Towards Model-Driven Engineering of Smart Contracts for Cyber-Physical Systems

Péter Garamvölgyi, Imre Kocsis, Benjámin Gehl, Attila Klenik
Dept. of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary
Email: ikocsis@mit.bme.hu

***Abstract*—Applications of Distributed Ledger Technologies (DLTs) in IoT and Cyber-Physical Systems (CPS) are rapidly emerging. However, developing correct and resilient smart contracts for these use cases is even less understood than it is for cryptocurrency-based contracts. This paper presents an initial approach for generating smart contracts for coordinating the usage of cyber-physical system elements from UML statecharts. While the current target platform is Ethereum, our approach can easily be extended to other blockchain platforms.**

I. Introduction

Cyber-Physical Systems (CPS) have been described as *the next computing revolution*, with the potential of transforming how we interact with the physical world around us [1]. With such a transformative technology, a great emphasis must be put on the security and correctness of complex CPS systems.

To ensure resilient control of complex CPS systems, we can leverage the recent advances in Distributed Ledger Technology (DLT) and smart contracts. We propose an approach where the logic implemented as a smart contract serves as a "digital twin" [2] of the CPS device, enforcing usage policies and intended cooperation patterns.

To support the development of such blockchain-assisted CPS systems, we propose a novel approach to smart contract development based on code generation from behavioral models, specifically statecharts. Effectively, we begin to explore the application of Model-Driven Development (MDD) for CPS-supporting smart contracts. In this paper, we demonstrate the utility of code generation from UML statecharts; other aspects of MDD such as formal verification of models, gradual refinement supporting code synthesis for different platforms, and so on, will be addressed in future work. More specifically, we use UML statecharts to model CPS asset state spaces and the various state transitions that can be requested on them through blockchain transaction invocations, and we translate such models into Solidity code, i.e. smart contracts for the Ethereum platform.

The rest of the paper is organized as follows. In section II we give an overview of DLT and CPS technologies and highlight some of their most important combinations. Section III presents our argument that for cyber-physical applications, the notions of smart contract correctness are much closer to logic correctness concepts in model-driven design than the cryptocurrency-based correctness notions that most research in "securing" smart contracts is addressing now. In section IV we give an overview of our statechart-based approach, while in section V we provide a detailed example. Section VI closes the paper with conclusions and an overview of planned future work.

II. Distributed ledgers and Cyber-Physical Systems

Blockchain-based distributed ledger (DL) technologies provide distributed, shared and cryptographically authentic ledgers without a central third party that has to be trusted. For the purposes of this paper (and referring to [3]) we define blockchain-based Distributed Ledger Technologies (DLT) as

- peer to peer networks,
- where the immutable and cryptographically secure sequence of transactions accepted by the system as a whole is interpreted and typically stored by the peers as a linked list of transaction blocks;
- blocks of transactions are accepted by the system using some distributed consensus mechanism over the blocks;
- transaction logic is typically client programmable through so-called smart contracts,
- and there's an incentive structure that, by driving the majority of the peers to behave honestly, secures the system with a sufficient probability.

While Bitcoin [4] was the first such system (with only limited programmability), today the DL architectural pattern that Bitcoin established is implemented by a very diverse set of technologies. Some of these are permissionless and public, global peer to peer networks, where anybody can commit computational resources (peers) towards taking part in maintaining the ledger by consensus. Also, anybody can use the system by issuing transactions – typically in a pseudonymous way. Loosely speaking, the base transaction type of such systems is wiring amounts of a so-called cryptocurrency among clients. This forms an integral part of securing the system – peers are incentivized to be honest by receiving amounts of the cryptocurrency when they take part in the consensus mechanism.

