ASPECTSOL: A Solidity Aspect-Oriented Programming Tool with Applications in Runtime Verification⋆

Shaun Azzopardi[2][0000−0002−2165−3698], Joshua Ellul[1][0000−0002−4796−5665], Ryan Falzon[3][0000−0001−7513−3658], and Gordon J. Pace[1][0000−0003−0743−6272]

[1] University of Malta, Msida, Malta
joshua.ellul@um.edu.mt and gordon.pace@um.edu.mt
[2] University of Gothenburg, Gothenburg, Sweden shaun.azzopardi@gu.se
[3] Hash Data ryan@hashdata.co

Abstract. Aspect-oriented programming tools aim to provide increased code modularity by enabling programming of cross-cutting concerns separate from the main body of code. Since the inception of runtime verification, aspect-oriented programming has regularly been touted as a perfect accompanying tool, by allowing for non-invasive monitoring instrumentation techniques. In this paper we present, AspectSol, which enables aspect-oriented programming for smart contracts written in Solidity, and then discuss the design space for pointcuts and aspects in this context. We present and evaluate practical runtime verification uses and applications of the tool.

Keywords: Aspect-Oriented Programming · Smart Contracts · Runtime Verification.

1 Introduction

Blockchain [11] and other distributed ledger technologies (DLTs) have enabled the management of digital assets without the need for a central authority, and with strong guarantees regarding immutability of transactions. Smart contracts residing on a blockchain, go a step further in that they handle the execution of logic on such a decentralised platform, thus ensuring faithful execution and enabling trustless operationalisation of protocols of behaviour between parties.

By being deployed on a platform with decentralised control, a blockchain or a variant thereof, these smart contracts come with the benefit of immutability — once written, the protocol cannot be modified unless in a manner that was

originally planned and built into the protocol itself. This autonomous and guaranteed computation platform provides a guarantee not granted by traditional centralised systems. Benefits rarely come without a related cost though, and in this case the cost emanates similarly from immutability.

Smart contracts are nothing more than executable code (running on a DLT platform), for which an unavoidable feature (as with any other software) is the presence of bugs. Immutable code may be the selling point of smart contracts, but immutable bugs are the cost. Add to this the fact that smart contracts typically deal with digital assets, making their correctness critical.

Although smart contracts are typically small programs, modularisation of code is a key measure to reduce potential errors. However, the way smart contracts call each other on platforms like Ethereum [13] is unlike that found in traditional systems, due to overheads, such as gas[4] costs, which can be prohibitive.

Beyond modularisation within the same smart contract, one would also desire to have the tools to encode features and transformations commonly in use across different smart contracts. One technique that has been used for such cross-cutting modularisation is that of aspect-oriented programming (AOP) [10]. In this paper we present an AOP tool for Solidity [6], one of the most commonly used smart contract programming languages. The tool is publicly available at `https://github.com/ryanfalzon/aspectSol`.

AOP has frequently been used as a tool for instrumentation of runtime monitoring and verification code, e.g. [7, 12, 4]. We show how our tool, AspectSol, can be effective in both instrumenting verification code for specifications, and injecting new features modularly, in a monitoring-oriented programming [2] style.

2   Aspect-Oriented Programming

The main idea behind AOP is to allow for the writing of cross-cutting features separate from the main system. Such an approach allows for the specification of *joinpoints* (points during the execution of the underlying system) where specified *advice* (specific instructions or code) will be weaved in. Joinpoints are typically specified using *pointcuts*, essentially specifying a set of joinpoints to be matched. Such an aspect-oriented specification can then be used to weave in the advice onto the existing system. Different joinpoint types are supported by different languages. For instance, for imperative languages, one allows hooking onto points such as the start and end of a function call, on an exception being raised, and updates of state.

---

[4] The notion of *gas* as a resource to be paid for to execute code is the most common way thorough which public blockchains motivate miners (the nodes in the decentralised network which process transactions and record them on the blockchain) to execute and record execution of smart contract code.

3   Smart Contract AOP: Design Considerations

The design of an AOP tool must necessarily take into consideration the nature of the programming language it is to be used for.

