Verified Abstract Interpretation of Digital Contracts

Fritz Henglein[1,2], Christian Kjær Larsen[1], Agata Murawska[1,2]

University of Copenhagen[1] and Deon Digital[2]

VPT 2021, 2021-03-28

Overview

- Domain-specific languages
- Contracts
- Verification
- Discussion

For more information, see Proc. Workshop on Trustworthy Smart Contracts, 2020.

Domain-specific languages (DSLs)

- DSL: Language for expressing specifications of data, processes and properties within a limited domain of discourse
  - "Little languages" (Jon Bentley, CACM programming pearls, 1986)
- Domain adequacy: Reflects domain ontology (concepts, objects, processes and their composition)
  - . . . and nothing else.
  - Facilitates use by domain experts (often $\neq$ programmers)
- Duality: Process+data
  - Interpreted/used in multiple ways, not fixed at language design time.
  - Facilitates analyzability (security, safety, synthesis, optimization, simulation, abstract interpretation . . .)
- Examples: text processing (regular expressions, Sed, awk, . . .), query processing (SQL), graph drawing (DOT, Neato, . . .), hardware design (VHDL, Verilog), business processes (BPEL, BPMN), financial and commercial contracts (MLFi, CSL, DAML, . . .), . . .

GPLs versus DSLs: Design duality

Does the syntax look familiar to a programmer?	Is the syntax readable by a domain expert?
Does the language have constructs and libraries for expressing efficient algorithms and data structures?	Does the language have no domain-irrelevant constructs?
Is the language expressive enough?	Is the language too expressive?
Is the language design stable?	Is the language evolvable?

GPLs versus DSLs

If programming is finding a needle (useful program) in a haystack (of useless, incorrect, bad or outright dangerous programs), then:

- GPL approach: Come up with a (necessarily big) single haystack that contains all needles for all domains (higher needle cardinality).
- DSL approach: Come up with small haystacks, each containing many needles for a particular domain (higher needle density).

Expressiveness versus analyzability:

- Programmer: I code algorithmic processes, let other people worry about analyzing them. Give me more expressiveness!
- Goethe: "In der Begrenzung zeigt sich der Meister."
- No analysis $\rightarrow$ little benefit of DSLs.
  - Use frameworks/libraries.

Contract

- **Agreement** between two or more parties
- Specification of (future) **obligations**, **permissions** and **prohibitions**
- Properties:
  - **Identifiable parties**: Required by law (AML, KYC) and for recourse (court action)
  - **Consideration**: Not one-sided exchange of resources (money, goods, assets, services)
  - **Confidentiality**: By default not disclosed to other parties (unless required by regulation)
- Examples: Sales, services, lease, financial (loan, bond, derivative), insurance, shareholder, mobility, transportation etc.

Contracts: Semantic model

- Basic observation about a contract: Given a *trace* of events (actions and observations), is it an *acceptable, complete execution* of the given contract or not?
- Extensional semantics: A contract *denotes* the set of acceptable, complete executions.
  - Any two specifications that denote the same set are *equivalent*.

Event e: A time-stamped, signed statement asserting that a real-world event has happened, with attached verifiable evidence.
$e \in E = \{\mathrm{Transfer}(\mathrm{Alice}, \mathrm{Bob}, 30 \cdot \mathrm{USD}, 2019\text{-}06\text{-}20), \ldots\}$.
(Evidence left out.)

Trace t: A sequence of events. $t \in T = E^*$.

Contract c: (Description of) a trace language (set of traces).
$c \in \mathcal{C} = 2^T$.

Contracts

Example (Exchange contract)

$$
\begin{aligned}
\text{phoneSale} = \\
&= (\text{Transfer}(\text{Alice}, \text{Bob}, 30 \cdot \text{USD}, T); \text{Transfer}(\text{Bob}, \text{Alice}, 1 \cdot \text{iPhone}, T')) \mid \\
&\quad T \leq \text{Tuesday} \wedge T' \leq T + 1 \cdot \text{day} \\
&+ (\text{Transfer}(\text{Bob}, \text{Alice}, 1 \cdot \text{iPhone}, T'); \text{Transfer}(\text{Alice}, \text{Bob}, 30 \cdot \text{USD}, T)) \mid \\
&\quad T' \leq \text{Tuesday} \wedge T \leq T' + 8 \cdot \text{day}
\end{aligned}
$$

Residual contracts (language derivatives)

Definition

Let $C \in 2^{E^*}$; ϵ denotes empty sequence.

- C is *terminated successfully* if $C = \{\epsilon\}$.
- C is *possibly terminated successfully* if $\epsilon \in C$.
- C is *satisfiable*, if $C \neq \emptyset$.
- e is *valid* for C if $e\backslash C$ is satisfiable.
- *Residual contract* of C under $e \in E$ is

$$e\backslash C = \{s \mid es \in C\}.$$

- e is *acceptable* for C if $e\backslash C$ is satisfiable.