Currently, Ethereum [5] is the most mainstream such platform with advanced smart contract support. In Ethereum, smart contracts deployed on the blockchain behave as program instances for the Ethereum Virtual Machine, and their functions

Electronic ISBN: ISBN:978-1-5386-6553-4 DOI: 10.1109/DSN-W.2018.00052

are triggered by transactions of the system. Clients pay by small amounts of the cryptocurrency as an execution fee for invoking smart contract functions. Smart contracts can have a state – immutably recorded on the ledger – and can also receive and send amounts of the cryptocurrency.

This way, various crypto-financial contracts and mechanisms can be implemented on top of the Ethereum network; from simple family purses to Decentralized Autonomous Organizations (DAOs) [6]. Smart contracts are also widely used to create ledgers and governing transaction logics for custom tokens - as part of, or based on, the Ether (the cryptocurrency of Ethereum) ledger. As a very wide range of concepts – from ownership through utility usage rights to identity – can be tokenized [7], the variety of such token contracts is becoming truly astonishing.

More generally, the Ethereum network also has a world computer interpretation. Programs can be deployed on a global, globally accessible, resilient and trustworthy P2P system; and methods of these programs can be invoked for a small fee (paid in the cryptocurrency of the network). Invocations and their effect on the program state are recorded immutably.

A. Application to Cyber-Physical Systems

Using this world computer is decidedly expensive, limiting program complexity and invocation frequency; at the same time, the above set of platform properties is a very powerful combination. Under the general interpretation, when managing the cryptocurrency or tokens does not have to be the focus of smart contracts, a wide range of application possibilities emerge in numerous application domains. For instance, for IoT security, [8] highlights five core use cases:

1) Scalable IoT discovery – the (contract) ledger is a trusted billboard for seed servers
2) Trusted communication – the (contract) ledger is a trusted message board
3) Semi-Autonomous Machine-to-Machine Operations – devices enter into temporary cooperation through smart contracts, following the contract rules; the blockchain also acts as a notary service (nonrepudiability of device agreements)
4) IoT Configuration and Update Controls – the (contract) ledger acts as a distributed system management service (ZooKeeper [9] is a good analogy in the data center management world)
5) Secure Firmware Image Distribution and Update – the smart contract acts as an update service

IoT can be understood as part of a large, emerging system class: Cyber-Physical Systems (CPS) [1]. CPS are Systems of Systems (SoS); they are dynamically composed systems of intelligent field devices and edge and cloud services, with multiple interconnected and hierarchical control loops. CPS are bringing an immense wave of innovation, from smart cities through industry to transportation. However, they bring about new risks, too, with the complex interconnection of systems the dependability and security assurance of which classically assumed a rather closed world.

As demonstrated by the above cited use cases, blockchain-based DLTs with the appropriate smart contracts can serve as various forms of middleware in such systems: importantly, point-to-point communication, publish/subscribe, orchestration, logging and operational management (including membership and directory services). A blockchain-based implementation of such middleware services has the same advantages that apply for cryptocurrency accounting and smart contracts in general.

- The service is resilient; truly effective denial of service attacks and security compromises are very infrequent.
- Service access is resilient, too; the system can be used through a large number of peers instead of a few service access points.
- The ledger serves as a transparent and immutable transaction log for the service actions.
- Actions are nonrepudiable.

These properties can effectively mitigate many problems that arise from the basic nature of CPS. For instance, if a blockchain-deployed ledger can be used to exchange system membership information (including cryptographic materials), then the dynamic in-field system formation of devices can be performed drastically simpler than using only classic in-field communication channels. Here we implicitly assume that the field devices only access a blockchain, but don't "run" it; approaches for that case are only at an early stage.

B. The argument for using private/consortial, permissioned blockchains

The downside of using open, permissionless blockchains in this manner is that the price to achieve a given level of transaction throughput and latency on public blockchains can be too high for many use cases. Depending on the transaction load of the blockchain, this price point is highly volatile. Even when transactions are correctly priced, public blockchains acknowledge new blocks of transactions rather slowly (the average block time for Ethereum is 15 seconds). Additionally, the globally shared nature of public DLTs may not be tolerable for certain applications.

