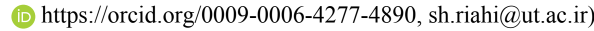
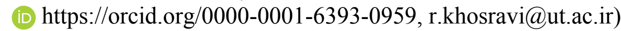
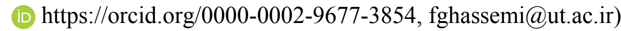
Journal of Universal Computer Science, vol. 30, no. 4 (2024), 462-501
submitted: 5/3/2023, accepted: 15/11/2023, appeared: 28/4/2024 *CC BY-ND 4.0*

Knowledge-Related Policy Analysis in an Inference-Enabled Actor Model

Shahrzad Riahi
(School of Electrical and Computer Engineering, College of Engineering,
University of Tehran, Tehran, Iran

https://orcid.org/0009-0006-4277-4890, sh.riahi@ut.ac.ir)

Ramtin Khosravi
(School of Electrical and Computer Engineering, College of Engineering,
University of Tehran, Tehran, Iran

https://orcid.org/0000-0001-6393-0959, r.khosravi@ut.ac.ir)

Fatemeh Ghassemi
(School of Electrical and Computer Engineering, College of Engineering,
University of Tehran, Tehran, Iran
School of Computer Science, Institute for Research in Fundamental Sciences,
PO. Box 19395-5746, Tehran, Iran

https://orcid.org/0000-0002-9677-3854, fghassemi@ut.ac.ir)

Abstract: People provide their information to distributed systems to receive the desired services. This information may be disclosed to the agents of the system as part of messages transmitted among them. As the agents of the system are smart, they can infer new information from their obtained information, that they may not be authorized to know. So preserving privacy in such systems is an important and yet challenging issue. We study the problem of analyzing the disclosure of private information in distributed asynchronous systems. Our approach to prevent private information disclosure is to require the system to follow knowledge-related policies defined for the system at design time. To achieve this, we construct a model of the system and assume the policies as the system properties and check whether these properties are satisfied in the system or not. In order to construct a model of the system, we extend the actor model, which is a well known reference model for distributed asynchronous systems, by enriching actors by the knowledge base and inference capability. As our knowledge-related policies should not be violated in any state of the system, we propose an efficient invariant model checking algorithm to verify the satisfaction of the policies in our actor model.

Keywords: Distributed systems, Knowledge, Inference, Privacy Policy, Formal Verification
Categories: F.4, M.4
DOI: 10.3897/jucs.103011

1 Introduction

Today, the use of distributed systems, as in Internet of Things (IoT) or microservice architectures, is growing and people provide their information to these systems to receive the desired services. In these systems, personal information is collected by various processes and devices. The information may be shared with many service providers to

be analyzed or reported for further objectives [Samani, 2015]. The information may also be provided to other parties for various intentions such as research or marketing. In such systems, if there is no sufficient control over the transmitted data among agents, a personal data breach may happen, so preserving privacy in such systems is an important and yet challenging issue. For example, a patient provides her personal information to a health care system for treatment, but if this information is sent to a research organization without the patient's consent, a personal data breach has occurred. General Data Protection Regulation (GDPR) [GDPR, 2020] defines personal data breach as "*a breach of security leading to the accidental or unlawful destruction, loss, alteration, unauthorized disclosure of, or access to, personal data transmitted, stored or otherwise processed*". As an example in the field of IoT, the use of smart home applications has provided useful facilities to users and improved their quality of life, however has also raised potential privacy challenges because of the vast amount of collected personal and sensitive data [Bugeja et al., 2021, Zeng and Roesner, 2019]. Home applications integrate smart locks, thermostats, switches, surveillance systems, and appliances and allow users to monitor and interact with their living spaces from anywhere. These applications have access to private data, such as information about when the user sleeps, who others are at home, or when others are at home that could lead to privacy issues if disclosed to unauthorized persons [Celik et al., 2019]. The information transmitted by an object probably will not cause any privacy issues on its own. However, when pieces of information from different objects are joined, grouped, and analyzed, they can reveal sensitive information [Chanal and Kakkasageri, 2020]. Therefore, it is necessary to have methods that can control such disclosure of information. Based on [Schneider, 2018], privacy cannot be enforced uniquely by technical means.

Moreover, in today's distributed systems, a large amount of data is produced and consumed by various components, which causes emerging network problems. To deal with these problems, computing paradigms such as edge computing and fog computing have been proposed and many efforts have been made to improve their performance. For instance, studies such as [Dong et al., 2023, Mohajer et al., 2022A, Mohajer et al., 2022B], have been done in the field of effective transmission strategies and optimization of resource allocation and energy consumption in edge computing. On the other hand, these new paradigms cause new privacy concerns. For instance, in edge computing, there may be honest but curious adversaries (such as edge data centers, infrastructure providers, services providers, or some users) that are usually authorized entities that intend to obtain more sensitive information while playing their role in the system [Zhang et al., 2018, Bi et al., 2020]. Such issues require more attention to privacy concerns in today's distributed systems.

Privacy violations can happen in different ways. Information collection, information processing, information dissemination, and invasions are different types of privacy violations [Solove, 2006]. Disclosure is a special form of information dissemination, which means "making private information known outside the group of individuals expected to know it" [Tschantz and Wing, 2009]. For example, knowing the salary information is allowed for members of a family, but it is not allowed for a colleague [Samani, 2015]. A useful method to prevent private information disclosure would be to define policies which control the disclosure of this information in the system and require the system to follow those policies. As stated in [Ronne, 2012], if the policies for protecting the privacy of individuals are violated, it is not only harmful to the individuals whose information is being disclosed, but can also be damaging to the organization that violates these policies. Therefore, having a framework with formal foundation to ensure that the system works according to its defined policies, is valuable.

Privacy is correlated with the interaction aspects of distributed systems [Samani, 2015]. Disclosure of personal information occurs when an agent receives information. The ways in which an agent can receive information about other agents can be classified into three categories: direct receive, indirect receive, or receive by inference [Riahi et al., 2017, Blanke, 2020]. The difference between direct and indirect receives is that in the first case, the owner of personal information directly sends its information to another agent, but in the second case an agent sends the personal information of another agent to a third one. In *receive by inference*, the agent infers other agents' personal information based on the information received previously from other agents. We introduce an approach that checks the disclosure of sensitive information as the result of inference, as well as direct and indirect receive, in the distributed systems.

We use model checking to analyze information disclosure, i.e., policies, in the presence of inference capability for the agents in the domain of distributed systems. We need a modeling notation that is suitable for specifying and analyzing such systems. We base our modeling approach on *Actor model* [Agha, 1985], which is a well known computation model for concurrent and distributed systems. An actor model consists of a set of active objects called *actors*, which encapsulate data, communicate via asynchronous message passing, and have no shared data. These characteristics are naturally suitable for modeling distributed systems in the real world. The actor model guarantees delivery of the messages, but the order in which the actors execute and the order of receiving messages, which are sent by different actors to a specific actor, are nondeterministic. This nondeterminism models the delays and effects of the network in sending messages. There are a number of actor-based programming and modeling languages like Erlang [Armstrong, 2007], Rebeca [Sirjani et al., 2004], Ptolemy II [Eker et al., 2003], and ABS [Johnsen et al., 2012a, ABS, 2022] proposed for different design concerns. For example, Rebeca and ABS are designed for analysis and code generation [Boer et al., 2017], while Erlang is optimized for efficient execution. However, modeling and analysis of privacy and information disclosure have not been the design concern of any of them. In the actor model, the actors' information can be disclosed among other actors as part of the transmitted messages, so it is essential to protect actors' private information from disclosure to unauthorized actors.

In [Riahi et al., 2017], we addressed the analysis of data disclosure policies in actor model by considering direct and indirect receives. The actor model used in that research does not support modeling the knowledge and inference capabilities of the actors. So, the receive by inference has not been addressed in that work. The existing actor modeling languages do not have the ability to model the knowledge and inference capabilities of the actors either. To tackle this problem, in the current paper, we propose *Inference-Enabled Actor* model (*Inferactor* for short) which enables us to model the actors' knowledge and inference capability and define policies that enable one to specify restrictions over the actors' knowledge to avoid disclosure of private information to unauthorized actors. As these policies are defined and checked on the actors' knowledge, we call them knowledge-related policies.

In Inferactor, we extend the Actor model with a knowledge-based logic. In this way, we made it possible for actors' knowledge to play a role in defining actors' behavior, which means that the actors can do something based on the knowledge they have or make inferences using their defined inference rules. Compared to the existing actor modeling languages, Inferactor enables modeling the knowledge and inference capabilities of the actors, but in terms of actor computation, it does not add much to the existing modeling languages and it is almost similar to Rebeca. To model the knowledge, inference capabilities, and knowledge-related policies, we define a knowledge-based logic based on

first-order epistemic logic [Fagin et al., 2003] and the knowledge-based logic presented in [Pardo et al., 2017]. As we present the formal syntax and semantics for defining policies, our policies are categorized as machine-readable privacy policies, according to [Morel and Pardo, 2020]. Our knowledge-related policies are invariant properties, so we provide an efficient invariant model checking algorithm to verify the satisfaction of the policies in an Inferactor model that checks the satisfaction of policies by reachable states constructed through a Breadth-First Search (BFS). Our algorithm is efficient in such a way that it does not maintain all the states visited during BFS, and in each state it checks the policies only for the actor whose knowledge has changed compared to the previous state. Our method can identify privacy violations from the system model in design time. Furthermore, it can be used for a system that has been implemented to identify privacy violations by analyzing the model of that system.

The main contribution of this paper is that we have presented a method that can model distributed systems in which the agents have the ability to infer, and can verify privacy requirements on the model with a policy-based method. To this end, the following items are proposed:

- We propose a formal model, called Inferactor, which enables us to model the knowledge and inference capabilities of the actors as well as their usual behavior. An overview of the main parts of Inferactor and the running example that will be used throughout this paper are introduced in Sect. 2. The formal description of Inferactor syntax and operational semantics are presented in Sect. 4 and Sect. 5, respectively.

- We define a knowledge-based logic to specify actors' knowledge and inference capabilities and to reason about the actors' knowledge. Our knowledge-based logic is based on first-order epistemic logic [Fagin et al., 2003] and the knowledge-based logic presented in [Pardo et al., 2017]. These logics cover more formulas than we need in our work. Therefore, we define a subset of these logics that is sufficient for modeling the knowledge and inference capabilities of the actors as well as providing the possibility of reasoning on the actors' knowledge to cover their inference capabilities, as our knowledge-based logic. The syntax and semantics of our defined knowledge-based logic are presented in Sect. 3 and Sect. 6, respectively.

- We define a subset of our knowledge-based logic to specify knowledge-related policies which made it possible to define policies globally for the entire system. The knowledge-related policies impose restrictions on information (about other actors and their knowledge) that an actor can access. The syntax and semantics of knowledge-related policies are presented in Sect. 7.

- We propose an efficient model checking algorithm to check the satisfaction of the policies and prove that our method is correct. Our algorithm is efficient in two respects 1) it does not maintain all the states visited during BFS, and 2) it only checks the policies for the actor of states whose knowledge has changed using the static information of the model. Our model checking algorithm is proposed in Sect. 8.

2 Inference-Enabled Actor Model

Due to the use of various IoT applications and services, a large amount of personal information, such as location, health, and energy consumption information, is available

to data consumers. In addition, data consumers can infer new information based on the information they have received. For example, specific appliances (such as medical devices) that are used by a person can be inferred by mining the signatures of that person's electricity consumption [Lisovich et al., 2010]. Data consumers can also combine multiple data items and/or with other information obtained from external data sources and thus infer new information based on their inference capabilities. For example, suppose Alice falsely claims that she lives alone. In this case, any data consumer who has access to both her location and electricity consumption metadata can infer that Alice lied by identifying the usage of some specific devices (such as microwaves and TVs) when Alice's location is not her home [Chaaya et al., 2019]. The inference capabilities increase the possible privacy risks, and in many studies, such as [Chaaya et al., 2019], [Wang et al., 2022], and [Yeom et al., 2018], the inference and the importance of considering it have been discussed. Therefore, it is very important to have methods to detect the information that is inferred. The modeling and analysis of information disclosure in the presence of actors' inference capabilities have not been the design concern of any of the existing actor models. Therefore, in our proposed model, we provide the ability to model the actors' inference capabilities in addition to the usual computations. To determine whether these inference capabilities can lead to the inference of new information, the information that users obtain directly or indirectly, should be kept. The usual way to do this is to define users' knowledge and store it in a knowledge base (Like in [Fagin et al., 2003, Pardo and Schneider, 2014, Pardo and Schneider, 2017]).

In this section, we define our model (based on Actor model [Agha, 1985]), called *Inference-Enabled Actor* model (*Inferactor* for short), which is enriched by actors' knowledge and inference capability. In addition to the basic properties of actors in Actor model, the actors in Inferactor have some new features including:

- Each actor has a knowledge base for keeping the knowledge obtained during model execution.
- An actor can use the knowledge saved in its knowledge base when handling messages.
- An actor can infer new knowledge from its obtained knowledge based on its defined inference rules.

An overview of the actor's structure in Inferactor, taking into account the actors' knowledge base and inference rules, along with their basic properties, is shown in Fig. 1. In this section, we informally describe the computation model of Inferactor, the concepts of knowledge and inference capabilities of the actors, and also the running example which will be used throughout this paper. The formal syntax and formal semantics of Inferactor are presented in the next sections.

2.1 Computation model of Inferactor

A model of a system consists of a set of entities that communicate with each other. Since our model is based on the Actor model, each entity is modeled by an actor. For example, in the running example (which will be described later in this section), there are four entities: consumer, smart meter, utility, and analyzer, and each of them is modeled as an actor. Like in the Actor model, there is no intra-object concurrency in Inferactor and each actor has only one execution thread as shown in Fig. 1. The actors have no shared data, and the only way to transfer data between the actors is to send messages to each

other. These actors may have their own personal information, and the actors' personal information may propagate to other actors as part of transmitted messages.

As described earlier, the actors communicate via asynchronous message passing. Communications among the actors are modeled by messages, and the required computations are modeled by the methods defined to serve the messages. The received messages are put in the message queue of the receiver actor. An actor takes a message from its message queue and executes the method defined to serve this message and then it proceeds by processing the next message in this queue (Methods and Message Queue in Fig. 1). A message may contain information about an actor in the system, so receiving and taking a message by an actor, can increase the knowledge of the actor.

By information about an actor, we mean the information whose subject actor is identifiable. The concept of identity of an individual person has been defined in [Pfitzmann and Hansen, 2010] as: *"an identity is any subset of attribute values of an individual person which sufficiently identifies this individual person within any set of persons"*. We model the identity of the actor by the actor's name. We assume the set *ID* which is the set of all actor identifiers in Inferactor model.

2.2 Knowledge and Inference Capabilities of the Actors

The actors may have initial knowledge and can obtain new knowledge through their interactions with other actors. We model a knowledge base for each actor to keep its obtained knowledge (Knowledge Base in Fig. 1). The knowledge base of each actor contains the knowledge that the actor directly obtains through interactions with other actors.

In addition to the knowledge that is explicitly in the knowledge bases of the actors, the actors may have inference capabilities that enable them to infer new knowledge that can be derived from their current knowledge in their knowledge bases. For example, if one knows the working hours of an employee in a project and the hourly basis for her, then she can infer the salary of that employee in that project. To model the inference capabilities of the actors, we explicitly specify the inference capabilities of each actor in the actor's body, in terms of inference rules (Inference Rules in Fig. 1). An inference rule specifies that an actor can infer new knowledge from its existing knowledge.

