software

MDPI

Article

SocialRbSC Language for Social Agents and Role Identity Development with Smart Contract Technology

Orçun Oruç *,† **and Uwe Aßmann** *,†

Software Technology Group, Technische Universität Dresden, Nöthnitzer Straße 46, 01187 Dresden, Germany
* Correspondence: orcun.oruc@tu-dresden.de (O.O.); uwe.assmann@tu-dresden.de (U.A.)
† These authors contributed equally to this work.

Abstract

Research on role-based modeling and agent-oriented programming has approached core concepts such as roles, contexts, agents, and agent types from several partially reconciled traditions: object-oriented role modeling (Object Teams, the Role Object pattern, powerJava, SCROLL), the OMG/MDA modeling tradition, and role-based collaboration. No single universally accepted definition has emerged. This paper addresses that situation by introducing SocialRbSC, a language and framework for specifying social agents, role-based coordination, and reasoning-oriented smart-contract behavior. SocialRbSC enables dynamic behavior adaptation through explicit role modeling, role-player bindings, and reasoning constructs such as beliefs, intentions, and goal–plan trees. Collaborative actions in changing environments can be represented through smart contracts that enforce role constraints and behavioral rules over shared blockchain state. The paper provides a precise language specification including its metamodel, concrete syntax grammar, well-formedness constraints, and the mapping from language constructs to generated Solidity smart-contract artifacts. Because implementing the BDI machinery directly as Solidity contracts is a deliberate and cost-bearing design decision, the paper also gives an operational semantics for beliefs, intentions, goal–plan trees, and the deliberation cycle, fixes the environment interface through which external information enters the belief base, states correctness properties for the construct-to-Solidity mapping, and reports a static-analysis-based security assessment of the generated artifacts. The runtime architecture is fully on-chain: roles, compartments, and the BDI reasoning machinery (beliefs, intentions, goal–plan trees, and the deliberation cycle) are implemented as Solidity contracts, while the Java annotation DSL and its code generator form the off-chain modeling and generation toolchain. The main contributions are: (1) a precisely specified language model for social agents, roles, and compartments, (2) an integration of role-based modeling with smart-contract programming, (3) a code-generation pipeline with compilable generated artifacts and stated correctness properties, (4) a reproducible gas-cost evaluation with absolute measurements, scaling experiments across agents, roles, and goal–plan-tree sizes, and a static-analysis-based security assessment of the access-control-hardened reference implementation, and (5) a publicly available software repository. The resulting approach provides a transparent, traceable, and tamper-resistant structure for context-aware social coordination on blockchain platforms.

Keywords: smart contract programming; role-based programming; agent-oriented programming; solidity programming language; code generation; metamodeling in software engineering

Check for updates

Academic Editor: Andrea Pinna

Received: 18 September 2026
Revised: 27 October 2026
Accepted: 16 November 2026
Published: 29 November 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Software systems increasingly operate in environments that change during execution, so they must adapt their behavior rather than only execute fixed object-oriented logic. In such settings, a purely object-centered design often makes coordination logic difficult to express, especially when multiple participants must collaborate under changing goals and constraints. Agent-oriented abstractions help address this problem by treating autonomous entities as explicit units of coordination. Each agent can pursue goals, react to environmental changes, and cooperate with other agents to solve a larger task. This makes agent-oriented design a relevant foundation for systems in which collaboration, reasoning, and context-sensitive behavior are central concerns.

1.1. Problem Statement

The ROSI project (https://tu-dresden.de/ing/informatik/sya/ps/research/projects/archive/RoSI access on 15 November 2026) enables us to realize the next-generation role-based language family for role-based software infrastructures. In this long-term project, role-based languages have been researched in different aspects such as context-sensitive applications, business process models with enterprise resource planning software, and dynamically adapted software systems. Common notations, formal foundations of role-based programming languages, role-based model editor, and role object model (CROM (https://github.com/Eden-06/CROM access on 15 November 2026)) have been researched over the last few years.

The formal definition of roles has been investigated from multiple perspectives over the past few decades. Within the ROSI project, related work has addressed business-process modeling, role-based programming languages, and role-based modeling languages. These perspectives are not mutually consistent: object-oriented role modeling treats a role as a dynamically acquired view on an object (the Role Object pattern [1], powerJava [2], SCROLL [3], and contextual languages such as Object Teams [4,5]), the OMG/MDA tradition treats roles as modeling concepts to be transformed into platform artifacts [6], and role-based collaboration formalizes roles as constraints over groups of agents [7]. Because these traditions disagree on what a role is, on whether a role has its own identity and state, and on how roles are bound, no single universally accepted definition has emerged. Building on that background, this paper focuses on roles together with closely related concerns such as trust, auditing, autonomy, and planning in complex role-based applications. To support that goal, the paper develops a domain-specific language for social role development.

In this paper, a social agent is an abstraction of an autonomous entity that perceives its environment, reasons about it, and acts within a social context. For example, a professor and a student may participate in the same collaboration while enacting different roles, maintaining different mental states, and pursuing different intermediate goals. When the state of one participant changes, the other participant may need to react, update its expectations, or adapt its plan. A goal is a mental representation of a process that can be achieved through agent anticipation [8]. This depicts the main limitation of conventional role modeling for the present problem: roles and compartments [9] structure participation and context, but they do not by themselves provide reasoning capabilities. In their traditional form, they organize relations between software entities, yet they do not explain how role players deliberate, revise goals, or coordinate adaptive behavior.

The paper is guided by the following research questions.

1. **RQ1 (Cooperation)**: Which are the most suitable role characteristics to realize with smart contracts?

2. **RQ2 (Coordination)**: How can agents with social and proactive skills coordinate smart contracts?

Based on the problem statement and the proposed language design, the study evaluates the following working hypotheses.

Hypothesis 1. *Smart-contract-based role systems are best supported by role characteristics that remain explicit and verifiable at runtime, namely compartment-scoped role activation, role-player binding, role-specific methods, and declarative preconditions over shared state.*

Hypothesis 2. *Agents can coordinate smart contracts more effectively when social and proactive behavior is represented through explicit reasoning structures such as beliefs, intentions, goal–plan trees, and deliberation cycles, even if those structures become the main source of execution cost.*

1.2. Contributions

The contributions of this paper are stated here, up front, rather than only at the end. They are listed below; Section 9 restates them in a single paragraph.

- **A unified language model** (addressing RQ1): SocialRbSC integrates roles, compartments, social agents, and smart contracts within one executable language model for context-dependent applications. The paper provides a precise language specification including its metamodel, concrete syntax, well-formedness constraints, an operational semantics for the reasoning layer, the environment interface through which external information enters the belief base, and an on-chain runtime architecture with an off-chain modeling and generation toolchain.
- **Explicit reasoning support** (addressing RQ2): The language introduces beliefs, intentions, goal–plan trees, and deliberation cycles as first-class constructs for coordinating proactive agent behavior, and the paper makes the cost of doing so on-chain explicit.
- **A code-generation pipeline with stated correctness properties**: The paper describes the three-stage pipeline (annotation processing, intermediate representation, Solidity emission) that maps language constructs to compilable smart-contract artifacts. It states the correctness properties of this mapping (binding soundness, access-control completeness, finite-state-machine safety, and precondition enforcement) and gives a proof sketch.
- **A reproducible evaluation and security assessment**: The paper reports absolute deployment and execution costs for the main modules, a baseline comparison against hand-written Solidity, an end-to-end medical-supply cycle, role-binding scaling up to 50 players, multi-agent/multi-role coordination up to 20 agents, and goal–plan-tree scaling up to 1000 nodes. The measurements show that role binding remains nearly constant per operation as the number of players grows and that the reasoning modules account for 80.1% of the module gas. The paper also checks the generated contracts against common Solidity security patterns, analyzes them with Slither, and reports that the remaining findings are Low severity or below, with one High-impact finding triaged as a false positive.
- **Open-source artifacts**: The SocialRbSC tooling, generated contracts, benchmark scripts, and example models are available in a public repository.
- **A basis for future optimization**: The results establish a clear direction for subsequent work on reducing reasoning cost and testing SocialRbSC in broader decentralized multi-agent settings.

https://doi.org/10.3390/software5040055

1.3. Limitations

- The primary language definition of SocialRbSC utilizes a smart contract language that supports object-oriented programming features by including inheritance, multiple inheritance, composition, message passing, data abstraction and type safety.
- Intelligent agents are illustrated through a use case that specifies the common specification requirements of the language; however, agent-communication standards such as FIPA or KQML are outside the scope of this paper.
- In our approach, roles, compartments, and social agents are all implemented as smart contracts. This also makes the limitations of SocialRbSC clearer, whatever the smart contract cannot do, the language cannot easily express either.

2. Background

The background of this paper comprises the intersection of agent-oriented programming, role-based modeling, and programmable ledgers. This intersection is still comparatively young in the literature, especially when smart contracts are treated not only as transactional artifacts but also as autonomous computational actors. In that direction, Ciatto et al. describe the idea of agent-oriented blockchains, where smart contracts can be interpreted through agent abstractions and endowed with more explicit autonomy [10]. This broader view motivates the SocialRbSC line of work: decentralized infrastructures offer transparency, accountability, and peer-to-peer coordination, but they also impose stricter scalability and execution constraints than conventional Web 2.0 platforms [11].

From the role-modeling perspective, the paper treats collaboration as a central systems concern. Complex collaborative software cannot usually be captured through a single abstraction alone, and role-based thinking offers a way to structure participants, permissions, and context-sensitive behavior [7]. In distributed collaborative systems, interaction is often layered across communication, coordination, organization, and interface concerns, which makes roles useful for representing how actors participate differently across contexts [12]. This is important here because the same core entity may need to operate under different permissions, responsibilities, and social expectations depending on the compartment or society where it acts.

The discussion also positions several implementation techniques for roles. Aspect-oriented mechanisms can attach behavior and state to an existing object identity and are therefore relevant when role behavior cuts across existing structures [13]. Trait-based techniques provide reusable units of behavior that can be composed without forcing a rigid inheritance tree, which is attractive for role composition, although method conflicts must still be resolved carefully [14,15]. This comparison helps explain why smart-contract-based roles also need compositional discipline, but within the tighter execution model of blockchain systems.

On the agent side, the paper builds on the view that software agents are useful abstractions for intelligent and distributed systems because they combine autonomy, proactiveness, and social ability [16]. Social agents extend this view by emphasizing trust, motivation, and socially situated interaction rather than isolated decision making [17]. The SODA methodology is particularly relevant here because it highlights societies, groups, roles, resources, and interaction rules as first-class design elements for agent systems [18]. This directly informs the paper's argument that role models and social structure should be designed together rather than added as an afterthought.

Finally, programmable ledgers provide the execution substrate for these ideas. Smart contracts operate over shared ledger state, and state-changing operations require consensus, which makes on-chain coordination trustworthy but comparatively expensive [11]. The background section therefore frames the main research problem as a balancing act: how to

combine the expressiveness of social and role-based agent models with the transparency and reliability of blockchain execution without ignoring the cost and scalability limits of on-chain computation [10,17].

3. Related Work

This section reviews neighbouring lines of work that motivate SocialRbSC: subject-oriented programming, aspect-oriented programming, ensemble-based programming, role modelling, agent-oriented programming, conceptual modelling with smart contracts, and recent developments in smart-contract engineering.

3.1. Subject-Oriented Programming

Subject-oriented programming aims to improve modularity to provide reusable and scalable software systems. The emphasis is on the subject, which can be thought of as modules in an object-oriented system. Harrison et al. emphasize a subject as a means of a collection of state and behavior specifications that reflect a particular form [19]. A subject can be activated and deactivated to compose another subject. In the case of subject activation, a particular subject creates an instance to execute the object, including the actual data. While deactivating a subject, the object of the subject is eliminated with its data logic. To combine other subjects, a composition rule should be defined in detail on how the components are to be combined [19]. Subjects are formed by compositions that can reside in different scopes through composition rules.

3.2. Aspect-Oriented Programming

The main algorithm in all object-oriented programming languages is to decompose a problem into objects and write the code according to a particular subproblem with these objects. According to Elrad et al. [20], subprograms that are relevant to a particular problem have a disadvantage, as they require both knowledge and cooperation on the part of the programmers of the calling components. Elrad et al. state that aspect-oriented programming offers an implicit invocation mechanism that invokes methods in code without explicitly defining concerns rather than objects or components [20]. These concerns can be disseminated to domain experts in different areas such as distributed computing, access control, synchronization, encryption, and redundant code analysis. Elrad et al. state that major issues regarding aspect-oriented programming include the complexity of verifying the aspect-oriented mechanism in source code, byte code, and object code, as well as the domain expertise required for specifying aspects in an object-oriented programming model [20].

The main aim of aspect-oriented programming is to modularize cross-cutting concerns in software systems. A concern is a specific requirement or consideration that must comply with the overall system goal [21]. System modules are represented through aspects to separate the codebase based on their main tasks, such as error handling, logging, and business logic. Separation of concerns is provided by aspects at runtime or compile time, reducing code duplication and making maintenance easier.

A more advanced technique is described as the superimposition of crosscutting behaviors, wherein crosscuts can be defined as a set of methods within a single object [22]. This can be achieved by message filters to separate concerns in a complex software application. Messages can be defined in terms of input filters by assigning filter types. Filter types, input filters, and output filters can be defined with a filter interface, allowing superimposing of multiple crosscutting concerns [22].

3.3. Ensemble-Based Programming

Although ensembles are closely related to the definition of collaboration and cooperation in context-based and subject-oriented programming, ensembles emphasize the

collaboration and cooperation of multiple components to achieve a common goal in different branches. Ensemble-based programming is realized in the concept of machine learning by combining various models (e.g., classifiers and predictors), abstract data structures in high-level programming languages, parallel and distributed computing, and representation of software components and modules. Main entities are named ensembles which are collections of individuals and each can operate in parallel with its neighbors [23]. The main difference between ensembles and other programming techniques is to admit the first-class computing concept because the ensemble can encompass objects with parent references in parallel execution [23].