Joinpoints. In particular, the choice of joinpoints, the candidate points in a program's execution where an aspect can trigger, is a crucial decision. Solidity is essentially an imperative language, and any AOP tool for the language will certainly borrow much from other tools for this class of languages, e.g. AspectC [3] and AspectJ [9]. Function entry and exit points are such joinpoints, which we adopt in AspectSol, allowing us to write aspects such as `before call-to Wallet.addFunds()` (just before the `addFunds()` function in the smart contract `Wallet` is called, on the caller's side) or `after call-to Wallet.withdrawFunds()` (at the end of the execution of the `withdrawFunds()` function in the smart contract `Wallet`, on the caller's side). In order to match on the callee's side, one would simply replace `execution-of` by `call-to`.

Given the imperative nature of Solidity, computation revolves around the notion of state and the points where the contract state is read or written — indicating relevant joinpoints in this context. In AspectSol we provide automated instrumentation of such points, e.g. `before set uint count` and `after get bool is_paid`. It is worth noting that read pointcuts only trigger when variables are read from within the smart contract being instrumented. This is unavoidable on public blockchains, where state can be read by external entities without having to explicitly call a smart contract.

Solidity's notion of computation failure through the use of reverted execution is unlike that of traditional exceptions. Rather than returning control to the current execution context with an exception flag, `revert` aborts all computation and returns the state of the contract to that which it was before computation started. This makes the use of such exceptions as joinpoints difficult (and expensive in terms of gas) to handle, and therefore not handled in AspectSol.

AspectSol allows the use of the wildcard symbol `*` throughout (for contract names, function names, parameter names, and types), but provides a means of capturing the matched name in order to allow references to it by using double square brackets, e.g. `set [[typevar]] [[varname]]` acts just like `set * *` (triggering whenever a variable is set) but provides access to `typevar` and `varname`, for instance to enable declaring an auxiliary variable of the same type and to access the variable's value in the advice.

Smart contract and language-specific considerations. AspectSol's salient features are particular to smart contracts in general and to Solidity in particular. Since smart contracts essentially encode a protocol between parties, the notion of such parties as active actors of transactions is at its very core. In order to facilitate aspects that use such notions, AspectSol provides a pointcut filter, `originating-from`, to trigger only when the call is made by a particular party, from a particular contract, or from a particular function. For instance, the following pointcut triggers at the start of calls to `depositMoney()` in the `Wallet` smart contract with `msg.sender` being `owners[0]`:

```
before
    execution-of Wallet.depositMoney()
    originating-from owners[0]
```

Another such feature is native cryptocurrency transfer for which it could be useful to implement pointcuts. In Solidity, sending funds is achieved through calling the `send` or `transfer` functions targeting the recipient's address, meaning that we can already capture such pointcuts as function calls, e.g. `before call-to [[recipient_address]].transfer()`. Receiving funds is, however, different in that functions can be tagged as `payable`, meaning that funds can be transferred to the smart contract whenever such a function is called. AspectSol allows for pointcut filtering based on such function tags, thus allowing us to capture calls to functions which transfer funds, e.g.:

```
after execution-of Wallet.*(*) tagged-with payable
```

Solidity modifiers allow functions to be tagged, which would result in changing their behaviour, e.g. to execute certain code before or after the function body. We treat modifiers similar to the `payable` tag, allowing aspects to capture functions which use (or do not use) certain modifiers. We similarly treat visibility annotations such as `public` and `internal` in the same manner.

Advice and aspects. Pointcuts are associated with executable advice by appending a subsequent executable block of code. Consider writing an aspect to make sure that no more than 100 deposits are performed to a wallet without a reconciliation process taking place. This can be achieved by keeping track of the number of deposits and checking that they have not exceeded 100 whenever a deposit takes place. In addition, we would need to declare the new variable which will be used to keep track of this number. This can be written as an aspect as follows:

```
aspect LimitDeposits {
    add-to-declaration Wallet {
        uint private number_of_deposits = 0;
    }

    before execution-of Wallet.addDeposit() {
        require (number_of_deposits < 100);
        number_of_deposits++;
    }
    before execution-of Wallet.reconciliation() {
        number_of_deposits = 0;
    }
}
```

Another mechanism particular to AspectSol is that of adding or removing tags by using pointcuts referring to a function and using `add-tag` or `remove-tag` to update the definition. For instance, if we want to make all public fields private, we can write the following:

```
update-definition Wallet.* tagged-with public {
    remove-tag public;
    add-tag private;
}
```

This approach can be used, for instance, to make a field private and add an appropriate getter and setter to the smart contract.

