Technical Report

Angerona - A Multiagent Framework for Logic Based Agents with application to Secrecy Preservation

Patrick Krümpelmann, Tim Janus,
Gabriele Kern-Isberner

3/2014

technische universität
dortmund

SFB 876 Verfügbarkeit von
Information durch Analyse unter
Ressourcenbeschränkung

Part of the work on this technical report has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A5.

Speaker: Prof. Dr. Katharina Morik
Address: TU Dortmund University
Joseph-von-Fraunhofer-Str. 23
D-44227 Dortmund
Web: http://sfb876.tu-dortmund.de

Contents

1 Introduction **3**

2 Answer Set Programming **5**

3 Conceptualization of Epistemic Agency for Secrecy **6**

3.1 General Agent Model 6

3.2 Epistemic Component for Secrecy 7

3.3 Functional Component for Secrecy 11

3.3.1 Basic Agent Cycle 12

3.3.2 Belief Change Operations and Secrecy 13

3.3.3 BDI Model for Epistemic Secrecy 15

4 The Angerona Framework **23**

4.1 The Agent Framework 24

4.1.1 Modular Agent Building Block 25

4.1.2 *ASML* based Agent's Cycle 28

4.1.3 Belief Base 30

4.1.4 Change Beliefs 31

4.2 The Multi-Agent Framework 33

4.2.1 The Phases of the Multi-Agent Simulation 33

4.2.2 Entities and Components 35

4.2.3 Report-System 35

4.2.4 Angerona's Graphical User Interface 37

4.3 Implementation of Secrecy Preserving Agents 39

4.3.1 Data Components 40

4.3.2 Functional Components 41

4.3.3 Secrecy Agent Cycle 43

4.4 The Tweety Library 44

5 Conclusion **45**

5.1 Future Work 46

5.1.1 Version aware Data Components 46

5.1.2 *ASML* based Operation Definition for Belief Bases 46

5.1.3 *ASML* for the Selection of the Operator Implementation 47
5.1.4 Annotation based Context Providers 47

A File Formats **51**
A.1 Core Configuration 51
A.2 Operator Sets and XML References 51
A.3 Simulation Template 52
A.4 Agent Configuration 54
A.5 Belief Base Configuration 54
A.6 Belief Base Files 55

B Angerona *Script Markup Language* **55**
B.1 Language Reference 56
B.1.1 *ASML* Command: assign 56
B.1.2 *ASML* Command: invoke operation 56
B.1.3 *ASML*: Control Structures 57
B.1.4 *ASML*: Iteration Structures 59
B.2 Context Provider Variables 59

1 Introduction

The agent paradigm is theoretically well developed in various instantiations and fills entire textbooks [37, 38]. Many agent programming languages and frameworks for the development of agents and multiagent systems exist [8]. A symbolic representation of an agent's knowledge allows for flexible representation, handling and exchange of information between software and human agents. A variety of logical formalisms with different expressivity and computational properties are available for knowledge representation [9, 35]. Yet no framework for developing logic based agents with advanced reasoning capabilities exists. In this work we present the ANGERONA framework which is designed to support the development of epistemic agents based on logical formalisms for knowledge representation and reasoning.

On the topic of secrecy a large body of work exists and diverse definitions of secrecy in various settings with different properties have been developed. For multiagent systems the main research focus herein lies on strong notions of secrecy of a whole (multiagent) system, for an overview see [13]. Secrecy is generally imposed by some global definition of secret information from a global complete view of the system. Various logics for secrecy or privacy have been developed for different purposes, that provide a high degree of expressiveness for specification, for an overview see [34]. While a substantial body of work on the definition of secrecy exists mechanisms for secrecy preservation in multiagent systems are lacking. Of our particular interest is the *inference problem* in which information is not only to be protected directly but also via possible inferences [11]. Hereby, prior information and background knowledge have to be considered in combination with some inference operator. Effective mechanisms for secrecy preserving query answering have been developed for the database setting, e. g. [27, 5]. In those, a censor component is used which checks the answer of the database to a user's query given the history of queries and answers, and the users assumed background knowledge. In case of secrecy violation the censor distorts the answer by refusing the answer or by lying.

In this work we consider secrecy and secrecy preservation from the point of view of an autonomous knowledge-based agent with incomplete and uncertain information, situated in a multiagent system. Moreover, we consider a dynamic system which demands for techniques to handle the dynamics of information and to dynamic secrecy preservation. Dynamic secrecy preservation allows for adaptation and potentially to a higher degree of availability of information than static access rights. The epistemic component of an agent is central to our approach. Secrecy of information and in particular the inference problem depend on the representation of information and the reasoning capabilities of the agents.

Agents reason under uncertainty about the state of the environment, the reasoning of other agents and possible courses of action. They pursue their goals by performing actions in the environment including the communication with other agents. On the one hand, the exchange of information with other agents is often essential for an agent in order to achieve its goals; especially if the agent is part of a coalition. On the other hand the agent is interested, or obliged, not to reveal certain information, its secrets. Restriction of communication leads to a loss in performance and utility of the individual

agent, coalitions and the whole multiagent system. That is, secrecy preservation can be seen as a special kind of maintenance goal [36] which requires reasoning about the information and reasoning capabilities of other agents. A good solution of the implied conflict between the agent's goal to preserve secrecy and its other goals is one that restricts communication just as much as necessary to still preserve secrecy. To achieve a possible violation of secrecy by an intended course of action has to be detected as early as possible and alternative courses have to be found.

In this work we define desirable properties of a notion of secrecy in our considered setting and an adequate notion of secrecy and secrecy preservation within a general epistemic agent model satisfying the desirable properties. We present a general epistemic agent model for secrecy independent of the underlying logic and apt for formalisms capable of dealing with incomplete information such as non-monotonic logics. Moreover, we design a secrecy-preserving agent, i.e. an agent that does not perform a secrecy violating action. This can be achieved by adding a censor operator which checks the agent's actions prior to execution and modifying them if necessary similar to the approaches from database theory [27, 6] and previous applications of these to a multiagent setting [7]. We argue that such an agent loses its autonomy since some foreign object modifies its actions. The performance of such an agent can be suboptimal as it might be much better off by taking secrecy concerns into consideration while deliberating and planning its actions. Here we present an agent model capable of the latter. The more secrecy preservation is integrated into the agent model the more options for utility optimization under secrecy constraints is possible.

For realizing agents, the BDI model [37, 25] has become a leading paradigm of the field. This model distinguishes between *Beliefs*, *Desires*, and *Intentions* as the main components of an agent's mind, the interactions of which determine its behavior. Roughly, *Beliefs* comprise (plausible) knowledge the agent has concerning the current situation and how the world works in general, *Desires* encode what she wishes to achieve and hence represent possible goals, whereas *Intentions* focus on the next actions the agent should undertake to achieve the current goal. The role of *Beliefs* in this scenario is to provide the agent with useful information to evaluate the current situation and to find reasonable and effective ways to achieve her goals. We base our agent model on the well-established belief, desires, intentions model and extend it by a partial planning mechanism and show that it satisfies the defined properties for secrecy preservation.

We consider answer set programming (ASP) [12] as a possible formalism for knowledge representation, reasoning and planning with limited resources and sketch the use of it within our framework. We implemented the general framework presented in this work. The result is a variable system for the development of knowledge-based agents. We implemented the proposed BDI model with an ASP based reasoning and deliberation component.

2 Answer Set Programming

We introduce extended logic programs and the answer set semantics as presented in [12]. An extended logic program consists of rules over a set of atoms $\mathcal{A}$ using strong negation $\neg$ and default negation not. A literal L can be an atom $A \in \mathcal{A}$ or a negated atom $\neg A$. The complement of a literal L is denoted by $\neg L$ and is A if $L = \neg A$ and $\neg A$ if $L = A$. Let $\mathcal{A}$ be the set of all atoms and Lit the set of all literals $Lit = \mathcal{A} \cup \{\neg A \mid A \in \mathcal{A}\}$. For a set of literals $X \subseteq Lit$ we use the notation $not\ X = \{not\ L \mid L \in X\}$ and denote the set of all default negated literals as $\mathcal{D} = not\ Lit$. $\mathfrak{L} = Lit \cup \mathcal{D}$ represents the set of all literals and default negated literals. A rule r is written as

$$L \leftarrow L_0, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n.$$

where the head of the rule $L = H(r)$ is either empty or consists of a single literal and the body $\mathcal{B}(r) = \{L_0, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n\}$ is a finite subset of $\mathfrak{L}$. The language of rules constructed over the set of atoms $\mathcal{A}$ this way is referred to as $\mathcal{L}^{asp}_{At}$. A finite set of sentences from $\mathcal{L}^{asp}_{At}$ is called an extended logic program $P \subseteq \mathcal{L}^{asp}_{At}$.

The body consists of a set of literals $\mathcal{B}(r)^+ = \{L_0, \ldots, L_m\}$ and a set of default negated literals denoted by $\mathcal{B}(r)^- = \{L_{m+1}, \ldots, L_n\}$. Given this we can write a rule as

$$H(r) \leftarrow \mathcal{B}(r)^+, not\ \mathcal{B}(r)^-.$$

If $\mathcal{B}(r) = \emptyset$ we call r a fact. A state S is a set of literals that does not contain any complementary literals L and $\neg L$. A state S is a model of a program P if for all $r \in P$ if $\mathcal{B}(r)^+ \subseteq S$ and $\mathcal{B}(r)^- \cap S = \emptyset$ then $H(r) \cap S \neq \emptyset$. The reduct P^S of a program P relative to a set S of literals is defined as

$$P^S = \{H(r) \leftarrow \mathcal{B}^+(r) \mid r \in P, \mathcal{B}^-(r) \cap S = \emptyset\}.$$

An answer set of a program P is a state S that is a minimal model of P^S. The set of all answer sets of P is denoted by $AS(P)$. Rule schemas can use variables which we denote by x, y, z and _ for the anonymous variable [12].

Two different inference relations are defined based on answer sets. An extended logic program P infers a literal L credulously, denoted by $P \models^c_{asp} L$, iff $L \in \cup AS(P)$. An extended logic program P infers a literal L skeptically, denoted by $P \models^s_{asp} L$, iff $L \in \cap AS(P)$. $P \models^\circ_{asp} S$ refers to any answer set inference relation. The answer set semantics defines the evaluation of queries $?L$ with $L \in Lit$ as *yes* if $P \models^\circ_{asp} L$, *no* if $P \models^\circ_{asp} \neg L$ and *unknown* else.

We also introduce nested logic programs (NLP) as defined in [20]. A NLP consists of rules that allow not quantified first order formulas to appear in the head and the body. In ASP the head can be a disjunction and as soon as the body has more than one element the elements are seen as conjunction. In NLP the head can also contain conjunctions and the body can also contain disjunctions. [20] also describes two default negations *not not a* and default negations in the head. These constructs are not used in this work and are therefore not described here.

The translation of FOL literals to ASP literals is trivial. A NLP rule with a conjunction in the head can be translated into multiple ASP rules:

$$\{a \wedge b \leftarrow c.\}_{NLP} \quad \{a \leftarrow c.\ b \leftarrow c.\}_{ASP}$$

and also a NLP rule containing a disjunction can be translated into multiple ASP rules:

$$\{a \leftarrow b \vee c.\}_{NLP} \quad \{a \leftarrow b.\ a \leftarrow c.\}_{ASP}$$

The translation of ASP rules into NLP rule is straightforward because NLP is a superset of ASP.