3.4. Role Modeling

Role modeling organizes code around the roles that objects play, but the literature offers several incompatible interpretations of what a role is. Steimann's foundational analysis already observed that roles can be represented in fundamentally different ways, namely as a separate entity, as a view on an object, or as a specialization, each with different consequences for identity and state [24]. The Role Object pattern realizes roles as first-class objects attached to a core object at runtime, which supports dynamic reclassification at the cost of explicit role-object management [1]; Riehle and Gross generalize this into role-model-based framework design and integration [25]. powerJava formalizes roles on an ontological foundation and separates the role from its player through explicit power and role dependencies [2]. SCROLL defines roles within a compositional multi-agent design and couples each role to the requirements it fulfills [3]. Object Teams and compartmentalized object-oriented programming treat roles as contextual, scoped views bound to a base object, where a role instance is meaningful only inside its enclosing team or compartment [4,5]. The OMG's model-driven architecture tradition, by contrast, treats roles as modeling concepts that are transformed into platform-specific artifacts rather than as runtime entities [6]. Role-based collaboration formalizes roles as constraints that can be assigned to agents within groups and environments [7]. Kühn et al. propose a family of role metamodels precisely because these underlying views differ across languages [9]. This diversity is the sense in which the literature has not reached universal agreement on roles, contexts, and agent types: role identity, role state, role binding, and role scoping are each defined differently by these traditions. SocialRbSC does not claim to resolve that debate; it commits to one explicit reading by means of compartment-scoped roles with explicit role-player bindings that are materialized on-chain, and it makes that reading precise, correct-by-construction, and executable.

3.5. Agent-Oriented Programming

Agent-oriented programming treats autonomous entities and their interactions as first-class abstractions. Wooldridge and Ciancarini survey the state of the art and describe agents as autonomous, reactive, and social entities that perceive and act in a shared environment [16]. Omicini's SODA framework organizes multi-agent systems into societies and infrastructures, separating application-level agents from the interaction infrastructure [18]. Dignum et al. argue that genuinely social agents require social and normative reasoning beyond purely reactive behaviour, motivating explicit models of beliefs and motivations [17]. Zhu consolidates two decades of role-based collaboration research in the E-CARGO model, which formalizes groups, roles, agents, and environments [7]. Ciatto et al. connect this line of work to blockchains by describing autonomous smart contracts as a form of agent, but they do not provide a modelling language or an end-to-end code-generation pipeline [10]. SocialRbSC builds on these ideas by making roles, compartments, and social agents explicit language constructs that are compiled to on-chain contracts.

3.6. Recent Developments

More recent work has shifted smart-contract engineering towards verification and systematic evaluation. Mavridou et al. propose VeriSolid, a correct-by-design approach that generates verified Solidity contracts from finite-state models [26]. Rouhani and Deters survey the security, performance, and applications of smart contracts, confirming the growing importance of contract-level abstraction and analysis [27]. Ante's bibliometric review documents the rapid expansion of smart-contract research and emphasizes the need for reproducible, comparable evaluations [28]. These developments reinforce the motivation for SocialRbSC: a language that combines role-based modelling with generative code generation and is evaluated through explicit, reproducible gas measurements.

3.7. Conceptual Modeling with Smart Contracts

A process of creating a simplified representation of smart contracts is essential concerning smart contract programming languages. Principally, the conceptual modeling aims at representing the real world in understanding, reasoning, and communicating in the view of complex smart contract application analysis and design [29]. Due to their deployment model, smart contracts are typically immutable after publication and must be designed under the scalability constraints of decentralized networks. Although the number of conceptual modeling tools is scarce, in essence, previous conceptual modeling tools address most of the concepts in conceptual modeling such as Business Process Modeling Notation (BPMN), Finite State Machine (FSM), and Unified Modeling Language.

3.7.1. Caterpillar

Caterpillar was designed as a BPMN-Based Solidity-oriented code generator. Design can be organized by Business Process Modeling Notation (BPMN) notation. In the context of business processes, the design of Caterpillar ensures the parties involved in a collaboration comply with a collaborative process model [30]. The authors utilize the blockchain technology with smart contracts for compliance by design, which means that parties execute steps of the business process by invoking a transaction, and transactions are checked by the blockchain decentralized authority in terms of the current state of processes, inputs, and outputs [30]. Main processes are represented with smart contracts according to compliance by design that ensures no participant can execute a transaction and does not accept the collaborative process model [30].

3.7.2. FSolidM

FSolidM is a finite-state-machine-based approach to conceptual modeling for Solidity code generation. A developer can define a smart contract as a finite state machine (FSM) to make the operational semantics of the contract easier to inspect. The tool created by Mavridou et al. [31] consists of three main parts: a formal model, a graphical editor, and a code generator [31]. *FSolidM* provides a way to validate that an FSM is well specified before generating Solidity code [31]. Attributes and language definitions can be configured through the graphical editor, which is integrated with parsing support for Solidity syntax checking [31].

3.7.3. iContractML

iContractML is a graphical tool we developed to generate smart contract code for multiple platforms, including Solidity, Hyperledger Composer, and Azure-based digital certificate systems. The tool's metamodel defines four core elements: participants, transactions, assets, and two types of relationships by explaining that TransRel for transaction links and AssetRel for asset links. Using these building blocks, iContractML provides template-based

code generation [32]. In the language metamodel, the relationship metaclass is referenced by three other classes: Element, Event, and SContract. SContract serves as the top-level container for any smart contract generated by the tool, regardless of the target platform. Events, meanwhile, use relationships to link participants, transactions, and assets together. Each event also stores two basic pieces of metadata: a name and a description [32].

4. Design of the SocialRbSC Framework and Language

Figure 1 presents the structural view of the SocialRbSC framework. At this level, the framework is organized around the relation between the society, the participating social agents, and the roles they enact inside a compartment. The diagram shows that coordination is not modeled as a flat exchange of method calls; instead, the framework introduces explicit entities for agent state, role binding, and interaction context. This is important because the framework is intended to combine role-based coordination with agent reasoning, so the central design concern is how those concepts remain connected in a single execution model.

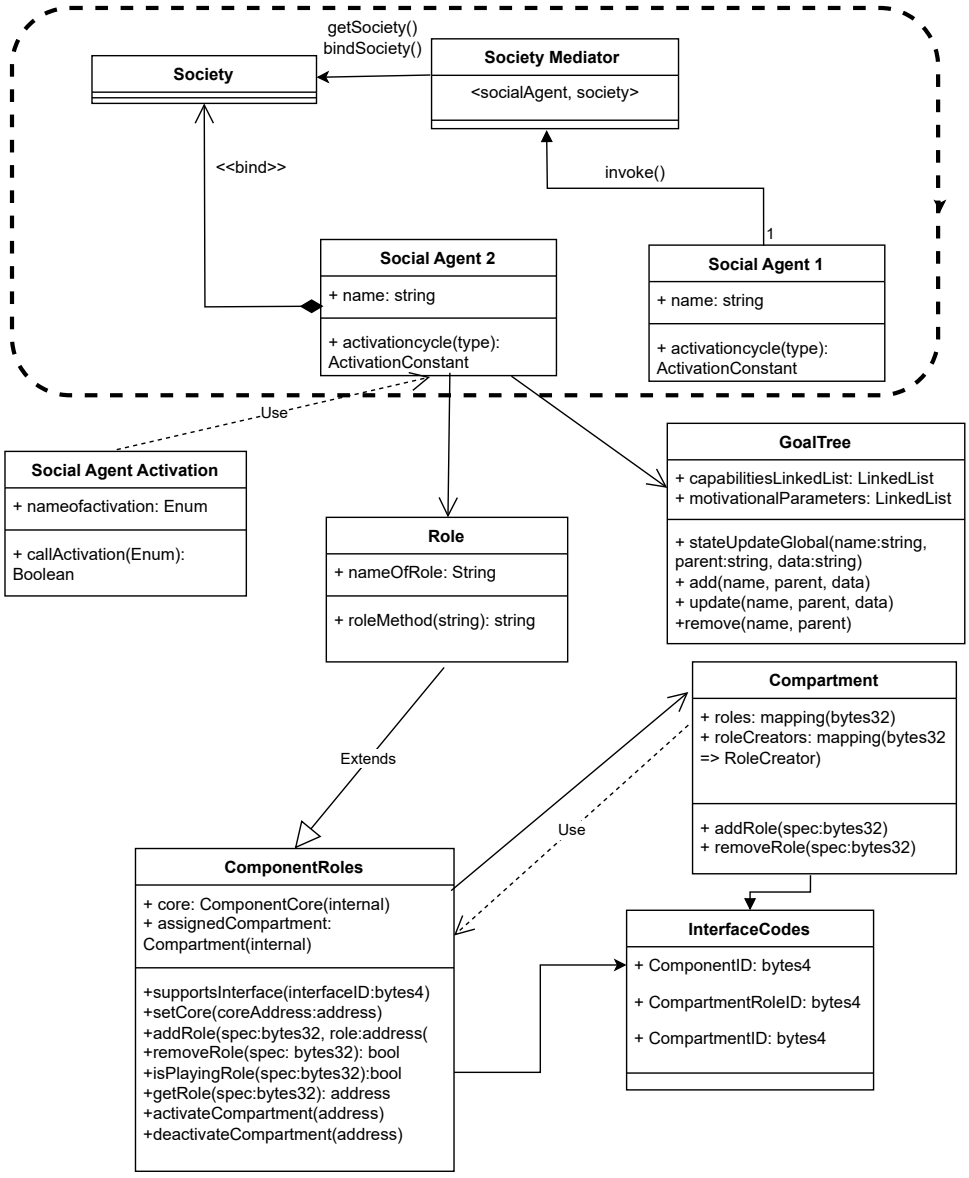


Figure 1. Agent-to-Agent GoalPlanTree Class Diagram.

The same figure also clarifies how reasoning is attached to social interaction. Goal–plan-tree elements represent the deliberative part of the agent, while the role and society elements represent the social context in which that deliberation is carried out. These elements show that an agent does not reason in isolation: beliefs, intentions, goals, and plans are evaluated in order to support actions that are meaningful inside a compartment and toward other agents. For that reason, the framework design can be read as a bridge between internal decision making and externally observable role behavior.

Figure 2 explains the package and component organization of the SocialRbSC language. The language is structured so that the social layer, the role layer, and the reasoning layer can be declared separately while still contributing to one smart-contract specification. In practical terms, this means that the language provides constructs for defining the society and compartments, for assigning and activating roles, and for expressing mental-state concepts such as beliefs, intentions, and deliberation steps.

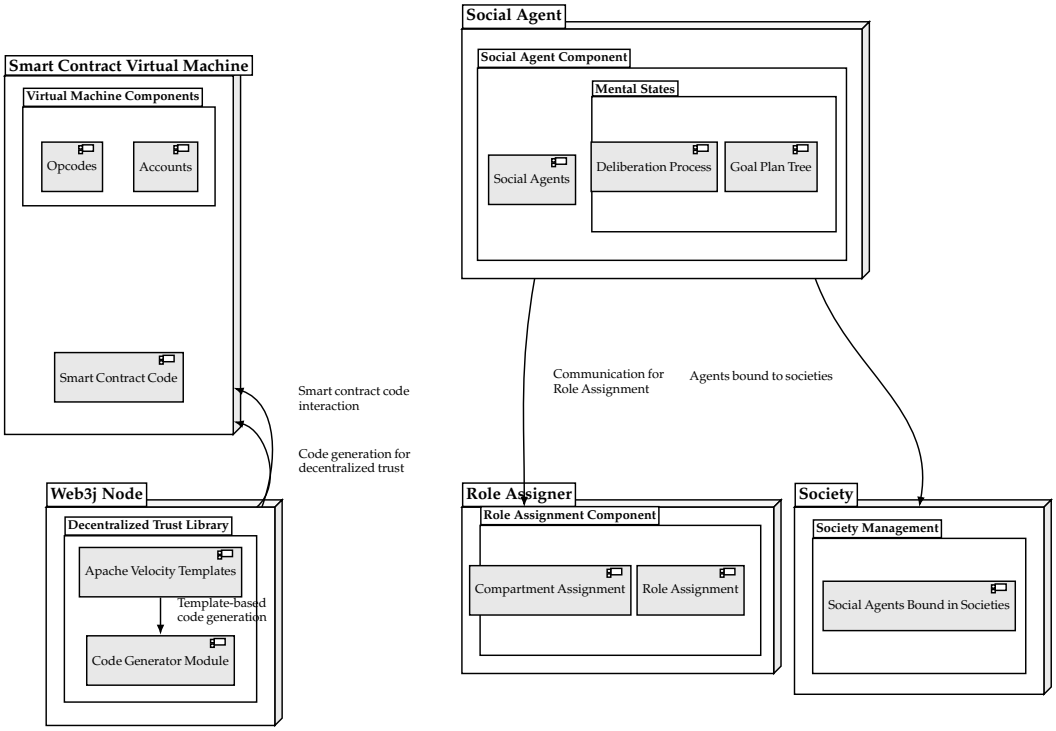


Figure 2. Packages and components of SocialRbSC Language.

From a design perspective, the package view shows how the language moves from declaration to execution. The upper-level packages define the social space in which agents participate, the middle packages describe role relationships and permissions, and the lower-level components support executable behavior such as preconditions, role methods, and reasoning-driven decisions. Explaining the language in this layered way is useful because it makes clear that SocialRbSC is not only a set of annotations; it is a structured language whose components are grouped according to the responsibilities they play in modeling social coordination on blockchain platforms.