The actor can save the obtained knowledge on its knowledge base, query the desired knowledge from its knowledge base and behave accordingly. We refer to these operations by save, query, and conditional operations.

2.3 Running Example

We describe a simple Inferactor model as the running example which will be used throughout this paper. Smart grid is one of the areas where privacy is a main and challenging issue. Due to the distributed nature of smart grids and the importance of protecting privacy of the people whose information is transmitted and revealed through the communications among different components of the smart grids, we define our running example in the domain of smart grid and smart metering. We first give a brief overview of the privacy concerns in smart grids, and then define the running example.

The most important mechanism in smart grids is smart metering. Smart metering is used to obtain information from, and control the behavior of consumer' devices and appliances [Fang et al., 2012]. The data collected by smart meters may be used to invade consumers' privacy. For example, analysis of energy consumption traces can

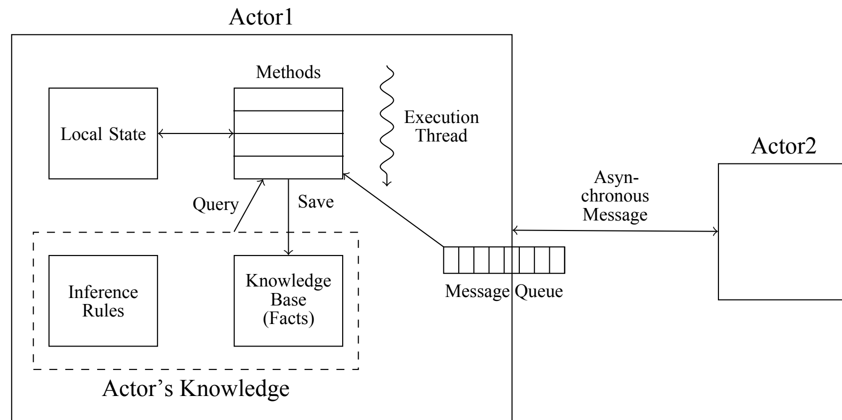


Figure 1: An overview of the actor's structure in Inferactor.

reveal the consumers' smart appliances (based on their load signatures). Data mining algorithms can also be used to invade the privacy of consumers by revealing their life-styles and economic status [Asghar et al., 2017]. Many techniques have been introduced for appliance load monitoring to infer information about consumers' behavior, habits or preferences [Giaconi et al., 2020]. This information includes the time they eat, when they watch TV, the periods that they are home, etc. [Cardenas and Safavi-Naini, 2012]. In addition, aggregating data from multiple sources in smart grids can reveal consumers' personal information.

Smart meters cannot transmit any sensitive data such as consumer name or address, and use a smart meter ID number to transmit sensitive data [Zabkowski and Gajowniczek, 2013]. The three major participants in the smart grid are consumers, utilities and third party service providers [Simmhan et al., 2011]. The information sent from the consumer's smart meter to the utility includes smart meter ID number, meter readings on different granularity levels, type of information transmitted, date and time, and payment details (for the customers using prepayment meter) [Zabkowski and Gajowniczek, 2013].

The smart meter data is also attractive to other people such as insurance companies (to determine premiums), marketers (to profile customers for targeted advertisements), advisory companies (to promote energy conservation and awareness), criminals (to identify best times for a burglary, or valuable appliances to steal) [Gunduzl et al., 2015].

Based on the above explanations, we define a simple scenario in smart grids, with an emphasis on inference capabilities of the actors and aggregating data from multiple sources. We assume each smart meter has a unique identifier, and each smart meter is related to the corresponding consumer by this identifier. In this scenario the consumers' smart meters send their electricity consumption (e-consumption for short) traces to a utility. The utility sends the smart meter identifier, e-consumption trace, and the city of some consumers to an analyzer to perform analysis on electricity consumption of consumers based on their cities. To do this, the utility first sends the cities of some consumers and then the e-consumption traces of those consumers to the analyzer. Upon receiving any of this information, the analyzer saves it in its knowledge base. The analyzer also sends a message to consumer to request consumer's personal information including consumer's name and smart meter identifier. The running example modeled in Inferactor

is presented in Sect. 4.

3 Knowledge-based Logic

We intend to model actors' knowledge and inference capabilities along with their other capabilities, so we need to define a logic which provides the formal semantics of actors' knowledge and inference capabilities, and also provides the ability to reason about the actors' knowledge. In addition, our policies are closely connected with the actors' knowledge. So, epistemic logic [Fagin et al., 2003, Meyer and van der Hoek, 1995], or the logic of knowledge, which is used for modelling and reasoning about knowledge, is the suitable logic for our model. Based on [Pardo and Schneider, 2017], epistemic logic provides great precision and granularity for modelling and reasoning about the knowledge of the agents in a system. Epistemic logic has played the main role in the modeling of knowledge in artificial intelligence and distributed computing [Hsu et al., 2001]. Many studies (such as [Hsu et al., 2001, Van Der Hoek and Verbrugge, 2002, Halpern and O'Neill, 2008, Pucella, 2013, Lehnherr et al., 2022]) have been done on the use of epistemic logic in various fields. The standard semantics of epistemic logic is defined over Kripke models (possible-worlds model).

Epistemic logic makes it possible to model knowledge and inference capabilities of the actors, and the privacy policies (by defining properties for actors' knowledge), so we define a knowledge-based logic (called $\mathcal{IKBL}$) based on epistemic logic [Fagin et al., 2003], and define the semantics of our knowledge-based logic over Inferactor.

3.1 Terms and Facts

We use the concept of term, to model the information in Inferactor model, as in studies like [Pardo et al., 2017] and [Pavlovic and Meadows, 2011]. A term is a piece of information about an actor. We denote the set of all terms in the model as $STerm$. Each term consists of a term name and a sequence of arguments.

A term that all its arguments have specified values is called fact. Facts are used to model the knowledge of the actors. The set of all facts in the model is denoted as $SFact$. For example, 'Alice is 30 years old' is a fact and we use the notation $age(\text{“Alice”}, 30)$ to denote this fact. Based on the definitions of facts and terms, we have $SFact \subseteq STerm$.

3.2 Knowledge-based Logic Syntax

The knowledge-based logic is used to specify the knowledge of the actors. The knowledge of an actor consists of the knowledge about data or about the knowledge of the other actors. The actors may have inference capabilities that enable them to infer new knowledge, based on their current knowledge. The inference rules are specified in the body of an actor, and then translated to logic formulas.

We define the knowledge-based logic $\mathcal{IKBL}$ for Inferactor, based on the knowledge-based logic presented in [Pardo et al., 2017] which itself is based on first-order epistemic logic [Fagin et al., 2003]. These logics cover more formulas than we need, including predicates to encode permissions and connections between agents and modal operators for common knowledge and distributed knowledge that are not needed in our work. Our knowledge-based logic is a subset of the logic defined in [Pardo et al., 2017] which is sufficient for modeling the knowledge and inference capabilities of the actors as well as providing the possibility of reasoning on the actors' knowledge to cover their inference capabilities.

Definition 3.1 (Knowledge-based Logic Syntax) *Given $i \in ID$, $x \in Var$ and $f \in STerm$, the syntax of the knowledge-based logic $\mathcal{IKBL}$ is defined as:*

$$\varphi ::= \psi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \forall x.\varphi$$
$$\psi ::= K_i\psi \mid f$$

We use the shorthand notation $\forall\{x, y, z\}.\varphi$ to represent $\forall x.\forall y.\forall z.\varphi$. The notation (modal operator) $K_i\psi$ denotes actor i knows ψ, and the Inferactor model satisfies $K_i\psi$ if and only if ψ is in the knowledge base of actor i, or actor i can infer ψ from its knowledge base and its inference rules, using the S5 axiomatisation of epistemic logic. The S5 axiomatisation of epistemic logic ([Fagin et al., 2003] and [Pardo and Schneider, 2014]) is based on the following set of axioms:

- A1. All (instances of) first-order tautologies
- A2. $(K_i\varphi \wedge K_i(\varphi \Rightarrow \psi)) \Rightarrow K_i\psi$ (Distribution Axiom)
- A3. $K_i\varphi \Rightarrow \varphi$ (Knowledge Axiom)
- A4. $K_i\varphi \Rightarrow K_iK_i\varphi$ (Positive Introspection Axiom)
- A5. $\neg K_i\varphi \Rightarrow K_i\neg K_i\varphi$ (Negative Introspection Axiom)

Along with the axioms, S5 includes the following three derivation rules:

- From φ and $\varphi \Rightarrow \psi$, infer ψ (Modus Ponens)
- From φ infer $K_i\varphi$, where φ must be provable from no assumptions (Necessitation)
- From φ infer $\forall x \cdot \varphi(x)$, where x does not occur in φ (Generalization)

We employ $\mathcal{IKBL}$ logic to model actors' knowledge and inference rules. The axiom system of the logic enables us to define the semantics of statements (query and conditional statements in Sect. 5) that deal with the knowledge of the actors. We also use a subset of this logic to define privacy policies (Sect. 7).

As an example of $\mathcal{IKBL}$ logic about how to infer knowledge based on existing knowledge and inference rules, suppose Alice knows that Bob and Charlie are classmates:

$$K_{Alice} classmates(Bob, Charlie)$$

Alice also knows that if two persons are classmates and one of them studies at a university, then the other one also studies at that university:

$$\forall x, y, u \cdot K_{Alice}(classmates(x, y) \wedge university(x, u) \Rightarrow university(y, u))$$

Now, if Alice knows that Bob's university is "U" (i.e., $university(Bob, U)$), she can infer that Charlie's university is also "U" (i.e., $university(Charlie, U)$).

The set of all well-formed $\mathcal{IKBL}$ formulae is called $\mathcal{F}_{\mathcal{IKBL}}$. We will define the semantics of the knowledge-based logic over the Inferactor in terms of a set of satisfaction relations in Sect. 6, after describing the formal syntax and semantics of Inferactor in the following two sections.

4 Formal Description of Inferactor Syntax

In this section, we first present the grammar of Inferactor and then provide an abstract specification of an Inferactor model's syntax. An Inferactor model consists of a number of actor declarations and a *main* block specifying initial messages to the actors.

Note that, like in Actor model, there is no intra-object concurrency in Inferactor and each actor has only one execution thread. To keep the presentation of Inferactor semantics simple, the current version of Inferactor is object-based, although we can easily extend the language to support classes of objects.

4.1 Notation

Here, we review the standard notations used in this paper for working with sequences and functions. The empty sequence is denoted by ϵ. The set of all finite sequences over elements of the set A is denoted by A^*. The ith element of a sequence $a \in A^*$ of length n ($1 \leq i \leq n$), is denoted by a_i, and $\langle a_1, \ldots, a_n \rangle$ is another presentation for the elements of a sequence $a \in A^*$ of length n. For two sequences $\sigma, \sigma' \in A^*$, $\sigma \oplus \sigma'$ denotes the sequence obtained by appending σ' to the end of σ, and $\langle h|T \rangle$ denotes a sequence which $h \in A$ is its first element and $T \in A^*$ consists of the elements in the rest of the sequence. For a function $f : X \to Y$, $f[x \mapsto y]$ denotes the function $\{(a, b) \in f | a \neq x\} \cup \{(x, y)\}$ and $f|_S$ denotes the function $\{x \mapsto f(x) | x \in S\}$ where $S \subseteq X$. $f[\eta]$ denotes the function $\{(a, b) \in f | a \notin dom(\eta)\} \cup \eta$ for two functions $f : X \to Y$ and $\eta : X \to Y$. For two sequences a and b of the same size n (assuming that the elements of a are distinct), the function $map(a, b)$ denotes the mapping of the elements of a into b, formally, $map(a, b) = \{a_i \mapsto b_i | 1 \leq i \leq n\}$.

4.2 Inferactor Syntax

The grammar of Inferactor in EBNF notation is shown in Fig. 2. In an actor's body, first the state variables are declared, then the constructor of the actor (for initializing variables and specifying the initial knowledge of the actor), some methods which serve the messages, and an inference block containing inference rules are defined, respectively.

In this grammar, $expr_k$ denotes boolean expressions defined by the knowledge-based logic $\mathcal{IKBL}$, and $expr$ denotes integer expressions defined over usual arithmetic operators (with no side effects), boolean expressions defined over usual relational and logical operators, or string constants enclosed in double-quotes.

Now, we present the running example modeled based on the above grammar. To keep the state space small, we only model one consumer, and ignore unnecessary interactions and data transmissions. Fig. 3 illustrates the running example. By execution of *main* block, the messages *sendECT* and *reqInfo* are put in the message queues of actors $smartmeter$ and $analyzer$, respectively, and the messages *sendConsumerCity* and *reqAnalysis* are put in the message queue of actor $utility$. The notation '!' denotes sending of a message to an actor. An actor takes a message from its message queue and executes the corresponding method to process it. When actor $smartmeter$ takes message *sendECT*, method *sendECT* (lines 6-8) is executed and message *getECT* is sent to $utility$. This message transmits smart meter identifier and the consumer's e-consumption trace to the utility (we abstract knowing the e-consumption trace data structure by assuming the fact $econs(i)$ which indicates knowing the consumer's e-consumption trace with the smart meter identifier i). When $analyzer$ takes message *reqInfo*, method *reqInfo* (lines

⟨*model*⟩ ::= ⟨*actor*⟩* ⟨*main*⟩

⟨*actor*⟩ ::= ‘actor’ ⟨*actor-id*⟩ ‘{’ ⟨*variables*⟩ ⟨*constructor*⟩ (⟨*method*⟩)* ⟨*infer-rules*⟩ ‘}’

⟨*variables*⟩ ::= (⟨*var-decl*⟩ ‘;’)*

⟨*var-decl*⟩ ::= ⟨*type*⟩ ⟨*var*⟩

⟨*type*⟩ ::= ‘string’ | ‘int’ | ‘bool’

⟨*constructor*⟩ ::= ⟨*actor-id*⟩ ‘{’ (⟨*assignment*⟩ ‘;’| ⟨*remember*⟩ ‘;’)* ‘}’

⟨*method*⟩ ::= ‘def’ ⟨*message*⟩ ‘(’ ⟨*param-list*⟩ ‘)’ ‘{’ ⟨*stat-list*⟩ ‘}’

⟨*infer-rules*⟩ ::= ‘inference’ ‘{’ (⟨*infer-rule*⟩)* ‘}’

⟨*param-list*⟩ ::= ϵ | ⟨*var*⟩ (‘,’ ⟨*var*⟩)*

⟨*stat-list*⟩ ::= (⟨*var-decl*⟩ ‘;’ | ⟨*statement*⟩ ‘;’)*

⟨*statement*⟩ ::= ⟨*assignment*⟩ | ⟨*send*⟩ | ⟨*remember*⟩ | ⟨*query*⟩ | ⟨*conditional*⟩

⟨*assignment*⟩ ::= ⟨*var*⟩ ‘=’ ⟨*expr*⟩

⟨*send*⟩ ::= ⟨*actor-id*⟩ ‘!’ ⟨*message*⟩ ‘(’ ⟨*arg-list*⟩ ‘)’

⟨*arg-list*⟩ ::= ϵ | ⟨*expr*⟩ (‘,’ ⟨*expr*⟩)*

⟨*remember*⟩ ::= ‘remember’ ‘(’ ⟨*fact*⟩ ‘)’

⟨*fact*⟩ ::= (‘K_’‘{’⟨*actor-id*⟩‘}’)* ⟨*fname*⟩ ‘(’ ⟨*arg-list*⟩ ‘)’

⟨*fname*⟩ ::= ⟨*identifier*⟩

⟨*query*⟩ ::= ‘forall’ ‘(’ ⟨*qfact*⟩ ‘)’ ‘{’ ⟨*stat-list*⟩ ‘}’

⟨*qfact*⟩ ::= ‘K_’‘{’⟨*actor-id*⟩‘}’)* ⟨*fname*⟩ ‘(’ ϵ | ((⟨*var*⟩ | ‘?’ ⟨*var*⟩) (‘,’ ⟨*var*⟩ | ‘,’ ‘?’ ⟨*var*⟩)* ‘)’

⟨*conditional*⟩ ::= ‘if’ ‘(’ ⟨*expr_k*⟩ ‘)’ ‘{’ ⟨*stat-list*⟩ ‘}’ (ϵ | ‘else’ ‘{’ ⟨*stat-list*⟩ ‘}’)

⟨*infer-rule*⟩ ::= (⟨*fact*⟩ (‘,’ ⟨*fact*⟩)*) ‘->’ ⟨*fact*⟩ ‘.’