3 Conceptualization of Epistemic Agency for Secrecy

In this section we start with a brief introduction to our general concept of an epistemic agent. Then we start the elaboration of our epistemic agent model for secrecy preservation. Herby start with the general concept and then concretize the model and finally define a concrete model based on the well-known BDI agent model.

3.1 General Agent Model

The central idea of our framework is that of a widely knowledge based, i. e. epistemic, agent. All processes of an agent should be based on the information available to it and the beliefs it forms thereupon by means of reasoning, formalized by its belief-operator. This contrasts our framework from other frameworks in which extra-logical components which are not available for reasoning control the agents behavior.

Definition 1 (Agent)**.** *An* agent *is a tuple* $(\mathcal{K}, \xi)$ *comprising of an epistemic state* $\mathcal{K}$ *and a functional component* ξ*. The set of all agents in the system is denoted by* $\mathfrak{A}$*.*

This notion fixes the very basic structure of an epistemic agent as illustrated in Figure 3. However, it hardly constrains the agent's type or realization. Any meaningful agent model has some data component and one input or output channel which alters the data component by some function. The data component is often a simple form of some memory for state variables or instructions. We do require that the data component is an epistemic one. That means, that it is based on a logic representation of information which comes along with reasoning capabilities which allow to infer more information based on the given representation. The assumptions we make here about the structure of an epistemic state are satisfied by virtually all formalisms used for knowledge representation.

The composition of an agent of an epistemic and a functional component is reflected by the plug-in architecture of the ANGERONA framework. It contains two independent plug-ins for the epistemic and the functional component respectively. The former is based on the TWEETY library for knowledge representation [30] which provides a rich framework and many plug-ins for knowledge representation formalisms. This allows to

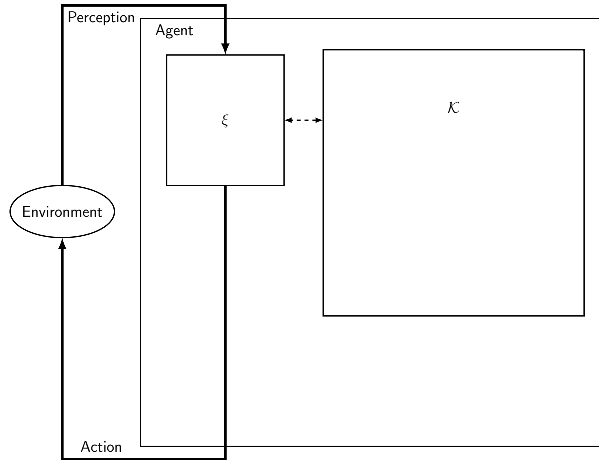

Figure 1: Basic structure of an epistemic agent

use any formalism already implemented, to develop our own, and to use those that might be implemented by others in the future.

In the following we concretize both components conceptually. That is, we develop a general framework for knowledge based agents with an extension towards secrecy preservation. For the functional component we define an abstract instance first. Then we develop the basic instance towards a BDI-based Agent model for secrecy preservation.

3.2 Epistemic Component for Secrecy

In this section we present an epistemic characterization of secrecy for autonomous agents in multiagent systems. We base our definition of secrecy on the view of a single agent that wants to maintain secrecy while interacting with other agents and its environment. It has incomplete and uncertain information. Thus it has to rely on its plausible but defeasible beliefs which result from its current epistemic state, and has to handle the dynamics of beliefs and secrets. As motivated above, an agent needs a definition of secrets that allows to define secrets individually with respect to each of its fellow agents. Moreover, each secret can vary in strength, that is, the agent wants to keep some information more secret than other.

To be more precise we consider scenarios having all of the following properties:

E1 subjective perspective of an autonomous resource-bound agent

E2 incomplete and uncertain information

E3 direct (communication) and indirect (effects in the environment) interaction with other agents

E4 dynamic, non-deterministic, and partially observable environment

With respect to the design of the agents we demand the following features:

A1 knowledge-based with logic-based epistemic state,

A2 inference operators that account for uncertainty,

A3 update operators for beliefs about the world, other agents, and secrets,

A4 resource-bounded reasoning about beliefs, changes and actions,

A5 goal directed, proactive behavior and planning.

We elaborate an agent model showing the features A1 to A5. For a comprehensible representation of main aspect of the secrecy scenario it is helpful to focus on the communication between two agents, the modeled agent which wants to defend its secrets, denoted by $\mathcal{D}$, from a potentially attacking agent, denoted by $\mathcal{A}$. For the illustration of our approach we use the following running example.

Example 1 (Strike Committee Meeting)**.** *Consider an employee* emp *working in a company for his boss* boss*. He wants to attend a strike committee meeting (scm) next week and has to ask his boss for a day off in order to attend. It is general knowledge that the agent* boss *puts every agent who attends the scm on her blacklist of employees to be fired next. Therefore the agent* emp *wants to keep its attendance to the scm secret.*

The epistemic state of an agent contains a complex representation of the agent's current belief state which might contain extra-logical information. Our concepts and notions of secrecy are independent of the used formalism we assume but we assume an underlying base-language $\mathcal{L}$, e. g. propositional logic, which the representation of the epistemic state makes use of. Belief operators can be applied to produce the set of currently held beliefs, the *belief set*, expressed in a language based on the same base language. The basic setup is illustrated in Figure 2. Three types of languages based on $\mathcal{L}$ are used, $\mathcal{L}_V$ for the view of the agent on other agents and its environment, $\mathcal{L}_S$ to define secrets and $\mathcal{L}_{BS}$ for the belief set of some view. This way we allow for formalisms that use different languages such as answer set programming [12], argumentation systems [4], or conditionals [28].

Definition 2 (Epistemic State)**.** *Let $\mathcal{D} \in \mathfrak{A}$ be some agent. The epistemic state of $\mathcal{D}$ is denoted by $\mathcal{K}_\mathcal{D}$. The set of all possible epistemic states of the agents in $\mathfrak{A}$ is denoted by Ω. Agent $\mathcal{D}$'s view on the world is given by $V_W(\mathcal{K}_\mathcal{D})$ and expressed in the language $\mathcal{L}_V$.The view agent $\mathcal{D}$ has on the view on the environment of agent $\mathcal{Y} \in \mathfrak{A}$ is given by $V_\mathcal{Y}(\mathcal{K}_\mathcal{D}) \subseteq \mathcal{L}_V$. $\mathcal{S}(\mathcal{K}_\mathcal{D})$ returns the secrets expressed in the language $\mathcal{L}_S$ of an agent with epistemic state $\mathcal{K}_\mathcal{D}$.*

To equip agents with reasoning facilities, a *belief operator* is defined which returns a set of beliefs given a belief base which represents the view on the world of an agent.

Definition 3 (Belief Operator)**.** *A belief operator $Bel : \mathcal{L}_V \to \mathcal{L}_{BS}$ is a function from the view language to the belief set language such that for a given view V we get $Bel(V) \subseteq \mathcal{L}_{BS}$. It satisfies the following two properties:*

- *Consistency: There does not exist $\phi \in \mathcal{L}$ and $W \in \mathcal{L}_V$ such that $\phi \in Bel(W)$ and $\neg\phi \in Bel(W)$.*

- *Propositional Consequence: For all $\phi, \psi \in \mathcal{L}$ such that ψ is a propositional consequence of ϕ, i.e., $\phi \vdash_{pl} \psi$, for all $W \in \mathcal{L}_V$ it holds*
  $\phi \in Bel(W)$ implies $\psi \in Bel(W)$.

The belief operator determines the currently held beliefs of the agent given some view. In a simple answer set programming setting beliefs are represented by a set of literals and a view by a logic program. The framework also allows for first order or modal logic, conditional information or defeasible rules, possibly augmented by some preference information. The belief operator determines how uncertainty of information is handled.

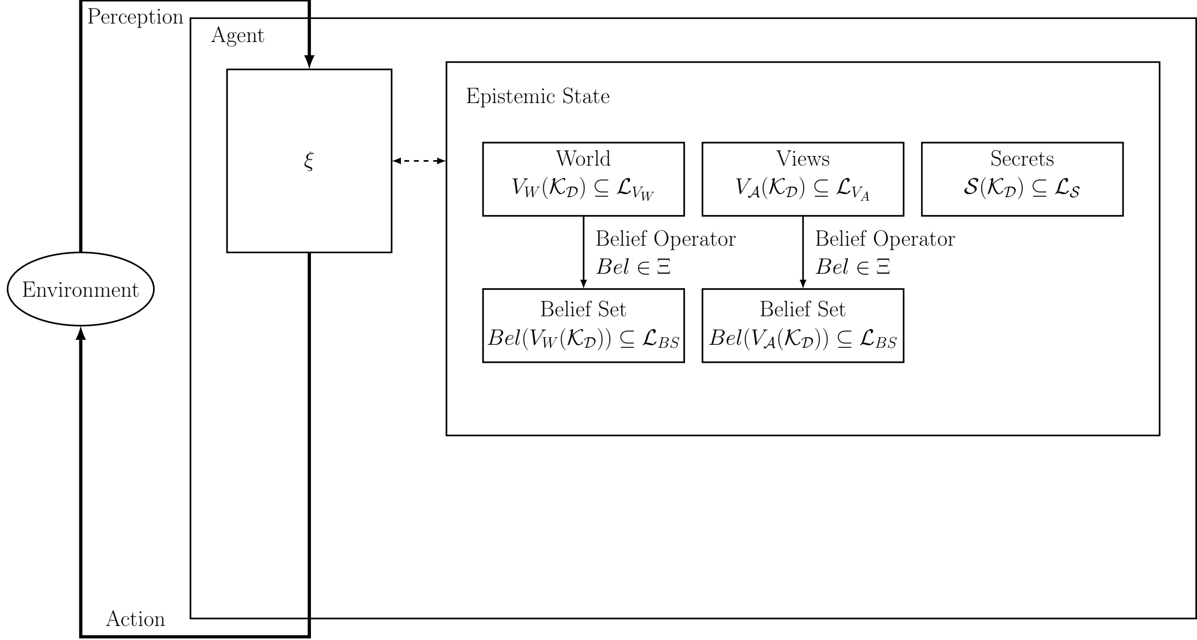

Figure 2: Basic Epistemic Agent

Given an epistemic state with incomplete and uncertain information various inferences are possible which are more or less credible. Different belief operators can be used which differ in the strength of belief associated to the resulting beliefs. We assume a structured family of belief operators from which an operator can be chosen.

Definition 4 (Belief-Operator Family)**.** *A* Belief-Operator Family *is a complete lattice $(\Xi, \preceq_\Xi\ )$ consisting of a set of Ξ of belief operators and a partial order $\preceq_\Xi$ on Ξ, We assume that the ignorant operator $Bel_\emptyset(V) = \emptyset$ is in any familiy, i. e. $Bel_\emptyset \in \Xi$. If $Bel \preceq_\Xi Bel'$, then for all $V \in \mathcal{L}_V$ it holds that $Bel(V) \subseteq Bel'(V)$. We read $Bel \preceq_\Xi Bel'$ as Bel' is at least as credulous as Bel.*

The notion of a belief-operator family captures a wide range of formalisms from purely qualitative ones to fully quantitive ones. We assume that for any two operators $Bel_1, Bel_2 \in \Xi$ such that $Bel_1 \preceq_\Xi Bel_2$ the information in the belief set of Bel_1 are at least as credible as those of Bel_2.