5. SocialRbSC Language Specification

This section provides a precise definition of the SocialRbSC language, covering its abstract syntax (metamodel), concrete syntax annotations, well-formedness constraints, execution semantics, and the mapping from language constructs to generated Solidity smart-contract artifacts. The section also clarifies the on-chain runtime architecture and the off-chain modeling and generation toolchain.

5.1. Metamodel and Abstract Syntax

The SocialRbSC language is defined by a metamodel organized into three layers: the social layer, the role layer, and the reasoning layer. Table 1 summarizes the core language constructs, their notation, and constraints.

Table 1. SocialRbSC Core Language Constructs.

Construct	Notation	Description	Constraints
Society	@Society (name = "…")	Top-level container for agent interaction space	Each society must contain at least one compartment
Compartment	@Compartment (name = "…", bindingRole = "…", activateCompartment = true | false)	Context scoping role activation and interaction	Compartments must be nested inside a society; role bindings must reference declared roles
SocialAgent	@SocialAgent (agentName = "…", name = "…", addRole, removeRole) @PlayableRoles("…")	Autonomous entity that enacts roles and reasons about goals	Agent must specify at least one playable role; name is an alias for agentName
Role	@Role (role name is the annotated class name) @RoleCardinality (maxInstances = n) @PlayingConstraint("…")	Behavioral position enacted by an agent within a compartment	Role cardinality must be a positive integer; playing constraints must be decidable at binding time
Belief	@Belief (name = "…", type = "…")	Agent's representation of the current state of the world	Belief types must be expressible as Solidity elementary or mapping types
Desire/Goal	@Desire (name = "…") @Priority(level = 1..3)	Desired state that the agent aims to achieve	Priority levels must be in the range [1, 3]
Intention	@Intention	Committed goal that the agent is actively pursuing	An intention must reference an existing desire
GoalPlanTree	@GoalPlanTree (root = "…") @Plan(name = "…") @Decomposition({…})	Tree-structured decomposition of goals into executable plans	Root goal must reference a declared desire; decomposition must be acyclic
Pre/ Postcondition	@Precondition ("…") @PreconditionVariable @Postcondition("…")	Guard evaluated before a role method executes, and condition asserted after it executes	Precondition expression must be evaluable on-chain without external oracle calls; postcondition must reference a declared event or state predicate
Role method	@RoleMethod (inside a @Role class) @RoleMethod(role = "…") (on a @SocialAgent method)	Role-bound operation: without an argument, @RoleMethod marks a method declared inside a @Role class; with role = "…", it marks a method on a @SocialAgent class that is valid only while the agent plays that role	The no-argument form must appear inside a @Role class; the argument form must appear on a @SocialAgent method and name a role listed in the agent's @PlayableRoles set
AccessControl	@AccessControl (restrictedTo = {"…"})	Declarative permission restricting method access to authorized role holders	Restricted-to set must contain at least one role declared in the enclosing compartment
Transition	@Transition (from = "…", to = "…")	Finite-state-machine transition for role lifecycle management	Both from and to states must be declared in the role's state enumeration
Properties	@Properties ({ @Property(name = "…", type = "…"), …})	Declares compartment-scoped shared state visible to all participating roles	Property types must be Solidity elementary or mapping types
Motivation	@MotivationalModel ({ @Motivation(…)}) @AvoidanceMotive	Motivational parameters (willingness, avoidance, affiliation) that steer plan selection	Motivational values are numeric; avoidance targets must name a declared hazard predicate
Deliberation	@DeliberationProcess (description = "…")	Marks the class that carries the perceive–reason–act cycle	At most one per society; must be attached to the class that declares the goal–plan tree

5.2. Concrete Syntax Grammar

The concrete syntax of SocialRbSC is defined as a set of Java annotations. The following grammar specifies the annotation structure in extended BNF notation and covers every construct used in this paper (Sections 3.4 and Appendix B); the complete grammar for the full annotation set is defined by the Java annotation interfaces in the software repository. The construct column of Table 1 is derived from the same metamodel, and the EBNF below

is generated from the same annotation interfaces, so the three views cannot drift apart (Listing 1).

Listing 1: SocialRbSC Annotation Grammar (EBNF).

```
SocialRbSCProgram    ::= Annotation* ClassDeclaration

Annotation           ::= SocietyAnnotation
                       | CompartmentAnnotation
                       | PropertiesAnnotation
                       | RoleAnnotation
                       | RoleCardinalityAnnotation
                       | PlayingConstraintAnnotation
                       | SocialAgentAnnotation
                       | PlayableRolesAnnotation
                       | BeliefAnnotation
                       | DesireAnnotation
                       | PriorityAnnotation
                       | IntentionAnnotation
                       | GoalPlanTreeAnnotation
                       | PlanAnnotation
                       | DecompositionAnnotation
                       | PreconditionAnnotation
                       | PreconditionVariableAnnotation
                       | PostconditionAnnotation
                       | RoleMethodAnnotation
                       | AccessControlAnnotation
                       | TransitionAnnotation
                       | DeliberationProcessAnnotation
                       | MotivationalModelAnnotation
                       | AvoidanceMotiveAnnotation

SocietyAnnotation    ::= '@Society' '(' 'name' '=' StringLiteral ')'

CompartmentAnnotation ::= '@Compartment' '('
                          'name' '=' StringLiteral
                          ('bindingRole' '=' StringLiteral)?
                          ('activateCompartment' '='
                              BooleanLiteral)?
                          ')'

PropertiesAnnotation ::= '@Properties' '(' '{'
                         Property (',' Property)*
                         '}' ')'
Property             ::= '@Property' '('
                         'name' '=' StringLiteral
                         (',' 'type' '=' StringLiteral)?
                         ')'

RoleAnnotation       ::= '@Role'
RoleCardinalityAnnotation ::= '@RoleCardinality' '('
                              'maxInstances' '=' IntegerLiteral ')'
PlayingConstraintAnnotation ::= '@PlayingConstraint' '('
```

```
                                StringLiteral ')'

SocialAgentAnnotation ::= '@SocialAgent' '('
                          'agentName' '=' StringLiteral
                          (',' ('addRole' | 'removeRole') '='
                              BooleanLiteral)*
                          ')'
PlayableRolesAnnotation ::= '@PlayableRoles' '(' '{'
                            StringLiteral (',' StringLiteral)*
                            '}' ')'

BeliefAnnotation    ::= '@Belief' '('
                        'name' '=' StringLiteral
                        (',' 'type' '=' StringLiteral)?
                        ')'
DesireAnnotation    ::= '@Desire' '(' 'name' '=' StringLiteral ')'
PriorityAnnotation  ::= '@Priority' '(' 'level' '=' ('1' | '2' |
    '3') ')'
IntentionAnnotation ::= '@Intention'

GoalPlanTreeAnnotation ::= '@GoalPlanTree' '('
                           'root' '=' StringLiteral
                           ')'
PlanAnnotation       ::= '@Plan' '(' 'name' '=' StringLiteral ')'
DecompositionAnnotation ::= '@Decomposition' '(' '{'
                            StringLiteral ('->' StringLiteral)?
                            (',' StringLiteral ('->' StringLiteral)
                                ?)*
                            '}' ')'

PreconditionAnnotation ::= '@Precondition' '(' StringLiteral ')'
PreconditionVariableAnnotation ::= '@PreconditionVariable'
PostconditionAnnotation ::= '@Postcondition' '(' StringLiteral ')'

RoleMethodAnnotation   ::= '@RoleMethod'
                           ('(' 'role' '=' StringLiteral ')')?

AccessControlAnnotation ::= '@AccessControl' '('
                            'restrictedTo' '=' '{'
                            StringLiteral (',' StringLiteral)*
                            '}' ')'
TransitionAnnotation ::= '@Transition' '('
                         'from' '=' StringLiteral ','
                         'to' '=' StringLiteral ')'

DeliberationProcessAnnotation ::= '@DeliberationProcess' '('
                                  'description' '=' StringLiteral
                                      ')'
MotivationalModelAnnotation ::= '@MotivationalModel' '(' '{'
                                Motivation (',' Motivation)*
                                '}' ')'
Motivation          ::= '@Motivation' '('
                        'type' '=' StringLiteral ','
                        'value' '=' NumberLiteral ')'
AvoidanceMotiveAnnotation ::= '@AvoidanceMotive'
```

Notation consistency. The canonical agent-name attribute is `agentName`; `name` is accepted as an alias and resolved to `agentName` by the annotation processor. All examples in this paper use the canonical form. `@RoleMethod` is a single annotation with two placements: without an argument it marks a method declared inside a `@Role` class, and with `role = ...` it marks a method on a `@SocialAgent` class that is valid only while that agent plays the named role. The named role must be listed in the agent's `@PlayableRoles` set.

5.3. Well-Formedness Constraints

A SocialRbSC specification is well-formed if and only if it satisfies the following constraints.

C1 (Society–Compartment containment): Every Compartment annotation must be enclosed within a class that carries a Society annotation. Compartments at the top level are not well-formed.

C2 (Role binding validity): A Compartment annotation that specifies a bindingRole must reference a @Role-annotated class declared in the same compilation unit.

C3 (Agent–Role consistency): Every role name in a SocialAgent's @PlayableRoles set must have a corresponding @Role declaration in the enclosing society.

C4 (Goal–Plan acyclicity): The decomposition graph defined by @Decomposition must not contain cycles. Each leaf node in the decomposition must be a primitive action (a method without further @Decomposition annotations).

C5 (Precondition scope): A @Precondition annotation must reference a @PreconditionVariable declared in the same compilation unit, and the precondition expression must be evaluable using only the blockchain state accessible from the generated contract.

C6 (Transition completeness): For every @Transition (from, to) annotation, both states must appear in the role's state enumeration, and the transition must not create unreachable states.

C7 (Cardinality enforcement): A @RoleCardinality (maxInstances = n) annotation constrains the generated contract to reject role bindings that would exceed n simultaneous players.

5.4. Execution Semantics of the Reasoning Layer

The abstract syntax of the reasoning layer is given in Table 1; this subsection fixes its execution semantics so that beliefs, intentions, goal–plan trees, and the deliberation cycle have a precise operational meaning on-chain. All definitions are finite and deterministic, as required by the gas-bound and determinism constraints of Section 5.10.

Belief base: A belief base is a finite partial function $B : \mathit{BeliefName} \rightharpoonup \mathit{Value}$, represented on-chain as a mapping. An update $B[b \mapsto v]$ replaces the value bound to b; expressions over beliefs are restricted to comparisons and arithmetic evaluable from on-chain state. A belief is therefore a *recorded* proposition rather than a logical formula, and the language deliberately does not attempt general belief revision.

Desires and intentions: A desire set $D = \{d_1, \ldots, d_n\}$ is a finite set of goals, each with a priority $p(d) \in \{1, 2, 3\}$ and a decidable activation condition $\mathit{act}(d)$ over B. An intention $i = \langle d, \pi \rangle$ pairs a committed desire d with a plan π selected for it. The intention stack $I = \langle i_1, \ldots, i_k \rangle$ is a finite sequence in which i_1 is the most recent entry (LIFO order). Selection is

$$\mathit{select}(D, B) = d^* \text{ such that } \mathit{act}(d^*) \wedge \neg \exists d \in D : \mathit{act}(d) \wedge p(d) > p(d^*),$$

i.e., the highest-priority active desire, with ties broken by declaration order; if no desire is active, no intention is pushed.

Goal–plan trees: A goal–plan tree is a finite, ordered, rooted tree $T = (V, E, \ell)$ in which every node is either an internal goal node with child list $E(v)$ or a leaf action node with label $\ell(v)$. The decomposition relation is acyclic (C4). A plan for d is a path from

a node labelled d to a leaf; the set of plans for d is $\Pi(d)$. Plan selection is a bounded depth-first search

$$solve(T, d, c) = \text{the first } \pi \in \Pi(d) \text{ whose guard holds, visiting at most } c \text{ nodes,}$$

where c is a static bound emitted by the generator. The bound guarantees that *solve* terminates and that its gas cost is $O(c)$, independent of the total tree size; the bound is the mechanism behind the bounded goal–plan-tree search listed among the on-chain components in Section 5.8.

Deliberation cycle: The deliberation cycle is the least fixed point of four relations over the agent state (B, I, s), where s is the role/agent state machine state:

$$perceive : (B, I, s) \to (B', I, s), \qquad reason : (B', I, s) \to (B', I', s),$$
$$act : (B', I', s) \to (B'', I', s'), \qquad done : (B'', I', s') \to (B'', I'', s'),$$

where *perceive* folds committed environment inputs and emitted events into B (Section 5.5), *reason* runs *select* and *solve* to push an intention, *act* executes the role method selected by the head intention subject to the binding, access-control, transition, and precondition guards of the mapping, and *done* pops the head intention once its postcondition holds. A single iteration is the composition $(B, I, s) \longrightarrow (B'', I'', s')$ whenever an intermediate (B', I') makes the four relations hold in sequence.

Properties: The relation terminates on well-formed specifications because V, D, and I are finite, the search bound c is finite, and every rule reads only on-chain state; it is therefore deterministic. It does not guarantee goal satisfaction or plan optimality; it guarantees only that any selected action is a leaf of a well-formed plan and is enforced by the same P1–P4 guards as any other role method. This is the precise sense in which the reasoning layer is specified operationally rather than as a refinement target in Section 5.7.

5.5. Environment Interface, External Inputs, and Belief Update

Because reasoning executes on-chain, an agent cannot observe the physical world directly. The language therefore fixes a narrow, verifiable interface through which external information enters the belief base, and nothing else may influence deliberation. Three channels are distinguished.

(1) On-chain state and events: The primary input is the ledger itself: shared compartment state (the @Properties of the compartment) and events emitted by other contracts. Reading these carries no trust assumption, because it is part of the state every node verifies. Re-deliberation can be triggered by observing an event such as `RoleBound` or `DeliveryStatusUpdated`; an off-chain watcher simply translates the observed event into a transaction that drives the next cycle.