Weaving considerations. We now move on to weaving considerations, for which different AOP tools take different strategies. For instance, AspectJ performs weaving at the byte-code level, but provides three points in time when it can be applied: compile time, pre-load time, or load time. AspectSol takes a source-code level weaving approach, thus transpiling from the original Solidity source code to produce updated Solidity code which takes the aspects into account. This simplifies testing the updated version of the smart contract more straightforward. The process flow of AspectSol is shown below:
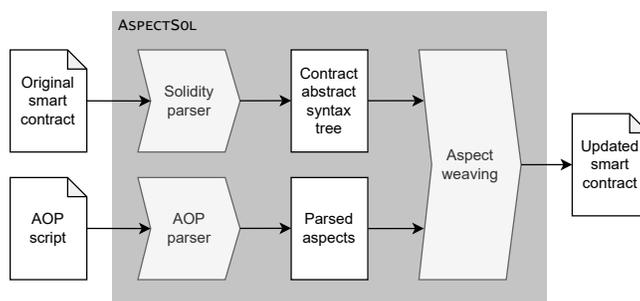
4 Runtime Verification Using AspectSol

In order to show the use of AspectSol, in particular for runtime verification instrumentation, and enable a qualitative assessment of the utility of the tool, we present three small case studies of smart contracts. For all three, we present verification code for runtime verification of desirable properties using AspectSol. All three examples can be found in the AspectSol repository.

Guarding against reentrancy attacks. Since a transaction invoking a smart contract executes in one go, with no interaction or interference from other blockchain transactions, as a programmer one sees smart contract functions as atomic. This eliminates having to consider concurrency issues, making design and implementation easier. Unless a function explicitly calls another function, no other function code will be executed before the termination of the current call. For example, consider a smart contract which implements a wallet in which owners may deposit funds and send them out to other users. In addition, one may have further functionality such as placing an amount of money in escrow for another user. Atomicity means that when the function `placeInEscrow(..)` is called, the programmer need not be concerned that there may be a concurrent call to withdraw funds, thus sufficing to check that there are enough funds available at the beginning of the call to `placeInEscrow(..)`.

However, reality is not so simple. If `placeInEscrow(..)` makes a call to another smart contract, that smart contract may call back and withdraw funds. The atomicity assumption thus breaks down. To make things worse, on Ethereum the

mechanism for a smart contract to send funds to another, is to invoke a special function `transfer` (or `send`) on the receiving smart contract. This is an opportunity for the receiver to call the original contract back, once again breaking the assumption of the atomicity of function calls. Such reentrancy was the source of bugs which led to the loss of the equivalent of millions of dollars.

One way in which reentrancy can be ruled out altogether is to use a Boolean flag to ensure such code is executed only once until a transaction is complete. Some developers do this manually where and when needed, whilst others advocate the use of a function modifier which uses a blanket rule to check that a `running` flag is false and set it to true upon entry to every function and reset it upon exit. Using AspectSol, we can refine the latter to be used only if and when control is yielded through a call to transfer funds as shown below:[5]

```
aspect SafeReenentrancy {
    add-to-declarations * { private bool running = false; }

    before execution-of *.* { require (!running); }
    before call-to *.transfer() { running = true; }
    after call-to *.transfer() { running = false; }
}
```

This is a universal solution in that it can be applied to *any* smart contract, without adding complexity in the code. In addition, gas costs are kept to a minimum, setting and resetting the flag only when transfers of funds are attempted.