⟨*main*⟩ ::= ‘main’ ‘{’ (⟨*send*⟩ ‘;’)* ‘}’

⟨*expr*⟩ ::= *Expressions over usual (side-effect free) operators.*

⟨*expr_k*⟩ ::= *Boolean expressions over knowledge-based logic* $\mathcal{IKBL}$

⟨*message*⟩ ::= ⟨*identifier*⟩

⟨*actor-id*⟩ ::= ⟨*identifier*⟩

⟨*var*⟩ ::= ⟨*identifier*⟩

⟨*identifier*⟩ ::= *A string of characters, numbers, and symbols*

Figure 2: The grammar of Inferactor Model in EBNF – the detailed syntax for expressions is omitted

29-31) is executed and message *sendInfo* is sent to $consumer$ to request name and smart meter identifier of $consumer$. Actor $consumer$ replies to this message by sending its name and smart meter identifier as the parameters of message *getInfo*.

When $utility$ takes *getECT*, only the received e-consumption trace is saved in its knowledge base ('remember' statement in line 44). By taking *sendConsumerCity*, $utility$ sends the city of $consumer$ to $analyzer$ (lines 53-58), and $analyzer$ saves this fact upon processing the message *inputCity* (lines 26-28). When $utility$ takes *reqAnalysis*, for each of its known consumer's e-consumption trace $econs(i)$, if it also knows the fact $k_{analyzer}city(i)$, then this e-consumption trace is sent to $analyzer$ (lines 46-52). Actor $analyzer$ saves the received information by taking message *inputECT* (lines 23-25).

Due to the concurrent execution of the actors, the order of processing the messages is non-deterministic. For example, after the execution of the main block, the actors $smartmeter$, $utility$ and $analyzer$ have messages in their message queues and the order in which these actors execute and process their messages is non-deterministic. As another example, the order in which *inputCity* and *getInfo* are received by $analyzer$ is non-deterministic and depends on the order in which the previous messages were sent.

We assume if $analyzer$ knows e-consumption trace of a smart meter, then it can infer the smart appliances related to that smart meter. This inference capability of $analyzer$ is modeled by the first inference rule (line 36) in the inference block of $analyzer$. We also assume if $analyzer$ knows the smart appliances related to a smart meter and the name of the owner of that smart meter, then it can infer the economic status of the owner (consumer). This inference capability of $analyzer$ is modeled by the second inference rule (lines 37-38) in the inference block of $analyzer$.

The economic status of a consumer is personal information, and we assume $analyzer$ is not allowed to know the economic status of $consumer$. So, one of the knowledge-related policies defined for the running example is "$analyzer$ is not allowed to know the economic status of $consumer$". We also define another policy "$utility$ is not allowed to know the name of the $consumer$ that is the owner of a $smartmeter$".

4.3 Abstract Syntax

In this section, we present an abstract specification of an Inferactor model's syntax.

4.3.1 Types of Variables

The variables in our model are typed variables, including *Int*, *Bool*, and *String* variables (as shown in the grammar of Inferactor). We assume the set *Var* contains all variables. By this assumption, the names of all variables, including state variables and local variables, in different actors must be unique. We can simply handle this problem by prefixing a variable name with the name of the actor and the name of the method this variable belongs to (the actor name for state variables and the actor name along with the method name for local variables of the methods). We assume the set *Val* contains all possible values that can be assigned to the variables or to be used within the expressions (i.e., $Val = \mathbb{Z} \cup \{\text{True}, \text{False}\} \cup \{\textit{The finite sequences of characters}\}$).

For simplicity, in the definition of the semantics of Inferactor, we ignore the type of variables and assume that the type of a variable is specified when that variable is declared and the type checking is done in static semantics.

```
1  actor smartmeter{
2   int id;
3   smartmeter{
4    id = 100;
5   }
6   def sendECT(){
7    utility!getECT(id);
8   }
9  }
10 actor consumer{
11  string name;
12  int meterid;
13  consumer{
14   meterid = 100;
15   name = "C1";
16  }
17  def sendInfo(){
18   analyzer!getInfo(
19     meterid,name);}
20 }
21 actor analyzer{
22  analyzer{}
23  def inputECT(x){
24   remember (econs(x));
25  }
26  def inputCity(i){
27   remember (city(i));
28  }
29  def reqInfo(){
30   consumer!sendInfo();
31  }
32  def getInfo(i,n){
33   remember (name(i,n));
34  }
35  inference{
36   econs(x) -> applst(x).
37   applst(x) , name(x,y)
38     -> economicstatus(y).
39  }
40 }
41 actor utility{
42  utility{}
43  def getECT(i){
44   remember (econs(i));
45  }
46  def reqAnalysis(){
47   forall (econs(?i)){
48    if (k_{analyzer} city(i)){
49       analyzer!inputECT(i);
50    }
51   }
52  }
53  def sendConsumerCity(){
54   analyzer!inputCity(100);
55   remember
56    (k_{analyzer} city(100));
57  }
58 }
59 main{
60  smartmeter!sendECT();
61  analyzer!reqInfo();
62  utility!sendConsumerCity();
63  utility!reqAnalysis();
64 }
```

Figure 3: The running example modeled in Inferactor

4.3.2 Term and Fact

As described earlier, we use the concept of term to model the information in Inferactor model. A term is a piece of information of an actor, and *STerm* denotes the set of all terms in the model. Each term consists of a term name and a sequence of its arguments. We define a term as a pair $(tn, \alpha) \in \mathit{DTName} \times \mathit{Expr}^*$, where tn is the term name and α is the sequence of its arguments. *Expr* denotes the set of integer-valued expressions defined over usual arithmetic operators (with no side effects), boolean expressions defined over usual relational and logical operators, or string constants enclosed in double-quotes.

A term that all its arguments have specified values is called fact. Facts are used to model the knowledge of the actors. A fact is defined as the pair $(tn, \alpha) \in \mathit{DTName} \times \mathit{Val}^*$, and the set of all facts in the model is denoted as *SFact*. For example, 'Alice is 30 years old' is a fact and we use the notation $\mathit{age}(\text{“}\mathit{Alice}\text{”}, 30)$ to denote $(\mathit{age}, \langle \text{“}\mathit{Alice}\text{”}, 30\rangle) \in \mathit{SFact}$. More generally, $f(v_1, ..., v_n)$ denotes $(f, \langle v_1, ..., v_n\rangle) \in \mathit{SFact}$.

In addition to this type of information, we want to keep the knowledge of an actor about the knowledge of the other actors. To model this type of information, we introduce 'compound term' and 'compound fact'. The set of all compound terms and the set of all compound facts in the model are denoted as *CTerm* and *CFact*, respectively. So, the set of all terms in the model is denoted as *Term*, and $\mathit{Term} = \mathit{STerm} \cup \mathit{CTerm}$, and the set of all facts in the model is denoted as *Fact*, and $\mathit{Fact} = \mathit{SFact} \cup \mathit{CFact}$.

A compound term is defined as a pair $(a, t) \in \mathit{ID}^* \times \mathit{STerm}$, in which a is a sequence of actor IDs and t is a term. Following the common notation used in epistemic logic, we use the notation $k_{a_1} k_{a_2} ... k_{a_n} t$ as an alternative to $(\langle a_1, a_2, ..., a_n \rangle, t)$. A compound term that all its arguments have specified values is called a compound fact, which is defined as a pair $(a, f) \in \mathit{ID}^* \times \mathit{SFact}$. For example, 'Bob knows Alice is 30 years old' is a compound fact, and we use the notation $k_{Bob}age(\text{“}Alice\text{”}, 30)$ to denote it.

In the rest of this paper, the word 'fact' refers to both types of fact. We refer to a term (a member of *Term*) using the notation $\overline{k}tn(\alpha)$, in which $\overline{k}$ is empty (for the members of *STerm*) or a sequence of k_{a_i} operators (for the members of *CTerm*), and $tn(\alpha) \in \mathit{STerm}$.

4.3.3 Inference Rules

An actor can infer a new fact based on its current known facts and its inference rules. For example, if one knows the working hours of an employee in a project and the hourly basis for her, then she can infer the salary of that employee in that project. We write this inference rule as:

$$workinghours(id, h), userhourlybasis(id, j) \to salary(id, h \times j).$$

Each inference rule is defined as the tuple $(lterms, rterm) \in 2^{\mathit{Term}} \times \mathit{Term}$, where $lterms$ contains the premised terms, and $rterm$ is the conclusion and must be a single term. We can also abstract the computation in inference rules, and rewrite the above inference rule as:

$$workinghours(id), userhourlybasis(id) \to salary(id).$$

The inference rules written in the body of the actors are translated to logic formulas. The inference rules are written in the body of the actors by the following syntax:

$$\overline{k}t_1(a_1, ..., a_m), ..., \overline{k}t_n(a'_1, ..., a'_p) \to \overline{k}t(a''_1, ..., a''_q)$$

We assume a set *TArgs* which contains the set of all arguments of the premised terms in the above inference rule, i.e., $\mathit{TArgs} = \{a_1, ..., a_m, ..., a'_1, ..., a'_p\}$. Since all the variables in the conclusion also appear in the premised terms, we define *TArgs* as the set of all arguments of the premised terms. Our goal is to analyze disclosure of the actors information due to the message passing among them, and there is no global knowledge in the model, so each actor can infer based on its own local knowledge. The above inference rule, written in the body of an actor $i \in \mathit{ID}$, is translated to $\mathcal{IKBL}$ formula as:

$$\forall \{v \in \mathit{TArgs} \cap \mathit{Var}\}.\overline{k}t_1(a_1, ..., a_m) \wedge ... \wedge \overline{k}t_n(a'_1, ..., a'_p) \Rightarrow \overline{k}t(a''_1, ..., a''_q)$$

For example, the inference rule defined for salary of an employee is translated to:

$$\forall id.workinghours(id) \wedge userhourlybasis(id) \Rightarrow salary(id)$$

Linking the available information from multiple data sources, based on their common information, can identify individuals and disclose sensitive information [Samani, 2015]. We can model this type of information disclosure (or linking attack) by defining suitable inference rules. For instance, we suppose an example, stated in [Halvorsen et al., 2022], about information disclosure associated with publishing COVID-19 infection rates in Denmark. In this example, a medical record contains 5 field: name, zip code, birthday, sex, and diagnosis. The medical record (called *MRec*) can be sent to others after anonymisation

by dropping the name field (called *AMRec*). It is assumed that users have not consented to their diagnosis being disclosed. There is also a public knowledge that can be available to everyone, which is represented by a public record (called *PRec*) and contains 4 fields: name, zip codes, birthdays, and sex. These records are modeled as follow:

$$MRec(name, zipcode, birthday, sex, diagnosis)$$
$$AMRec(zipcode, birthday, sex, diagnosis)\ PRec(name, zipcodes, birthdays, sex)$$

By joining the information of anonymized medical record (*AMRec*) and public record (*PRec*), the name of the person to whom the medical record belongs is revealed, and as a result, it is determined whether that person is ill or not. We model this linking attack by defining an inference rule:

$$AMRec(z, b, s, d) \wedge PRec(n, z, b, s) \Rightarrow MRec(n, z, b, s, d)$$

We assume the set *IRule* is the set of all inference rules in the model.

4.3.4 Methods

Each method is defined as the tuple $(m, p, lv, b) \in MName \times Var^* \times Var^* \times Stat^*$, where m is the name of the message the method is used to serve, p is the sequence of the names of the formal parameters, lv is the set of local variables defined in the scope of the method, b contains the sequence of statements comprising the body of the method, and *MName* is the set of all method names. The set *Mtd* subsumes all method declarations in the model.

4.3.5 Actors

Each actor is an instance of the type $Actor = ID \times 2^{Var} \times 2^{Mtd} \times 2^{IRule}$. An actor $(id, vars, mtds, inferrules)$ has the identifier *id*, the set of variables *vars*, the set of methods *mtds*, and the set of inference rules *inferrules*. The set *vars* contains the state variables (defined at the beginning of the actor's body) and the local variables (defined in the actor's methods) of the actor.

4.3.6 Statements

We first define the statements that are common in actor models, like assignment, and message sending statements. The assignment and send statements are defined as bellow:

- $Assign = Var \times Expr$ is the set of assignment statements. We use the notation $var := expr$ as an alternative to $(var, expr)$.
- $Send = ID \times MName \times Expr^*$ is the set of send statements. We use the notation $x!m(v)$ as an alternative to (x, m, v).

In addition to the above statements, we define three other statements to deal with the knowledge of the actors in Inferactor. These statements, which are used to save a fact in the knowledge base of an actor, query the parameters of a fact from the knowledge of an actor, and choose different path of execution based on the knowledge of an actor, are defined as bellow:

- $Save$ statement. The actors in Inferactor can save a fact in their knowledge-base. For example, an actor can save the fact 'Alice is 30 years old' by execution of the statement remember $(age(\text{“}Alice\text{”}, 30))$. Formally, *Save* = *Term* is the set of statements which save a fact in the knowledge base of the actor. We use the notation remember $(\overline{k}tn(\alpha))$ for save statement which adds the fact $\overline{k}tn(\alpha)$ to the knowledge base of the actor.

- *Query* statement. The actors in Inferactor can query the arguments of a fact from their knowledge bases. For example, an actor can query the age of Alice by the statement forall $(age(\text{“}Alice\text{”}, ?x))$ which assigns the age of Alice to variable x. Formally, *Query* = *Term* × *Stat** is the set of query statements. The fact that is the query statement parameter is called the input fact, and a variable in the arguments of the input fact whose value is set by the query statement, is called output variable and marked with '?' symbol before its name. We use the notation forall $(\overline{k}tn(\alpha))\ \sigma$ as an alternative to $(\overline{k}tn(\alpha), \sigma)$. This statement assigns some values to the arguments which are prefixed by ? (output variables), based on the knowledge of the actor. As the number of query results can be different (zero, one, or more), we use a loop structure for handling a query. In each iteration of this loop, the corresponding values are assigned to the output variables based on one of the possible query results. For example, an actor can query those who are 30 years old by the statement forall $(age(?x, 30))$, and in each iteration, the name of one who is 30 years old is assigned to variable x.