A basic and well-known family of operators would be $\Xi_a = \{Bel_{skep}, Bel_{cred}\}$ consisting of a skeptical and a credulous belief operator for some formalism. The order on these

operators can be based on the subset relation, i. e. $Bel_i \preceq_\Xi Bel_j$ if and only if $Bel_i(V) \subseteq Bel_j(V)$ for all $V \subseteq \mathcal{L}_V$. Assuming that an agent believes credulously everything that it believes skeptically we get $Bel_{skep} \preceq_\Xi Bel_{cred}$.

A richer hierarchy is given by quantitive approaches as by probabilistic formalisms where a family of operators is given by $\Xi_b = \{Bel_x \mid x \in [0, 1]\}$. Each Bel_x includes every sentence which is believed with probability $\geq x$ and $Bel_x \preceq_\Xi Bel_y$ if and only if $x \geq y$.

Based on this general agent model we define our notion of secrecy. In realistic settings the information to be kept secret is neither global, nor uniform, nor static. Secrets are not global in their content as an agent has different secrets with respect to different agents.

Example 2 (*strike committee meeting*)**.** *In our example, the employee wants to keep his attendance to the scm secret from his boss but not from other employees that want to attend the meeting.*

They are also not uniform with respect to their strength. That is, an agent wants to keep some information more secret than other. These differences in strength of secrets arise naturally from the value of the secret information to a specific agent. The value of secret information depends on the severeness of the negative effects, or the cost, for the agent resulting from disclosure of the secret information. These costs can differ widely and consequently the agent is interested in not revealing secret information to different degrees. These degrees of strength are expressed by assigning belief operators to the information.

Example 3 (*strike committee meeting*)**.** *The employee does not even want his boss to be suspicious about him attending the scm (secrecy with respect to a credulous belief operator). He also does not want that other employees that are against the strike to know that he attends the scm. However, with respect to the latter he considers it sufficient that they do not know for sure that he attends (secrecy with respect to a skeptical belief operator).*

Secrets are also not static, they arise, change and disappear during runtime of an agent such that it has to be able to handle these changes adequately.

Example 4 (*strike committee meeting*)**.** *If the employee realizes that his boss overheard his phone call with the strike committee he should give up his corresponding secret since it has been revealed already.*

Apart from being aware of its secrets at any time, an agent has to act such that it avoids revealing its secrets. It shall only reveal secret information if it considers it necessary in order to achieve its goals; which depends on the strength of the secret and the utility of the goals. That is, an agent has to take its secrets into account while acting and, moreover, while planning its intended course of action.

These considerations lead to the following formulation of properties of secrecy:

S1 secrets can be specific with respect to an agent

S2 secrets can vary in strength

S3 secrets can change over time, they can be adopted, strengthened, weakened or dropped

S4 secrecy accounts for uncertainty about beliefs of other agents, state of the world, effects of actions

Next we give a notion of secrets satisfying the properties given above. Most importantly, the information to be kept secret has to be expressed in the belief set language. Moreover, the agent from which the information has to be kept secret has to be defined and lastly the strength of the secret has to be expressed. Note that by assuming a more credulous belief operator of the attacker leads to a stronger protection of secret information of the defender with respect to the attacker. That is, if the defender reveals some information a credulous attacker might infer some secret information while a skeptical one with the same information might not. In the former case the defender should not have revealed the information. That is why we can specify the strength of a secret by the selection of a belief operator under which it should be protected. Hence in our framework secrets are triples specifying the information to be kept secret, the belief operator to use and the agent towards which the agent holds the secret.

Definition 5 (Secrets)**.** *A* secret *is a triple* $(\Phi, Bel, \mathcal{A})$ *which consists of a formula* $\Phi \in \mathcal{L}_{BS}$*, a belief operator* $Bel \in \Xi$ *and an agent identifier* $\mathcal{A} \in \mathfrak{A}$*. The set of secrets of agent* $\mathcal{D}$ *is denoted by* $\mathcal{S}(\mathcal{K}_\mathcal{D})$*. The set of all such triples defines the language* $\mathcal{L}_S$*.*

The semantics of a secret is that if agent $\mathcal{D}$ holds the secret $(\Phi, Bel, \mathcal{A}) \in \mathcal{S}(\mathcal{K}_\mathcal{D})$, it does not want that agent $\mathcal{A}$ believes Φ by use of the belief operator Bel, i. e. $\Phi \notin Bel(V_\mathcal{A}(\mathcal{K}_\mathcal{D}))$. Since agents only have local views, secrecy is based on the subjective view $V_\mathcal{A}(\mathcal{K}_\mathcal{D})$ agent $\mathcal{D}$ has on the beliefs of another agent $\mathcal{A}$.

Example 5 (*strike committee meeting*)**.** *We can formalize the secrets of the employee as* $\mathcal{S}(\text{emp}) = \{(\mathsf{attend_scm}, Bel_{cred}, \text{boss})\}$ *using the proposition* attend_scm *with the obvious meaning.*

Definition 6 (Safe Epistemic State)**.** *An epistemic state* $\mathcal{K}_D$ *is* safe *if and only if it holds that* $\Phi \notin Bel(V_\mathcal{A}(\mathcal{K}_\mathcal{D}))$ *for all* $(\Phi, Bel, \mathcal{A}) \in \mathcal{S}(\mathcal{K}_\mathcal{D})$*. We denote the set of all possible safe epistemic states by* $\Lambda \subseteq \Omega$*.*

The condition $\Phi \notin Bel(V_\mathcal{A}(\mathcal{K}_\mathcal{D}))$ in the definition of a safe state refers to the world view of an attacker maintained by $\mathcal{D}$.

3.3 Functional Component for Secrecy

For an agent, the dynamics of an environment are reflected by the perceptions it receives and influenced by actions it performs. The functional component of an agent deals with the incoming perceptions, updates the current epistemic state accordingly, and determines the next action to be performed.

3.3.1 Basic Agent Cycle

For the general model we use a functional component ξ consisting of a change operator $\circ$ and an action operator act, i. e. $\xi = (\circ, \mathsf{act})$. We assume a set of possible actions Act and a set of possible perceptions Per. . Since both, actions and perceptions imply epistemic changes, actions and perceptions are generalized to pieces of information $\tau \in \Gamma = Act \cup Per$.

In each agent-cycle the next action $a \in Act$ of an agent is determined by the current epistemic state $\mathcal{K} \in \Omega$ such that an agent is modeled by an action operator $\mathsf{act} : \Omega \to Act$. Each agent $\mathcal{D} \in \mathfrak{A}$ is assigned a specific action operator $\mathsf{act}_{\mathcal{D}}$. We denote the set of all action operators by $\mathbb{A}$. Updates to an epistemic state that are implied due to the execution of an action or the incorporation of a perception are realized by a belief update operator.

Definition 7 (Change Operator). *We assume an operator $\circ$ which changes an epistemic state $\mathcal{K} \in \Omega$ given some information $\tau \in Act \cup Per$, such that $\mathcal{K} \circ \tau = \mathcal{K}'$ with $\mathcal{K}' \in \Omega$.*

The concept of the change operator taking epistemic states and new information is very versatile. A belief change operator has to consider the type of epistemic state, adequate operators for actions and for perceptions, different types of sequences of pieces of information, the estimated effects on the views on other agents and the dynamics of secrets. These tasks call for different types of operations, e. g, update and revision operators [15, 19]. Here we use the term change operator as a general one and abstract from these types and assume the proposed operator to deal with these appropriately.

Up to this point we defined an epistemic state and a safety condition on it with respect to secrecy for a fixed point in time. We want to define secrecy of an autonomous agent in a dynamic environment. An agent preserves secrecy if it never performs any action that leads to an unsafe epistemic state. An agent starts with an initial epistemic state which is changed by perceptions from the environment and the agents actions. The new epistemic state is then given by the change of its current state by the information it gets from its environment in form of a perception and by the information which action it has performed. For an agent $\mathcal{D}$ with action function $\mathsf{act}_{\mathcal{D}}$ and some perception $p \in Per$ an agent cycle results in a new epistemic state determined by $\mathcal{K}_{\mathcal{D}} \circ p \circ \mathsf{act}_{\mathcal{D}}(\mathcal{K}_{\mathcal{D}} \circ p)$. The set of all possible successive epistemic states of agent $\mathcal{D}$ is determined by the set of initial epistemic states $\Lambda^0_{\mathcal{D}}$ and all respective successor states for all possible perceptions and corresponding actions of $\mathcal{D}$. i. e.

$$\begin{aligned}\Omega_{\circ,\mathsf{act}}(\Lambda^0_{\mathcal{D}}, Per) &= \{\mathcal{K} \mid \mathcal{K} = \mathcal{K}^0 \circ p_0 \circ \mathsf{act}_{\mathcal{D}}(\mathcal{K}^0 \circ p_0) \circ \ldots \\ &\quad , p_0, \ldots, p_i \subseteq Per, i \in \mathbb{N}_0, \mathcal{K}_0 \in \Lambda^0_{\mathcal{D}}\}.\end{aligned}$$

An agent preserves secrecy if its action and change operators are such that no unsafe epistemic state can be reached by any possible progression of epistemic states.

Definition 8 (Secrecy-preserving Agent). *Let $\mathcal{D} = (\mathcal{K}_{\mathcal{D}}, (\circ_{\mathcal{D}}, act_{\mathcal{D}})) \in \mathfrak{A}$ be an agent. Let $\Lambda^0_{\mathcal{D}}$ be a set of safe initial epistemic states and Per be a set of perceptions. We call $\mathcal{D}$* secrecy *preserving with respect to $\Lambda^0_{\mathcal{D}}$ and Per if and only if for all $\mathcal{K}_{\mathcal{D}} \in \Omega_{\circ,\mathsf{act}}(\Lambda^0_{\mathcal{D}}, Per)$ it holds that $\mathcal{K}_{\mathcal{D}}$ is safe.*

We require an agent with an initial safe epistemic state to act such that secrecy is not violated for all possible perceptions it might receive. That is, we formulate secrecy mainly as a requirement on the operator *act*. The determination of the effects of perceptions and actions, however, is dependent on the agents change operator such that the latter sets the conditions for the *act* operator. Therefore we discuss desirable properties of change operators in the next section.

3.3.2 Belief Change Operations and Secrecy

Secrecy preservation as of Definition 8 is heavily dependent on the epistemic state of the agent since we consider subjective secrecy from the respective agent's perspective. In a dynamic setting an agent has to adapt its epistemic state and thereby also its view on other agents and its secrets. This makes the notion of secrecy dependent on the agent's way of information processing and adaptation of its own beliefs. The properties of the change operator are hence significant for the resulting definition of secrecy preservation. To make the importance of this operator clear, a completely ignorant agent would never subjectively violate secrecy as it would ignore the violation of secrecy. We discuss desirable properties of an change operator and their effect on the resulting notion of secrecy.