(2) Signed off-chain assertions: Information that originates outside the blockchain (sensor readings, delivery confirmations, prices, certificates) enters as a transaction carrying a signed assertion. The generated contract recovers the signer from the assertion with ECDSA (`ecrecover`) [33,34] and accepts the update only if the recovered address is an authorized source for that belief; the assertion, the signer, and the block timestamp are then stored, so the value is attributable and auditable. Source identity is thus cryptographically verified without trusting a central oracle for correctness: an oracle can only assert what its key signs, and a false or withheld assertion is attributable to that key.

(3) Committed inputs: For large or private payloads, the contract stores only a commitment $commit = H(payload)$ together with the signer, and the payload is revealed later (commit–reveal) or retained off-chain for audit. This keeps gas bounded while preserving verifiability.

Trust model: The reasoning machinery is not trusted with unverifiable inputs: *perceive* accepts only state reads, recovered-signature assertions, and hash commitments. Consequently an off-chain party cannot cause an action that its signature does not authorize, and a dishonest sensor can at worst produce a signed, attributable false belief. In the reference artifacts, the helper `verifyValidatorSignature` is a placeholder that must be instantiated with `ecrecover`; the interface (signed payload, authorized-source check, event) is fixed by the language, while the concrete scheme (single key, threshold or multi-signature, or a TLS-attested oracle) is a deployment choice. Section 8 explains why this verifiable interface is preferable to trusting an off-chain reasoner.

5.6. Mapping from Language Constructs to Solidity Artifacts

The SocialRbSC code generator produces one Solidity smart contract per Society, Compartment, and Role construct, plus supporting library contracts. Table 2 defines the mapping.

Table 2. Mapping from SocialRbSC Constructs to Generated Solidity Artifacts.

SocialRbSC Construct	Generated Solidity Artifact	Description
@Society	Mediator contract (e.g., ConcreteMediator)	Registry contract that stores agent-to-society bindings; exposes bindAgent, unbindAgent, getAgentFromSociety
@Compartment	Compartment contract (e.g., MedicalSupplyMission)	Core contract managing role bindings, access-control modifiers, state variables, and role-specific functions
@Role	Role behavior contract (e.g., DeliveryProviderRole)	Contract encapsulating role-specific state, FSM status, and behavior functions; references its compartment via an owner address
@RoleMethod	Public/external function in the role or agent contract	Without an argument, a role-contract function with onlyAgent, onlyCompartment, and onlyInRole modifiers; with role = "…", a role-guarded agent function
@Precondition	require statement at function entry	Translated to a Solidity require() evaluating the precondition expression
@AccessControl	onlyInRole modifier	Generated as a modifier that checks agentRoles[msg.sender][role].isActive
@Transition	Enum-based state machine with require guards	Generated as an enum for role states, with require checks ensuring valid from/to transitions
@SocialAgent	EOA or contract address stored in mediator	The agent is represented by its blockchain address; reasoning state (beliefs, intentions) is stored on-chain in the social-agent engine contracts
@GoalPlanTree	GoalPlanTree and LinkedList contracts	Goal–plan trees are evaluated on-chain; the selected action is enforced by the role contract

5.7. Correctness Properties of the Mapping

The mapping in Table 2 is not only a syntactic translation; we state the properties that the generator preserves. Let G be the code generator and let $C = G(S)$ be the Solidity artifacts emitted for a well-formed specification S (one satisfying C1–C7). We consider four correctness properties.

P1 (Binding soundness): A call that binds a player p to a role r succeeds only when the caller is authorized to manage bindings, p does not already hold an active binding for r, and the binding is recorded; otherwise the transaction reverts. On success, the binding is stored before the event is emitted, so the number of recorded players for r is exactly the number of successful binding operations.

P2 (Access-control completeness): For every method annotated with `@AccessControl`, a call to the generated function reverts unless the caller holds an active role in the restricted set R (enforced by the generated `onlyInRole` modifier).

P3 (Finite-state-machine safety): For every `@Transition(from =` s_1`, to =` s_2`)`, the generated function is executable only when the role's current state is s_1, and after a successful call the state is exactly s_2; no transition outside the declared enumeration is reachable.

P4 (Precondition enforcement): For every method annotated with a precondition φ, the generated function reverts whenever φ does not hold before the method body executes, so the state change is not observable unless φ held.

Proposition: If S is well-formed (C1–C7) and $C = G(S)$, then C satisfies P1–P4.

Proof sketch. By structural induction over the generator rules of Table 2. The rule for a binding operation emits an authorization guard, a duplicate-binding guard, and a state write that precedes the event, which yields P1; the well-formedness constraints C2 and C3 guarantee that the referenced roles and agents exist, so the guards are well-defined. The rule for access control emits an `onlyInRole`-style modifier from the restricted-to set, and C4 ensures that this set is non-empty and declared, which yields P2. The transition rule emits an enumeration together with a guard requiring the current state to equal `from` before writing `to`, which yields P3; C6 guarantees that both states are declared. The precondition rule emits a `require` at function entry, and C5 guarantees that the expression is evaluable from on-chain state, which yields P4. Each rule only strengthens the guard of the operation it emits, so the induction is preserved under composition.

Proposition deliberately covers the enforceable, state-based part of the language. The deliberative part (goal–plan trees and the deliberation cycle) is specified operationally rather than as a refinement target: we do not claim goal satisfaction or plan optimality of deliberation, only that the action it selects is enforced by the same binding, access-control, transition, and precondition guards. The four properties are exercised by the accompanying test suite and complemented by the static analysis reported in Section 5.11.

5.8. On-Chain Runtime Architecture and Off-Chain Toolchain

A central design decision in SocialRbSC is that every runtime component, including role binding, access control, and the BDI reasoning machinery, executes on-chain as Solidity contracts, while the language toolchain (the Java annotation DSL, the code generator, and the web3j-based interaction layer) runs off-chain at development and deployment time. Table 3 summarizes this split.

Table 3. On-chain vs. Off-chain Components of the SocialRbSC Architecture.

Component	Location	Rationale
Role binding and activation	On-chain (compartment contract)	Binding must be verifiable by all participants; state changes require consensus
Access control (modifier checks)	On-chain (compartment contract)	Permission enforcement must be deterministic and non-bypassable
Precondition evaluation	On-chain (require statements)	Preconditions guard state-changing operations and must be atomically checked
Finite-state machine transitions	On-chain (role contract)	Role lifecycle must be transparent and auditable through event logs
Belief storage and update	On-chain (SocialAgentBelief contract)	Beliefs are stored in an on-chain mapping so that deliberation is deterministic and auditable
Goal–plan tree reasoning	On-chain (GoalPlanTree contract)	Plans are stored in an on-chain tree structure (LinkedList-backed); search is bounded to keep gas predictable

Table 3. *Cont.*

Component	Location	Rationale
Intention selection	On-chain (SocialAgentIntention contract)	The intention stack is maintained on-chain so selection is verifiable and tamper-resistant
Deliberation cycle execution	On-chain (SocialAgentDeliberationCycle contract)	The full perceive–reason–act loop runs on-chain over the agent state machine
Event emission for coordination	On-chain (contract events)	Events provide a tamper-resistant notification mechanism that clients can observe
Cryptographic identity verification	On-chain (signature verification)	Verifying agent identities through ECDSA [33,34] signatures is cheap and reliable on-chain
Modeling and annotation DSL	Off-chain (Java, compile time)	Annotations describe the model; the DSL itself executes no code on-chain
Code generation	Off-chain (compile time)	The annotation processors and Velocity templates emit Solidity source files
Java-to-Solidity interaction	Off-chain (web3j runtime)	A web3j-based harness deploys and invokes the generated contracts from Java

In this architecture, the entire runtime model (society, compartments, roles, beliefs, intentions, goal–plan trees, and the deliberation cycle) is implemented as Solidity contracts and executes in the EVM. The off-chain Java toolchain is used only to model the system with annotations, generate the Solidity source code, and drive deployment and interaction through a web3j harness. Because reasoning is on-chain, deliberation consumes gas; this is the cost quantified in the evaluation of Section 7 and is the primary trade-off of making coordination deterministic and auditable.

5.9. Code Generation Pipeline

The SocialRbSC code-generation pipeline consists of three stages.

Stage 1 (Annotation Processing): The SocialRbSC annotation processor scans the annotated Java source files and extracts the language model into an abstract syntax tree (AST) conforming to the metamodel in Table 1. During this stage, the processor validates well-formedness constraints C1–C7 and reports errors for any violations.

Stage 2 (Intermediate Representation, IR): The validated AST is lowered into an intermediate representation that separates structural concerns (role bindings, access control, state transitions) from reasoning concerns (beliefs, goals, plans), all of which are ultimately emitted as on-chain contracts. The IR is a platform-independent representation that can target multiple smart-contract platforms.

Stage 3 (Solidity Code Generation): The IR is traversed to emit Solidity source files. For each society, the generator produces a mediator contract; for each compartment, a compartment contract with RBAC modifiers; for each role, a role behavior contract with FSM state management; and for each precondition, a Solidity require statement. Reasoning components (belief management, intention selection, goal–plan-tree reasoning, and the deliberation cycle) are emitted as Solidity contracts that import the framework's social-agent engine. A Java wrapper (web3j) is generated alongside the Solidity to drive deployment and interaction from the modeling language.

5.10. Constraints on Generated Code

The generated Solidity contracts inherit the following constraints from the SocialRbSC language model.

- **Gas bound:** Every precondition expression must be evaluable within the gas limit of a single transaction. Recursive precondition evaluation and unbounded loops are not permitted.
- **Determinism:** All on-chain functions must be deterministic. Randomness or oracle-dependent values must not appear in any generated contract.
- **State visibility:** Compartment, role, and agent state that must be auditable is declared as public or internal; the social-agent engine stores beliefs and intentions in mappings exposed through getters.
- **Event completeness:** Every state-changing function must emit at least one event so that off-chain clients and the web3j harness can observe state transitions.

5.11. Security Properties and Static Analysis of Generated Contracts

Smart contracts that encode role bindings and behavioral rules are security-critical, so the generated artifacts are designed against established Solidity security patterns and then analyzed with a static analyzer.

Established security patterns: Table 4 maps the patterns relevant to this paper, namely access control on role-binding transitions, checks–effects–interactions, reentrancy, arithmetic safety, input validation, and event completeness, to their concrete realization in the generated contracts.

Table 4. Established Solidity Security Patterns and Their Realization in the Generated Contracts.

Security Pattern	Realization in Generated Contracts
Access control on role-binding transitions	Binding and unbinding are privileged operations: the generated compartment contract (`MedicalSupplyMission`) guards `bindAgentToRole` and `unbindAgentFromRole` with an `onlySociety` modifier, and the mediator contract (`ConcreteMediator`) guards `bindAgent` and `unbindAgent` with the same modifier. The first role player cannot be bound by an arbitrary caller, and duplicate bindings are additionally rejected by `roleNotInUse`.
Access control on role-restricted operations	`onlyInRole(role)` checks `agentRoles[msg.sender][role].isActive`; role-behavior contracts define an `onlyAgent` modifier (only the bound player) and an `onlyCompartment` modifier (only the owning compartment).
Checks–effects–interactions (CEI)	Every state-changing function writes state before emitting its event, and no outbound call occurs between a check and its effect. In particular, the binding and unbinding functions contain no external calls at all.
Reentrancy	The reference artifacts make no external calls and transfer no value, so no reentrancy surface is present; checks–effects–interactions is the primary and, here, sufficient defense. The generator is designed to emit a `nonReentrant` guard whenever a role method is bound to an external call (none of the reference artifacts perform external calls).
Arithmetic safety	Contracts target Solidity ≥ 0.8, so overflow and underflow revert automatically in checked arithmetic.
Input validation	Zero-address checks on agent registration; finite-state-machine guards reject transitions that are not declared for the current state.
Event completeness	Each state-changing function emits an event (`RoleBound`, `RoleUnbound`, `DeliveryStatusUpdated`, `DeliveryConfirmed`, ...), producing a tamper-resistant audit trail.

Static analysis: The seven artifacts (the three generated contracts, the three hand-written baselines, and the goal–plan-tree benchmark) were analyzed with the Slither static-analysis framework [35] (v0.11.6) using solc 0.8.13. Table 5 summarizes the findings.

After triage, the 18 remaining findings are all of Low severity or below (Informational and Optimization hints); none is High or Medium. Slither reports *no* reentrancy, unchecked-call or unchecked-transfer, arbitrary-send, or access-control findings on the generated contracts. The analysis is reproducible with the command recorded in the supplementary repository, and the same command can be applied to the full framework;

symbolic-execution tools such as Mythril are complementary and are planned as future work for the value-transfer-bearing extensions of the language.

Table 5. Slither Static-Analysis Findings for the Generated and Baseline Artifacts.

Detector	Impact	Count	Assessment
solc-version	Informational	4	Known advisories for the pinned 0.8.13/`^0.8.0` range; the affected features are not used
naming-convention	Informational	4	Style only (constructor and function parameters)
constable-states	Optimization	4	Variables could be marked `constant`; no security effect
immutable-states	Optimization	4	Deploy-time addresses could be `immutable`; no security effect
missing-zero-check	Low	2	Constructor addresses; the deployment harness always passes non-zero addresses
uninitialized-state	High (triaged)	1	False positive: an element-level write to the `supplyInventory` mapping is not counted as initialization; the mapping is populated through `registerSupply` and the finding is suppressed with a justified inline directive

6. Case Studies with SocialRbSC Programming Language

The complete SocialRbSC code example for Listing A1 is provided in Appendix A. The listing is best read as a social-agent/smart-contract scenario that proceeds from high-level coordination to local execution. The @Society annotation defines the overall interaction space, in this case a marketplace in which negotiation takes place. The adjacent @DeliberationProcess annotation indicates that coordination is not limited to method invocation; instead, the agent's decisions pass through an explicit reasoning step.