Enforcing properties. Consider the wallet smart contract, with a property that states: *No more than 1000 outgoing payments, or 100 ether[6] may be sent from the wallet, unless the smart contract is first verified by a trusted regulator.* If `sendFunds()` is the function provided by the smart contract to send funds to third parties, and `verifyWallet()` is the function used by the regulator to verify the owner of the wallet, we can encode runtime checks to ensure adherence to the specification using the following aspect:

```
aspect WalletVerification {
    add-to-declarations Wallet {
        private bool is_verified = false;
        private uint number_of_payments = 0;
        private uint sum_of_payments = 0;
    }

    before execution-of Wallet.sendFunds(payable address dst, uint amount) {
        if (!is_verified) {
            require (number_of_payments < 1000);
            require (sum_of_payments + amount <= 100 ether);
            number_of_payments ++;
            sum_of_payments += amount;
        }
    }
    after execution-of Wallet.verifyWallet() {
        is_verified = true;
    }
}
```

---

[5] In practice, we would also need to do this for the `send` function.
[6] Ether is the native cryptocurrency used in Ethereum.

Note that `require` stops and reverts the computation altogether if the condition fails. In this manner, any attempt to send funds beyond the set limits is aborted, effectively enforcing the property.

Adherence to a business process flow. Finally, we show how AspectSol can be used to runtime verify that a smart contract can only be used as intended. Consider a casino smart contract which allows the casino to open bets on a coin toss, enabling a player to place a guess, and then resolve the bet. The expected business process flow of the smart contract is shown below:
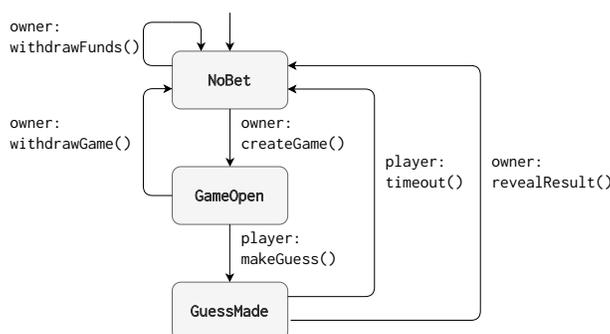
Transitions are tagged by a pair `party:function`, denoting that the transition is taken when the identified party calls the named function. Furthermore, we take the semantics of the notation to indicate that invoking a function when in a state with no outgoing transition tagged with that function (and calling party) should fail. For instance, the function `withdrawFunds()` should only succeed when in mode `NoBet` and called by the `owner` of the casino, thus disallowing the owner from taking funds to leave the player without their winnings after they guessed correctly. We can encode such a business process flow to ensure that the implementation does not diverge from the expected behaviour using aspects as follows:[7]

```
aspect CasinoBusinessProcessFlow {
    add-to-declarations Casino {
        enum Mode { NoBet , GameOpen , GuessMade }
        private Mode mode = Mode.NoBet;
    }

    before execution-of Casino.createGame() {
        require (mode == Mode.NoBet);
        require (msg.sender == owner);
    }
    after execution-of Casino.createGame() {
        mode = Mode.GameOpen;
    }
    ...
}
```

It is worth noting that such AspectSol code would typically be generated by a runtime verification tool from the graph-based specification. In practice, we

---

[7] The full code of the example can be found in the tool repository.

can make the specification notation richer by, for instance, allowing conditions on the transitions to specify properties such as allowing `createGame()` only if there are sufficient funds in the smart contract. Similarly, we can deal with failure in a manner other than simply by disallowing violating calls, e.g. rewarding the player with a win if the casino attempts a disallowed action after the player has placed a bet. The fact that the specification is made independent of the code (which we do not even show here) is the strength of using aspect-oriented programming to specify such properties.

5   Discussion and Conclusions

The only other aspect-oriented programming tool specifically designed for smart contracts which we are aware of is that discussed in [8] — also designed for Solidity but adopts the same pointcuts used in traditional imperative and object oriented language AOP tools. In contrast, we chose to reassess relevant pointcuts in the context of smart contracts. In terms of weaving approach, it appears to be similar to that used in AspectSol, although they make more extensive use of modifiers to instrument code. Direct comparison is, however, not possible since their tool is not available. Many other aspect-oriented programming tools can be used for smart contracts written for platforms which support traditional languages, but these approaches do not specifically address concepts specific to smart contracts. In particular for runtime verification, having native notions of digital asset transfers, parties, and access to modifiers and other tags can be particularly useful.