The change operator changes the world view, agent views, and the secrets of an agent for incoming perceptions and actions. The main difference between perceptions and actions is that perceptions represent actions performed by other agents while actions represent the actions the agent under consideration has performed. Both types of inputs thus have different semantics and lead to different changes. Moreover, both types might not be represented in the same language as the the belief components of the agent. Classically belief change operators, however, satisfy categorical matching, i.ė. domain and codomain are equal. To facilitate the use of classic belief change operators which are well developed we split our change operator into sub-operations. For the interpretation of the input and translation to the language of the respective belief component we introduce translation operators

$$t_W : \Gamma \rightarrow \mathcal{L}_W \text{ and } t_V : \mathfrak{A} \times \Gamma \rightarrow \mathcal{L}_V.$$

Example 6. *A perception might represent an act of communication between agents and comprise of information about the sender of the information, the addressees, a timestamp, and some logical content. The translation function has to process this complex information into some sentence or set of sentences in the target language.*

The result of the translation is then used to change the respective component by use of specific revision operators

$$*_{\mathcal{D}} : \mathcal{L}_W \rightarrow \mathcal{L}_W \text{ and } *_{\mathcal{A}} : \mathcal{L}_V \rightarrow \mathcal{L}_V.$$

The latter operator reflects the revision strategy of agent $\mathcal{A}$. Secrets are updated on the basis of the agent's updated beliefs and views such that the update operator for secrets is dependent on these as well as on the incoming information, i. e.

$$*_S : \mathcal{L}_S \times \mathcal{L}_W \times \mathcal{L}_V \times \Gamma \rightarrow \mathcal{L}_S.$$

We assume that we can define the changes of the $\circ$ operator to the components by suboperations. Formally we postulate that the effect of $\circ$ can be expressed by means of the component specific operators, such that

$$V_W(\mathcal{K} \circ \tau) = V_W(\mathcal{K}) *_{\mathcal{D}} t_W(\tau), \quad (1)$$

$$V_{\mathcal{A}}(\mathcal{K} \circ \tau) = V_{\mathcal{A}}(\mathcal{K}) *_{\mathcal{A}} t_V(\mathcal{A}, \tau) \text{ and} \quad (2)$$

$$\mathcal{S}(\mathcal{K} \circ \tau) = \mathcal{S}(\mathcal{K}) *_{\mathcal{S}} (V_W(\mathcal{K} \circ \tau), V(\mathcal{K} \circ \tau), \tau). \quad (3)$$

For secrecy preservation it is necessary that the agent does not give up any secrets upon reflecting its own actions since it would be able to perform arbitrary actions without violating secrecy by abandoning its secrets. Thus, the agent must not be able to preserve a safe epistemic state by modifying its secrets for which we propose the following property.

Secrets-Invariance$_\circ$ If $\tau \in Act$ then $\mathcal{S}(\mathcal{K} \circ \tau) = \mathcal{S}(\mathcal{K})$

The *Secrets-Invariance* property is restricted to inputs that are actions. These actions are those of the agent itself and perceptions reflect changes in the environment or actions of other agents. For the latter the postulate should not hold. That is, an agent should not be able to ignore the fact that a secret has been revealed due to changes in the environment or actions of other agents. This is expressed in the following property.

Acknowledgment$_\circ$ If $\tau \in Per$ then $\mathcal{K} \circ \tau$ is safe.

The changes to the set of secrets in order to achieve a safe epistemic state should be minimal. That is, a secret should not be weakened without a reason, i. e. it it is violated, it should not be strengthened and it should be weakened minimally.

Min-Secrecy-Weakening$_\circ$ If $(\Phi, Bel, \mathcal{A}) \in \mathcal{S}(\mathcal{K})$ and $(\Phi, Bel', \mathcal{A}) \in \mathcal{S}(\mathcal{K} \circ \tau)$ with $Bel \neq Bel'$ then $\Phi \in Bel(V_{\mathcal{A}}(\mathcal{K}_{\mathcal{D}} \circ \tau))$ and there is no Bel'' such that $\Phi \notin Bel''(V_{\mathcal{A}}(\mathcal{K}_{\mathcal{D}} \circ \tau))$ with $Bel'' \succ Bel'$.

Another secrecy relevant property of belief change arises from the changes to views of other agents. An agent should not be able to preserve secrecy by ignoring the effects of its own actions on the beliefs of potentially attacking agents. In particular the information for some agent $\mathcal{A}$ contained in an action of $\mathcal{D}$ should be incorporated into $\mathcal{D}$'s view on $\mathcal{A}$. This is formulated by the next property.

Awareness$_\circ$ If $\tau \in Act$ then $t_W(\tau) \in V_W(\mathcal{K} \circ \tau)$ and for each $\mathcal{A} \in \mathfrak{A}$ $t_V(\mathcal{A}, \tau) \in V_{\mathcal{A}}(\mathcal{K}_{\mathcal{D}} \circ \tau)$.

There might very well be actions which are not visible to all agents and therefore should also not affect the view on all agents.

Example 7. *If agent $\mathcal{D}$ communicating privately with some agent $\mathcal{A}$ it should change its view on $\mathcal{A}$ but not its view on other agents.*

This is achieved by use of appropriate translation operators which select the relevant information for each agent. For agents that are not affected by the information the transformation function returns the empty set.

A more in-depth analysis of update operations depends on the actual formalization of actions and perceptions and the resulting transformation operators and their properties. Given the formalization given so far we can state that it satisfies the properties defined above.

Observation 1. *The satisfaction of* Secrets-Invariance$_\circ$, Acknowledgment$_\circ$, Min-Secrecy-Weakening$_\circ$ *and* Awareness$_\circ$ *of the belief change operator of an agent corresponds to the satisfaction of (P1), (P5) and (S3).*

Presupposing appropriate translation operators satisfying Awareness$_\circ$ means that the corresponding revision operator $*$ of has to satisfy success.

Success$_*$ $\Phi \in V * \Phi$

Proposition 1. *Given a belief change operator $\circ_\mathcal{B}$ constructed as defined in (*). If the corresponding inner change operator $*$ satisfies* Success $\Phi \in V * \Phi$, *then $\circ_\mathcal{B}$ satisfies* Awareness$_{\circ_\mathcal{B}}$.

Proof. By the construction given in (1)-(3) we have that $V_W(\mathcal{K} \circ \tau) = V_W * t_W(\tau)$ and $V_\mathcal{A}(\mathcal{K} \circ \tau) = V_{\mathcal{D},\mathcal{A}} * t_V(\mathcal{A}, \tau)$ such that $t_W(\tau) \in V_W(\mathcal{K} \circ \tau)$ if and only if $t_W(\tau) \in V_W * t_W(\tau)$ and $t_V(\tau, \mathcal{A}) \in V_\mathcal{A}(\mathcal{K} \circ \tau)$ if and only if $t_V(\tau, \mathcal{A}) \in V_{\mathcal{D},\mathcal{A}} * t_V(\mathcal{A}, \tau)$. Given that $*$ satisfies Success it holds that $t_W(\tau) \in V_W * t_W(\tau)$ and $t_V(\tau, \mathcal{A}) \in V_{\mathcal{D},\mathcal{A}} * t_V(\mathcal{A}, \tau)$ and therefore also $t_W(\tau) \in V_W(\mathcal{K} \circ \tau)$ and $t_V(\tau, \mathcal{A}) \in V_\mathcal{A}(\mathcal{K} \circ \tau)$ such that *Awareness$_\circ$* is satisfied. □

In the following we elaborate on the functional component of the agent model and its aspects for epistemic secrecy.

3.3.3 BDI Model for Epistemic Secrecy

The framework of epistemic secrecy presented thus far focuses on the definition of epistemic secrecy and leaves the functional component of an agent abstract. In the following we instantiate the functional component with a BDI agent model. We make an agent's desires and intentions explicit parts of its epistemic state $\mathcal{K}$, which was only partially defined before. Correspondingly, the update operator $\circ$ does not only realize the update of the belief base, views and secrets of the agent but also updates its desires and intentions.

Definition 9 (BDI-State)**.** *We define a* BDI-State *as an epistemic state $\mathcal{K} = \langle\{V_W, \mathcal{V}, \mathcal{S}\}, \Delta, \mathcal{I}, \mathcal{KH}\rangle$. It consists of a world view V_W, a set of secrets $\mathcal{S}$, a set of agent views $\mathcal{V} = \{\mathcal{V}_\mathcal{Y} \mid \mathcal{Y} \in \mathfrak{A}\}$, a set of desires Δ, a set of intentions $\mathcal{I}$, and the agent's know-how $\mathcal{KH}$. We refer to the first component as the beliefs $\mathcal{B}(\mathcal{K})$, or $\mathcal{B}$ if un-ambigouos, of the agent and denote the set of all possible beliefs by $\mathfrak{B}$. It must hold that $V_W(\mathcal{B}) = V_W(\mathcal{K}) = V_W \subseteq \mathcal{L}_V$, $V_\mathcal{Y}(\mathcal{B}) = V_\mathcal{Y}(\mathcal{K}) = V_\mathcal{Y} \subseteq \mathcal{L}_V$ and $\mathcal{S}(\mathcal{K}_\mathcal{D}) \subseteq \mathcal{L}_S$.*

Hence a BDI-State is safe if and only if its belief component is safe. In the following we elaborate the $\Delta$and $\mathcal{I}$ components of a BDI-State and generalize the notion of a secrecy-preserving agent to account for the agent's current state of desires and intentions.

We denote $\mathfrak{D}$ as the set of all desires or possible goals an agent may have. These are expressible in the belief-set language $\mathcal{L}_{BS}$ where a desire is some element of the language that an agent wants to be satisfied in its beliefs. An agent maintains a subset $\Delta \subseteq \mathfrak{D} \subseteq \mathcal{L}_{BS}$ which resembles its current set of desires. Desires can be dynamically generated and motivated by the agents motives. For the sake of simplicity of representation of the topic of this paper we leave this aspect out of consideration and confer to [18] for details. Every agent has at each moment one selected desire that is the goal it currently pursues, denoted by $\mathit{selected}(\Delta)$. For the sake of simplicity of presentation we assume here that an agent has only one (top-level) goal at the same time. An extension to multiple goals would be easily possible.

An agent also maintains a set of intentions $\mathcal{I} \subseteq \mathfrak{I}$ where $\mathfrak{I} \subseteq \mathcal{L}_{BS}$ denotes the set of all possible intentions. Intentions describe the aims of the agent that it currently pursues in order to fulfill its selected goal. These are more concrete than desires. Every time an agent selects a new goal from its desires, this goal becomes its top-level intention. At any time the set $\mathcal{I}$ correlates directly to the currently pursued goal $\mathit{selected}(\Delta)$ and contains the next subgoals the agent wants to be satisfied in order to fulfill it. A subset of the intentions are atomic in the sense that they are directly satisfiable by the execution of one single action a. We assume here that this action is unique[1]. We denote this set of *atomic intentions* by $\mathcal{I} \subseteq \mathfrak{I}$. For each atomic intention $I \in \mathsf{AtInt}$ we denote the corresponding action by $\alpha(I)$. For a set of atomic intentions $\mathcal{I}' \subseteq \mathsf{AtInt}$ we define $\alpha(\mathcal{I}') = \{\alpha(I) \mid I \in \mathcal{I}'\}$ and for a sequence $\alpha((I_1, \ldots, I_n)) = (\alpha(I_1), \ldots, \alpha(I_n))$. The set of all actions corresponding to the set of atomic intentions are the *capabilities* $\mathcal{C} = \alpha(\mathsf{AtInt})$ of the agent.

Example 8. *An exemplary atomic intention is door_opened and the corresponding action is $\alpha(\mathit{door_opened}) = \mathit{open_door}$. Consequently performing the action open_door leads to the satisfaction of the intention door_opened.*

A stack of intentions is modeled by assuming a linear order $\preceq$ on $\mathcal{I}$ which reflects the order of executability, i. e. if $I \preceq I'$ then I has to be executed before I'. The order dependence of subgoals is often partial and the agent has to decide the order of incomparable subgoals. By assuming a linear order we assume that this decision is already reflected in the order. Given an intention I we denote by $\mathsf{pre}(I)$ the sequence of intentions that have to be satisfied for the satisfaction of I. More formally, we demand that for $I \in \mathcal{I}$, $\mathsf{pre}(I) = (I_1, \ldots, I_n, I)$ is such that $I_i \preceq I_{i+1}$ for $1 \leq i < n$ and $\{I_1, \ldots, I_n\} = \{J \in \mathcal{I} \mid J \preceq I\}$. This representation can be extended to multiple goals by multiple stacks.