Private/consortial, permissioned [10] blockchains can address most of these concerns, at the expense of some of the system-level resilience. In this world, the Hyperledger project [11] of the Linux Foundation collects the leading platform technologies, as e.g. Hyperledger Fabric [12] and Hyperledger Composer [13] (although the Ethereum software can be also used to build private and/or permissioned blockchains). Hyperledger Fabric and Composer support building blockchain systems with true identity and permission management (including participation in transaction-consensus). As such, these systems don't need a native cryptocurrency to function.

Note that none of the CPS use cases discussed here require the smart contract to have a cryptocurrency balance; the blockchain is used as a distributed communication, computation and storage "fabric", with special extra-functional properties.

III. "Correct" CPS smart contracts with Model-Driven Development?

Today, smart contract development is a rather challenging task. The programming models of the various blockchain platforms are new and not always straightforward to newcomers. And the risk associated with programming faults is rather high: smart contracts handle "money" in the public blockchains and the distributed ledger is immutable (no "rollbacks"). The used programming languages are not necessarily up to the task, either; the insufficiencies of the Solidity language arguably heavily contributed to the infamous "DAO hack" [14] on the Ethereum network.

As a consequence, testing, code analysis and formal verification tools are rapidly appearing [15]. Some research has been done on automatic generation of smart contracts from formal models [16] [17]. There also seems to be a consensus that Domain Specific Languages are needed to properly mitigate risks, particularly non-Turing complete and functional languages, where formal verification is possible. In some cases, even mechanized formal verification is feasible [18]. In the Bitcoin world Ivy [19] is a good example, by constraining the expressiveness of Bitcoin script as well as at the same time introducing financial asset and transaction notions as first-class concepts, it renders large classes of potential smart contract vulnerabilities simply impossible by design. Simplicity is another notable approach for designing smart contract languages that are secure by design [20].

However, most of the focus seems to fall on preventing the "stealing" or otherwise unintended manipulation of crypto-assets. From this point of view, CPS smart contracts form a subset where the appropriate notion of "correctness" is very different. Informally, CPS smart contract correctness will have to incorporate at least the following notions.

- Making possible only the intended sequences of transactions.
- Not being able to reach a state where no further intended transactions can be initiated (i.e. freedom from deadlocks).
- Strict adherence to transaction initiation rights and roles.
- Robustness: reacting with specified behavior for erroneous input (and to some extent, internal – programming – errors).

Similar requirements are regularly encountered in general-purpose development, and particularly in the development of embedded, safety-critical and business critical systems; and addressed through Model-Driven Development [21] and model analysis. Using appropriate models to design and reason about complex systems highlights the important logic and hides the details. This, and the potential of formally verifying the correctness of such models, can help prevent many programming and design errors and enable the developer to build more robust, resilient systems. The contributions of this paper fit this pattern and can be interpreted as an initial foray into applying MDD techniques for the design of systems that incorporate blockchain-based components.

Here, we examine the application of a subset of UML statecharts [22] for modeling CPS cooperations and automatically generating smart contracts from the model. Using state machine-like languages (and other high-level formalisms) for simulation, formal analysis, and code generation has a solid and mature literature in various domains. Such domains include, but are not limited to: e-voting systems [23]–[25]; mission-critical embedded systems [26]; distributed control systems [27]; and production control systems [28], [29].

However, existing approaches require careful consideration when applying them to distributed ledger technologies (DLT) as the target platform. The unique operational semantics of DLTs must be taken into account when using high-level modeling formalisms (e.g., UML statecharts) to describe the behavior of deployed applications. An evaluation of the differences, similarities, and applicability of traditional approaches is planned as part of our ongoing work.