Intuition: C expresses what remains to be done in a contract. $e\backslash C$ expresses what remains to be done after e has happened.
In language theory *residual* is called *(left) derivative*.

Contract manager

- Contract manager: Interpreter *consistent* with denotational semantics.
  - Not *determined* by it. Variations: logging, event validation, escrow management, transaction management, incompleteness, ...
  - Component-oriented: identiy/authorization manager(s), resource manager(s), information provider(s), time service, ledger system, ...

Monitoring contract manager

- *Monitoring contract manager*:
  1. Receive contract C_0 and launch new process.
  2. $C := C_0$;
  3. Receive event e.
     - ★ Validate e. If e valid and $e \backslash C$ is satisfiable then $C := e \backslash C$, return "accepted" and inform contract parties; otherwise return "rejected".
  4. Receive query request.
     - ★ Return log of validated events (history); return C (future).
  5. Receive terminate request.
     - ★ If C is possibly terminated successfully, return "success" and exit process;
     - ★ otherwise, return "breach because of premature contract termination" and exit process.

Transactional contract execution (static escrow)

Like monitoring execution, except resource transfers are delayed until termination (static escrow).

1. Receive contract C_0 and launch new process.
2. $C := C_0$; $t := 0$.
3. Receive event e (*not* validated).
   - If $e \backslash C$ is satisfiable then $C := e \backslash C$ and $t := t + \mathrm{eff}(e)$ and return "accepted"; otherwise return "rejected".
4. Receive query request.
   - Return (C, t).
5. Receive terminate request.
   - If C is possibly terminated successfully, then submit t to trusted resource manager for validation and effecting.
     - ★ If validation/effecting succeeds, return "success" and exit process;
     - ★ otherwise, return "breach because of insufficient resources" (no transfers are effected) and exit process.
   - Otherwise, return "breach because of premature contract termination" and exit process.

Contracts: Compositional specification

- Idea: Put contracts together from subcontracts
- What are the primitives and combinators?
  - Primitives: *resource transfers* (who is required to give what to whom by when?), success (nothing to do), fail, others (not included here)
  - Subcontract combinators: one of them, all of them, one after the other, repeatedly
- Contract Specification Language (CSL):

$$
\begin{aligned}
c ::= {} & \mathsf{Success} \mid \mathsf{Failure} \mid c_1 + c_2 \mid c_1 \parallel c_2 \mid c_1 ; c_2 \mid \\
& \mathsf{Transfer}(A_1, A_2, R, T \mid P).c \mid f(\boldsymbol{a}) \\
D ::= {} & \{ f_i[\boldsymbol{X_i}] = c_i \}_i \\
r ::= {} & \mathsf{letrec}\ D\ \mathsf{in}\ c
\end{aligned}
$$

- Andersen, Elsborg, Henglein, Larsen, Simonsen, Compositional Specification of Commercial Contracts (2006).
  - Basis for *Deon Digital CSL*, financial instrument issuance, life-cycling

Contracts: Relational semantics

- Idea: Specify inductively when a trace satisfies a CSL contract

$$\frac{}{\delta \vdash_D \epsilon : \mathsf{Success}} \qquad \frac{\delta \vdash_D s : c_1}{\delta \vdash_D s : c_1 + c_2} \qquad \frac{\delta \vdash_D s : c_2}{\delta \vdash_D s : c_1 + c_2}$$

$$\frac{\delta \vdash_D s_1 : c_1 \quad \delta \vdash_D s_2 : c_2 \quad (s_1, s_2) \rightsquigarrow s}{\delta \vdash_D s : c_1 \parallel c_2} \qquad \frac{\delta \vdash_D s_1 : c_1 \quad \delta \vdash_D s_2 : c_2}{\delta \vdash_D s_1\, s_2 : c_1; c_2}$$

$$\frac{Q[\![P]\!]^{\delta'} = \mathsf{true} \quad \delta' \vdash_D s : c \quad (\delta' = \delta, \{A_1 \mapsto a_1, A_2 \mapsto a_2, R \mapsto r, T \mapsto t\})}{\delta \vdash_D \mathsf{transfer}(a_1, a_2, r, t)\, s : \mathsf{Transfer}(A_1, A_2, R, T \mid P).c}$$

$$\frac{\boldsymbol{X} \mapsto \boldsymbol{v} \vdash_D s : c \quad f(\boldsymbol{X}) = c \in D \quad v = Q[\![\boldsymbol{a}]\!]^{\delta}}{\delta \vdash_D s : f(\boldsymbol{a})}$$

Contract properties

- Not all *possible* formal contracts are *fair* ("real") contracts.
  - Wrong participants, information providers, resource types (formulation errors).
  - Lack of *consideration* (fair exchange/fair value for all parties).
- Idea: Check properties of *general importance*.
- Given any contract:
  - Who may be *participating*?
  - Who may be *obliged* to do something? (Do we have their signature?)
  - Are *all* executions *fair*? (Do they have consideration? Might one party get unfair advantage?)