- $Conditional$ statement. The actors in Inferactor can select different path of execution based on the facts they know. For example, actor A is responsible for collecting data in a research. When A receives a data item (d), if the sender (actor B) has already registered, this data will be sent to the researcher (actor C). Otherwise, the sender is first asked to register. This scenario can be modeled by if $(registered(B))$ $\{C!newdata(d);\}$ else $\{B!registerationreq();\}$. Formally, $Cond = Expr_k \times Stat^* \times Stat^*$ is the set of conditional statements, where $Expr_k$ denotes the set of boolean expressions defined over the knowledge-based logic $\mathcal{IKBL}$. We use the notation if $(expr_k)\ \sigma$ else σ' as an alternative to $(expr_k, \sigma, \sigma')$, which determines whether $expr_k$ holds based on the knowledge of the actor this statement belongs to, and based on that, one of the two sequences of statements σ or σ' is executed.

We also define another statement, called $endm$, which is implicitly added to the end of each method. This statement executes after the last statement of the method, and removes the formal parameters and the local variables added by this method, from the set of variables of the actor this method belongs to. So, the set of statements in Inferactor is defined as $Stat = Assign \cup Send \cup Save \cup Query \cup Cond \cup \{endm\}$.

4.3.7 Inferactor Model

The main block is specified by $Send^*$, and consists of a sequence of message send statements. Note that since there may be more than one message to the same actor, the send statements are ordered in a sequence and not just a set of statements.

Having the above definitions, the set of Inferactor models is specified by $2^{Actor} \times Send^*$, where the second component corresponds to the main block. The set *Inferactor* is the set of all Inferactor models.

4.3.8 Auxiliary Functions

We define the following auxiliary functions to be used in defining the formal semantics:

- $body : ID \times MName \to Stat^*$, where $body(x, m)$ returns the body of the method m of the actor identified by x, appended by the special statement $endm$, which denotes the end of the method.
- $params : ID \times MName \to Var^*$, where $params(x, m)$ returns the list of formal parameters of the method m of the actor identified by x.
- $lvars : ID \times MName \to 2^{Var}$, where $lvars(x, m)$ returns the names of the local variables of the method m of the actor identified by x.
- $svars : ID \to 2^{Var}$ where $svars(x)$ returns the names of the state variables of the actor identified by x.
- $init : 2^{Var} \to (Var \to \{0, False, \epsilon\})$, where $init(vars)$ is a function mapping the variable names in the input set $vars$ to their initial values (zero, false, or empty string) depending on their types.
- $constr_v : ID \to (Var \to Val)$, where $constr_v(x)$ updates $init(svars(x))$ based on the assignment statements of the constructor of actor x.
- $constr_k : ID \to 2^{Fact}$, where $constr_k(x)$ returns the set of initial knowledge of actor x based on the remember statements of its constructor.
- $q_0 : ID \times Send^* \to Msg^*$, where $q_0(x, \sigma)$ returns the message queue for actor identified by x, results from the sequence of send statements σ. This function is used to construct the initial message queues of the actors from the sequence of send statements in the main block of an Inferactor.
- $irules : ID \to 2^{IRule}$, where $irules(x)$ returns the set of inference rules of the actor identified by x.
- $kn : ID \to 2^{Fact}$, where $kn(x)$ returns the set of facts are in the knowledge-base of the actor identified by x.

4.3.9 Static Semantics

The following rules define the well-formedness of Inferactor model which is hard to (or cannot be) described in the Inferactor grammar, but must be statically checked. Some of these rules are similar to the usual well-formedness rules in such formal models (like in [Khamespanah, 2018]), and others are specially defined for Inferactor. Note that these rules can be formally defined in abstract syntax, but as the formal definition of these rules is not in line with the goal of this paper, we only provide the informal definitions of them.

- Unique Identifiers. The actor identifiers are unique within an Inferactor model.
- Unique State Variables. The names of the state variables of an actor are unique within all variables used in that actor, including state variables, message parameters, local variables, and variables in the arguments of the terms in inference block.

– Unique Methods. The names of the methods of an actor are unique within all method names in that actor.

– Unique Parameters and Local Variables. The names of the formal parameters and the local variables of a method are unique within that method, and different from the state variable names of the enclosing actor.

– Well-Typed Receiver. The receiver of a message has a method with the same name as the message.

– Well-Formed Arguments. Well-formedness is defined for arguments of messages and terms:

  - The list of actual arguments passed to a message send statement conforms to the list of formal parameters of the corresponding method, in both length and type.

  - The list of arguments of a term in a query statement conforms to the list of the arguments of the corresponding facts saved in the knowledge base of the actor.

– Limitations of Main Block. The arguments of the send statements in the main block can only be constant expressions.

– Limitation of Inference Block. There is no need to declare the variables in the arguments of the terms in inference block. The names of the variables in inference block of an actor are different from the state variable names of the enclosing actor.

– Well-Typed Assignments. The assignments are well-typed, i.e., two sides of an assignment statement have to be of the same type.

– Well-Typed Expressions. The expressions are well-typed, i.e., all variables and constants used in an expression have to be of the same type.

5 Formal Description of Inferactor Semantics

The goal of this paper is to find potential unauthorized accesses to information. We check statically whether an Inferactor model will possibly run into a state where information is not accessed based on specified policies. Therefore, we present the operational semantics of Inferactor, that describes how the model actually runs dynamically, so that we can statically check such a situation using our proposed analysis solution. In this section, we describe the formal semantics of Inferactor models in terms of transition systems. Before that, we make a few definitions and assumptions.

In order to convert the facts stored in the actors' knowledge base into the knowledge defined by $\mathcal{IKBL}$, we define a function which converts $(\langle a_1, a_2, ..., a_n\rangle, t)$ to $K_{a_1} K_{a_2} ... K_{a_n} t$.

As the main focus is on the message passing and interleavings of actors' execution, we abstract away the semantics of expressions by assuming the function $\mathit{eval}_v : \mathit{Expr} \rightarrow \mathit{Val}$ evaluates an expression within a specific context $v : \mathit{Var} \rightarrow \mathit{Val}$. We assume eval_v is overloaded to evaluate a sequence of expressions: $\mathit{eval}_v(\langle e_1, e_2, \ldots, e_n\rangle) = \langle \mathit{eval}_v(e_1), \mathit{eval}_v(e_2), \ldots, \mathit{eval}_v(e_n)\rangle$.

We also define another function $\mathit{eval}_x : \mathit{Expr}_k \rightarrow \{\mathit{True}, \mathit{False}\}$ to evaluate a knowledge-based logic expression based on the knowledge of actor x (its knowledge base and its inference rules) under the axiom system S5.

As defined earlier, the actors can execute a query (*Query* statement) on their knowledge. The type of a query result is $Var \rightarrow Val$, which is a mapping of values to the output variables. We abstract the execution of the query by assuming a function $qresult_x : Term \rightarrow (Var \rightarrow Val)^*$ which returns the sequence of all query results according to the knowledge of actor x under the axiom system S5. We use the notation $qresult_x(\overline{k}tn(\alpha))$ for computing the set of all results for $\overline{k}tn(\alpha)$ according to the knowledge of actor x.

5.1 States

We assume actors communicate via message passing and queue their incoming messages in a FIFO mailbox. We define the type for the messages as $Msg = MName \times (Var \rightarrow Val)$. In a message $(m, a) \in Msg$, m is the name of the message and a is a function mapping argument names to their values. The mailbox of an actor is defined as a sequence of messages, written as Msg^*. The actor's knowledge base is represented by a set of facts defined in Sect. 4.3.

The global state of an Inferactor system is represented by a function $s : ID \rightarrow (Var \rightarrow Val) \times Msg^* \times Stat^* \times 2^{Fact}$, which maps an actor's identifier to the local state of the actor. The set of all global states is called *State*. The local state of an actor is defined by a tuple like (v, q, σ, κ), where $v : Var \rightarrow Val$ gives the values of the state variables and local variables of the actor, $q : Msg^*$ is the mailbox of the actor, $\sigma : Stat^*$ contains the sequence of statements the actor is going to execute to finish the service to the message currently being processed, and $\kappa : 2^{Fact}$ is the knowledge base of the actor.

The knowledge of an actor includes both the facts saved in its knowledge base (κ), and the facts which can be inferred by inference rules ($irules(x)$). Note that, the inferred facts are not saved in the knowledge base.

5.2 Transitions

The transitions between the states occur as the results of actors' actions including: taking a message from the mailbox, executing a statement, and ending the execution of a method. The set of all transitions is defined as $Tran : State \times Label \times State$ where $Label : (ID \times Mtd) \cup \{\tau\}$. The transitions for taking a message are labeled by members of $ID \times Mtd$ and the other transitions are labeled with τ. We use the notation $s \xrightarrow{l} s'$ to denote $(s, l, s') \in Tran$. The SOS rules for the transitions are defined as follow.

5.2.1 Message take

The actors take one message, execute the corresponding method to the end, and then take another message. By taking message m by actor x, the formal parameters of message m and the local variables of method m are added to the variables of actor x. The label for taking message m by actor x is defined by the notation $x : m$ which denotes (x, m).

$$\frac{s(x) = (v, \langle (m, a) | T \rangle, \epsilon, \kappa)}{s \xrightarrow{x:m} s[x \mapsto (v \cup a \cup init(lvars(x, m)), T, body(x, m), \kappa)]} \quad \text{(message take)}$$

5.2.2 Assignment

When an actor executes an assignment statement $\mathit{var} := \mathit{expr}$, the value of variable *var* is updated by the output of function $\mathit{eval}_v(\mathit{expr})$.

$$\frac{s(x) = (v, q, \langle \mathit{var} := \mathit{expr} | \sigma\rangle, \kappa)}{s \xrightarrow{\tau} s[x \mapsto (v[\mathit{var} \mapsto \mathit{eval}_v(\mathit{expr})], q, \sigma, \kappa)]} \quad \text{(assignment)}$$

5.2.3 Message send

When the actor x executes the send statement $y!m(e)$, message m is appended to the message queue of actor y and the formal parameters of the method m of actor y are set by the output of function $\mathit{eval}_v(e)$.

$$\frac{s(x) = (v, q, \langle y!m(e) | \sigma\rangle, \kappa) \wedge s(y) = (v', q', \sigma', \kappa') \wedge p = \mathit{params}(y, m)}{s \xrightarrow{\tau} s[x \mapsto (v, q, \sigma, \kappa)][y \mapsto (v', q' \oplus \langle (m, \mathit{map}(p, \mathit{eval}_v(e)))\rangle, \sigma', \kappa')]} \quad \text{(send)}$$

5.2.4 End of method

When the actor x executes the endm statement, the formal parameters and the local variables of the method which were previously added to the variables of x, are removed. We define this action by the projection of variables of x on its state variables ($v|_{\mathit{svars}(x)}$ denotes this projection).

$$\frac{s(x) = (v, q, \langle \mathsf{endm}\rangle, \kappa)}{s \xrightarrow{\tau} s[x \mapsto (v|_{\mathit{svars}(x)}, q, \epsilon, \kappa)]} \quad \text{(end-of-method)}$$

5.2.5 Save

By execution of remember $\overline{k}tn(\alpha)$, the fact $\overline{k}tn(\alpha)$ is added to the knowledge base of the actor this statement belongs to.

$$\frac{s(x) = (v, q, \langle \mathsf{remember}\ \overline{k}tn(\alpha) | \sigma\rangle, \kappa)}{s \xrightarrow{\tau} s[x \mapsto (v, q, \sigma, \kappa \cup \{\overline{k}tn(\mathit{eval}_v(\alpha))\})]} \quad \text{(save)}$$

5.2.6 Query

By executing the query statement forall $\overline{k}tn(\alpha)$ do σ' by actor x, first the sequence of all results (sequence γ) is computed by function $\mathit{qresult}_x(\overline{k}tn(\alpha))$ based on the knowledge of actor x. Then, for each result in this sequence, one iteration of statements included in query statement is executed. We specify these iterations by defining a function $\mathit{qloop} : ((\mathit{Var} \to \mathit{Val})^* \times \mathit{Stat}^*) \to ((\mathit{Var} \to \mathit{Val}) \times \mathit{Stat}^*)$, which iterates until the γ sequence is empty.

$$\frac{s(x) = (v, q, \langle \mathsf{forall}\ \overline{k}tn(\alpha)\ \mathsf{do}\ \sigma' | \sigma\rangle, \kappa) \wedge \gamma = \mathit{qresult}_x(\overline{k}tn(\alpha))}{s \xrightarrow{\tau} s[x \mapsto (v, q, \langle \mathit{qloop}(\gamma, \sigma') | \sigma\rangle, \kappa)]} \quad \text{(query)}$$

$$\frac{s(x) = (v, q, \langle qloop(\langle \eta | T \rangle, \sigma') | \sigma \rangle, \kappa)}{s \xrightarrow{\tau} s[x \mapsto (v[\eta], q, \sigma' \oplus \langle qloop(T, \sigma') \rangle \oplus \sigma, \kappa)]} \quad \text{(qloop-iterate)}$$

$$\frac{s(x) = (v, q, \langle qloop(\epsilon, \sigma') | \sigma \rangle, \kappa)}{s \xrightarrow{\tau} s[x \mapsto (v, q, \sigma, \kappa)]} \quad \text{(qloop-end)}$$

5.2.7 Conditional

By executing conditional statement if $expr_k$ then σ else σ' by actor x, the knowledge-based logic expression $expr_k$ is evaluated in the context of the knowledge of actor x. If $expr_k$ is evaluated to true, then the sequence of statements σ is executed, and otherwise, the sequence of statements σ' is executed.

$$\frac{s(x) = (v, q, \langle \text{if } expr_k \text{ then } \sigma \text{ else } \sigma' | \sigma'' \rangle, \kappa) \wedge eval_x(expr_k) = \text{True}}{s \xrightarrow{\tau} s[x \mapsto (v, q, \sigma \oplus \sigma'', \kappa)]} \quad \text{(conditional}_T\text{)}$$

$$\frac{s(x) = (v, q, \langle \text{if } expr_k \text{ then } \sigma \text{ else } \sigma' | \sigma'' \rangle, \kappa) \wedge eval_x(expr_k) = \text{False}}{s \xrightarrow{\tau} s[x \mapsto (v, q, \sigma' \oplus \sigma'', \kappa)]} \quad \text{(conditional}_F\text{)}$$

5.3 Transition System

The transition system semantics for an Inferactor model *IM* is defined as $TS(IM) = (State, \rightarrow, Label, s_0)$, where:

- *State* is the set of global states defined in previous subsection,
- $\rightarrow$ is the transition relation *Tran* defined in previous subsection,
- *Label* is the set of transition labels defined in the previous subsection, and
- s_0 is the initial state.

In the initial state, the state variables of all actors have their initial values (depending on their types), the knowledge bases of the actors are empty sets, and the message queues of the actors contain the messages specified in the main block. So, the initial state for the actor x is defined as $s_0(x) = (constr_v(x), q_0(x, \sigma), \epsilon, constr_k(x))$, where σ is the sequence of send statements specified in the main block.

The transition system semantics we have defined so far, is the small-step transition system semantics. Another type of transition system semantics, which is also used in actor-based languages like Rebeca [Sirjani et al., 2004] and Ptolemy [Eker et al., 2003], is big-step transition system semantics. In big-step semantics, it is assumed that when an actor is in the middle of processing a message, the other actors are not doing anything. In this case the execution of the methods are non-preemptive, i.e., when an actor takes a message, it executes the entire body of the corresponding method before starting execution of another method.