We define an operator which reflects the changes of the agent's beliefs given some intention. This operator can be used by the agent to simulate the epistemic effects of the satisfaction of intentions prior to committing to them which is crucial for secrecy preservation. Since intentions are expressed in that same base language as the beliefs of an

[1]This is no real restriction since for each alternative action for an atomic intention a distinct atomic intention can be added.

agent, standard revision operators for the language framework can be used.

Definition 10. *The* revision by intentions operator *is an operator* $\diamond : \mathfrak{B} \times \mathfrak{I} \to \mathfrak{B}$ *which satisfies the following two properties:*
Success: *For each* $Bel \in \Xi$ *and* $I \in \mathfrak{I}$ *it holds that* $I \in Bel(\mathcal{B} \diamond I)$.
Consistency: *For each* $Bel \in \Xi$ *and* $I \in \mathfrak{I}$ *it holds that* $Bel(\mathcal{B} \diamond I)$ *is consistent.*

The revision by a sequence of intentions $(I_1, \ldots, I_n)$ *is defined as the iterated revision* $\mathcal{B} \diamond (I_1, \ldots, I_n) = (\ldots((\mathcal{B} \diamond I_1) \diamond I_2) \diamond \ldots \diamond I_n)$.

The revision by intentions operator reflects the changes of the satisfaction of an intention. The actual actions necessary to satisfy the intention and effects thereof are not considered but only the state of beliefs if the considered intention is satisfied.

For resource-bounded agents in highly dynamic environments it is seldom possible nor effective to generate complete plans [21, 10]. On the one hand the time available might not be sufficient to generate complete plans and on the other hand complete plans may be invalidated before completion by the changing environment. In fact, classic BDI agents do not perform look-ahead planning at all. The BDI model we present here is capable of look-ahead planning of varying degree of detail and reasoning about plans and action. We assume that each partial plan can be assigned a level of detail δ from an partially ordered set Δ.

Definition 11 (Plan)**.** *Let* I *be an intention,* Bel *a belief operator and* $\delta \in \Delta$ *an element of a complete lattice* $(\Delta, \preceq_\delta)$*. A* partial plan $\pi^{Bel}_{I,\delta}$ *for* I *with respect to* Bel *and level of detail* δ *is a sequence of intentions* $\pi^{Bel}_{I,\delta} = (I_1, \ldots, I_n) \subseteq \mathfrak{I}$ *such that* $I \in Bel(\mathcal{K} \diamond (I_1, \ldots, I_n))$ *where the certainty of the success of the plan is dependent on the belief operator* $Bel \in \Xi$.

The parameter δ denotes the level of detail of a plan given some measure of detail which might reflect the amount of steps that are planned ahead or the amount of consecutive atomic intentions. We assume that a *complete plan* that consists only of atomic intentions, i.e. $\pi^{Bel}_{I,\delta_\top} = (I_1, \ldots, I_n) \subseteq \mathsf{AtInt}$ is the plan with the highest level of detail $\delta_\top$. As noted above for simplicity of presentation we assume here that an agent has only one top-level intention *selected*(Δ) such that in every situation $\mathcal{I} = \{\pi^{Bel}_{selected(\Delta),\delta}\}$. For the general case we assume that the set of intentions can be partitioned into plans for a set of top-level intentions $\mathcal{I}^\top = \{I_1, \ldots, I_n\}$, i.e. $\mathcal{I} = \{\pi^{Bel}_{I_1,\delta_1}, \ldots, \pi^{Bel}_{I_n,\delta_n}\}$.

In the abstract epistemic model of secrecy the epistemic state $\mathcal{K}$ is revised by the update operator $\circ$ and the action of the agent is determined by the *act* operator based on the current epistemic state of the agent. An agent cycle is modeled by these two operators. In the following we develop an agent model and cycle which realize these abstract operators by means of a set of suboperations. The functional-component $\xi = (\circ, act)$ is implemented such that $\circ$ is implemented by the suboperations (belief update, opt, intention update,) and *act* is implemented by the execute operation. The operators for the functional component are defined as follows, whereby $\mathcal{P}(\xi)$ denotes the power-set of ξ. The operator

$$\mathsf{belief\ update} : \mathfrak{B} \times \Gamma \to \mathcal{P}(\mathfrak{B})$$

updates the beliefs of the agent. The operator

$$\mathsf{opt} : \mathfrak{B} \times \mathcal{P}(\mathfrak{D}) \times \mathcal{P}(\mathfrak{I}) \rightarrow \mathcal{P}(\mathfrak{D})$$

determines the current options of the agent given its current epistemic state and current intentions. Given its options the agent has to determine what it intents to do given its beliefs, desires and currently pursued intentions.

$$\mathsf{intention\ update} : \mathfrak{B} \times \mathcal{P}(\mathfrak{D}) \times \mathcal{P}(\mathfrak{I}) \rightarrow \mathcal{P}(\mathfrak{I})$$

The operator

$$\mathsf{execute} : \mathcal{P}(\mathfrak{I}) \rightarrow Act$$

returns an action that corresponds to a minimal atomic intention: $\mathsf{execute}(\mathcal{I}) = \alpha(min_{\preceq}(\mathcal{I}))$. An agent cycle consists of the successive execution of the belief update, opt, intention update and execute operators as illustrated in Figure 3. The abstract update operator $\circ$ up-

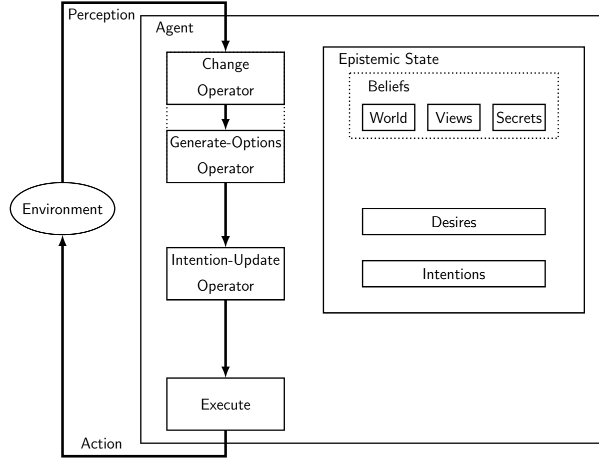

Figure 3: BDI agent model

dates the entire epistemic state and is realized in the agent model not only through the belief update operator but through the operators belief update, opt, and intention update. The general update operator $\circ$ in the BDI context updates the belief, desires and intentions of an agent. These suboperations are realized by the corresponding BDI operators presented above and summarized in Figure 4. The belief update operator follows the

```
1 Input: Epistemic State 𝒦, new information τ
2 Output: New Epistemic State 𝒦'
3 𝒦 ∘ τ
4         ℬ' := belief update(ℬ, τ)
5         Δ' := opt(ℬ', Δ, ℐ)
6         ℐ' := intention update(ℬ', Δ', ℐ)
7 return 𝒦' = ⟨ℬ', Δ', ℐ', 𝒦ℋ⟩
```

Figure 4: General Update Operator implementation

structure of the change operator given in Section 3.3.2 which is depicted in Figure 5. We

```
1 Input: Beliefs 𝓑, new information τ
2 Output: New Beliefs 𝓑'
3 belief update(𝓑, τ)
4         V'_W := V_W *_𝓓 t_W(τ)
5         𝓥' := ∅
6         for each V_𝓐 ∈ 𝓥
7                 𝓥' := 𝓥' ∪ V_𝓐 *_𝓐 t_V(𝓐, τ)
8         𝓢' := 𝓢(𝓚) *_𝓢 (V'_W, 𝓥', τ)
9 return 𝓑' := {V'_W, 𝓥, 𝓢'}
```

Figure 5: Belief Update implementation

assume that the respective operators satisfy *Constant-Secrets*$_\circ$*, Awareness*$_\circ$*, Success*$_*$*, Acknowledgment*$_\circ$. We already described the opt and intention update operators. The act operator is implemented by the execute operator such that $\mathsf{act}(\mathcal{K}) = \mathsf{execute}(\mathcal{I}(\mathcal{K}))$.

Our goal is to design a secrecy-preserving agent, i.e. an agent that does not perform any secrecy violating action. As argued before: the more closely secrecy preservation is integrated into the agent model the more options for utility optimization under secrecy constraints is possible. To this end we want to determine all parts of the BDI model and cycle that can contribute to secrecy preservation. In the first stage of an agent cycle the agent selects a goal to pursue. At this point the agent does not account for all effects of acting towards a goal and is not aware of the necessary actions, especially in dynamic environments. However, the agent can already check whether a goal violates secrecy in principle while deliberating on what to do. That is, it considers if the satisfaction of the goal in itself entails the revelation of a secret. Since this is not sufficient to guarantee preservation of secrecy the agent has to check for principally secrecy violating subgoals and for secrecy violating sequences of intentions, i. e. plans, in the means-end-reasoning phase.

We extend the definition of a safe epistemic state towards a safe BDI-state for a definition of a secrecy-preserving agent which lead to stricter notions of a safe state.

Definition 12 (Safe BDI-State)**.** *A BDI-state* $\mathcal{K} = \langle\{V_W, \mathcal{S}, \mathcal{V}\}, \Delta, \mathcal{I}, \mathcal{KH}\rangle$ *is* bdi-safe *iff* $\mathcal{B}$ *is safe and for all* $I \in \mathcal{I} : \mathcal{B} \diamond \mathsf{pre}(I)$ *is safe.*

The definition of a bdi-safe state assures that the sequence of actions up to the satisfaction of the intention no secrecy violating action is required for the satisfaction of the intention. This demands the agent to consider its current plans, which might be incomplete, to determine whether these violate secrecy. We can strengthen the notion of a bdi-safe state by requiring a *level of detail* of the current plans.

Definition 13 (δ-safe State)**.** *A BDI-state* $\mathcal{K}$ *with* $\mathcal{K} = \langle\{V_W, \mathcal{S}, \mathcal{V}\}, \Delta, \mathcal{I}, \mathcal{KH}\rangle$ *is* δ-safe *iff* $\mathcal{B}$ *is safe and* $\mathcal{I}$ *is a plan with detail of at least* δ *and for all* $I \in \mathcal{I}$ $\mathcal{B} \diamond \mathsf{pre}(I)$ *is safe.*

Preserving Secrecy The act operator is defined by the execute operator which solely depends on the current state of the intentions of the agent. If an agent is in a strictly safe BDI-State $\mathcal{K}$, then the satisfaction of all of its intentions in the planned order does

not violate secrecy. This holds in particular for any atomic intention. Since the action it performs directly corresponds to its first atomic intention $\mathsf{act}(\mathcal{K})$ does not cause violation of secrecy such that $\mathcal{K} \circ \mathsf{act}(\mathcal{K})$ is safe. If an agent maintains a strictly safe BDI-State, it adapts dynamically to its perceptions and update its beliefs and intentions accordingly such that $\mathsf{act}(\mathcal{K})$ cannot return an action which violates secrecy.