The structural layer then specifies how participation is organized. Two @Compartment declarations define the contexts in which interactions occur and bind the buyer and seller roles before any exchange takes place. Here the role model becomes concrete: @SocialAgent identifies the participating agents, and @Role specifies the behavioural position each agent occupies in the negotiation. The annotations thus define a hierarchy in which societies contain compartments, compartments activate role contexts, and agents join those contexts by adopting roles.

The reasoning layer follows. @Belief captures what the agent currently takes to be true about the situation, whereas @Intention captures the state the agent aims to change, in this case the current price. These annotations are significant because they demonstrate how SocialRbSC supports proactive coordination: agents do not merely occupy roles, but also maintain internal reasoning states that guide their decisions.

Execution control is addressed last. A shared variable marked @PreconditionVariable represents state that must be checked before an action can execute. The @Precondition annotation then acts as a guard, verifying that the transfer amount is valid. Once the guard is satisfied, @RoleMethod identifies the operation that executes, updating the negotiated amount within the seller's role context.

In summary, the example traverses the annotation language in the same order in which the program runs: it first declares the social context, then organizes participants and roles, then establishes the reasoning state, and finally executes role-specific behaviour under explicit conditions.

Interpretation of Appendix A Medical Supply Chain Code Example

Listing A2 in Appendix A extends the preceding marketplace example to a healthcare delivery scenario, illustrating how SocialRbSC scales from a small negotiation model to a realistic multi-role coordination problem.

At the top level, the class *MedicalSupplyDeliverySystem* acts as a container for the domain model. The annotation @Society (name = "HealthcareNetwork") defines the social environment in which hospitals, delivery providers, validators, and coordinators interact. Within that society, @Compartment(name = "MedicalSupplyMission") creates an operational mission context, so that all delivery-specific rules, state, and permissions reside inside a concrete compartment rather than in a global scope.

A @Properties block attached to the mission compartment defines the shared mission state that is visible to every participating role. The listing defines three properties: deliveryDeadline (the deadline for the delivery), requiredSupplies (the required materials), and hospitalLocation (the destination). The compartment is therefore not only a structural wrapper but also the data context that drives the use case.

The role classes specify each participant's responsibilities. The @Role annotation marks *DeliveryProviderRole* as the role responsible for transporting supplies, and @RoleCardinality(maxInstances = 1) ensures that at most one active provider occupies the role per mission. The @PlayingConstraint annotation adds a semantic rule: the same object cannot play the same role in multiple compartments at the same time. Within this role, several annotations interact:

@Precondition defines what must hold before a delivery can be accepted.

@Postcondition defines what the intended state looks like after acceptance.

@AccessControl restricts status updates to authorized role holders.

@Transition specifies the legal state change from *INTRANSIT* to *DELIVERED*.

Taken as a whole, these annotations define both who has authority and how the delivery role evolves over its lifecycle. The validator and coordinator roles illustrate two further coordination patterns. *ValidatorRole* uses @AccessControl(restrictedTo = "Validator") so that only an authorized validator can confirm a delivery, and its @Postcondition states that the validation intention is satisfied once receipt confirmation succeeds. *OrderCoordinatorRole*, by contrast, matches supply requests to available delivery providers: its @Belief annotation captures what the coordinator knows about available inventory, and its @Postcondition indicates that a matching event should be emitted when the assignment logic succeeds. In short, the validator certifies correctness, while the coordinator organizes allocation.

The two social agents in the listing show how roles connect to autonomous behaviour. @SocialAgent (agentName = "UAVDeliveryAgent") defines a delivery-capable agent that can enact several roles, listed in @PlayableRoles. Its @Belief fields describe what the agent currently knows, such as battery level, position, and route availability. The @Desire annotations state what the agent aims to achieve, and @Priority orders those goals so that the framework can reason about trade-offs. The @MotivationalModel and @AvoidanceMotive annotations add motivational semantics: the agent should not only deliver efficiently but also avoid hazardous zones. Finally, @GoalPlanTree, @Plan, and @Decomposition show how the main goal decomposes into smaller executable steps within the deliberation cycle.

DoctorAgent represents the requester side of the workflow. Through @PlayableRoles, the doctor agent can act as requester, validator, or collaborator depending on the situation. Its @Belief field stores the current supply status, and the method requestSupplies(...) is marked with @RoleMethod (role = "SupplyRequester"), meaning that the operation is valid only when the agent acts in that role. The accompanying @Postcondition states that a request event must be emitted after execution.

Taken as a whole, Appendix B shows that SocialRbSC definitions are not merely documentation annotations; they specify social structure, execution constraints, reasoning state, and blockchain-related behaviour within a single unified language model.

7. Results

7.1. Benchmark Design and Execution Environment

The evaluation comprises five complementary experiments. (i) A module-level benchmark measures the deployment and execution cost of the five main SocialRbSC modules: *CompartmentInitiator*, *ComponentCore*, *DeepRolePlayer*, *GoalPlanTree*, and *SocialAgentDeliberationCycle*. (ii) A baseline comparison contrasts the SocialRbSC role contracts against minimal hand-written Solidity that implements the same operations without any framework support. (iii) An end-to-end multi-agent experiment executes a complete medical-supply coordination cycle, and (iv) a scalability experiment measures how binding cost evolves as the number of role players grows. (v) A multi-agent, multi-role coordination experiment and a goal–plan-tree scaling experiment assess larger configurations. All contracts were compiled with Solidity 0.8.13 (optimizer enabled, 200 runs) and executed on a local Hardhat (v2.29.1) Ethereum Virtual Machine. Because gas consumption is deterministic for a fixed compiler version and bytecode, repeated executions of the same input sequence produce identical values; the reported figures are therefore exact measurements rather than statistical averages, and the only residual variation (a few gas) comes from address-dependent calldata, not from run-to-run noise. The measurement scripts (`scripts/measure-gas.js` and `scripts/measure-evaluation.js`) are available in the supplementary repository to support independent reproduction.

Table 6 reports the absolute gas values (in gas units) and the percentage share for each module.

Table 6. Absolute Gas Consumption by SocialRbSC Module.

Module	Deployment	Execution	Total	Share
CompartmentInitiator	564,047	22,570	586,617	6.40%
ComponentCore	580,896	23,471	604,367	6.60%
DeepRolePlayer	463,143	165,433	628,576	6.86%
GoalPlanTree	3,601,662	1,426,520	5,028,182	54.88%
SocialAgentDeliberationCycle	1,837,955	475,952	2,313,907	25.26%
Total	7,047,703	2,113,946	9,161,649	100%

Figure 3 summarizes the relative gas usage visually.

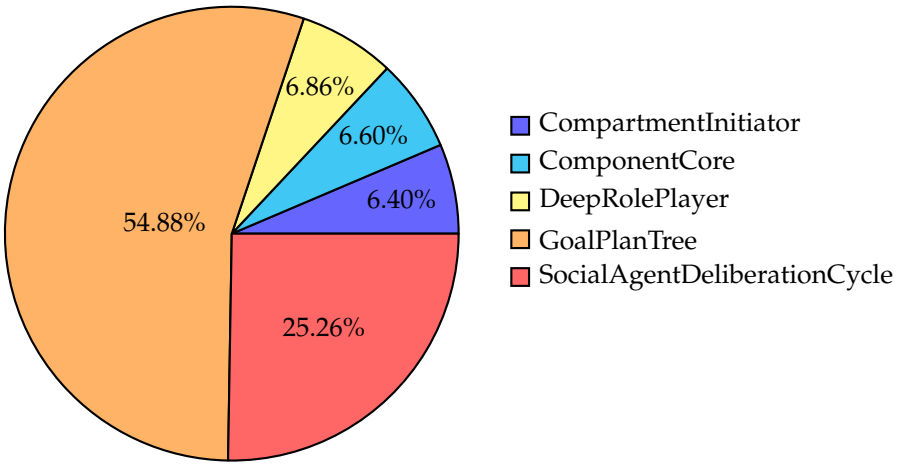


Figure 3. Contract Gas Usage as Percentage of Modules in SocialRbSC Framework.

7.2. Analysis of Gas Cost Distribution

The most direct result from this distribution is that the framework spends most of its on-chain cost on reasoning-oriented behavior rather than on role binding infrastructure. Taken

together, *GoalPlanTree* and *SocialAgentDeliberationCycle* consume 7,342,089 gas (80.1% of total), which indicates that planning and deliberation dominate execution cost in the current implementation. By contrast, *DeepRolePlayer* contributes 628,576 gas (6.9%), suggesting that role-player management is comparatively lightweight within this benchmark.

This result is relevant for both research questions. First, it shows that the role-oriented constructs of SocialRbSC, such as compartments, component management, and role-player binding, can be implemented on-chain without becoming the dominant source of gas consumption in the reported benchmark. In that sense, the measurements provide evidence for Hypothesis 1: explicit role characteristics that can be represented as stateful and verifiable smart-contract operations are practical building blocks for the language.

Second, the distribution shows where social and proactive coordination is actually realized in the framework. The dominant shares of *GoalPlanTree* and *SocialAgentDeliberationCycle* indicate that coordination depends mainly on goal reasoning and deliberation logic rather than on the structural role layer alone. This provides evidence for Hypothesis 2 and suggests that SocialRbSC operationalizes agent coordination, albeit at a measurable computational price.

7.3. Baseline Comparison

To quantify the overhead introduced by the role abstractions, the SocialRbSC role contracts were compared against minimal hand-written Solidity that implements the same operations without any framework support. The baseline for role binding uses a single nested mapping together with a role-member array; the baseline for the role lifecycle uses a bare enumeration with `require` guards and no role checks or motivational guards. Table 7 reports the results.

Table 7. Baseline Comparison: Hand-Written Solidity vs. SocialRbSC Role Contracts.

Operation	Hand-Written	SocialRbSC	Overhead
Role binding (one agent)	89,998	161,807	+79.8%
Role lifecycle (three transitions)	100,058	89,205	−10.8%
Role registry/compartment deployment	161,345	1,266,714	+685.1%
Role-behaviour deployment	182,141	501,965	+175.6%

The richer role binding, which stores a complete `RoleBinding` record, maintains a role-member list, and enforces a binding constraint, costs approximately 80% more gas than a minimal binding (161,807 vs. 89,998 gas). The role lifecycle, by contrast, is slightly cheaper in the SocialRbSC contract than in the hand-written baseline (89,205 vs. 100,058 gas for three transitions, i.e., −10.8 %), because the generated role contract writes a smaller payload per transition while still enforcing role, precondition, and motivational checks. Deployment shows the cost of the framework's structural scaffolding: the compartment, which subsumes the baseline role registry, costs 1,266,714 gas versus 161,345 gas (+685.1%), and the role-behaviour contract costs 501,965 gas versus 182,141 gas for the minimal FSM (+175.6%). Because the hand-written baseline has no compartment abstraction, its role registry is compared with the SocialRbSC compartment that subsumes it. The abstraction therefore adds cost mainly to the binding operation and deployment rather than to the execution of role behaviour; the additional binding cost includes the access-control check that restricts binding to the owning society, which the minimal hand-written baseline does not perform.

7.4. End-to-End Multi-Agent Evaluation

To complement the isolated module measurements, a complete medical-supply coordination cycle was executed on a single compartment with two social agents (a delivery

UAV and a validator) and one society mediator. The cycle comprises contract deployment, role binding for both agents, mediator registration, a two-step delivery-status progression, and validator confirmation. Table 8 reports the gas of each phase.

Table 8. End-to-End Medical-Supply Coordination Cycle.

Phase	Gas
Deployment (compartment + mediator + role)	2,112,911
Bind *DeliveryProvider* role	161,807
Bind *Validator* role	161,711
Mediator agent registration	47,437
Status transition to *IN_TRANSIT*	49,768
Status transition to *DELIVERED*	32,705
Validator confirmation	32,255
Operations subtotal	485,683
End-to-end total	2,598,594

The single-cycle cost is dominated by one-time deployment; the recurring coordination operations total approximately 0.49 M gas for a two-agent mission. A complete coordination cycle is therefore executable within the gas limit of a single Ethereum block and well below typical block gas limits.

7.5. Scalability Analysis

To assess how the framework behaves as the number of participants grows, the delivery-provider role was bound for $N \in \{1, 5, 10, 25, 50\}$ distinct agents on a compartment. Table 9 reports the total binding cost and the average per-binding cost.

Table 9. Scaling of Role Binding with the Number of Role Players.

Role Players (N)	Total Binding Gas	Gas per Binding
1	161,807	161,807
5	740,623	148,125
10	1,464,170	146,417
25	3,634,763	145,391
50	7,252,414	145,048

The average per-binding cost stabilizes at approximately 145,000 gas and is essentially independent of N, so the total cost grows linearly with the number of participants. Role binding therefore scales as $O(1)$ per operation and $O(N)$ in aggregate, indicating that the role layer does not exhibit the super-linear growth that would make large multi-agent deployments impractical.

7.6. Multi-Agent, Multi-Role, and Goal–Plan-Tree Scaling

The isolated and single-cycle measurements are complemented by two scaling experiments that exercise multiple agents and roles and growing goal–plan trees.

Multi-agent, multi-role coordination: A compartment was populated with A delivery providers and A validators, i.e., $2A$ agents across two distinct roles, and every provider then ran a full coordination cycle (two status transitions plus validator confirmation). Table 10 reports the results.

Table 10. Multi-Agent, Multi-Role Coordination (A providers and A validators).

A	Agents	Bindings	Cycles	Total	Per Agent
1	2	323,530	114,728	1,704,972	438,258
5	10	1,480,838	573,604	3,321,156	410,888
10	20	2,927,488	1,147,244	5,341,446	407,473

The per-agent cost of a full round converges to approximately 407,000 gas, so coordination cost grows linearly in the number of participating agents and roles; the only larger term is the one-time compartment deployment (1,266,714 gas), which is amortised over the mission.