In Inferactor, as the knowledge of the actors can only be increased, there is no possibility for a knowledge-related policy to be violated in the middle of the execution of

a method, but be satisfied at the end of the execution of that method, so switching from the small-step to the big-step semantics has no effect on the evaluation of the policies. Since we want to keep the state space small, we define the big-step transition system semantics for Inferactor. In this case, many of the unnecessary interleavings that were created in small-step semantics, are removed and the state space becomes significantly smaller.

5.3.1 Big-step Semantics

The big-step transition system semantics for an Inferactor model IM is defined as $T(IM) = (State, \hookrightarrow, Label, s_0)$. In this definition, *State*, *Label* and s_0 are the same as in the small-step semantics. To define the big-step transition relation $\hookrightarrow$, we act in the same way as the authors in [Khamespanah, 2018] used to define the big-step semantics of Timed Rebeca. An actor is idle, if it is not in the middle of processing of a message, and a global state s is idle, if all the actors are idle in s. An idle state is defined as:

$$idle(s) \iff \forall x \in ID \cdot s(x) = (v, q, \epsilon, \kappa)$$

As in [Khamespanah, 2018], we use the notation $idle(s, x)$ and $idle(s)$ to denote the actor identified by x is idle in state s, and the state s is idle, respectively. Note that the outgoing transitions from an idle state can only be the "message take" transitions.

The transition relation $\hookrightarrow \subset State \times Label \times State$, which occurs between two idle state, specifies an actor takes a message from its message queue and executes the corresponding method completely to the end. In this case, the state of the system is updated at the end of the execution of a method. Two global states s and s' are in relation $\hookrightarrow$ iff both of them are idle, and there is a path between s and s' in $TS(IM)$ such that the first transition on the path is taking a message m by an actor x and all other transitions in this path are related to the execution of the statements of method m, and so all other actors are idle throughout this path. Formally, $s \overset{x:m}{\hookrightarrow} s'$ iff

- $idle(s) \wedge idle(s')$, and
- $\exists s_1, s_2, \ldots s_k \in State, x \in ID \cdot s = s_1 \overset{x:m}{\longrightarrow} s_2 \rightarrow \ldots \rightarrow s_k = s' \wedge (\forall n, 1 < n < k \cdot \neg idle(s_n, x)) \wedge (\forall y \in ID \setminus \{x\}, 1 \leq j \leq k \cdot idle(s_j, y))$

It is notable that in big step semantics, there is no need to keep the local variables of the actor's methods in the local state of the actor. Therefore, in big step semantics, the variable part of the local state of the actor (i.e., function v in (v, q, σ, κ)) only contains the state variables of the actor.

6 Knowledge-based Logic Semantics over Inferactor

The syntax of the knowledge-based logic $\mathcal{IKBL}$ has been defined in Sect. 3, and the Inferactor syntax and semantics have been defined in Sect. 4 and Sect. 5, respectively. Now, we define the semantics of knowledge-based logic $\mathcal{IKBL}$. Since there is no global knowledge in Inferactor model and the inference rules are defined and applied locally to an actor's knowledge base, we define the semantics of knowledge-based logic $\mathcal{IKBL}$ over an actor in Inferactor model.

Definition 6.1 (Knowledge-based Logic Semantics) *For a given Inferactor model IM, a state $s \in T(IM)$, actors $i, j \in ID$, a logic formulae $\varphi \in \mathcal{F}_{\mathcal{IKBL}}$, a fact $f \in STerm$ and $x \in Var$, the satisfaction of the logic formulae φ by the actor's state $s(i)$ is defined as:*

$$
\begin{aligned}
s(i) \models f &\iff (irules(i), kn(i)) \vdash f \\
s(i) \models K_j\varphi &\iff (irules(i), kn(i)) \vdash K_j\varphi \\
s(i) \models \varphi_1 \wedge \varphi_2 &\iff s(i) \models \varphi_1 \wedge s(i) \models \varphi_2 \\
s(i) \models \neg\varphi &\iff s(i) \not\models \varphi \\
s(i) \models \forall x.\varphi &\iff \forall v \in Val.\ \varphi[v \mapsto x] \in Fact \Rightarrow s(i) \models \varphi[v \mapsto x]
\end{aligned}
$$

The knowledge of an actor includes the facts in its knowledge base and the facts which are derived from the knowledge base by applying its inference rules, using the axiom system S5. Given the set of facts $\eta \in 2^{Fact}$ and a set of inference rules $\nu \in 2^{IRule}$, we write $(\nu, \eta) \vdash \varphi$ to denote φ can be derived from ν and η under the axiom system S5.

In [Pardo et al., 2017], two assumptions, called knowledge consistency and self-awareness are defined for the knowledge base of the agents. We assume these assumptions for the actors' knowledge. Knowledge consistency means that, φ and $\neg\varphi$ cannot be derivable from an actor knowledge. Knowledge consistency is held in Inferactor, because the actors cannot save the negation of a fact in their knowledge bases, and the inference rules do not allow the actors to infer the negation of the facts. Self-awareness means the actors aware of their knowledge, i.e., if φ is derivable from the knowledge of an actor, $K_i\varphi$ is also derivable ($(irules(i), kn(i)) \vdash \varphi \Rightarrow (irules(i), kn(i)) \vdash K_i\varphi$). Self-awareness is held in Inferactor, based on the second derivation rule presented in Sect. 3.2.

7 Knowledge-Related Policies

Privacy policies help users understand how companies collect, use and share their data [Reidenberg et al., 2015]. Based on [He and Anton, 2003], "*Capturing and modeling privacy requirements in the early stages of system development is essential to provide high assurance of privacy protection to both stakeholders and consumers*". Privacy policies may be defined based on the privacy preferences of data owners or based on the constraints defined according to laws, regulations, and best practices adopted by organizations to handle personal data [Masellis et al., 2015]. After the privacy policies are specified based on the privacy requirements in the system, they are modeled using a privacy policy specification language. We present our policy specification language in this section.

Since we intend to check the satisfaction of the policies in the system, we assumed the policies as system properties, and by providing formal semantics for them, on the one hand, we have prevented ambiguity in the definition of policies, and on the other hand, we have provided the possibility of using formal verification techniques to check the satisfaction of them in the model.

Knowledge-related policies are used for reasoning about the knowledge of the actors. In fact, knowledge-related policies define restrictions on the actors' knowledge, such as who is not allowed to know certain information about another actor, the conditions under which they are allowed or not, and so on. For example, the policies defined for the

running example are "$analyzer$ is not allowed to know economic status of $consumer$", and "$utility$ is not allowed to know the name of the $consumer$ that is the owner of a $smartmeter$". We can also define more complex policies for the running example such as "$analyzer$ is not allowed to know both of appliances list of a $smartmeter$ and the name of the $consumer$ that is the owner of that $smartmeter$ in the same time". Privacy policies are often described in natural language. To enable formal verification of these policies, they need to be expressed explicitly and unambiguously [Tokas and Owe, 2020]. There have been some earlier approaches in the literature to incorporate epistemic logic to specify security and privacy policies, such as in [Pardo and Schneider, 2014], [Pardo et al., 2017] and [Moezkarimi et al., 2022]. We also use epistemic logic to specify our policies, and define a subset of it that is suitable for specifying our knowledge-related policies. The formal syntax and semantics of our knowledge-related policies are defined in this section.

7.1 Syntax of Knowledge-Related Policies

Privacy policies are defined for actors' knowledge, so facts alone are not used in describing policies, and a fact must be considered within the scope of an actor's knowledge. We therefore define a subset of $\mathcal{IKBL}$ logic to describe policies in which the possibility of defining fact alone is excluded from $\mathcal{IKBL}$. The syntax of the knowledge-related policies, based on the knowledge-based logic $\mathcal{IKBL}$, defined for Inferactor is presented in Def. 7.1. To specify actors' information, in knowledge-related policies, irrespective to the values of their arguments, we allow facts with the symbol '_' in their arguments. In this case, we mean the fact with any value for the symbols of '_' in its arguments and define $Fact_{_} = DTName \times (Val \cup Var \cup \{_\})^*$.

Definition 7.1 (Knowledge-Related Policy Syntax) *Given $i \in ID$, and $f \in Fact_{_}$, the syntax of knowledge-related policies is defined as:*

$$\Phi ::= \pi \mid \Phi \wedge \Phi \mid \neg\Phi \mid \forall x.\Phi$$
$$\pi ::= K_i\pi \mid K_i f$$

The set of all knowledge-related policies is denoted by *KnPolicy*. It is notable that policies in form of $\Phi_1 \Rightarrow \neg\Phi_2$ are also definable by our policy language (we can write this type of policies as $\neg(\Phi_1 \wedge \Phi_2)$).

Based on the defined syntax for knowledge-related policies, the defined policies for the running example are specified as $\neg K_{analyzer} economicstatus(\text{"C1"})$ (analyzer is not allowed to know economic status of consumer whose name is "$C1$"), $\neg K_{utility} name(100, _)$ (utility is not allowed to know the name of the consumer that is the owner of the smartmeter with the meterid of 100), and $\forall m.\neg(K_{analyzer} applst(m) \wedge K_{analyzer} name(m, _))$ (analyzer is not allowed to know both of appliances list of a smartmeter and the name of the consumer that is the owner of that smartmeter in the same time).

7.2 Semantics of Knowledge-Related Policies

The formal semantics of knowledge-related policies in terms of satisfaction relations over Inferactor is presented in Def. 7.2.

Definition 7.2 (Knowledge-Related Policy Semantics) *For a given actor $i \in ID$ in Inferactor model IM, the state $s \in T(IM)$, the fact $f \in Fact_$, and a knowledge-related policy $\varphi \in KnPolicy$, the satisfaction relation $\models \subseteq State \times KnPolicy$ is defined as:*

$$\begin{aligned} s \models K_i f &\iff (\mathit{irules}(i), kn(i)) \vdash f \\ s \models K_i \pi &\iff (\mathit{irules}(i), kn(i)) \vdash \pi \\ s \models \Phi_1 \wedge \Phi_2 &\iff s \models \Phi_1 \wedge s \models \Phi_2 \\ s \models \neg\Phi &\iff s \not\models \neg\Phi \\ s \models \forall x.\Phi &\iff \forall v \in Val.\ \Phi[v \mapsto x] \in Fact \Rightarrow s \models \Phi[v \mapsto x] \end{aligned}$$

And the satisfaction of a set of knowledge-related policies KnPset : 2^{KnPset} by an Inferactor model IM is defined as:

$$IM \models KnPset \iff \forall s \in T(IM), \forall \pi \in KnPset \cdot s \models \pi$$

If a defined policy contains a fact $t(\alpha, _)$, this fact is first translated to $\mathcal{IKBL}$ formula $\forall x \cdot t(\alpha, x)$ and then the satisfaction of the policy is checked.

For example, suppose an actor j within a recommendation system that has also access to some external data sets (some scenarios of privacy violation within the recommendation databases are illustrated in [Sitti et al., 2017]) and a given policy $\neg K_j phone(\text{“Alice”}, _)$. We assume actor j with the knowledge base and set of inference rules as follow:

$kn(j) = \{postalcode(\text{“Alice”}, 12345), age(\text{“Alice”}, 30), profile(0987, 12345, 30)\}$
$irules(j) = \{postalcode(x, y), age(x, z), profile(w, y, z) \rightarrow phone(x, w).\}$

The arguments of *profile* fact are the phone number, postal code, and age of a user. According to the knowledge base and inference rule actor j knows, it can infer the fact $phone(\text{“Alice”}, 0987)$. So, based on the definition of knowledge-related policy semantics, the defined policy is not satisfied in this example.

8 Verification of Knowledge-Related Policies

For analyzing knowledge-related policies, we need to analyze the disclosure of actors' information in the actor system, i.e., to determine the information that each actor can access in the system. As we have explained in [Riahi et al., 2017], an actor can receive private information in three ways: directly, indirectly, or by inference. The difference between direct and indirect receive is that in the first case, the owner of the private information directly sends its information to another actor, but in the second case an actor sends the private information of another actor to a third one. In receiving by inference, the actor infers other actors' private information based on its knowledge base and its set of inference rules.

In [Riahi et al., 2017], we addressed the analysis of data disclosure policies in actor models by addressing direct and indirect receives, but as the actor model in [Riahi et al., 2017] was not designed for modeling the knowledge and inference capabilities of the actors, the receive by inference has not been considered in that work.

As knowledge-related policies should never be violated, we can check them as invariant properties. To verify invariant properties, they must hold for all reachable states [Baier and Katoen, 2008]. We use Breadth First Search (BFS) for constructing the state space, and for each state, we check whether the knowledge of each actor (considering its inference power) satisfies the knowledge-related policies upon its construction. We propose an invariant model checking algorithm that is efficient in two respects 1) we

only maintain a subset of visited states during our BFS, 2) we only check the policies for those actors of states whose knowledge has changed using the static information of the model.

8.1 Efficient Algorithm for Knowledge-Related Policy Verification

We use BFS to construct the state space, starting from the initial state. When a state is generated, we verify the policies and then compute its next states. We maintain those states that their next states have to be computed, in a list called active state list, denoted by *ActvSt*. We keep all the generated states in a set called *VisitedSt*. Initially, we add the model's initial state (as defined in Section 5.3) to the active state list, which should satisfy the given policies. We explore the state at the head of the active state list and compute all its next states using our SOS rules. The newly generated states, not visited before, are added to the active state list to be explored if they satisfy the policies. The visited set grows as the states are explored which may prohibit our invariant checking in cases with a large state space. To tackle this problem, we only keep a subset of visited states, those that may be reached again on other paths during BFS to improve the space efficiency of the algorithm. We approximate the reachability of a state in the future using a heuristic based on the knowledge of actors of the state (the knowledge of the state for short). As actors in our model do not forget their acquired knowledge, facts are only added to their knowledge bases and cannot be removed; when the knowledge of a visited state is less than the knowledge of active states, i.e., *ActvSt*, it cannot be reached again as the next state of any active state. A state s can *possibly reach* the state r, if and only if the knowledge of each actor in s is a subset (or equal) of the knowledge of that actor in r, denoted by $s \sqsubseteq r$.

Definition 8.1 ($\sqsubseteq$ relation) *For a given Inferactor model IM with the set of actors ID, and two states $s, r \in T(IM)$, the formal definition for $\sqsubseteq$ relation is defined as below:*

$$s \sqsubseteq r \iff \forall x \in ID \cdot s(x) = (v, q, \sigma, \kappa) \wedge r(x) = (v', q', \sigma', \kappa') \Rightarrow \kappa \subseteq \kappa'$$

Trivially, $\sqsubseteq$ relation is a pre-order relation. From $s \sqsubseteq r$, it is possible that s reach r over a path over which its knowledge increases.

We keep a state in *VisitedSt* as long as it is not completely explored yet or is possibly reachable again. Formally, a state s is possibly reachable when there exists at least one active state s' where $s' \sqsubseteq s$. So, a state s can be removed from *VisitedSt*, if $\nexists s' \in$ *ActvSt* $\cdot\, s' \sqsubseteq s$. Please note that each state is added to both *ActvSt* and *VisitedSt* upon its creation. So, a state cannot be removed from *VisitedSt* while it belongs to *ActvSt* (as the mentioned condition does not hold). In other words, *ActvSt* $\subseteq$ *VisitedSt*. We prove that our method is correct.

