasoning about the correctness of the smart contracts by human becomes even more challenging. With our approach, the challenges can be mitigated by performing simulation, formal verification and testing.

5 Conclusion and Future Work

The paper presented a trusted smart contract development approach and a toolbox to synthesize smart contracts from Simulink Stateflow models. The approach captures Stateflow models, performs simulation and formal verification on the models, and auto-synthesizes Solidity smart contracts and generates test cases to ensure the correctness of the smart contract model. The smart contract synthesis process is also described. The synthesized smart contract was trusted and highly assured with evidences of formal proofs and test results. Furthermore, the development process enables the designers focus on the contract design in an user-friendly interface, and automates the verification and synthesis work, which helps to detect and fix issues to reduce the rework time. The proposed development process and toolbox have been used extensively at GE Research for blockchain smart contract development and have proven to be much more effective and efficient than traditional development process.

Our future work will focus on improving the toolbox by supporting formal verification that includes interacting devices, and optimizing the automated smart contract code generation tool for example generating blockchain smart contract specific functions such as payable functions.

References

1. N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” Principles of Security and Trust, vol. 10204, pp. 164-186, March, 2017.
2. Understanding the DAO attack, http://www.coindesk.com/understanding-daohack-journalists/
3. G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto, “Validation of decentralised smart contracts through game theory and formal methods,” Programming Languages with Applications to Biology and Security, vol. 9465, pp. 142-161, 2015.
4. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, “Formal verification of smart contracts: Short paper,” Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91-96, October, 2016.
5. A. Mavridou, and A. Laszka, “Designing secure Ethereum smart contracts: A finite state machine based approach,” arXiv preprint arXiv:1711.09327, November, 2017.
6. I. Sergey, and A. Hobor, “A concurrent perspective on smart contracts,” International Conference on Financial Cryptography and Data Security, pp. 478-493, April, 2017.
7. Stateflow, https://www.mathworks.com/products/stateflow.html.
8. SimulinkDesign Verifier, https://www.mathworks.com/products/sldesignverifier.html.
9. Remix, https://remix.ethereum.org.
10. Truffle, https://github.com/trufflesuite/truffle.