Definition 14. *An* intention update *operator is* δ-secrecy-aware *if for all* $\mathfrak{B}$, $\mathfrak{D}$, $\mathfrak{I}$: $\mathfrak{B} \circ \mathsf{intention\,update}(\mathfrak{B}, \mathfrak{D}, \mathfrak{I})$ *is* δ*-safe.*

Given the previous results we elaborate on the implementation of secrecy-aware intention update operators in the following. To ensure secrecy preservation the agent has to make sure that all subgoals and actions towards their realization do not violate secrecy. While doing so it also has to take changes in the environment into consideration. To this end a whole set of intentions, i. e. a plan, has to be considered when considering secrecy and not only a single (atomic) intention. For this check define an operator which is capable of simulating the effect of the satisfaction of intentions and whole sequences of intentions and to determine if confidentiality would be violated.

Definition 15 (Violates operator)**.** *Given a set $\mathcal{VD}$ of degrees of violation with a partial order $\preceq_{\mathcal{VD}}$, such that $(\mathcal{VD}, \preceq_{\mathcal{VD}})$ forms a complete lattice. The violates operator is a function* $\mathsf{violates} : \mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{I}) \to \mathcal{VD}$.

The violates operator can be used by other operators to simulate effects prior to commitment to an output. The operator allows for simulating the effects of arbitrary (ordered) sets of intentions and not only for atomic intentions or the corresponding actions respectively. Therefore the violates operator can be used by the intention update operator to check whether a high level intention necessarily violates secrecy.The earlier secrecy violation is determined the smaller is the risk of violating it and the less is the amount and cost of replanning. Note that it is easily possible to extend the violates operator to evaluating the severeness of secrecy violation. The general form of the violates operator determines a degree of violation rather than a binary evaluation of secrecy. This is useful for complex planning and notions of recovery safe agents. For any violates operator the strict preservation of secrecy corresponds to the degree of violation $\top = \sqcup(\mathcal{VD})$. A basic violates operator is a binary one.

Definition 16 (Violates operator)**.** *Let $\mathcal{B}$ be a beliefs component and $\mathcal{I}$ a set of intentions. Let*
$\mathcal{VD}_{binary} = (\{\textit{true, false}\}, \{(\textit{true, false}), (\textit{false, false}),$
$(\textit{true, true})\})$ *and* $\mathsf{violates}(\mathcal{B}, \mathcal{I}) = \begin{cases} \textit{true} & \textit{if } \mathcal{B} \circ \mathcal{I} \textit{ is safe} \\ \textit{false} & \textit{if } \mathcal{B} \circ \mathcal{I} \textit{ is not safe} \end{cases}$

In means-end reasoning a BDI agent typically has, besides its beliefs about the state of the world, structural beliefs, or know-how, in form of a plan library [38]. This plan library supplies subgoals to a given goal which, iteratively performed, generates a plan. Abstractly we formalize this by means of a *subgoal-generation operator* as illustrated in Figure 7. We implement this planing component by the an approach to know-how based on the one presented in [32, 17]. The know-how is a general approach that allows

for means-end reasoning similarly to many other approaches used in agent programming languages [8]. Unlike classic approaches it allows an agent to reason about its know-how on the same level as with any other beliefs. For secrecy preservation alternative, secrecy preserving, plans can be expressed in the agents know-how.

Definition 17 (Know-How Statement)**.** *A* know-how statement σ *is a tuple*

$$\sigma \quad = \quad (a, (s_1, \ldots, s_n), \{c_1, \ldots, c_m\})$$

with goals $a, s_1, \ldots, s_n \in \mathfrak{I}$, $a \notin \{s_1, \ldots, s_n\}$ and literals $c_1, \ldots, c_m$. The goal a is called the target*, the goals $s_1, \ldots, s_n$ are called the* sub-targets*, and the elements $c_1, \ldots, c_m$ are called the* conditions *of σ.*

Besides the belief base and the know-how base, a record describing the current state of the plan deliberation is needed to fully describe an agent's intentional state. Also to keep track of already failed plan deliberations we use a notion of intention tree similar to e. g. [29, 39] which generalizes the notion of an intention stack.

Definition 18 (Intention tree)**.** *An* intention tree $\mathcal{T}_\mathcal{I}$ *is a tree with two sorts of nodes with the root node being a goal and for every node K it holds that (a) iff K is a goal, then every child of K is a know-how statement σ with $goal(\sigma) = K$, (b) iff K is a know-how statement σ, then the (ordered) children of K are the subgoals of σ. Furthermore there is exactly one node, that is additionally labelled* "current"*, which is also returned by the operator* $\mathsf{curr}(\mathcal{T}_\mathcal{I})$*. Furthermore the empty tree $\mathcal{T}_\mathcal{I}^\emptyset$ is also an intention tree with* $\mathsf{curr}(\mathcal{T}_\mathcal{I}^\emptyset)$ *being undefined.*

Informally speaking, an intention tree $\mathcal{T}_\mathcal{I}$ captures the current state of the pursuit of a goal. The initial state of an intention tree is described by a root node R that is labeled with "*current*".

In Figure 6 we give the pseudo-code of the intention update operator based on know-how. For a complete operationalization of the basic know-how means-end reasoning procedure via state transition systems and an implementation in ASP we refer to [33]. The intention update can be divided into two operators. The deliberation operator updates the current goals of the agent based on its current desires, beliefs and intentions. Hereby the feasibility of desires and their priority as well as the commitment and reconsideration of current intentions have be handled adequately. Since intention reconsideration, and desire generation and prioritization is a rather complex task in itself, we do not go into details on these here. and confer to [26] and [18] respectively.

The means_end_reasoning operator determines all know-how statements for $\mathsf{curr}(\mathcal{I})$ whose conditions are satisfied (19), whose subgoals are consistent with the current set of intentions (20) and whose subgoals are not generally violating secrecy in the context of the current partial plan (21). If there are no know-how statements satisfying all conditions, then the $\mathsf{curr}(\mathcal{I})$ cannot be satisfied and it has to be backtracked to an appropriate point in the intention tree which is determined by a backtracking operator (9). Otherwise the resulting know-how statements are added as children of $\mathsf{curr}(\mathcal{I})$ (11) and on of them is selected (12). The subgoals of the selected know-how statement replace its intention (13).

```
1  intention update(B, Δ, I)
2     deliberation(B, Δ, I)
3     means_end_reasoning(B, I, δ)
4
5  means_end_reasoning(B, I, δ)
6  do
7     kh= subgoalgen(B, I, curr(T_I), δ)
8     if kh = ∅ then
9        backtrack
10    else
11       children(I) := kh
12       selected(kh) = select kh statement
13       I = I \ {I} ∪ {s_1, ..., s_n}
14       curr(T_I) = select non atomic element of I
15 until I has reached detail δ
16
17 subgoalgen(B, I, I, δ)
18    kh = get all kh statements for I
19    kh = kh \{I | {c_1, ..., c_m} ⊈ Bel(W * pre(I) \ I)}
20    kh = kh \{I | B * (pre(I) \ I) ∪ {s_1, ..., s_n} inconsistent}
21    kh = kh \{I | violates(B, pre(I) \ I) ∪ {s_1, ..., s_n}}
22 return kh
23
24 violates(B, I)
25    B_temp := B ◇ I
26    if B_temp is safe then
27       return false
28    else return true
```

Figure 6: The intention update operator

The curr mark is set to another non-atomic intention according to the planning strategy. These steps, (7) to (14), are repeated until the desired detail of planning is reached. The binary violates operator creates a temporary copy of the current beliefs of the agent and simulates the revision by the given set of intentions.

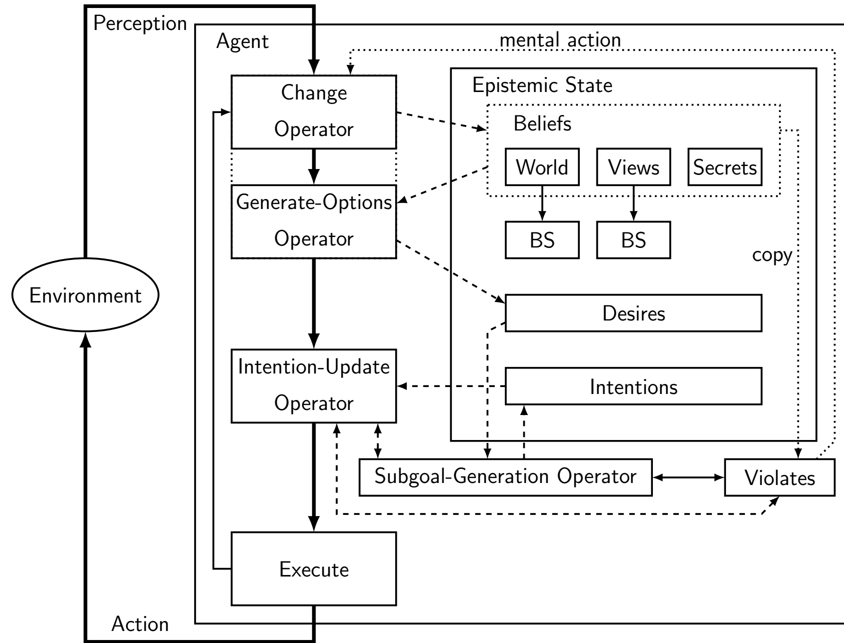

Figure 7: Agentmodel with Violates and Subgoal-Generation Operator

4 The Angerona Framework

The Angerona framework is a Java based framework for the implementation of multi-agent systems, with a strong focus on knowledge based agents. We can distinguish between the pure *agent framework* for the development of knowledge based agents and the *multiagent framework* which contains the infrastructure to run a multiagent system based on agents from the *agent framework*.

The architecture of the *agent framework* is based on the concepts described in the previous parts of this report. It has a similar structure such that the same levels of abstraction can be realized which makes for a very versatile and flexible framework. The resulting framework allows the development of knowledge based agents that can be based on a variety of logical formalisms. The functional component can be freely defined such that very different agent concepts can be implemented. The secrecy agent model that is described in Section 3.2 and Section 3.3 is implemented as an agent model in Angerona. That means the flow of the agent cylce is defined by ordering different *operation types*. The implementation of the operators in the agent cycle can be exchanged. Moreover, the ASP instantiation as knowledge representation mechanism is fully implemented.

The *multiagent framework* provides the infrastructure for the implementation and the integration of an environment simulation. It provides mechanisms for the communication between the agents and their interaction with the environment and it also provides a graphical user interface for visualizing the simulation, the agents with their knowledge and for controlling the simulation.

This section first describes the agent framework of Angerona in Section 4.1. In Section 4.2 we elaborate on the overall multiagent framework. Then the structure of the epistemic

and the functional component for the BDI Model for Epistemic Secrecy and the way these concepts are realized in Angerona are elaborated in Section 4.3 according to the concept descriptions of Section 3.2.

According to the different layers of abstraction and the resulting structure Angerona can be used on different levels. In this work we distinguish between three roles, which represent different perspectives on the Angerona framework and can be used to describe several use-cases for the framework.

- The *user* uses Angerona to perform simulations. She changes the configuration files and uses different plug-ins to customize her simulations. She does not changes or creates agent models or knowledge representation mechanisms.
- The *plug-in developer* implements plug-ins to extend the functionality provided by the Angerona framework. He can add new knowledge representation mechanisms or agent models. He does not change the Angerona core code.
- The *core developer* works on the Angerona core code to adapt the framework to make it more flexible and usable for external *plug-in developer* and *user*.

4.1 The Agent Framework

In Section 3.1 a general definition of a knowledge based agent is given in Definition 1. According to this, an agent comprises of $(\mathcal{K}, \xi)$: the functional component ξ and the epistemic state $\mathcal{K}$ which represents the knowledge of the agent.