Furthermore, we make our algorithm efficient when it checks whether a newly created state satisfies the knowledge-related policies. The knowledge of an actor can only be increased when a remember statement is executed, so the knowledge-related policies should only be checked in those states created as the consequence of executing a method that contains the remember statement. In addition, in the destination state of a transition, only the knowledge of the actor that caused this transition may be increased and the knowledge of other actors remains unchanged. So, in each state, we only need to check the satisfaction of the policies that are related to this actor. We define two auxiliary functions to be used in model checking algorithm:

- $msgremember$: *Inferactor* $\rightarrow 2^{MName}$. The $msgremember(IM)$ returns the set of all method names in *IM* which includes a remember statement.

– $policyset : 2^{KnPolicy} \times ID \rightarrow 2^{KnPolicy}$. The $policyset(pset, x)$ returns the set of all knowledge-related policies in $pset$ that are related to actor x.

We explain Algorithm 1 in detail. For a given Inferactor model *IM* and a set of knowledge-related policies *KnPset*, first it creates the initial state (line 3). The initial state is added to *ActvSt* if it satisfies *KnPset* (lines 4-7). Then, until *ActvSt* is not empty, a state is taken (and removed) from the head of *ActvSt*, and all its possible transitions are added to the set *Trset* (lines 9-10). For each transition tr (made as a result of the execution of a method belonging to actor x) and its destination state s, if s has been created previously (i.e. $s \in$ *VisitedSt*), nothing else is done about s. Otherwise (i.e. $s \notin$ *VisitedSt*), s is added to *VisitedSt* (line 13-14) and it is determined whether s satisfies the set of policies *policyset*$(KnPset, x)$ or not (lines 15-20), and then s is added to *ActvSt* (line 21). For each state r in the visited states, the algorithm checks whether r can be removed from the set based on the condition we define for removing a state (lines 22-26). If *removable* flag in line 23 is set to *false*, it means that there exists at least one state in *ActvSt* which can reach r, then r cannot be removed. But, if *removable* flag is set to *true*, state r is removed from *VisitedSt* (lines 27-28).

In Algorithm 1, the function *Dest*(tr) returns the destination state of transition tr. The function L_{mname} : *Tran* $\rightarrow$ *MName* returns the method name part of a transition label, and the function L_{sender} : *Tran* $\rightarrow$ *ID* returns the sender part of a transition label. For example, for transition tr with label $A : m$, we have $L_{mname}(tr) = m$ and $L_{sender}(tr) = A$.

The correctness of the defined condition for removing a state from the state space during the construction of the state space, is proved by Lemma 1.

Lemma 1 *If a state is removed by Algorithm 1 from the state space of an Inferactor, then there is no possibility to create it again in the rest of the state space construction.*

Proof. We prove this lemma by contradiction. We assume the state r is removed by Algorithm 1, but there is a state w which reaches r, so, r will be created again later. So, it holds that $r \sqsubseteq w$. Two cases can be distinguished:

– At the time r is removed, $w \in$ *ActvSt*: according to our condition for removing a state in line 25, r could not be removed.

– w has created after r was removed: in this case, at the time r was removed, there was a state $u \in$ *ActvSt* which later led to the creation of w directly or through some other states, i.e., $u \rightarrow \ \dots \ \rightarrow w$.

  When there is a transition $x \rightarrow y$, we can conclude $x \sqsubseteq y$ and the relation $\sqsubseteq$ is a transitive relation, so:

$$u \rightarrow \ \dots \ \rightarrow w \Rightarrow u \sqsubseteq w$$

  From our assumption that w could have transition to r, we have $w \sqsubseteq r$, and because of the transitivity property of $\sqsubseteq$, we have:

$$u \sqsubseteq w \wedge w \sqsubseteq r \Rightarrow u \sqsubseteq r$$

  At the time r was removed, $u \in$ *ActvSt* $\wedge\ u \sqsubseteq r$, so, according to line 25 of Algorithm 1, r could not be removed.

In both cases, r could not be removed and this contradicts our assumption. Therefore, the initial assumption, that there is a state w which can reach r and so r will be created again later, must be false. □

Here, we discuss the efficiency of our proposed algorithm from two respects:

Algorithm 1 Efficient knowledge-related policies analysis algorithm

Input: The Inferactor model *IM* and the set of knowledge-related policies *KnPset*
Output: Whether the Inferactor model *IM* satisfies the set of knowledge-related policies *KnPset*, and the counterexample is returned

1: $ActvSt \leftarrow \langle\rangle$
2: $VisitedSt \leftarrow \emptyset$
3: $s_0 \leftarrow ComputeInitialState(IM)$
4: **For each** $\varphi \in KnPset$
5:     **If** $s_0 \not\models \varphi$
6:         **return** *false*
7: $add(ActvSt, s_0)$
8: **While** $ActvSt \neq \emptyset$ *do*
9:     $u \leftarrow head(ActvSt)$ //u is removed from *ActvSt*
10:     $Trset \leftarrow ComputeAllNextTransitions(u)$
11:     **For Each** $tr \in Trset$ *do*
12:         $s \leftarrow Dest(tr)$
13:         **If** $s \notin VisitedSt$
14:             $VisitedSt \leftarrow VisitedSt \cup \{s\}$
15:             $x \leftarrow L_{sender}(tr)$
16:             **If** $L_{mname}(tr) \in msgremember(IM)$
17:                 **For each** $\varphi \in policyset(KnPset, x)$
18:                     **If** $s \not\models \varphi$
19:                         $counterexample = (x, \varphi, s)$
20:                         **return** *false*
21:             $ActvSt \leftarrow add(ActvSt, s)$
22:     **For each** *state* $r \in VisitedSt$
23:         $removable = true$
24:         **For each** *state* $w \in ActvSt$
25:             **If** $w \sqsubseteq r$
26:                 $removable = false$
27:         **If** $removable == true$
28:             $VisitedSt \leftarrow VisitedSt \setminus \{r\}$
29: **return** *true*

- Algorithm 1 only checks policies of the states in which the knowledge of an actor has increased as a consequence of executing a method with a remember statement (line 16) and only checks the policies for the actor whose knowledge has changed (line 17). We assume that N is the total number of states in the model, N_1 is the number of states created as a result of executing a method m, which includes remember statements (i.e., $m \in msgremember(IM)$), and N_2 is the number of states created as a result of executing a method $m \notin msgremember(IM)$ (we have $N = N_1 + N_2$). Therefore, instead of performing $O(N \times |KnPset|)$ checks, the number of checks whether a state satisfies a policy has the following order:

$$O(N_1 \times max_{x \in ID}(|policyset(KnPset, x)|))$$

- Algorithm 1 does not maintain all the states visited during BFS. We will not keep the states that have been checked before and do not affect the further construction of the state space. In this way, the number of states and as a result the order of the space required to run the algorithm is reduced. The number of states that are removed from the state space based on our algorithm depends on how to increase the knowledge of the actors in the system model. For an iteration of the algorithm, assume that n_1 is the number of states like $r \in VisitedSt$ that there exists a state

$w \in$ *ActvSt* such that $w \sqsubseteq r$, and n_2 is the number remaining states in *VisitedSt* (we have $|VisitedSt| = n_1 + n_2$). In this iteration, n_2 states are removed from *VisitedSt* which improves the memory consumption of the algorithm.

To show the steps of the execution of Algorithm 1, we first define a more simpler and abstract scenario for the running example, with an emphasis on the non-deterministic order of messages due to the concurrent execution of the actors. In this scenario, the electricity consumers send their electricity consumption to a utility, and they expect this information is not disclosed to anyone else. The utility may send this information to an analyzer to receive the result of the analysis. In this case, the utility performs an operation on the information, for example, adds a fixed value to it, so the analyzer does not have access to the actual consumer's information. Also, to test how the analyzer and the utility interact, the analyzer can send its information to the utility and then the next steps can be done on it. The important issue in this scenario is that if the analyzer information is among the information that the utility sends to the analyzer, then the analyzer can compare the value she sent to the utility and the value she received, and in this way she can also find the actual values of other consumers' information.

To keep the state space small, we only model one consumer and ignore some unnecessary interactions. Fig. 4 illustrates the second scenario for the running example. By execution of the *main* block, messages $m4$ and $m5$ are put in the message queues of actors $consumer$ and $analyzer$, respectively. When actor $consumer$ takes message $m4$, method $m4$ (lines 3-5) is executed and message $m1$ is sent to $utility$. When $analyzer$ takes message $m5$, method $m5$ (lines 9-11) is executed and message $m2$ is sent to $utility$. Due to the concurrent execution of actors, the order of sending $m1$ and $m2$ to $utility$ is non-deterministic and as the result the order of messages in the mailbox of $utility$ is non-deterministic. When $utility$ takes $m1$, only the received information is saved in its knowledge-base ('remember' statement in line 23). By taking $m2$, first $utility$ saves the received information and then sends all its known information (by sending message $m3$ for each information) to $analyzer$ (lines 27-29). So, the order of receiving $m1$ and $m2$ in $utility$ affects the number of data sent to $analyzer$ by $utility$. Actor $analyzer$ saves the received information by taking message $m3$. We assume if $analyzer$ knows $known(\text{``}d1\text{''})$ and $known(\text{``}d2\text{''})$ then it can infer $known(\text{``}d3\text{''})$. This inference capability of $analyzer$ is modeled by the inference rule in lines 16-17. The knowledge-related policy for this example is "analyzer is not allowed to know $know(\text{``}d3\text{''})$.

The complete state space constructed for this example by assuming the big-step semantics (described in Sect. 5.3) is presented in Fig. 5. Each state in Fig. 5, has three columns specifying the state of actors $consumer$, $analyzer$ and $utility$, respectively from left to right, and the number in top left of each state specifies the order of creating the states. The actor's message queue is shown by $\langle\rangle$ notation, and the actor's knowledge base is shown by $\{\}$ notation. Note that none of the actors in the example have state variable, so the states in Fig. 5 do not contain state variable part.

The executing of Algorithm 1 for each transition in the state space presented in Fig. 5 is shown in Table 1. The leftmost column specifies the order of creating the transitions. The rightmost column indicates whether the set of knowledge-related policies (*KnPset*) is satisfied in the current point of the model checking algorithm. Here, $KnPset = \{\neg K_{analyzer} known(\text{``}d3\text{''})\}$.

When state 2 is created and the transition from state 1 to state 2 is added to the state space (row 1), the visited states are 1 and 2, and both of them are active. At this point, *KnPset* is satisfied. By adding the transition from 6 to 9, first, the set of visited states is $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and the set of active states is $\{7, 8, 9\}$, but since the knowledge

```
1  actor consumer{
2   consumer{}
3   def m4(){
4    utility!m1("d1");
5   }
6  }
7  actor analyzer{
8   analyzer{}
9   def m5(){
10   utility!m2("d2");
11  }
12  def m3(x){
13   remember (known(x));
14  }
15  inference{
16   known("d1"), known("d2")
17     -> known("d3").
18  }
19 }
```

```
20 actor utility{
21  utility{}
22  def m1(x){
23   remember (known(x));
24  }
25  def m2(x){
26   var y;
27   remember (known(x));
28   forall (known(?y)){
29    analyzer!m3(y);
30   }
31  }
32 }

34 main{
35  consumer!m4();
36  analyzer!m5();
37 }
```

Figure 4: The running example (second scenario) modeled in Inferactor

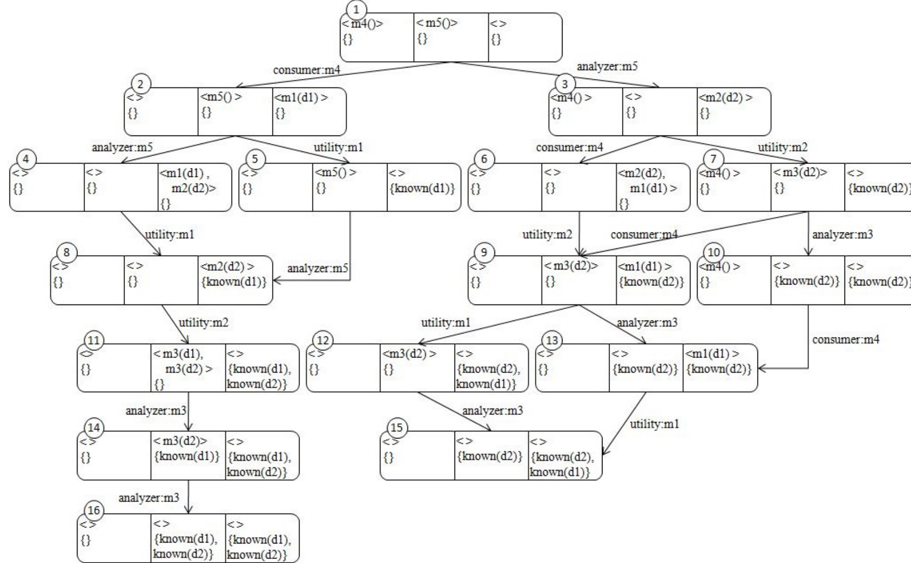


Figure 5: The state space constructed for running example

of state 1 is less than the knowledge of the states in active state list, 1 cannot be reached again from any active state, then 1 is removed from the *VisitedSt*. By the same reasoning, states 2, 3, 4, and 6 are also removed from *VisitedSt*, so *VisitedSt* is updated to $\{5, 7, 8, 9\}$. Finally, when state 16 is created (row 19), the knowledge base of actor $analyzer$ is updated to $\{known(\text{"}d1\text{"}), known(\text{"}d2\text{"})\}$ and based on the inference rule defined for $analyzer$ (lines 15-18 of Fig. 4), $analyzer$ can infer $known(\text{"}d3\text{"})$. Since $analyzer$ is not allowed to know $known(\text{"}d3\text{"})$, a policy violation occurs and the $counterexample$ is set to $(analyzer, \neg K_{analyzer} known(\text{"}d3\text{"}), 16)$.