Goal–plan-tree scaling: The full framework's GoalPlanTree contract is not part of the extracted benchmark artifacts, so plan storage and bounded search were measured with a reduced on-chain goal–plan-tree micro-benchmark (`GoalPlanTreeBenchmark.sol`) that mirrors the framework's child-linked-list structure and bounded depth-first search. Table 11 reports, for tree sizes $T \in \{10, 50, 100, 500, 1000\}$, the cumulative insertion cost, the insertion cost per node, and the cost of a full bounded traversal.

Table 11. Goal–Plan-Tree Scaling: Storage and Bounded Search.

Nodes (T)	Insert Total	Insert/Node	Traverse	Traverse/Node
10	1,124,909	112,491	95,494	9549
50	5,577,777	111,556	286,000	5720
100	11,144,027	111,440	524,141	5241
500	55,687,215	111,374	2,429,621	4859
1000	111,367,715	111,368	4,812,349	4812

Insertion cost per node is essentially constant (≈111,400 gas), and traversal cost per node decreases to ≈4800 gas as fixed overhead is amortised, so both storage and bounded search grow linearly, i.e., $O(T)$. A full traversal of a 1000-node tree costs about 4.8 M gas, comfortably within a single block, and because search is capped at the static bound c of Section 5.4, deliberation cost is bounded independently of the total tree size.

7.7. Evidence for the Working Hypotheses

To make the relationship between claims and evidence explicit, Table 12 maps each working hypothesis to the experiments that test it, the observable outcome, and the limits of that evidence.

Table 12. Hypothesis-to-Evidence Mapping.

Hypothesis	Experiments	Observable Outcome	Strength and Limits
H1 (RQ1): explicit, verifiable role characteristics are the ones best supported by smart contracts	Module benchmark (Table 6), baseline (Table 7), binding scalability (Table 9), multi-agent/multi-role (Table 10)	The role/binding layer is a minority of the module total; binding is $O(1)$ per operation up to 50 players and 20 agents; the role lifecycle is cheaper than the hand-written baseline	*Supports* H1 for the measured, state-based role characteristics; does not establish suitability for role features not modelled here (e.g., temporal or ownership roles)
H2 (RQ2): explicit reasoning structures coordinate smart contracts effectively, at a measurable cost	Module benchmark (Table 6), end-to-end cycle (Table 8), goal–plan-tree scaling (Table 11)	A full perceive–reason–act cycle executes within a block; reasoning dominates gas (80.1%); plan storage and bounded search are linear and bounded	*Supports* H2 in the sense of technical feasibility and cost transparency; does not establish that on-chain reasoning outperforms off-chain reasoning on decision quality or latency

The table states the scope precisely. H1 is supported for the explicit, state-based role characteristics that were measured; H2 is supported for the feasibility of explicit on-chain coordination and for the claim that its cost is transparent and controllable. Neither hypothesis is established as a general superiority claim: the experiments are deterministic single-host measurements rather than a population study, no statistical-significance claim is made, and the remaining threats are listed in Section 7.8.

7.8. Threats to Validity

The following threats to validity should be considered when interpreting the gas measurements. **Internal validity:** The measurements were obtained on a local EVM instance rather than a public testnet, so network latency, congestion, and gas-price variation are not captured. **External validity:** Although the evaluation includes a baseline comparison, an end-to-end lifecycle, and a scalability study up to 50 role players, the scenarios remain limited in diversity and were executed with a single compiler configuration; results may differ for other coordination patterns, larger agent populations, and public networks. **Construct validity:** Gas consumption is a proxy for computational cost but does not capture developer effort, code maintainability, or security properties. All benchmark scripts, raw measurements, and analysis code are available in the supplementary repository to support independent reproduction and extension.

Table 13 makes the validation results explicit by aligning each functional requirement of the medical supply scenario with the SocialRbSC mechanism that fulfills it and with the corresponding concrete case-study requirements. The mapping shows that the framework does not address the case study only at an abstract modeling level; rather, role assignment, deliberative autonomy, compartment-based collaboration, verifiable identity, decentralized authorization, and motivational behavior all have direct counterparts in the implemented scenario. In this sense, the table provides compact evidence that the proposed language features are not isolated technical constructs, but operational elements that collectively support the end-to-end coordination demands of the case study.

Table 13. Case Study Validation: Functional Requirements vs. SocialRbSC Features.

Requirement	SocialRbSC	Medical Supply Case Study
Multi-Agent Role Assignment	Role Object Pattern, Player contracts, mediator binding	Multiple UAVs dynamically acquire Delivery Provider role
Social Agent Autonomy	Goal–Plan Tree, on-chain BDI deliberation	UAV re-plans delivery route based on weather updates
Compartmentalized Collaboration	Compartment contracts, CompartmentInitiator, nested compartments	Each hospital maintains independent supply mission compartment
Trustworthy Role Identity	Smart contract deployment, cryptographic verification, audit trail	Role assignments verified via Etherscan, complete transition history
Decentralized Authorization	Role-based access control, Finite State Machine (FSM) state enforcement, atomic transitions	Only Validator role can confirm delivery; no central authority required
Motivational Parameters	BDI motivational factors (willingness, avoidance, affiliation)	UAV exhibits avoidance of hazard zones; hospitals show affiliation to preferred suppliers

The architectural implication is therefore narrow but important: optimizing planning and deliberation logic is likely to yield larger gas savings than optimizing the role abstraction layer alone. At the same time, the chart by itself does not prove optimality; it identifies where the current implementation spends gas and which parts of the language architecture deserve further empirical evaluation.

7.9. Reproducibility and Replication Package

The complete toolchain, generated artifacts, benchmark contracts, test suite, and scripts are available in a public repository, and a tagged release provides a stable version for citation. Table 14 lists the pinned dependency versions actually used for the measurements in this paper, and the benchmark script reproduces every gas value in Tables 7–11 as well as the static-analysis result of Section 5.11.

Table 14. Pinned Dependencies and Versions for Reproduction.

Component	Version Used	Role in Reproduction
Node.js/pm	24.12/11.6	JavaScript runtime for the Hardhat harness
Hardhat	2.29.1 (`package.json` pins `^2.17.0`)	In-process EVM, compilation, deployment
ethers.js	5.8.0 (`^5.7.2`)	Contract interaction in the measurement scripts
solc	0.8.13, optimizer enabled, 200 runs	Compiler for all contracts and artifacts
Slither	0.11.6	Static analysis of the generated, baseline, and benchmark artifacts
JDK/Maven	17/3.9	Java annotation DSL, processors, and code generator (JDK 11 also supported)
web3j/Ganache	as declared by the generator/7.9.0	Java–Solidity bridge and integration tests

The exact command sequence that reproduces these results is listed in Appendix C.

Expected outputs: Running `scripts/measure-evaluation.js` prints five labelled sections (A–E) whose values match the tables in this paper; the Slither run reports the findings listed in Table 5. The Java toolchain additionally provides unit tests (`mvn test`) and Ganache-based integration tests (`mvn verify`); the repository records the last successful build and test output alongside the scripts so that a reader can compare results without re-running the entire Java pipeline.

8. Discussion

The discussion is organized with the two research questions introduced in the problem statement and around the working hypotheses in Hypotheses 1 and 2.

RQ1. Which role characteristics are the most suitable to realize with smart contracts? The use-case model, together with the SocialRbSC annotations, suggests that the most appropriate characteristics are those that can be clearly defined based on on-chain state and confirmed deterministically during runtime. Specifically, role activation within a compartment, role binding to players, role-specific functions, and declarative preconditions are all compatible with the smart-contract paradigm because they can be stated as transparent state updates and enforceable restrictions. The gas cost distribution, which shows that these fundamental role abstractions can be implemented without appreciably raising execution overhead, also supports this viewpoint. The reasoning layer accounts for a greater proportion of the overall cost than the role infrastructure.

RQ2. How can agents with social and proactive skills coordinate smart contracts? The proposed technique involves modeling agent coordination using explicit reasoning mechanisms rather than relying on implicit assumptions in contract calls. To do this, SocialRbSC incorporates beliefs, intents, goal–plan trees, and deliberation cycles into role-based interactions. The class and deployment diagrams show how social agents, roles, and compartments are structurally linked, whereas gas consumption data shows that deliberation and planning are the key operational costs. As a result, proactive coordination is possible, but it involves assigning the majority of the computing resources to reasoning-intensive components.

Why On-Chain Reasoning?

The most distinctive design decision of SocialRbSC is that beliefs, intentions, goal–plan trees, and the deliberation cycle are implemented as Solidity contracts rather than as off-chain components. That decision is also the most debatable one from a practical standpoint, so it deserves an explicit justification rather than being presented as a fait accompli.

The choice is motivated by the property that makes on-chain coordination worthwhile in the first place: verifiability. If deliberation runs off-chain, coordination depends on a

trusted runtime whose decisions cannot be independently checked; participants must trust that each agent reasoned correctly and reported its decision truthfully. By executing the BDI machinery on-chain, the state that determines collective behaviour, which are beliefs, the intention stack, and the selected plan, becomes part of the shared, tamper-resistant ledger, and every deliberation step is auditable through events. For settings in which trust between participants is the central problem, such as the medical-supply scenario, this is a real gain: no participant has to trust another participant's reasoning, only a deterministic contract that everyone can inspect.

The cost of that gain is quantified in Section 7: the goal–plan-tree and deliberation-cycle modules dominate gas consumption (together 80.1% of the module benchmark in Table 6). Three design constraints keep this cost bounded. First, deliberation is bounded: goal–plan trees are finite and acyclic (C4) and plan search is bounded, so reasoning has a predictable per-transaction cost. Second, reasoning reads only on-chain state, which keeps deliberation deterministic and removes the oracle non-determinism that an off-chain reasoner would reintroduce. Third, the role and binding layer that must be verifiable by all participants is deliberately cheap (Table 7), so the expensive reasoning can be reduced without weakening the coordination guarantees.

A practical alternative is a *hybrid* architecture in which deliberation runs off-chain and only the resulting commitment (the selected action together with a justification hash) is recorded and verified on-chain. This would reduce gas substantially but reintroduces a trusted reasoner and weakens auditability. SocialRbSC therefore treats fully on-chain reasoning as the transparent end of a design spectrum and hybrid commitment as the scalable end; choosing between them is an application-level trade-off that the language makes explicit rather than hides. Quantifying the hybrid variant and adding off-chain reasoning with on-chain verification to the benchmark is the most important item of future work, and the current design is intentionally structured so that the reasoning layer can be swapped without changing the role, compartment, and binding contracts.

In summary, this discussion provides preliminary validation for both primary hypotheses. First, the investigated role qualities can be enforced through smart-contract state mechanisms and defined preconditions without incurring prohibitive gas costs, thereby supporting RQ1. While this approach shifts the primary computational challenge from role assignment to the planning and reasoning phase, RQ2 is supported within the scope of the measured scenarios (Section 7.7), as coordination is successfully modeled through explicit, reasoned procedures. The broader conclusion is that SocialRbSC effectively unifies social coordination mechanisms with trustworthy, executable logic; however, future research must prioritize mitigating the computational cost of reasoning-intensive components and rigorously validating the framework in larger, multi-agent deployments.

9. Conclusions

The main purpose of this paper is to introduce a language-centered approach to assembling social-agent-based applications with roles, compartments, and reasoning constructs. In essence, the paper proposes a first version of SocialRbSC together with a code-generation-oriented toolchain that extends earlier role concepts from [9,24] with explicit support for social agents and reasoning structures.

The conclusion can be stated directly in terms of the research questions. For **RQ1**, the study finds that the most suitable role characteristics for smart contracts are the characteristics that remain explicit, stateful, and verifiable during execution: compartment-scoped role activation, role-player binding, role-specific methods, and declarative preconditions. These constructs fit the smart-contract model because they can be audited through state transitions and do not dominate gas consumption in the reported implementation.

For **RQ2**, the study finds that agents with social and proactive skills can coordinate smart contracts by representing beliefs, intentions, goal–plan trees, and deliberation cycles as first-class elements of the language. The results show that this coordination strategy is technically feasible, but the cost of such feasibility is that reasoning-intensive modules consume most of the measured gas budget. SocialRbSC therefore demonstrates that intelligent coordination can be embedded into role-based smart contracts, while also revealing that planning and deliberation are the main targets for future optimization.

Taken into consideration as a whole, the study provides initial support for both working hypotheses: SocialRbSC can realize core role abstractions in a way that is compatible with smart-contract execution, and it can coordinate those abstractions through explicit social reasoning. The broader implication is that trustworthy, role-based coordination on blockchain platforms is achievable, but practical deployments will depend on reducing the cost of reasoning-heavy modules and evaluating the language in larger multi-agent scenarios.

Summary of Contributions

The contributions of this work are stated in full and up front in Section 1.2. In summary, SocialRbSC (i) provides a unified, precisely specified language model for social agents, roles, and compartments; (ii) makes beliefs, intentions, goal–plan trees, and deliberation cycles first-class constructs while exposing their on-chain cost; (iii) defines a code-generation pipeline with stated correctness properties for the construct-to-Solidity mapping; (iv) reports a reproducible gas evaluation together with a security assessment of the access-control-hardened generated contracts; (v) publishes the toolchain and artifacts; and (vi) identifies reasoning cost as the primary target for future optimization.