In Angerona this general model is reflected by a plug-in architecture, which is build up of *plug-ins*, *modules*, *data components* and *operators*. A *plug-in* defines a category that can be extended, like a knowledge representation mechanism and consists of one or multiple *modules*. A *module* is either a *data component* or an *operator*. Both, *data components* and *operators* are explained in Section 4.1.1.

Implementation Detail. *Plug-ins are used to load code during runtime, for this the* Java *implemention uses the* Java Simple Plugin Framework (*JSPF*) *[24].* JSPF *heavily uses the* Java *features annotations and reflection to provide code contained in external jar files at runtime. Annotations allow to mark* Java *classes and interfaces with meta information which can be read using reflection. Reflection allows to get source information about classes during runtime. Getting the list of methods of a class is an example for reflection. For further details see [22, 23]. The plug-in architecture creates a clear modularization and allows us to easily exchange the implementation of the epistemic and functional components. Thus, a* plug-in developer *can change the behavior of the framework without changing the core framework code.*

Angerona uses an entity-component system [2] to describe it's agents, thereby Angerona differentiates between *operators*, which are functional components and *agent components*,

[2] <http://www.gamedev.net/page/resources/_/technical/game-programming/understanding-component-entity-systems-r3013

Module	Category	Functionality
Belief base	*KR*	A belief base with a specific knowledge representation mechanism
Reasoner	*KR*	Reasons the belief-set by using a belief base as input
Change	*KR*	Changes a belief base if the agent learns new information
Translator	*KR*	Translates perceptions to the language of the *KR*
Operator	*functional*	An operator of the functional component
Component	*data*	An extension of the agent model (additional data like motivations or know-how [32] for example)
Environment-Behavior	simulation	Changes the simulation process and the communication with external software
View	*GUI*	Adds an UI element that shows internal data

Table 1: Plugin Architecture divided into Modules

which are the data components of the agent. This leads to a strict separation of data and functionality of the agent, such that the behavior of an agent is defined by thoses components. The belief base of an agent is a specialized data component and is explained in full detail in Section 4.1.3. The plug-in architecture allows a *plug-in developer* to extend and exchange those components.

Table 1 shows all the modules that can be provided by a *plug-in developer* through a plug-in. Beside the modules corresponding to the *belief base plugin* and the *operator plugin* all the other modules that can be part of a plug-in are defined. The *simulation* plug-in can be used to adapt the multi-agent simulation, such that it communicates with external software for example and the *GUI* plug-in is used to create custom UI-Views for data-components for example.

Section 4.1.1 explains how a set of Operator modules is used to define the behavior of an agent and how it can be used to realize the concepts of Section 3.1. Then the ANGERONA *Script Markup Language* (*ASML*), which is used to define the agent cylce, is explained in Section 4.1.2. Section 4.1.3 describes how the modules of the *KR* [3] category can be used to implement different knowledge representation mechanism that are uniformly usable by an agent. In Section 4.1.4 the interaction between the *KR* modules is explained that altogether form the change beliefs operation. The Environment-Behavior of the *simulation* plug-in and the Views of the *GUI* plug-in are not part of the agent framework but of the multi-agent framewore and are therefore explained in Section 4.2.

4.1.1 Modular Agent Building Block

This Section introduces the concept of *operation types*, which are used to differentiate *operators* on an abstract level. That means we distinguish between an abstract level, which only defines *operation types* to define an agent model and an instantiation level,

[3] *KR* is a shortcut for knowledge representation

which instantiates *operators* with a specific *operation type*. In ANGERONA a set of *operation types* and an execution order on that set is used to define an agent model. After the functional component, that consists of *operation types*, *operators* and the *agent cycle*, is introduced the Section also introduces the *agent components* and shows that a *belief base* is a special *agent component*.

To give the *user* a high flexibility ANGERONA allows the definition of custom agent models. Therefore a concept called *operation type* is used.

Definition 19 (An *operation type*)**.** *is a triple* (n, in, re) *containing a unique name* n*. A set of input parameters* in *and return type* re*. A set* O *of* operation types *is called* consistent *iff* $\forall (n_1, in_1, re_1), (n_2, in_2, re_2) \in O : n_1 \neq n_2$

Every *operator* has an *operation type*. If two *operators* have the same *operation type*, that means an *operation type* with the same name, they are called *interchangeable*.

Implementation Detail. *If an agent contains a set of* interchangeable *operators then* ANGERONA*'s default policy is to use the preferred operator that is selected in the configuration file, see the Appendix A.4 for more details.*

Example 9. *If the operators A and B both have the operation type* subgoal generation*. But another operator C has the operation type* violates*. Then A and B are* interchangeable *but neither A and C nor B and C are* interchangeable*.*

Given the definition of *operation types* building an agent model is straightforward by defining the used *operation types* and an execution order.

Listing 1: Simple BDI Agent Cycle in Pseudo Code

```
1 BeliefUpdate()
2 DesiresGeneration()
3 IntentionSelection()
```

Listing 1 shows a basic BDI agent cycle in pseudo code using the three operations BeliefUpdate, DesireGeneration and IntentionSelection. All three have different *operation types*:

Example 10 (Possible *operation types* for the BDI cycle)**.** *The following list shows exemplary* operation types *for the pseudo code listing above.*

- *BeliefUpdate* $\rightarrow$ *('brf',* $\Omega \times p$*,* Ω*)*
  with operation type *'brf' expecting the agent's epistemic state and a perception as input and outputs the new epistemic state of the agent.*

- *DesireGeneration* $\rightarrow$ *('desgen',* $\Omega \times \Delta$*,* Δ*)*
  with operation type *'desgen' expecting the agent's epistemic state and its desires as input and outputs the new desires of the agent*

- *IntentionSelection* $\rightarrow$ *('intsel',* $\Omega \times \Delta \times \mathcal{I}$*,* $\mathcal{I}$*)*
  with operation type *'intsel' expecting the agent's epistemic state, its desires and intentions as input and outputs the new selected intention*

With the *operation types* in mind one can think of two levels for defining the functional component of an agent in ANGERONA. On the first level *operation types* and an execution order for them are defined. On the second level operators using one of the former defined *operation type* are implemented. So the first level defines an abstract agent model and the second level instantiates the agent model.

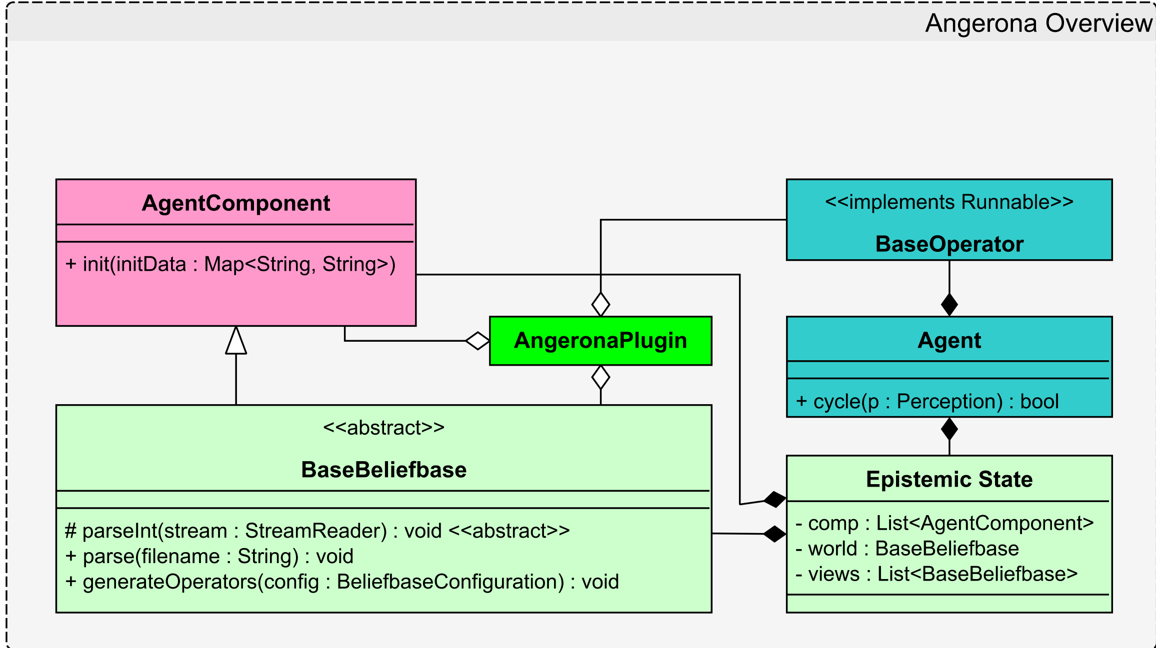

Figure 8: Agent, Components and Plugins

Figure 8 shows a UML diagram of the ANGERONA framework on a high abstraction level. The functional component ξ is provided by a set of operators extending the **BaseOperator** class. The **Epistemic State** of an agent contains a world belief base and a list of views on the other agent's beliefs. It also contains a list of *agent components* like desires, motivations or know-how [32].

Every belief base is also an *agent component* but a belief base is a specialization because it depends on *operators* which define it's functionality. The belief base encapsulates the data of a knowledge representation mechanism like ASP or ordinal conditional functions (OCF) [28]. The UML diagram also indicates that the operators that are part of the belief base are defined in a belief base configuration file which's format is explained in full detail in Appendix A.5.

Implementation Detail. *The UML diagram also shows that a* plug-in developer *who implements a belief base module has to implement a parse method to read the beliefs from a string or a file. The available parsers in the* ANGERONA *framework use JavaCC [16] to generate the parsing code.*

The agent data components referred to as **AgentComponent** in the UML diagram have an initalization method that gets invoked during the initalization of the agent. The

method's parameters are key-value pairs which are defined in *data* tags in the simulation template XML file. The format of the simulation template XML file is fully defined in Appendix A.3.

4.1.2 *ASML* based Agent's Cycle

Today scripting languages allow to change the behavior of a complex software system without changing its core code base. The change of the behavior caused by changing a scripting file does not imply rebuilding the software. This allows fast prototyping because the time intensive rebuilding process vanishes. Another advantage of some of the scripting languages is that they are not as complex as an object oriented programming language like JAVA.

Scripting languages can be differentiated by their type. There are shell languages, text processing languages, application specific languages etc. The application specific languages have two sub types: The popular embeddable languages like Python, Perl, Ruby, PHP etc. and the domain specific languages which are specialized to a single application. Although the embeddable languages are more often used they have a lot of functionality which might not be needed in every implementation and they are also harder to learn than domain-specific languages with their small set of specialized commands. Therefore the ANGERONA framework uses a domain-specific language with a specialized feature and command set.

The ANGERONA *Script Markup Language* (*ASML*) is an XML format which defines a scripting language for the ANGERONA framework. It is a domain-specific scripting language and it is used to define the agent's cycle. There are also plans to use *ASML* to implement the operations of the belief bases in the future.

ASML allows a simple invocation of operations by providing the unique name of the *operation type*. In addition control structures like if-then-else and loops allow the *user* to decide how the script behaves by considering runtime variables. This section starts by explaining the data structure which is internally used to allow the script interpreter to access runtime variables. Then it explains the operation invocation of *ASML* with the help of an example agent cycle script defined in *ASML*. This section aims to give an understanding of the important concepts of *ASML* and its execution in the ANGERONA framework. For a complete list of commands and a reference to write *ASML* scripts refer to Appendix B.