Currently, to the best of our knowledge, our statechart-based development in the context of DLTs is a novel approach. A conceptually similar state machine-based approach is described in [17], but there the emphasis is put on "financial" contracts. BPMN-based smart contract execution is an active area of research [30], [31] in the context of applications orchestrating business collaborations, but statecharts are more suited to express the envisioned CPS scenarios.

IV. Expressing ledger-based CPS interactions with statecharts

The idea we are pursuing here is to design CPS interaction patterns using standard, verifiable UML statecharts. Then, we will proceed to generate a smart contract from the model, which acts as a digital twin of the actual CPS devices. Our approach is generic enough to be used with a number of different DLT targets, including Ethereum EVM and Hyperledger Fabric or Composer. Our current implementation supports Ethereum smart contracts; extensions for code generation to multiple platforms – including Hyperledger ones – are being developed.

Our core idea is that we express the state of a cyber-physical system using UML statechart simple states and composite states (including history states). Then, state transitions in this model reflect either the observed or intended state transitions of the reflected system. These transitions will map to distinct transactions of the blockchain platform we generate code for. We introduce a notion of access control via transaction guard expressions. These special guards specify roles that are allowed to initiate that transition either to reflect the observed state changes of the reflected system or to initiate a state transition in it by "stepping" the model. Specific actors are mapped to roles during smart contract instantiation and this mapping can be changed dynamically.

Note that our approach in itself does not necessarily ensure that the physical world and its smart contract based reflection remain in synchrony; for instance, in terms of the example that follows in the next section, whether the door of a secure facility opens when the smart contract governing its states

statechart	Solidity	notes
state	enum value	
transition	contract method	this is the public interface of the contract
transition	enum value	this is the inner representation different instances of the same transition (e.g. different guards) might be separate enum values
composite state	enum value	modelled as a normal state automatic transition to start state on entry inner state names are encoded (e.g. mycomposite_innerstate1)
actions (entry/exit/transition)	method body	
guards	require or if/else expressions	
history	contract variable for composite states	
timed transitions	N/A	smart contracts are passive, so timed transitions do not make sense
access control i.e. who can send what	user-defined code or modelling language extensions	

Table I: UML-Solidity mappings

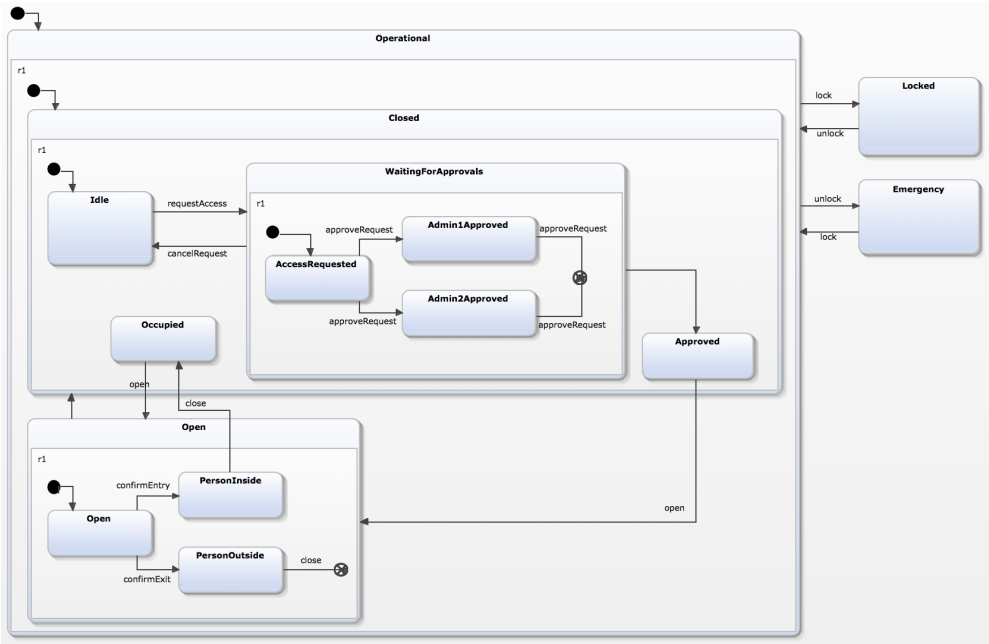

Figure 1: UML statechart model (actions and guards omitted for clarity)