One valid question is whether one really needs aspect-oriented programming in Solidity, given it provides modifiers, which allow for tagging functions whose behaviour will be changed accordingly, e.g. by adding code before or after the main body of the function. Indeed, some simple use of aspect-oriented programming, e.g. injecting advice at the start or the end of a function can be done using modifiers. However, this has severe limitations in that one cannot inject code on the caller's side, or around specific calls to external functions — functionality which AspectSol provides. Similarly, tag-based filtering and manipulation is a powerful tool which cannot be replicated using modifiers. Finally, modifiers reside within a particular smart contract, and thus lose advantages of separation-of-concerns between the business logic and the cross-cutting aspects, and of reuse. Despite modifiers being a powerful programming construct, they do not replace the role an aspect-oriented tool can provide.

We have presented AspectSol, an aspect-oriented programming tool for Solidity, designed specifically for smart contracts and going beyond traditionally used aspects, pointcuts, and advice for imperative and object-oriented languages. Although Solidity is an imperative language, smart contract notions of value flow, and interacting parties provide an opportunity for more domain specific aspect-oriented programming. In particular, we have argued and showed how the tool is particularly suited to instrument runtime monitoring and verification

code into smart contacts, and we are currently in the process of redesigning ContractLarva [5, 1] to use AspectSol for instrumentation.

References

1. Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring smart contracts: ContractLarva and open challenges beyond. In: Colombo, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 113–137. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_8

2. Chen, F., Rosu, G.: Java-MOP: A monitoring oriented programming environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 546–550. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_36

3. Coady, Y., Kiczales, G., Feeley, M.J., Smolyn, G.: Using aspectC to improve the modularity of path-specific customization in operating system code. In: Tjoa, A.M., Gruhn, V. (eds.) Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001. pp. 88–98. ACM (2001). https://doi.org/10.1145/503209.503223

4. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D.D., Fantechi, A. (eds.) Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5596, pp. 135–149. Springer (2008). https://doi.org/10.1007/978-3-642-03240-0_13

5. Ellul, J., Pace, G.J.: Runtime verification of Ethereum smart contracts. In: 14th European Dependable Computing Conference, EDCC 2018, Iaşi, Romania, September 10-14, 2018. pp. 158–163. IEEE Computer Society (2018). https://doi.org/10.1109/EDCC.2018.00036

6. Ethereum: Solidity. Online Documentation `http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html` (2016)

7. Havelund, K.: Runtime verification of C programs. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop, FATES 2008, Tokyo, Japan, June 10-13, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5047, pp. 7–22. Springer (2008). https://doi.org/10.1007/978-3-540-68524-1_3

8. Hung, C., Chen, K., Liao, C.: Modularizing cross-cutting concerns with aspect-oriented extensions for Solidity. In: IEEE International Conference on Decentralized Applications and Infrastructures, DAPPCON 2019, Newark, CA, USA, April 4-9, 2019. pp. 176–181. IEEE (2019). https://doi.org/10.1109/DAPPCON.2019.00033

9. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001,

Proceedings. Lecture Notes in Computer Science, vol. 2072, pp. 327–353. Springer (2001). https://doi.org/10.1007/3-540-45337-7_18

10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer (1997). https://doi.org/10.1007/BFb0053381

11. Nofer, M., Gomber, P., Hinz, O., Schiereck, D.: Blockchain. Business & Information Systems Engineering **59**(3), 183–187 (2017)

12. Shin, H., Endoh, Y., Kataoka, Y.: ARVE: aspect-oriented runtime verification environment. In: Sokolsky, O., Tasiran, S. (eds.) Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4839, pp. 87–96. Springer (2007). https://doi.org/10.1007/978-3-540-77395-5_8

13. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)