Author Contributions: Conceptualization, O.O. and U.A.; methodology, O.O. and U.A.; software, O.O.; validation, O.O.; formal analysis, O.O.; investigation, O.O.; resources, O.O.; data curation, O.O.; writing—original draft preparation, O.O.; writing—review and editing, O.O. and U.A.; visualization, O.O.; supervision, U.A.; project administration, U.A.; funding acquisition, U.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The SocialRbSC annotation processor, code generator, Solidity benchmark contracts, test cases, measurement scripts, and example models are available in the public Git repository at https://github.com/zointblackbriar/SocialRBSC (access on 15 November 2026). The exact revision used for the reported results is recorded by a release tag and commit identifier in that repository. The repository contains the SocialRbSC language toolchain (Java annotation DSL, four annotation processors, and the Velocity/JavaPoet/web3j code generator), the `SOLMASFramework` Solidity framework with its Hardhat test suite, gas report, and the extracted benchmark contracts, the example models under `SocialRbSC-Examples`, and an Ecore-style metamodel diagram under `ecore_model_diagram`. Installation instructions and pinned dependency versions are given in Section 7.9, and the exact reproduction commands are listed in Appendix C.

Acknowledgments: This work is funded by German Research (DFG) within the Research Training Group Role-Based Infrastructures for Continuous-Context-Sensitive Systems (GRK 1907).

Conflicts of Interest: The authors declare that they have no financial or personal relationships that could be perceived as influencing the research presented in this paper. No funding or support from any commercial or competing entities was received for this study. The authors have no direct or indirect financial interests, affiliations, or involvement with any organizations that could affect the objectivity or outcomes of the research.

Appendix A

This appendix provides the code-oriented material referenced in the main text. It contains the core SocialRbSC example (source-level DSL code), the medical supply chain case study, and the generated Solidity counterparts used to illustrate how the language maps to executable smart-contract artifacts.

Status of code listings. Listings A1 and A2 are SocialRbSC source-level specifications written in the annotation-based DSL; method bodies marked with comments such as "// Negotiation logic" are illustrative pseudocode placeholders that must be completed by the developer for each concrete deployment. The three Solidity listings are available verbatim in the supplementary repository at the following paths:

- `SOLMASFramework/contracts/paper_snippets/ConcreteMediator.sol`
- `SOLMASFramework/contracts/paper_snippets/MedicalSupplyMission.sol`
- `SOLMASFramework/contracts/paper_snippets/DeliveryProviderRole.sol`
- `SOLMASFramework/contracts/paper_snippets/GoalPlanTreeBenchmark.sol`

These contracts compile without error with `npx hardhat compile`. Placeholder helper functions (e.g., `calculateWeight`) are stubs that must be implemented with domain-specific logic for production deployment.

Appendix B. SocialRbSC Code Example

The full SocialRbSC code example for Listing A1 is shown below.

Listing A1: SocialRbSC source-level DSL specification (method bodies are illustrative; compilable versions in supplementary repository).

```
@Society(name=''Marketplace'')
@DeliberationProcess(description=''Deliberation Process'')
class SocietyAssignment1 {
        @Compartment(bindingRole=''SocialAgentSeller'',
            activateCompartment=true)
        @Compartment(bindingRole=''SocialAgentBuyer'',
            activateCompartment=true)
        interface Compartment1 {
        }

        @Belief
        class currentPriceIs100 {
                public void  beliefFunction() {
                }
        }

        @Intention
        class proposeReduction90Euro {
                public void  intentionFunction() {
                }
        }

        @SocialAgent(agentName=''SocialAgentBuyerA'')
        @Role
        class SocialAgentBuyer {
                public void  negotiate() {
                        // Negotiation logic
                }
        }
```

```
28
29          @SocialAgent(agentName=``SocialAgentSellerA'')
30          @Role
31          class SocialAgentSeller {
32                  public void adjustOffer() {
33                          moneyToBeSent += 1;
34                  }
35          }
36
37          @PreconditionVariable
38          public static int moneyToBeSent;
39
40          @Precondition
41          public boolean hasMoneyToBeSent() {
42                  return moneyToBeSent > 0;
43          }
44
45          @RoleMethod
46          public void increaseTheAmount() {
47                  moneyToBeSent += 1;
48          }
49  }
```

Appendix B.1. Medical Supply Chain Case Study Code Snippet General-Purpose Programming Language

Listing A2: SocialRbSC source-level DSL specification for medical supply chain (illustrative method bodies; full implementation in repository).

```
1   package socialrbsc.examples.medsupply;
2
3   import socialrbsc.lang.annotations.*;
4   import socialrbsc.solidity.generator.*;
5
6
7   public class MedicalSupplyDeliverySystem {
8
9     @Society(name = ``HealthcareNetwork'')
10    public class HealthcareSociety {
11
12      @Compartment(name = ``MedicalSupplyMission'')
13      @Properties({
14        @Property(name = ``deliveryDeadline'', type = ``uint256''),
15        @Property(name = ``requiredSupplies'', type = ``bytes32[]''
              ,
16        @Property(name = ``hospitalLocation'', type = ``address'')
17      })
18      public class MissionCompartment {
19
20        @Role
21        @RoleCardinality(maxInstances = 1)
22        @PlayingConstraint(``Objects may only play this role in one
              compartment at a time'')
23        public class DeliveryProviderRole {
```

```
24
25          @Precondition(``belief.capacity >= required_supplies.
                weight'')
26          @Postcondition(``intention.goal = DELIVER_SUPPLIES'')
27          public void acceptDelivery(SupplyPackage supplies) {
28            // Role-specific behavior
29            this.currentPayload = supplies;
30            this.status = DeliveryStatus.ACCEPTED;
31          }
32
33          @AccessControl(restrictedTo = {``DeliveryProvider''})
34          public void updateDeliveryStatus(Location currentLocation)
                {
35            this.lastKnownLocation = currentLocation;
36            this.status = DeliveryStatus.IN_TRANSIT;
37          }
38
39          @Transition(from = ``IN_TRANSIT'', to = ``DELIVERED'')
40          public void completeDelivery() {
41            this.deliveryTime = System.currentTimeMillis();
42            this.status = DeliveryStatus.DELIVERED;
43          }
44        }
45
46        @Role
47        @RoleCardinality(maxInstances = 1)
48        public class ValidatorRole {
49
50          @AccessControl(restrictedTo = {``Validator''})
51          @Postcondition(``intention.satisfied = true'')
52          public boolean confirmReceipt(
53            SupplyPackage supplies,
54            Certificate signedByValidator
55          ) {
56            if (!verifySignature(signedByValidator)) {
57              throw new IllegalArgumentException(``Invalid signature
                    '');
58            }
59            return true;
60          }
61        }
62
63        @Role
64        public class OrderCoordinatorRole {
65
66          @Belief(name = ``availableInventory'', type = ``mapping(
                bytes32 => uint256)'')
67          private Map<String, Long> inventory;
68
69          @Postcondition(``event.SupplyMatched emitted'')
70          public DeliveryAssignment matchSupplyWithDelivery(
71            SupplyRequest request,
72            List<DeliveryProvider> availableProviders
73          ) {
74            // Coordinator logic
```

```
            return assignBestProvider(request, availableProviders);
          }
        }
    }

@SocialAgent(agentName = ``UAVDeliveryAgent'')
    @PlayableRoles({``DeliveryProvider'', ``InventoryMonitor'', ``
        RoutePlanner''})
    @MotivationalModel({
      @Motivation(type = Affiliation.class, value = 0.8),
      @Motivation(type = Avoidance.class, target = ``
          hazardous_zones'', value = 0.9)
    })
    public class UAVAgent {

      @Belief(name = ``currentBattery'', type = ``uint256'')
      private long batteryLevel;

      @Belief(name = ``currentLocation'', type = ``Location'')
      private Location location;

      @Belief(name = ``availableRoutes'', type = ``Route[]'')
      private List<Route> routes;

      @Desire(name = ``deliverSuppliesOnTime'')
      @Priority(level = 1)
      private Goal primaryGoal;

      @Desire(name = ``minimizeBatteryUsage'')
      @Priority(level = 2)
      private Goal secondaryGoal;

      @Desire(name = ``avoidHazards'')
      @Priority(level = 3)
      @AvoidanceMotive
      private Goal avoidanceGoal;

      @GoalPlanTree(root = ``deliverSuppliesOnTime'')
      @Plan(name = ``OptimizedDeliveryPlan'')
      @Decomposition({
        ``navigateToHospital -> minimizeBatteryUsage'',
        ``avoidWeatherHazards -> avoidHazards'',
        ``validateDelivery -> completeMission''
      })
      public void deliberate() {
        // BDI deliberation cycle implementation
        this.updateBeliefs();
        this.evaluateGoals();
        this.selectIntention();
        this.executeIntention();
      }
    }

    @SocialAgent(agentName = ``DoctorAgent'')
```

```
    @PlayableRoles({''SupplyRequester'', ''Validator'', ''
        Collaborator''})
    public class DoctorAgent {

      @Belief(name = ''currentSupplyStatus'', type = ''
          SupplyInventory'')
      private SupplyInventory supplies;

      @Desire(name = ''ensureMedicalSupplyAvailability'')
      @Priority(level = 1)
      private Goal primaryGoal;

      @RoleMethod(role = ''SupplyRequester'')
      @Postcondition(''event.SupplyRequested emitted'')
      public SupplyRequest requestSupplies(
        String[] requiredItems,
        long deadline
      ) {
        // Request creation logic
        return new SupplyRequest(requiredItems, deadline);
      }
    }
  }
}
```

Listing A3: Generated Solidity—Society Mediator Contract (compilable; see repository for full test suite).

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.13;

interface Mediator {
    function bindAgent(address _agentAddress, string memory
        _agentName) external;
    function unbindAgent(string memory _agentName) external;
    function getAgentFromSociety(string memory _agentName)
        external view returns(address);
}

contract ConcreteMediator is Mediator {
    mapping(string => address) public assignedAgentsToSociety;
    bool public isMediator;

    // Society owner: the only authority allowed to register or
        remove agents.
    address public society;

    constructor() {
        society = msg.sender;
    }

    modifier onlySociety() {
        require(
            msg.sender == society,
            ''Only the society can manage agent bindings''
```

```
        );
        _;
    }

    function bindAgent(
        address _societyAddress,
        string memory _socialAgentName
    ) public override onlySociety {
        require(
            _societyAddress != address(0x0),
            ''a social agent should have an address''
        );
        assignedAgentsToSociety[_socialAgentName] =
            _societyAddress;
    }

    function unbindAgent(string memory _socialAgentName) public
        override onlySociety {
        delete assignedAgentsToSociety[_socialAgentName];
    }

    function getAgentFromSociety(
        string memory _agentName
    ) public view override returns(address) {
        address agentAddress = assignedAgentsToSociety[_agentName
            ];
        require(agentAddress != address(0x0), ''Agent not found in
             the society'');
        return agentAddress;
    }
}
```

Listing A4: Generated Solidity—Compartment with RBAC (compilable; helper-function bodies are minimal stubs).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface ICompartmentCore {
    function bindAgentToRole(address agent, string memory role)
        external;
    function hasRole(address agent, string memory role) external
        view returns (bool);
}

contract MedicalSupplyMission is ICompartmentCore {

    enum DeliveryStatus { PENDING, ACCEPTED, IN_TRANSIT, DELIVERED
        , CONFIRMED, FAILED }

    struct Supply {
        bytes32 id;
        string name;
        uint256 quantity;
        uint256 weight;
```

```
    }

    struct RoleBinding {
        address agent;
        string roleType;
        uint256 bindingTime;
        bool isActive;
    }

    struct TotalWeight {
        uint256 grams;
        uint256 itemCount;
    }

    // Society/compartment owner: the only authority allowed to bind or
    // unbind role players (access control on role-binding transitions).
    address public society;

    bytes32 public missionId;
    uint256 public deliveryDeadline;
    bytes32[] public requiredSupplies;
    address public hospitalLocation;

    mapping(address => mapping(string => RoleBinding)) public agentRoles;
    mapping(string => address[]) public roleMembers;
    // Populated through registerSupply(); Solidity mappings require no
    // explicit initialization.
    // slither-disable-next-line uninitialized-state
    mapping(bytes32 => Supply) public supplyInventory;
    mapping(address => DeliveryStatus) public deliveryStatus;

    event RoleBound(
        address indexed agent,
        string indexed role,
        uint256 timestamp
    );
    event RoleUnbound(
        address indexed agent,
        string indexed role,
        uint256 timestamp
    );
    event DeliveryStatusUpdated(
        address indexed provider,
        DeliveryStatus newStatus,
        uint256 timestamp
    );
    event DeliveryConfirmed(
        address indexed validator,
        address indexed provider,
        uint256 timestamp
    );
```

```
69
70      constructor() {
71          society = msg.sender;
72      }
73
74      modifier onlySociety() {
75          require(
76              msg.sender == society,
77              ''Only the society can manage role bindings''
78          );
79          _;
80      }
81
82      modifier onlyInRole(string memory requiredRole) {
83          require(
84              agentRoles[msg.sender][requiredRole].isActive,
85              ''Agent does not have required role''
86          );
87          _;
88      }
89
90      modifier roleNotInUse(address agent, string memory role) {
91          require(
92              !agentRoles[agent][role].isActive,
93              ''Agent already plays this role''
94          );
95          _;
96      }
97
98      modifier avoidHazardZones(string memory zone) {
99          // Motivational constraint: Avoidance of hazardous zones
100         require(
101             !isClassifiedAsHazard(zone),
102             ''Avoidance motive prevents entry to hazardous zone''
103         );
104         _;
105     }
106
107     function bindAgentToRole(
108         address agent,
109         string memory role
110     ) public override onlySociety roleNotInUse(agent, role) {
111         agentRoles[agent][role] = RoleBinding({
112             agent: agent,
113             roleType: role,
114             bindingTime: block.timestamp,
115             isActive: true
116         });
117
118         roleMembers[role].push(agent);
119         emit RoleBound(agent, role, block.timestamp);
120     }
121
122     function hasRole(
123         address agent,
```

```
        string memory role
    ) public view override returns (bool) {
        return agentRoles[agent][role].isActive;
    }

    function registerSupply(
        bytes32 id,
        string memory name,
        uint256 quantity,
        uint256 weight
    ) public onlySociety {
        supplyInventory[id] = Supply(id, name, quantity, weight);
    }

    function unbindAgentFromRole(
        address agent,
        string memory role
    ) public onlySociety {
        require(
            agentRoles[agent][role].isActive,
            ''Agent does not hold this role''
        );

        agentRoles[agent][role].isActive = false;
        emit RoleUnbound(agent, role, block.timestamp);
    }

    function acceptDelivery(
        address provider,
        bytes32[] memory suppliesList
    ) public onlyInRole(''DeliveryProvider'') {
        require(
            msg.sender == provider,
            ''Only the provider can accept the delivery''
        );

        TotalWeight memory packageWeight = calculateWeight(
            suppliesList);
        require(
            checkUAVCapacity(provider, packageWeight),
            ''Insufficient UAV capacity''
        );

        deliveryStatus[provider] = DeliveryStatus.ACCEPTED;
        emit DeliveryStatusUpdated(provider, DeliveryStatus.
            ACCEPTED, block.timestamp);
    }

    function updateDeliveryStatus(
        address provider,
        DeliveryStatus newStatus,
        string memory zone
    ) public onlyInRole(''DeliveryProvider'') avoidHazardZones(
        zone) {
        require(
```

```
            msg.sender == provider,
            ''Only the provider can update their status''
        );
        require(
            newStatus == DeliveryStatus.IN_TRANSIT ||
            newStatus == DeliveryStatus.DELIVERED,
            ''Invalid status transition''
        );

        deliveryStatus[provider] = newStatus;
        emit DeliveryStatusUpdated(provider, newStatus, block.
            timestamp);
    }

    function confirmDelivery(
        address provider,
        bytes calldata signedValidation
    ) public onlyInRole(''Validator'') {
        require(
            deliveryStatus[provider] == DeliveryStatus.DELIVERED,
            ''Delivery must be in DELIVERED state''
        );
        require(
            verifyValidatorSignature(signedValidation),
            ''Invalid validator signature''
        );

        deliveryStatus[provider] = DeliveryStatus.CONFIRMED;
        emit DeliveryConfirmed(msg.sender, provider, block.
            timestamp);
    }

    // Helper functions (domain-specific logic to be completed per
         deployment)
    function calculateWeight(bytes32[] memory supplies)
        private view returns (TotalWeight memory) {
        uint256 total = 0;
        for (uint256 i = 0; i < supplies.length; i++) {
            total += supplyInventory[supplies[i]].weight;
        }
        return TotalWeight(total, supplies.length);
    }

    function checkUAVCapacity(address uav, TotalWeight memory
        weight)
        private view returns (bool) {
        // Capacity check - domain-specific; returns true by
            default
        return weight.grams > 0;
    }

    function isClassifiedAsHazard(string memory zone)
        private pure returns (bool) {
        // Hazard classification - domain-specific; returns false
            by default
```

```
        return false;
    }

    function verifyValidatorSignature(bytes calldata signature)
        private pure returns (bool) {
        // Signature verification - domain-specific; returns true
            by default
        return signature.length > 0;
    }
}
```