specifies it to do so depends on the correct functioning of the door electronics. Also, when certain transactions are used to ”step” the smart contract state space based on observations, missing observations and different interleavings of observation sequences may introduce challenges. Most of these issues can be addressed by making the ”reflection” nature of the smart contract explicit at the model level (e.g. modeling the state of waiting for door open approval and proceeding with further transactions only when the notification is received). In the longer run, it seems to be unavoidable that we have to work out a suitable operational semantics for statecharts executed by smart contract platforms (for instance, instead of introducing event queues, the proper approach seems to be to allow only one single outstanding transition request at any time). Support for parallel regions is under development, however, timed transitions cannot be mapped to ledger based execution.

The mapping between model elements and Solidity constructs is mostly straightforward. Possible states are represented as enumerations, and contracts keep track of the current state using contract variables. Transitions are triggered by transactions sent to the smart contract, i.e. each transition type has an associated public method on the contract. Actions are general Solidity code-segments, while guards are bool expressions that can be evaluated using a series of conditional statements.

The tool we chose to implement this system is YAKINDU Statechart Tools [32]. We implemented a code generator module for generating Solidity code directly from our YAKINDU model. The output is a partially implemented smart contract that keeps track of the actual state and receives and validates state transitions as Ethereum transactions. The generated smart contract can be manually extended to add functionality that we cannot model in our statechart, e.g. registering addresses for

the actual roles, raising events, etc.

Table I shows the mapping from statechart elements to the corresponding Solidity constructs. Our investigations have shown that all statechart elements can be mapped to Solidity smart contract elements in a fairly straightforward way. For example, states are represented as enum values, separate transition types each have a corresponding public method, guards are logical tests, etc.

V. An example: access of a secure facility

The example scenario is set in a high-security facility. The task is to implement the control mechanism of the entrance to one of the rooms in the facility. The doors open or close by monitoring the state of their governing blockchain contract.

The administration of the facility has defined the following security protocol for entering the room:

1) Request access
2) Approve access
3) Open door
4) Confirm entry
5) Close door

And the corresponding exit protocol:

1) Open door
2) Confirm exit
3) Close door

Note that the entry request must be approved before the room can be accessed. In our scenario, we require the approval of two designated administrators. Furthermore, we require that the requestor confirm entry and exit before closing the door (e.g. via using a secure identity card reader that is able to initiate blockchain transactions on their behalf). Generally, all actions of the access protocol have to go through the smart contract which defines whether the participant is allowed to request that give action in the current state of the whole security system.

Moreover, we have two additional security measures. First, the security team can block entry to and exit from the room at any time. For example, once the security personnel noticed an unauthorized entry to the room, they can lock the door, preventing the intruders from escaping. Second, a designated emergency team can open the door at any time. For example, in case of an incidental fire, the fire brigade can enter the room without going through the security protocol.

A. Model description

Figure 1 shows our UML statechart model for the scenario discussed above, created using the YAKINDU statechart modeling tool. For clarity, only states and transitions are included in this figure.

The model has three main states: Operational, Locked, and Emergency. Operational is the normal state, Locked is the state where normal entry to the room is blocked by the security team, while Emergency is a state triggered by the designated emergency team where the door can always be opened.

Inside Operational, a typical execution of the entry protocol would look like this:

1) The requestor requests access to the room. The state changes from Idle to WaitingForApprovals.
2) Both administrators approve the requestor's access request. The state changes from WaitingForApprovals to Approved.
3) The requestor pushes the open button on the door. The door initiates an `open` transition on the smart contract from its own address. If this transaction succeeds, the state changes from Approved to Open and the door opens.
4) The requestor confirms her entry and closes the door. Again, the door first initiates a `close` transition and only closes physically if it succeeded.