Step	Transition	*VisitedSt*	*ActvSt*	*KnPset*
1	$1 \to 2$	$\{1,2\}$	$\langle 1,2\rangle$	✓
2	$1 \to 3$	$\{1,2,3\}$	$\langle 2,3\rangle$	✓
3	$2 \to 4$	$\{1,2,3,4\}$	$\langle 2,3,4\rangle$	✓
...	...	...	...	✓
8	$5 \to 8$	$\{1,2,3,4,5,6,7,8\}$	$\langle 6,7,8\rangle$	✓
9	$6 \to 9$	$\{5,7,8,9\}$	$\langle 7,8,9\rangle$	✓
10	$7 \to 9$	$\{5,7,8,9\}$	$\langle 7,8,9\rangle$	✓
...	...	...	...	✓
17	$12 \to 15$	$\{10,13,14,15\}$	$\langle 13,14\rangle$	✓
18	$13 \to 15$	$\{14\}$	$\langle 14\rangle$	✓
19	$14 \to 16$	$\{\}$	$\langle\rangle$	$\times$

Table 1: The execution of Algorithm 1 for the running example

9 Related Work

There are a number of actor-based programming and modeling languages, such as Erlang [Armstrong, 2007], Rebeca [Sirjani et al., 2004], Ptolemy II [Eker et al., 2003], and Abstract Behavioral Specification (ABS) [Johnsen et al., 2012a, ABS, 2022], have been presented for modeling, designing, and reasoning about a wide range of concurrent and distributed systems. Compared to the existing actor modeling languages, our Inferactor model enables modeling the knowledge and inference capabilities of the actors, but in terms of actor computation, it does not add much to the existing modeling languages and is almost similar to Rebeca. In this way, we have made it possible to reason about the knowledge of the actors using Inferactor model. The ABS language supports the modeling of time-sensitive and resource-sensitive behaviors. In [Johnsen et al., 2012b], a Real-Time ABS model for resource-aware applications is presented to model virtualized systems in a cloud environment. In [Kamburjan et al., 2016], ABS has been extended to express scheduling policies based on required communication ordering, to verify communication correctness in a concurrency model of the core ABS language. In [Kamburjan et al., 2019], ABS has been extended with Hybrid Active Objects for modeling cyber-physical systems. Ptolemy II [Eker et al., 2003] is a software framework that supports actor-based modeling and design. In [Baldwin et al., 2004], an extension of Ptolemy II has been used for the modeling and simulation of wireless sensor networks. In [Lasnier et al., 2013], Ptolemy II is used as part of a framework for distributed simulation of cyber-physical systems. In [Kamaleswaran and Eklund, 2011], Ptolemy II has been used for simulating real-time components and generating the respective Java code for interactive hypothesis testing of a clinical decision support system. Rebeca [Sirjani et al., 2004] is an actor-based modeling language used to model concurrent and distributed systems. Some extensions of Rebeca are Timed Rebeca [Reynisson et al., 2014] and Probabilistic Timed Rebeca [Jafari et al., 2014] which extend Rebeca to model real-time systems and to capture probabilistic behaviors, respectively. In [Moradi et al., 2020], Timed Rebeca has been used for security analysis of cyber-physical systems at the design phase. Erlang [Armstrong, 2007] is an actor programming language designed for concurrent real-time distributed applications, and supports a huge number of independent actors. As we aim to analyze knowledge-related policies, we propose Inferactor (based on Actor model [Agha, 1985]), which enables us to model the actors' knowledge and inference capability.

As stated in [Tschantz and Wing, 2009], formal methods are useful in finding security vulnerabilities at the code level, especially when combined with static analysis techniques. Our proposed approach to analyzing knowledge-related policies belongs to the category

of static analysis techniques because it verifies all the execution paths of a program without actually executing it. Static analysis techniques have been widely applied to various types of properties. In the context of privacy, static analysis of privacy makes it possible to detect potential leaks during design time before the system is deployed [Ferrara and Spoto, 2018]. In [Ferrara and Spoto, 2018] the focus is on the GDPR and static analysis. They have combined taint analyses and backward-slicing algorithms to produce reports useful for GDPR compliance. [Wang, 2022] has introduced a static analyzer, via abstract interpretation, to support compliance verification between a program and a policy. This approach encodes policies as abstract values and uses the standard Python interpreter to perform abstract interpretation. In [Tokas et al., 2022], a type and effect system is developed for an object-oriented, distributed modeling language supporting both asynchronous and synchronous method interactions. The privacy policies for data types and methods are encoded within the program. This approach checks statically a program's compliance with privacy policies at the compile time using the type system. In [Baramashetru et al., 2023], privacy concepts have been integrated into a core active object language. The syntax and operational semantics of this language are presented and it is proven that all the executions following the defined operational semantics are privacy compliant. In [Kouzapas and Philippou, 2015] and [Kokkinofta and Philippou, 2016], a framework for statically ensuring that a privacy policy is satisfied by an information system is developed. This framework is based on the π-calculus and accompanied by a type system and a privacy language for capturing privacy requirements and expressing privacy policies, respectively. This framework performs type checking of the system against a typing and produce a permission interface and if the produced permission interface satisfies a policy, the system satisfies that policy. In [Samani, 2015], a framework for modeling and analyzing privacy concerns in distributed systems has been presented. This framework is applied to the interaction protocols. In this framework, depending on the sequence of messages in the interaction protocol and the identified privacy concern, an adequate protection mechanism, such as anonymization or encryption, is applied. In [Riahi et al., 2017], an approach for data disclosure policies analysis in the actor model has been presented. The model checking and data dependence analysis are used to ensure that the actors do nothing that violates the defined policies. Among the studies reviewed in this field, [Tokas et al., 2022] and [Riahi et al., 2017] are the most relevant to our work. [Tokas et al., 2022] defines privacy policies for data types and methods and checks their compliance using a type system. In contrast, we define policies for the knowledge of the actors and check their satisfaction through model checking. The actor model used in [Riahi et al., 2017] does not support modeling the knowledge and inference capabilities of the actors. While, we have proposed Inferactor model which supports the modeling of actors' knowledge and inference capability.

Access control is a useful defense mechanism organizations can deploy to meet legal compliance with data privacy. The most widely used access control strategies fall into Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) categories [Golightly et al., 2023, You et al., 2023]. Access control policies prevents unauthorized access to data. While, our privacy policies describe the required privacy of the system and we check the satisfaction of these policies in the system by verifying them in the system model. Since we consider secondary access to data (or data usage), access to data through inference, and access to data about the knowledge of the other actors, we need to model actors' knowledge and their inference capabilities. Our defined privacy policies can be considered as generalized access and usage control policies related to the knowledge of the actors.

In the following, we review the works that addressed the knowledge of the agents in

the areas close to our concern in this paper. In [Lager, 2019], a web logic programming language, called Web Prolog, has been introduced, which extends Prolog [Sterling and Shapiro, 1986] with some features of actor programming language Erlang [Armstrong, 2007] to provide some primitives that support concurrency, distribution and intraprocess communication. Web Prolog is a programming language with many details, but we introduce a modeling language and try to make it as simple as possible for modeling and analyzing the actor systems, and easy to learn for the users.

One group of work considers the knowledge of agents in social networks. For example in [Pardo and Schneider, 2014], a formal privacy policy framework (PPF) based on epistemic logic [Fagin et al., 2003] to specify and reason about privacy policies in social networks has been presented. Then in [Pardo et al., 2017], the authors have extended the PPF framework by considering a deductive engine for agents which enables the agents to perform knowledge inferences. They provide a set of template operational semantic rules for each social network to model its dynamics. These rules are conditioned by the privacy policies, specified by epistemic formulae. In [Pardo and Schneider, 2017], the problem of verifying knowledge properties over social network models (SNMs) has been considered. In [Moezkarimi et al., 2022], an epistemic framework is introduced for social networks to verify privacy in social networks. This framework takes into account the ability of agents to infer knowledge by considering the sequence of observed messages, while in our approach the actors infer knowledge based on their obtained information.

Modeling the knowledge of agents has also been considered in evaluation of security protocols. For example in [Chen et al., 2008], formal analysis of secure transaction protocols has been studied. The authors have proposed a logical framework, including the axioms and inference rules, and a verification model, including the inference engine and the knowledge base, to enable protocol analysis. In [Pavlovic and Meadows, 2011] and [Pavlovic and Meadows, 2012], a logical framework for reasoning about security of protocols has been presented. The authors have proposed Actor-Network Procedure (ANP) for formalizing security ceremonies and the Procedure Derivation Logic (PDL) to reason logically about ANPs. The authors in [González-Burgueño and Ölveczky, 2019] have extended ANPs to support explicitly specifying the nodes' capabilities for heterogeneous devices and have also modify PDL to take into account the knowledge of participants at different points in time. Their model does not have an executable formal semantics and they use a non-automatic manner for proving the properties. The authors in [González-Burgueño and Ölveczky, 2021] have defined ANPs with capabilities (ANP-Cs), which extends ANPs with an explicit specification of the capabilities of the different nodes. They have defined two variations of PDL logic called PDL-CK and PDL-CK$_L$. PDL-CK supports global reasoning about the system and PDL-CK$_L$ supports local reasoning from the perspective of a single node, based only on what that node can observe. In [Kamkuemah, 2022], an approach to analyze security protocols has been introduced which uses epistemic logic (to specify security requirements of a protocol) and Z [Spivey and Abrial, 1992] specification language (to specify the protocols). This approach considers assumptions about the cryptographic primitives and the capabilities of adversaries that attack the protocols, and reasons algebraically that protocols achieve the security properties under the assumption. Since we want to support concurrency and its resulting nondeterminism in our model, this approach is not suitable for us.

Table 2 presents a comparison of the closest work to ours based on a set of metrics. These metrics include target systems, modeling formalism, knowledge modeling, supporting inference capability, policy specification, and the method to analyze privacy policies in the system. As our domain of focus is asynchronous message passing distributed systems, one can model the inference capability of computing agents (actors) based

Reference	System type	Model	Knowledge modeling	Inference	Policy specification	Analysis method
[Pardo et al., 2017]	Social network service (SNS)	Social network model (SNM)	✓	✓	✓	Model checking
[Riahi et al., 2017]	Distributed system (Actor model)	Rebeca	×	×	✓	Model checking and data dependence analysis
[Tokas et al., 2022]	Distributed system (active object paradigm)	A small language based on the active object paradigm	×	×	✓	Static type checking (define a type and effect system)
Current work	Distributed system (Actor model)	Inferactor	✓	✓	✓	Model checking

Table 2: Comparison with the most related work

on communicated information embedded in messages to analyze the effect of inferred knowledge by agents. In the approach in the first row of Table 2, the template operational rules of the framework must be concretized with respect to the features of the given instance of social network. The privacy policies defined for a given social network are embedded within the concrete operational semantic rules. When policy rules are changed, the semantic rules have to be revised accordingly. In our approach, the semantic rules of our framework are not affected by the instance of the distributed system. The second and third rows do not consider the knowledge and inference capabilities of agents, so they cannot find privacy violations due to inferred knowledge achieved by agents through communication.

10 Conclusions and Future Work

One of the most important reasons for privacy violations in distributed systems is related to the lack of control over the information that agents of the system transmit to each other. In this paper, we provide a formalism to model the knowledge of the actors along with their other characteristics. In this way, we can speak about the actors' knowledge in presence of their inference capabilities, as well as their asynchronous communications and interleaving of their executions.

We considered a knowledge base for each actor and defined some operations through some statements to manipulate/use it. The knowledge base of each actor contains the knowledge that is directly obtained through interactions with other actors. The inference capability of an actor, which enables the actor to infer new knowledge based on its current knowledge (using S5 axiomatization of epistemic logic), is explicitly specified in the actor's body. We defined knowledge-based logic $\mathcal{IKBL}$ to specify the actors' knowledge and inference capabilities. We extended $\mathcal{IKBL}$ to specify knowledge-related policies. Such policies can be considered as generalized access and usage control policies related to the knowledge of the actors by imposing restrictions on the knowledge of an actor about the knowledge of the other actors. We proposed an efficient model checking algorithm, to check the satisfaction of knowledge-related policies for an Inferactor. Using our method, we can ensure that in a distributed asynchronous system there is no knowledge-related policy violation.

We intend to develop an automatic tool for model checking the defined policies in Inferactor models. This tool includes an epistemic engine to support the epistemic

features in addition to the usual characteristics of the actor model (including computation and communication). The tool enables us to elaborate more complex examples. We also aim to define another type of inference rule to consider the inference capabilities of the actors based on the received messages in addition to the known facts. In this case, in addition to making inferences based on knowledge, actors can also make inferences based on the behavior they observe from the other actors. This type of inference rules for an actor can be defined based on the received messages of that actor or based on the information from messages exchanged among other actors. In the second case, since the view of the actors is local and they only know about the interactions in which they participate, a mechanism is needed to inform an actor about the messages sent between the other actors. This extension is beneficial to reason about scenarios in which agent has observability capabilities due to their special roles and so accesses in the system. The purpose of using private information is an important aspect of privacy policies which complements the access and usage policy rules [Tschantz et al., 2012]. We intend to enrich our approach with the specification and analysis of purpose-based privacy policies. The specification of purpose-based privacy policies requires the definition of the purpose model, and the analysis of these policies requires that the conformance of the system model with the purpose model is also defined.

References

[ABS, 2022] Abstract Behavioral Specification (ABS) language, http://abs-models.org.

[Agha, 1985] Agha, G.A.: “Actors - a model of concurrent computation in distributed systems”; MIT Press series in artificial intelligence, MIT Press (1985).

[Armstrong, 2007] Armstrong, J.: “Programming Erlang, Software for Concurrent World”; Pragmatic Bookshelf (2007).

[Asghar et al., 2017] Asghar, M.R., Dán, G., Miorandi, D., and Chlamtac, I.: “Smart Meter Data Privacy: A Survey”; IEEE Communications Surveys and Tutorials, Vol. 19, No. 4, (2017), 2820–2835.

[Baier and Katoen, 2008] Baier, C. and Katoen, J.P.: “Principles of Model Checking”; MIT Press, Cambridge, Massachusetts, London, England (2008).

[Baldwin et al., 2004] Baldwin, P., Kohli, S., Lee, E.A., Liu, X., and Zhao, Y.: “Modeling of Sensor Nets in Ptolemy II”; In proceedings of Information Processing in Sensor Networks (IPSN), April (2004), 359–368.

[Baramashetru et al., 2023] Baramashetru, C., Tapia Tarifa, S.L., and Owe, O.: Integrating Data Privacy Compliance in Active Object Languages. Research report http://urn.nb.no/URN:NBN:no-35645. (2023).

[Bi et al., 2020] Bi, M., Wang, Y., Cai, Z., and Tong, X.: A privacy-preserving mechanism based on local differential privacy in edge computing. China communications, Vol. 17, Issue 9, (2020), 50–65.

[Blanke, 2020] Blanke, J.M.: “Protection for ‘Inferences Drawn’: A Comparison Between the General Data Protection Regulation and the California Consumer Privacy Act’’; Global Privacy Law Review, Vol. 1, Issue 2, (2020), 81–92.

[Boer et al., 2017] Boer, F.D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., and Yang, A.M.: “A survey of active object languages”; ACM Computing Surveys (CSUR), Vol. 50, No. 5, (2017), 1–39.

[Bugeja et al., 2021] Bugeja, J., Jacobsson, A., and Davidsson, P.: “PRASH: a framework for privacy risk analysis of smart homes”; Sensors, Vol. 21, Issue 19, p.6399 (2021).

[Cardenas and Safavi-Naini, 2012] Cardenas, A. and Safavi-Naini, R.: "Security and Privacy in the Smart Grid"; Handbook on Securing Cyber-Physical Critical Infrastructure (2012).

[Celik et al., 2019] Celik, Z.B., Fernandes, E., Pauley, E., Tan, G., and McDaniel, P.: "Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities''; ACM Computing Surveys (CSUR), Vol. 52, Issue 4, Article No. 74, (2019), 1–30.

[Chaaya et al., 2019] Chaaya, K.B., Barhamgi, M., Chbeir, R., Arnould, P., and Benslimane, D.: Context-aware system for dynamic privacy risk inference: Application to smart IoT environments. Future Generation Computer Systems, Vol. 101, (2019), 1096–1111.

[Chanal and Kakkasageri, 2020] Chanal, P.M. and Kakkasageri, M.S.: "Security and privacy in IOT: a survey''; Wireless Personal Communications, Vol. 115, (2020), 1667–1693.

[Chen et al., 2008] Chen, Q., Zhang, C., and Zhang, S.: "Secure Transaction Protocol Analysis: Models and Applications"; Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg (2008).

[Dong et al., 2023] Dong, S., Zhan, J., Hu, W., Mohajer, A., Bavaghar, M. and Mirzaei, A.: Energy-efficient hierarchical resource allocation in uplink-downlink decoupled NOMA HetNets. IEEE Transactions on Network and Service Management, (2023).