Appendix B.2. Generated Counterpart of Solidity Medical Supply Chain Case Study

Listing A5: Generated Solidity—Role-Based Behavior Contract (compilable; helper functions are stubs for domain-specific logic).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IDeliveryProviderRole {
    function acceptDelivery(bytes32[] memory supplies) external;
    function updateStatus(string memory zone) external;
    function completeDelivery() external;
}

contract DeliveryProviderRole is IDeliveryProviderRole {

    enum DeliveryStatus { PENDING, ACCEPTED, IN_TRANSIT, DELIVERED
        }

    address public compartment;
    address public uavAgent;
    DeliveryStatus public currentStatus;
    bytes32[] public currentPayload;

    event DeliveryAccepted(
        address indexed agent,
        bytes32[] supplies,
        uint256 timestamp
    );
    event StatusUpdated(
        address indexed agent,
        DeliveryStatus newStatus,
        uint256 timestamp
    );
    event DeliveryCompleted(
        address indexed agent,
        uint256 deliveryTime
    );

    modifier onlyCompartment() {
        require(msg.sender == compartment, ''Only compartment can
            call'');
        _;
```

```
    }

    modifier onlyAgent() {
        require(msg.sender == uavAgent, ''Only assigned agent can
            call'');
        _;
    }

    constructor(address _compartment, address _uavAgent) {
        compartment = _compartment;
        uavAgent = _uavAgent;
        currentStatus = DeliveryStatus.PENDING;
    }

    function acceptDelivery(
        bytes32[] memory supplies
    ) public override onlyAgent {
        require(
            currentStatus == DeliveryStatus.PENDING,
            ''Role must be in PENDING state''
        );

        // Precondition check: capacity check
        require(
            checkBatteryCapacity(),
            ''Insufficient battery for delivery''
        );

        currentPayload = supplies;
        currentStatus = DeliveryStatus.ACCEPTED;

        emit DeliveryAccepted(msg.sender, supplies, block.
            timestamp);
    }

    function updateStatus(
        string memory zone
    ) public override onlyAgent {
        require(
            currentStatus == DeliveryStatus.ACCEPTED,
            ''Must have accepted delivery first''
        );

        // Motivational constraint: Avoidance of hazardous zones
        require(
            !isHazardZone(zone),
            ''Avoidance motive: Cannot enter hazardous zone''
        );

        currentStatus = DeliveryStatus.IN_TRANSIT;
        emit StatusUpdated(msg.sender, DeliveryStatus.IN_TRANSIT,
            block.timestamp);
    }

    function completeDelivery() public override onlyAgent {
```

```
        require(
            currentStatus == DeliveryStatus.IN_TRANSIT,
            ''Must be in transit to complete''
        );

        // Postcondition: Record completion timestamp
        currentStatus = DeliveryStatus.DELIVERED;
        emit DeliveryCompleted(msg.sender, block.timestamp);
    }

    // Helper functions for BDI integration (domain-specific logic
        to be completed per deployment)
    function checkBatteryCapacity() private pure returns (bool) {
        return true;
    }

    function isHazardZone(string memory zone) private pure returns
        (bool) {
        return false;
    }
}
```

Appendix C. Reproduction Commands

Listing A6 lists the command sequence that reproduces the gas measurements reported in Tables 7–11 and the static-analysis result of Section 5.11. The pinned dependency versions that these commands rely on are given in Table 14.

Listing A6: Reproduction Commands.

```
git clone https://github.com/zointblackbriar/SocialRBSC
cd SocialRBSC
mvn clean install          # Java annotation DSL, 4 processors,
    Velocity/JavaPoet generator
cd SOLMASFramework
npm ci                     # install pinned Hardhat/ethers
    dependencies
npx hardhat compile        # compiles the extracted benchmark
    contracts
npx hardhat run scripts/measure-evaluation.js      # reproduces
    Tables 7-11
slither contracts/paper_snippets --solc solc-0.8.13 # reproduces
    the security section
```

References

1. Bäumer, D.; Riehle, D.; Siberski, W.; Wulf, M. The Role Object Pattern. In Proceedings of the 4th Conference on Pattern Languages of Programs (PLoP '97), Monticello, NY, USA, 3–5 September 1997.
2. Baldoni, M.; Boella, G.; van der Torre, L. powerJava: Ontologically Founded Roles in Object Oriented Programming Languages. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06), Dijon, France, 23–27 April 2006*; Association for Computing Machinery: New York, NY, USA, 2006; pp. 1414–1418.
3. Herlea, D.; Jonker, C.M.; Treur, J.; Wijngaards, N.J.E. Specification of Behavioural Requirements within Compositional Multi-Agent System Design (SCROLL). In *Engineering Societies in the Agents World IV (ESAW 2003)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3071, pp. 8–26.

4. Herrmann, S. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World (Net.ObjectDays 2002)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2591, pp. 248–264.
5. Herrmann, S. A precise model for contextual roles: The programming language ObjectTeams/Java. *Appl. Ontol.* **2007**, *2*, 181–207.
6. Object Management Group. *Model Driven Architecture (MDA) Guide, Version 1.0.1*; OMG Document omg/2003-06-01; Object Management Group: Boston, MA, USA, 2003.
7. Zhu, H. *E-CARGO and Role-Based Collaboration: Modeling and Solving Problems in the Complex World*; John Wiley & Sons: Hoboken, NJ, USA, 2021.
8. Castelfranchi, C. Modelling social action for AI agents, *Artif. Intell.* **1998**, *103*, 157–182. https://doi.org/10.1016/S0004-3702(98)00056-3.
9. Kühn, T.; Leuthäuser, M.; Götz, S.; Seidl, C.; Aßmann, U. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*; Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J., Eds.; Springer International Publishing: Cham, Switzerland, 2014; pp. 141–160. ISBN 978-3-319-11245-9.
10. Ciatto, G.; Maffi, A.; Mariani, S.; Omicini, A. Towards Agent-Oriented Blockchains: Autonomous Smart Contracts. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Social Good*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 29–41.
11. Yuan, M.J. *Building Blockchain Apps*; Addison-Wesley Professional: Boston, MA, USA, 2019.
12. Anumba, C.; Ugwu, O.; Newnham, L.; Thorpe, A. A Multi-Agent System for Distributed Collaborative Design. *Logist. Inf. Manag.* **2001**, *14*, 355–367.
13. Richardson, J.; Schwarz, P. Aspects: Extending Objects to Support Multiple, Independent Roles. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, CO, USA, 29–31 May 1991*; Association for Computing Machinery: New York, NY, USA, 1991; pp. 298–307.
14. Ducasse, S.; Nierstrasz, O.; Schärli, N.; Wuyts, R.; Black, A.P. Traits: A Mechanism for Fine-Grained Reuse. *ACM Trans. Program. Lang. Syst.* **2006**, *28*, 331–388.
15. Schärli, N.; Ducasse, S.; Nierstrasz, O.; Black, A.P. Traits: Composable Units of Behaviour. In *ECOOP 2003: Object-Oriented Programming*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 248–274.
16. Wooldridge, M.; Ciancarini, P. Agent-Oriented Software Engineering: The State of the Art. In *Agent-Oriented Software Engineering*; Ciancarini, P., Wooldridge, M.J., Eds.; Springer: Berlin/Heidelberg, Germany, 2001; pp. 1–28.
17. Dignum, F.; Hofstede, G.J.; Prada, R. From Autistic to Social Agents. In Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), Paris, France, 5–9 May 2014.
18. Omicini, A. SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems. In *Agent-Oriented Software Engineering*; Ciancarini, P., Wooldridge, M.J., Eds.; Springer: Berlin/Heidelberg, Germany, 2001; pp. 185–193.
19. Harrison, W.; Ossher, H. Subject-oriented programming: A critique of pure objects. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Washington, DC, USA, 26 September–1 October 1993*; Association for Computing Machinery: New York, NY, USA, 1993; pp. 411–428.
20. Elrad, T.; Filman, R.E.; Bader, A. Aspect-oriented programming: Introduction. *Commun. ACM* **2001**, *44*, 29–32.
21. Laddad, R. *AspectJ in Action: Enterprise AOP with Spring Applications*; Simon and Schuster: New York, NY, USA, 2009.
22. Bergmans, L.; Aksit, M. Composing multiple concerns using composition filters. *Commun. ACM* **2001**, *44*, 51–57.
23. Ungar, D.; Adams, S.S. Harnessing emergence for manycore programming: Early experience integrating ensembles, adverbs, and object-based inheritance. In *Proceedings of the ACM International Conference Companion on Object-Oriented Programming Systems, Languages, and Applications Companion, Reno/Tahoe, NV, USA, 17–21 October 2010*; Association for Computing Machinery: New York, NY, USA, 2010; pp. 19–26.
24. Steimann, F. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data Knowl. Eng.* **2000**, *35*, 83–106. https://doi.org/10.1016/S0169-023X(00)00023-9.
25. Riehle, D.; Gross, T. Role Model Based Framework Design and Integration. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98), Vancouver British, CO, Canada, 18–22 October 1998*; Association for Computing Machinery: New York, NY, USA, 1998; pp. 117–133.
26. Mavridou, A.; Laszka, A.; Stachtiari, E.; Dubey, A. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security (FC 2019)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11598, pp. 446–465.
27. Rouhani, S.; Deters, R. Security, Performance, and Applications of Smart Contracts: A Systematic Survey. *IEEE Access* **2019**, *7*, 50759–50779.
28. Ante, L. Smart contracts on the blockchain: A bibliometric analysis and review. *Telemat. Inform.* **2021**, *57*, 101519.
29. Soffer, P.; Rolland, C. Combining intention-oriented and state-based process modeling. In *Proceedings of the International Conference on Conceptual Modeling, Klagenfurt, Austria, 24–28 October 2005*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 47–62.

30. López-Pintado, O.; García-Bañuelos, L.; Dumas, M.; Weber, I.; Ponomarev, A. Caterpillar: A business process execution engine on the Ethereum blockchain. *Softw. Pract. Exp.* **2019**, *49*, 1162–1193. https://doi.org/10.1002/spe.2702.
31. Mavridou, A.; Laszka, A. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In *Financial Cryptography and Data Security: FC 2018 International Workshops*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2018; Volume 10958.
32. Hamdaqa, M.; Met, L.A.P.; Qasse, I. iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Inf. Softw. Technol.* **2022**, *144*, 106762. https://doi.org/10.1016/j.infsof.2021.106762.
33. Wood, G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum Project Yellow Paper. 2014. Available online: https://ethereum.github.io/yellowpaper/paper.pdf (accessed on 15 November 2026).
34. Johnson, D.; Menezes, A.; Vanstone, S. The Elliptic Curve Digital Signature Algorithm (ECDSA). *Int. J. Inf. Secur.* **2001**, *1*, 36–63. https://doi.org/10.1007/s102070100002.
35. Feist, J.; Grieco, G.; Groce, A. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '19), Montreal, QC, Canada, 27 May 2019*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 8–15.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.