The exit protocol works in a similar way.

By only executing the action after its validity has been established (by checking whether the corresponding transition is valid or not), we ensure consistency between the door and its digital twin. To have stronger guarantees about this consistency, we could extend the model by implementing multi-step open/close flows (e.g. first check validity, then close, then confirm action).

Notice in the above description that in our model the origin of state transitions plays an important role: only certain actors can initiate certain transitions. To model this, we leveraged YAKINDU's modelling DSL. We defined variables representing the actual roles (e.g. `requestor`, `admin1`) and role collections (`security`, `emergency`). As YAKINDU's DSL has limited support for user-defined types, we defined these variables as strings and encoded the actual types in their names (e.g. `requestorActor`, `securityActorCollection`).

Once we have defined these roles, we use user-defined operators and guards to specify who can initiate the given state transition. Figure 2 shows a portion of the model with all the actions and guards included. Note the use of the special identifier `sender`. This denotes the sender of the transaction associated with the corresponding state transition. In Ethereum, for example, this corresponds to `msg.sender`.

Due to space constraints, we cannot show the output of our Proof-of-Concept Ethereum code generator; an example can be found at: https://gist.github.com/benjamingehl/aa7ddea2690df45994e80c2537150305

VI. Conclusion and future work

We have presented an approach to generate Ethereum smart contracts from UML statecharts with the intention of controlling the usage and interactions of CPS elements. While our approach is partial and the mapping does not yet follow the common operational semantics based way, it already presents a compelling argument for pursuing the application of classic MDD methods at the very least in DLT applications that are detached from cryptocurrencies and are intended to manage nontrivial state spaces.

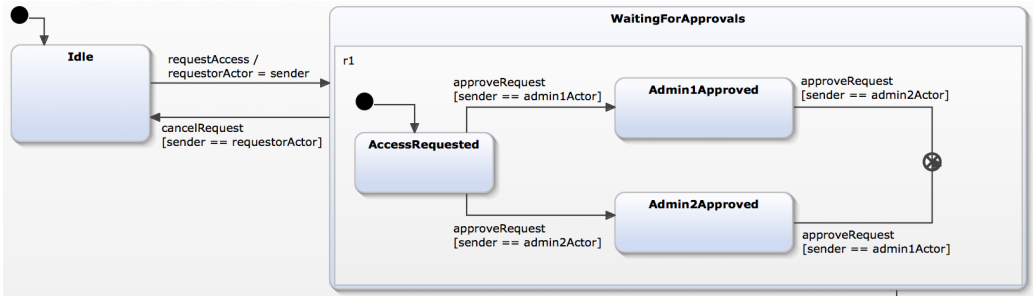

Figure 2: Modelling guards

Future work will first extend the scope of the approach and map out the modeling styles and necessary model properties that are required for provably correct functionality in various cases. Also, we will explore the application of the large set of existing statechart analysis tools. Support for multiple DLT platforms is already under development.

REFERENCES

[1] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Proceedings of the 47th design automation conference*. ACM, 2010, pp. 731–736.

[2] S. Boschert and R. Rosen, "Digital twin the simulation aspect," in *Mechatronic Futures*. Springer, 2016, pp. 59–74.

[3] M. Gray *et al.*, "Introducing Project Bletchley." [Online]. Available: https://github.com/Azure/azure-blockchain-projects/blob/master/bletchley/bletchley-whitepaper.md (accessed on 2018-04-10).

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[5] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014. [Online]. Available: http://www.cryptopapers.net/papers/ethereum-yellowpaper.pdf

[6] U. Chohan, "The decentralized autonomous organization and governance issues," 2017.