[Eker et al., 2003] Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y.: "Taming heterogeneity - the Ptolemy approach"; Proceedings of the IEEE, Vol. 91, Issue 1, January (2003), 127–144.

[Fagin et al., 2003] Fagin, R., Halpern, J.Y., Moses, Y., and Vardi, M.Y.: "Reasoning About Knowledge"; MIT press Cambridge (2003).

[Fang et al., 2012] Fang, X., Misra, S., Xue, G., and Yang, D.: "Smart Grid - The New and Improved Power Grid: A Survey"; IEEE Communications Surveys and Tutorials (2012).

[Ferrara and Spoto, 2018] Ferrara, P. and Spoto, F.: "Static Analysis for GDPR Compliance"; The Second Italian Conference on Cyber Security (ITASEC18), CEUR Workshop Proceedings, Vol. 2058, (2018).

[GDPR, 2020] General Data Protection Regulation (GDPR), available at: https://gdpr-info.eu/.

[Giaconi et al., 2020] Giaconi, G., Gund, D., and Poor, H.V.: "Smart Meter Data Privacy"; CoRR abs/2009.01364 (2020).

[Golightly et al., 2023] Golightly, L., Modesti, P., Garcia, R., and Chang, V.: "Securing Distributed Systems: A Survey on Access Control Techniques for Cloud, Blockchain, IoT and SDN"; Cyber Security and Applications, Vol. 1, p.100015 (2023).

[González-Burgueño and Ölveczky, 2019] González-Burgueño A. and Ölveczky, P.C.: "Formalizing and Analyzing Security Ceremonies with Heterogeneous Devices in ANP and PDL"; 8th IPM International Conference on Fundamentals of Software Engineering (FSEN), (2019), 96–110.

[González-Burgueño and Ölveczky, 2021] González-Burgueño, A. and Ölveczky, P.C.: "Formalizing and analyzing security ceremonies with heterogeneous devices in ANP and PDL"; Journal of Logical and Algebraic Methods in Programming, Vol. 122, August (2021).

[Gunduzl et al., 2015] Gunduzl, D., Kalogridis, G., and Mustafa, M.: "Privacy in Smart Metering Systems"; 7th IEEE International Workshop on Information Forensics and Security (IEEE WIFS 2015), Rome, Italy (2015).

[Halpern and O'Neill, 2008] Halpern, J. and O'Neill, K.: "Secrecy in multiagent systems"; ACM Transactions on Information and System Security, Vol. 12, Issue 1, No. 5, (2008), 1–47.

[Halvorsen et al., 2022] Halvorsen, L., Steffensen, S.L., Rafnsson, W., Kulyk, O., and Pardo, R.: "How Attacker Knowledge Affects Privacy Risks: An Analysis Using Probabilistic Programming"; In Proceedings of the 2022 ACM on International Workshop on Security and Privacy Analytics, (2022), 55–65.

[He and Anton, 2003] He, Q. and Anton, A.I.: A framework for modeling privacy requirements in role engineering. In Proc. of REFSQ, Vol. 3, (2003), 137–146.

[Hsu et al., 2001] Hsu, TS., Liau, CJ., and Wang, DW.: "A Logical Model for Privacy Protection"; In: International Conference on Information Security, Springer, Berlin, Heidelberg (2001), 110–124.

[Jafari et al., 2014] Jafari, A., Khamespanah, E., Sirjani, M., and Hermanns, H.: "Performance Analysis of Distributed and Asynchronous Systems using Probabilistic Timed Actors"; Electronic Communication of the European Association of Software Science and Technology 70 (2014).

[Johnsen et al., 2012a] Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: "ABS: a core language for abstract behavioral specification"; Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, Austria, November 29-December 1, 2010, Revised Papers 9, Springer Berlin Heidelberg, (2012), 142–164.

[Johnsen et al., 2012b] Johnsen, E.B., Schlatte, R., and Tapia Tarifa, S.L.: "Modeling resource-aware virtualized applications for the cloud in Real-Time ABS''; In Formal Methods and Software Engineering: 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, Springer Berlin Heidelberg, (2012), 71–86.

[Kamaleswaran and Eklund, 2011] Kamaleswaran, R. and Eklund, M.: "A method for interactive hypothesis testing for Clinical Decision Support Systems using Ptolemy II"; In: 24th Canadian Conference on Electrical and Computer Engineering (CCECE), IEEE (2011), 001278–001281.

[Kamburjan et al., 2016] Kamburjan, E., Din, C.C., and Chen, T.C.: "Session-based compositional analysis for actor-based languages using futures''; In Formal Methods and Software Engineering: 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Springer International Publishing, (2016), 296–312.

[Kamburjan et al., 2019] Kamburjan, E., Mitsch, S., Kettenbach, M., and Hähnle, R.: "Modeling and verifying cyber-physical systems with hybrid active objects''; arXiv preprint arXiv:1906.05704, (2019).

[Kamkuemah, 2022] Kamkuemah, M.N.: "An analysis of security protocols for lightweight systems"; Ph.D. dissertation, Stellenbosch University, (2022).

[Khamespanah, 2018] Khamespanah, E.: "Modeling, Verification, and Analysis of Timed Actor-Based Models"; Ph.D. dissertation, Reykjavik University School of Computer Science (2018).

[Kokkinofta and Philippou, 2016] Kokkinofta, E. and Philippou, A.: "Type Checking Purpose-Based Privacy Policies in the π-Calculus"; Web Services, Formal Methods, and Behavioral Types (WS-FM 2014/WS-FM 2015), Springer International Publishing Switzerland (2016), 122–142.

[Kouzapas and Philippou, 2015] Kouzapas, D. and Philippou, A.: "Type Checking Privacy Policies in the Π-calculus"; FORTE 2015: Formal Techniques for Distributed Objects, Components, and Systems, (2015), 181–195.

[Lager, 2019] Lager, T.: "Intro to Web Prolog for Erlangers"; In proceedings of the 18th ACM SIGPLAN International Workshop on Erlang (Erlang '19), (2019), 18–29.

[Lasnier et al., 2013] Lasnier, G., Cardoso, J., Siron, P., Pagetti, C., and Derler, P.: "Distributed simulation of heterogeneous and real-time systems"; In: 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications, (2013), 55–62.

[Lehnherr et al., 2022] Lehnherr, D., Ognjanović, Z., and Studer, T.: "A Logic of Interactive Proofs"; In: Artemov S., Nerode A. (eds) International Symposium on Logical Foundations of Computer Science. Lecture Notes in Computer Science, Vol. 13137, Springer, Cham (2022), 143–155.

[Lisovich et al., 2010] Lisovich, M.A., Mulligan, D.K., and Wicker, S.B.: Inferring personal information from demand-response systems. IEEE Security and Privacy, Vol. 8, Issue 1, (2010), 11–20.

[Masellis et al., 2015] Masellis, R.D., Ghidini, CH., and Ranise, S.: A Declarative Framework for Specifying and Enforcing Purpose-Aware Policies. In: Foresti, S. (eds.) Security and Trust Management. LNCS, vol. 9331, Vienna, Austria, (2015), 55–71.

[Meyer and van der Hoek, 1995] Meyer, J.J.Ch. and van der Hoek, W.: “Epistemic Logic for AI and Computer Science”; volume 41 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (1995).

[Moezkarimi et al., 2022] Moezkarimi, Z., Ghassemi, F. and Mousavi, M.R.: “A policy-aware epistemic framework for social networks”; Journal of Logic and Computation, Vol. 32, Issue 6, September (2022), 1234–1271.

[Mohajer et al., 2022A] Mohajer, A., Daliri, M.S., Mirzaei, A., Ziaeddini, A., Nabipour, M., and Bavaghar, M.: Heterogeneous computational resource allocation for NOMA: Toward green mobile edge-computing systems. IEEE Transactions on Services Computing, Vol. 16, Issue 2, (2022), 1225–1238.

[Mohajer et al., 2022B] Mohajer, A., Sorouri, F., Mirzaei, A., Ziaeddini, A., Rad, K.J. and Bavaghar, M.: Energy-aware hierarchical resource management and backhaul traffic optimization in heterogeneous cellular networks. IEEE Systems Journal, Vol. 16, Issue 4, (2022), 5188–5199.

[Moradi et al., 2020] Moradi, F., Abbaspour Asadollah, S., Sedaghatbaf, A., Čaušević, A., Sirjani, M., and Talcott, C.: “An actor-based approach for security analysis of cyber-physical systems”; In International Conference on Formal Methods for Industrial Critical Systems, Springer, Cham (2020), 130–147.

[Morel and Pardo, 2020] Morel, V. and Pardo, R.: “SoK: Three Facets of Privacy Policies”; In WPES (2020).

[Pardo and Schneider, 2014] Pardo, R. and Schneider, G.: “A Formal Privacy Policy Framework for Social Networks”; In: Giannakopoulou D., Salaün G. (eds) Software Engineering and Formal Methods (SEFM 2014), LNCS, Vol. 8702, Springer, Cham (2014).

[Pardo and Schneider, 2017] Pardo, R. and Schneider, G.: “Model Checking Social Network Models”; In proceedings of the Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF’17, Vol. 256, (2017), 238–252.

[Pardo et al., 2017] Pardo, R., Balliu, M., and Schneider, G.: “Formalising Privacy Policies in Social Networks”; Journal of Logical and Algebraic Methods in Programming, Vol. 90, (2017), 125–157.

[Pavlovic and Meadows, 2011] Pavlovic, D. and Meadows, C.: “Actor-network procedures: Modeling multi-factor authentication, device pairing, social interactions”; CoRR abs/1106.0706 (2011).

[Pavlovic and Meadows, 2012] Pavlovic, D. and Meadows, C.: “Actor-Network Procedures”; Distributed Computing and Internet Technology (ICDCIT), LNCS, Vol. 7154, (2012), 7–26.

[Pfitzmann and Hansen, 2010] Pfitzmann, A. and Hansen, M.: “A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management”; (2010) available at: http://www.maroki.de/pub/dphistory/2010_Anon_Terminology_v0.34.pdf.

[Pucella, 2013] Pucella, R.: “Knowledge and Security”; arXiv preprint, arXiv: 1305.0876 (2013).

[Reidenberg et al., 2015] Reidenberg, J.R., et al.: Disagreeable privacy policies: Mismatches between meaning and users’ understanding. Berkeley Technology Law Journal, Vol. 30, No. 1, (2015), 39–88.

[Reynisson et al., 2014] Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingolfsdottir, A., and Sigurdarson, S.H.: “Modelling and Simulation of Asynchronous Real-Time Systems Using Timed Rebeca”; Science of Computer Programming, Vol. 89, (2014), 41–68.

[Riahi et al., 2017] Riahi, Sh., Khosravi, R., and Gassemi, F.: “Purpose-based Policy Enforcement in Actor-based Systems”; 7th International Conference on Fundamentals of Software Engineering (FSEN), LNCS, Springer (2017), 196–211.

[Ronne, 2012] Ronne, J.: “Leveraging Actors for Privacy Compliance”; In: Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions (AGERE! 2012), ACM (2012), 133–136.

[Samani, 2015] Samani, A.: “Privacy in Cooperative Distributed Systems: Modeling and Protection Framework”; Ph.D. dissertation, The University of Western Ontario (2015).

[Schneider, 2018] Schneider, G.: “Is Privacy by Construction Possible?”; 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9 (2018).

[Simmhan et al., 2011] Simmhan, Y., Kumbhare, A.G., Cao, B., and Prasanna, V.: “An Analysis of Security and Privacy Issues in Smart Grid Software Architectures on Clouds”; 4th International Conference on Cloud Computing, IEEE (2011).

[Sirjani et al., 2004] Sirjani, M., Movaghar, A., Shali, A., and de Boer, F.: “Modeling and verification of reactive systems using Rebeca”; Fundamenta Informaticae 63, (2004), 385–410.

[Sitti et al., 2017] Sitti, S., Riyana, S., and Riyana, N.: “Scenario of privacy violation within the recommendation databases”; International Conference on Digital Arts, Media and Technology (ICDAMT), (2017), 383–388.

[Solove, 2006] Solove, D.J.: “A Taxonomy of Privacy”; University of Pennsylvania Law Review, Vol. 154, No. 3, (2006), 477–560.

[Spivey and Abrial, 1992] Spivey, J. M. and Abrial, J.: “The Z notation”; Prentice Hall Hemel Hempstead, (1992).

[Sterling and Shapiro, 1986] Sterling, L. and Shapiro, E.: “The Art of Prolog: Advanced Programming Techniques”; MIT Press, Cambridge MA (1986).

[Tokas and Owe, 2020] Tokas, S. and Owe, O.: “A formal framework for consent management”; In Formal Techniques for Distributed Objects, Components, and Systems. 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, Proceedings 40, Springer International Publishing (2020), 169–186.

[Tokas et al., 2022] Tokas, S., Owe, O., and Ramezanifarkhani, T.: “Static checking of GDPR-related privacy compliance for object-oriented distributed systems”; Journal of Logical and Algebraic Methods in Programming, 125:100733 (2022).

[Tschantz and Wing, 2009] Tschantz, M. and Wing, J.: “Formal Methods for Privacy”; In: 2nd World Congress on Formal Methods, Springer-Verlag, Berlin, Heidelberg (2009), 1–15.

[Tschantz et al., 2012] Tschantz, M.C., Datta, A., and Wing, J.M.: Formalizing and enforcing purpose restrictions in privacy policies. IEEE Symposium on Security and Privacy, (2012), 176–190.

[Van Der Hoek and Verbrugge, 2002] Van Der Hoek, W. and Verbrugge, R.: “Epistemic logic: A survey”; Game theory and applications, Vol. 8, (2002), 53–94.

[Wang et al., 2022] Wang, D., Ren, J., Wang, Z., Zhang, Y. and Shen, X.S.: PrivStream: A privacy-preserving inference framework on IoT streaming data at the edge. Information Fusion, Vol. 80, (2022), 282–294.

[Wang, 2022] Wang, L.: Towards Privacy-Preserving and Regulation-Compliant Data Analysis. Ph.D. thesis, University of California, Berkeley, (2022).

[Yeom et al., 2018] Yeom, S., Giacomelli, I., Fredrikson, M. and Jha, S.: Privacy risk in machine learning: Analyzing the connection to overfitting. In 2018 IEEE 31st computer security foundations symposium (CSF), (2018), 268–282.

[You et al., 2023] You, M., Yin, J., Wang, H., Cao, J., Wang, K., Miao, Y., and Bertino, E.: “A knowledge graph empowered online learning framework for access control decision-making”; World Wide Web, Vol. 26, Issue 2, (2023), 827–848.

[Zabkowski and Gajowniczek, 2013] Zabkowski, T. and Gajowniczek, K.: "Smart metering and data privacy issues"; Information Systems in Management, Vol. 2(3), (2013), 239–249.

[Zeng and Roesner, 2019] Zeng, E. and Roesner, F.: "Understanding and Improving Security and Privacy in Multi-User Smart Homes: A Design Exploration and In-Home User Study"; In USENIX Security Symposium, August (2019), 159–176.

[Zhang et al., 2018] Zhang, J., Chen, B., Zhao, Y., Cheng, X., and Hu, F.: Data security and privacy-preserving in edge computing paradigm: Survey and open issues. IEEE access, Vol. 6, (2018), 18209–18237.