Static analysis by abstract interpretation

- Idea:
  - Analyze a contract by abstractly interpreting contracts in a suitable abstract domain $A = (L, \ldots)$.
  - Ensure that interpretation is *sound* wrt. semantics.
  - Extract analysis result (superset of participants, upper bound on unfairness, etc).
- Define $\beta : T \to L$ for *abstract interpretation* of single trace.
- Define abstract combinators $+^\sharp, \ldots$ for concrete combinators $+, \ldots$, respectively, such that

$$\begin{aligned} \beta(s_1) \sqsubseteq A[\![c_1]\!] &\Rightarrow \beta(s_1) \sqsubseteq A[\![c_1]\!] +^\sharp A[\![c_2]\!] = A[\![c_1 + c_2]\!] \\ \beta(s_2) \sqsubseteq A[\![c_2]\!] &\Rightarrow \beta(s_2) \sqsubseteq A[\![c_1]\!] +^\sharp A[\![c_2]\!] = A[\![c_1 + c_2]\!] \end{aligned}$$

  and similarly for other combinators.
- Challenge: Formalization and mechanization of abstract interpretation of infinite unfolding of recursion.
  - Employ Schmidt (1995): Coinductive interpretation of abstract derivation trees. Prove and mechanize its soundness.

Soundness of abstract interpretation framework

Theorem

If certain properties for the abstract operators $+^\sharp, \ldots$ *in relation to* β *are satisfied (see paper) then*

$$\forall s \in T, (\delta \vdash^D s : c) \wedge \delta \sim m \Rightarrow \beta(s) \sqsubseteq A[\![c]\!]_m.$$

Mechanization

The following is mechanized in Coq:

- **Multiple semantics of CSL**: denotational, relational, small-step (online, monitoring); proofs of their equivalence
- **Static analysis framework** based on abstract interpretation of natural semantics
  - Example analyses: participation, fairness (see paper)
- **Mechanization** of semantics, equivalences, abstract interpretation framework, its soundness, example analyses in Coq

Paper at `http://fc20.ifca.ai/wtsc/WTSC2020/WTSC20_paper_8.pdf`

- Coq source code at `ayertienna.github.io/csl_formalization_wtsc20.zip`

Discussion: Digital contracts versus smart contracts

- Ethereum-style smart contract: contract, control and settlement conflated as single-threaded program in GPL
- Digital contract management:
  "smart contract = contract + control + settlement"[1]
  - Domain-specific/"small" language for contracts of their own
  - Same contract, running on multiple platforms (Corda, Fabric, Ethereum, etc)
  - Multiple contracts, same contract manager ("generic" smart contract)
  - Same contract, multiple contract managers:
    - ★ Same contract, multiple implementation techniques (centralized, TEEs, zero-knowledge, etc.)
    - ★ Same contract, multiple instrumentations (with/without logging, with/without escrow/collateral, etc)
  - Prove properties about a contract without having to prove properties about the implementation of its contract manager.

[1] Kowalski, *Algorithm = Logic + Control*, CACM 1979

Discussion: Digital contracts versus smart contracts

- Digital contract versus smart contract analysis/verfication: DSL/little language analysis versus GPL/big language analysis.
  - analysis of digital contracts (CSL, MLFi, . . .) *excluding* contract manager (Haskell, Kotlin, OCaml, etc.; GPL/big language) vs.
  - analysis of smart contract (Solidity/EVM, Kotlin, Go, etc.; GPL/big language).

CSL semantics: Discussion

- Three different inductive reasoning principles = three different static analysis/abstract interpretation frameworks:
  - Induction on proof of containment ($\cong$ induction on the legal parses) of event sequences (grammar view)
  - Induction on abstract syntax of contracts (hierarchical view)
  - Induction on length of traces (automata view)
    - ★ Basis for contract life-cycle management (run-time monitoring), optimization and compilation
    - ★ Automatically makes any contract analysis based on denotational or relational semantics applicable not only to original contracts, but to residual ("running") contracts in any state.
- Semantics-driven language design: What do/should contracts denote? In which domains? How are denotations combined? Can they be freely ("orthogonally") combined?
  - Systematic derivation of automata-view (operational) semantics from grammar-view/denotational semantics.
- Reference semantics for correctness of contract managers, e.g. early matching, routed matching, greedy matching, escrow management, netting, etc.

CSL semantics: Summary

- CSL with denotational semantics: Compositional reasoning on syntax of contracts (reasoning on sets of traces)
- CSL with relational semantics: Inductive reasoning on containment relation
- CSL with monitoring semantics (computing residual contracts): Inductive reasoning on length of trace
  - Basis for contract life-cycle management (run-time monitoring), optimization and compilation
  - Automatically makes any contract analysis based on denotational or relational semantics applicable not only to original contracts, but to residual ("running") contracts in any state.