[7] D. Siegel, "The Token Handbook." [Online]. Available: https://hackernoon.com/the-token-handbook-a80244a6aacb (accessed on 2018-04-10).

[8] Cloud Security Alliance, Blockchain/Distributed Ledger Technology Working Group, "Using Blockchain Technology to Secure the Internet of Things," 2008. [Online]. Available: https://downloads.cloudsecurityalliance.org/assets/research/blockchain/Using_BlockChain_Technology_to_Secure_the_Internet_of_Things.pdf (accessed on 2018-04-10).

[9] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9. Boston, MA, USA, 2010.

[10] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain Technology Overview (NISTIR 8202)." [Online]. Available: https://csrc.nist.gov/CSRC/media/Publications/nistir/8202/draft/documents/nistir8202-draft.pdf (accessed on 2018-04-10).

[11] "Hyperledger." [Online]. Available: https://www.hyperledger.org (accessed on 2018-04-10).

[12] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[13] "Hyperledger Composer." [Online]. Available: https://hyperledger.github.io/composer/unstable/reference/reference-index.html (accessed on 2018-04-10).

[14] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[15] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016, pp. 91–96.

[16] C. K. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," in *Foundations and Applications of Self* Systems, IEEE International Workshops on*. IEEE, 2016, pp. 210–215.

[17] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," *arXiv preprint arXiv:1711.09327*, 2017.

[18] A. K. Ilya Sergey and A. Hobor, "Scilla: a smart contract intermediate-level language," 2018.

[19] "Ivy: A high-level language and IDE for writing Bitcoin smart contracts." [Online]. Available: https://github.com/ivy-lang/ivy-bitcoin (accessed on 2018-04-10).

[20] R. OConnor, "Simplicity: A new language for blockchains," *arXiv preprint arXiv:1711.03028*, 2017.

[21] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[22] M. von der Beeck, "Formalization of UML-statecharts," in *International Conference on the Unified Modeling Language*. Springer, 2001, pp. 406–421.

[23] R. Tiella, A. Villafiorita, and S. Tomasi, "Specification of the control logic of an eVoting system in UML: the ProVotE experience," in *Proceedings of the 5th International Workshop on Critical Systems Development Using Modeling Languages*. Citeseer, 2006.

[24] ——, "FSMC+, a tool for the generation of Java code from statecharts," in *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. ACM, 2007, pp. 93–102.

[25] A. Villafiorita, K. Weldemariam, and R. Tiella, "Development, formal verification, and evaluation of an e-voting system with VVPAT," *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 4, pp. 651–661, 2009.

[26] K. Wagstaff, K. Peters, and L. Scharenbroich, "From protocol specification to statechart to implementation," *Jet Propulsion Laboratory Technical Report CL08-4014*, 2008.

[27] T. Tomura, K. Uchiro, S. Kanai, and S. Yamamoto, "Object-oriented design pattern approach for modeling and simulating open distributed control system," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 1. IEEE, 2001, pp. 211–216.

[28] H. J. Köhler, U. Nickel, J. Niere, and A. Zündorf, "Integrating UML diagrams for production control systems," in *Proceedings of the 22nd international conference on Software engineering*. ACM, 2000, pp. 241–251.

[29] H.-J. Köhler, U. Nickel, J. Niere, and A. Zündorf, "Using UML as a visual programming language," Technical Report tr-ri-99-205, University of Paderborn, Paderborn, Germany, Tech. Rep., 1999.

[30] H. I. Projects. (2017) Contract-based business process execution. [Online]. Available: https://www.hyperledger.org/blog/2017/09/07/interning-with-hyperledger-5-interns-share-their-experiences-and-advice

[31] L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber, "Optimized execution of business processes on blockchain," in *International Conference on Business Process Management*. Springer, 2017, pp. 130–146.

[32] "YAKINDU Statechart Tools." [Online]. Available: https://www.itemis.com/en/yakindu/state-machine (accessed on 2018-04-10).