The Interpreter Context The *ASML* interpreter is responsible to run a script and reports errors, for this every interpretation of *ASML* runs in a specific context. The context is similar to the scope term used by programming languages like JAVA. Such a scope defines which variables or attributes are accessible on a specific part of the source-code, like a class-method definition for example. In every *ASML* script a global context exists which is empty in the default case. But several objects like agents or belief bases can act as a *context provider*. These context providers have the responsibility to fill an *ASML* context with variables which are accessible in *ASML* scripts by an identifier. If the agent is used as *context provider* then its world belief base can be accessed through

an *ASML* script by '$world'. When scripting the agent's cycle with *ASML* the agent is used as *context provider*.

Implementation Detail. *The variables in a context given by a* context provider *are references which exist in the* JAVA *scope and are not deallocated after the script execution. That means changes on variables that are exposed by a* context provider *are accessible for another* ASML *script. But every variable created by an* ASML *script is only temporary valid until the script execution ends and can therefore not be used by a second* ASML *script that is executed later.*

Operation Invocation The key feature of *ASML* is the invocation of operators, thus allowing the definition of agent cycles like the simple agent cycle in Listing 1. For this only the operation type and the required input parameter of the operation type has to be given. The selection is handled internally by ANGERONA. Therefore every agent has one preferred operator for every set of operators of a specific operation type. The initial preferred operator is given by the agent configuration file and can be changed by a *plug-in developer*.

Listing 2: Simple BDI Agent Cycle in*ASML*

```
<if condition="$in.perception"> <operation type="brf">
            <param name="information">$in.perception</param>
            <param name="beliefs">$beliefs</param>
        <operation>
</if>
<operation type="desgen">
    <param name="information">$in.perception</param>
<operation>
<operation type="intsel">
    <output>action</output>
<operation>
<execute action="$action" />
```

The Listing 2 shows the pseudo code of Listing 1 in *ASML*. An interpreter parses such an XML file and executes every *ASML* command in the order it occurs in the XML file. The *if* tag has a condition *attribute* and if this condition is true then the *ASML* code between the opening and closing *if* tag is executed.

The *operation* tag has an *operation type* name as attribute and it's inner elements define a parameter map. This map is given as string key-value pairs. If a required parameter is missing an error is logged and a default output value is generated by the operator allowing the script to proceed. Every operation can save its output value in the context by providing a name for the output value in the output element, like the *intsel operation type* in the listing. This allows later commands like 'execute' to use the output values of previous operations.

'execute' is a context depending command which makes an agent execute an action. It does not invoke an *operator* like the *operation* element but is invokes the agent's perform action method and therefore the context has to be provided by an agent otherwise the 'execute' command throws an error.

4.1.3 Belief Base

A belief base is also an *agent component* but it is treated in another way because it adapts a knowledge representation method in a general way. The default belief base in Angerona is represented as a disjunctive logic program (DLP) under the answer set semantics. An alternative implementation uses OCFs [28] for knowledge representation. Belief bases using other representations like bayesian networks, a set of FOL formulas or other types of logic programs are also possible.

Every knowledge representation is handled by the same interfaces and therefore the different representation methods are interchangeable for an Angerona agent. The agent knows three operations that have to be implemented by a *plug-in developer* as an *operator* with a specific *operation type*: reasoning, translation and change. The following list explains the operations in more detail:

- The **Reasoner** determines the belief set of a belief base. Therefore it instantiates a belief operator as defined in Definition 3. The belief set is represented as set of NLP facts.
  Input: Belief base
  Output: Set of literals as NLP facts.

  Alternatively the **Reasoner** processes a query by giving an answer value. The query is given as a NLP fact.
  Input: Belief base and NLP fact representing the query
  Output: Answer value either true, false, reject or unknown

- The **Translator** translates a given perception or a given NLP program in the belief base that represents the input parameter. This *operator* is important to communicate with other agents that might use a different knowledge representation mechanism.
  Input: A NLP program or a perception
  Output: A belief base

- The **Change Operation** changes the agent's belief base by another beliefbase. An implementation instantiates a revision $*_{\mathcal{D}}$ for an agent's belief base component as defined in Section 3.3.2 on page 13 .
  Input: A belief base
  Output: A belief base

The reasoner, translator and change *modules* are all parts of the belief base plug-in. This means that those three modules are implemented for one specific belief base type and therefore for one specific knowledge representation mechanism. The belief base might have several reasoners that define different behaviors like a credulous and a skeptic reasoner. Those *reasoners* represent the *belief operators* defined in Definition 3 on page 8 and multiple *reasoners* can be ordered like a *belief operator family* in Definition 4

Angerona uses the greatest common denominator of the different knowledge representation mechanisms to allow the agents to use different representation methods inter-

changeable. Nevertheless a *plug-in developer* can decide to extend the basic interfaces to provide more enhanced features.

Example 11. *The ASP* reasoners *are extended, such that they can provide the answer-sets that are processed during the reasoning.*

The above example shows that although ANGERONA has a small general interface it allows the definition of more complex interfaces for specialized knowledge representation mechanisms or agent models. The main disadvantage one faces when extending the ANGERONA base interfaces is that the functional components of the agent model may depend on a specific knowledge representation mechanism.

An agent A in the ANGERONA framework contains one belief base for every agent A' in the simulation. The world belief base represents the current information avaible to A. For every other agent A' one belief base exists that represents the view of A on the knowledge of A', this represents the agent's epistemic state as defined in Definition 2 on page 8. This allows the agent to use different belief base modules for different agents. The world belief base and all views of an agent together form the agent's beliefs that conceptually refer to the beliefs in a BDI model as defined in Definition 9 on Page 15.

Example 12. *Alice decides to represents the belief base of Bob using ASP but wants to represents the belief base of Claire using OCFs.*

As the example illustrates there might be cases where the knowledge of different agents has to be be represented by different representation mechanisms. The prediction of the behavior of other agents and the effects of actions on other agents' beliefs is better if the belief base module that represents the view on the other agent's belief base works in the same way like the belief base module that is used by the other agent's world belief base. The optimal case is that both belief base modules are equal, that means they have the same initial knowledge and use the same *operator* implementations for translation, reasoning and changing the belief base. With those assumptions the prediction of the other agents belief base is correct and therefore optimal. Nevertheless in most cases the used belief base module of other agents and their initial knowledge is unknown and has to be estimated.

4.1.4 Change Beliefs

ANGERONA's core uses $\mathcal{L}_{Base}$ facts and rules to represent knowledge but the agent's belief bases can use other knowledge representation mechanisms. DLPs under the answer set semantics and ordinal conditional functions (OCFs) [28] are used to represent the knowledge in most of the currently considered examples. There might be further knowledge representation mechanisms added in the future and this section explains the concept that allows uniform belief changes for different knowledge representation mechanisms.

A specific *operation type* called *UpdateBeliefs* is used by the agents to process the information in a perception and add this information to the agent's beliefs. This *operation type* uses a perception as input parameter and an epistemic state as output. An instantiation of the *ChangeBeliefs operation type* acts as mediator, that means it manages the

sub-operations like selecting affected belief bases and the correct translation operator, for the change process of the agent's beliefs.

In Section 3.3.2 a formal definition of an agent's belief change operation is given. Although it is possible for a *plug-in developer* to implement a change belief operation which does not stick to the formal concept it is recommended to use the standard implementation of the change beliefs operator. This standard implementation acts as a mediator by selecting the affected belief bases on the used operators. Therefore the *translator* represents the t function and the *beliefs operator* represents the $*$ function defined in Section 3.3.2.

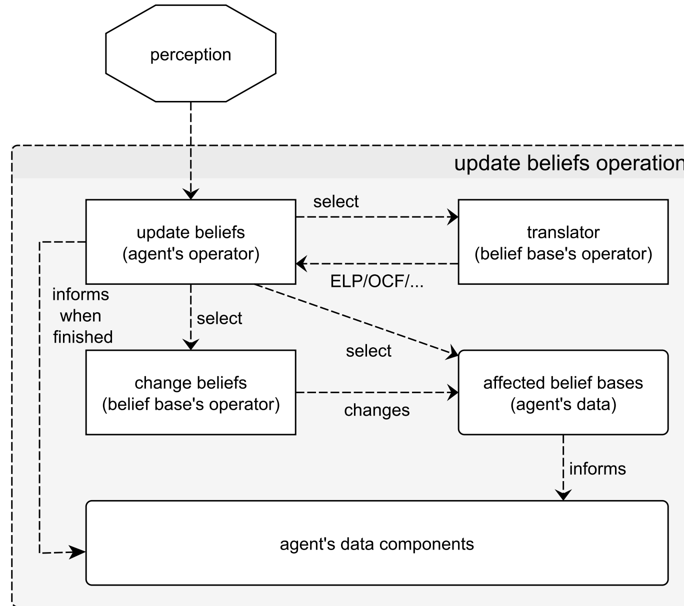

Figure 9: Change-Beliefs in Detail.

Figure 9 illustrates a change of an agent's beliefs using the standard implementation. The Figure contains all used operators. The most important operator is the *change beliefs operator* because it is the mediator for the operation. It has to select the affected belief bases, the used *translator* and *change belief base operator*. It delegates the output of the *translator* to the specific *change belief base operator*.

In the base implementation the *change beliefs operator* selects every affected belief base. Then it selects the appropriated *translator* which translates the perception or the set of $\mathcal{L}_{Base}$ literals into the same representation as the belief base. After that the preferred *change belief base operator* is selected and invoked. These three steps are repeated for every affected belief base. After the complete change of beliefs has been done the data components get informed about the entire change by a *changed beliefs event* and can also update their content corresponding to to the perception which causes these changes and the changed beliefs.

Example 13 (Perception causing multiple Belief Base Changes)**.** *Imagine Employee, Boss and Alice are in the same room. Employee tells Alice that he wants to attend the strike-committee meeting. When Boss receives the information he has to update the views*

on Alice's and Employee's belief bases because he learned that both know that Employee wants to attend the strike committee meeting.

4.2 The Multi-Agent Framework

Comparing the performance of different agent models and knowlege representation mechanisms is an important goal in the research of knowledge representation and non-monotonic reasoning.

ANGERONA uses the concept of a *simulation* to perform experiments to measure the performance of different agent models and knowledge representation mechanisms.

This section introduces the multi-agent framework of ANGERONA. Section 4.2.1 starts by explaining the different phases of the multi agent framework. Then Section 4.2.2 explains the terms *entity* and *component* and their meaning for the ANGERONA framework. On the basis of the entities and components Section 4.2.3 explains the ANGERONA report facility which is used to show the progress of the simulation using the UI of ANGERONA. The UI used by the ANGERONA multi agent framework is highly customizable. The ANGERONA plug-in architecture allows the *plug-in developer* to implement custom UI views to show the content of a data component or to provide alternative views for a data component. In Section 4.2.4 a short explanation of the main UI controls of the ANGERONA multi-agent system is given.

4.2.1 The Phases of the Multi-Agent Simulation

The ANGERONA framework uses several statements like: 'all plug-ins which are defined in the core configuration file and are valid are also loaded' and to decide if such a statement is true at the current point in time ANGERONA uses different phases.

Those phases are the bootstrap, the simulation initialization and the simulation running phase. Before the bootstrap phase nothing is initialized nor loaded and therefore no statement is true. Every statement valid after the bootstrap phase is also valid during and after the initialize simulation phase and every statement valid after the initialized simulation phase is also valid during the running simulation phase.

ANGERONA starts with the bootstrap phase where it loads all found resources. Resources are XML configuration files for the agent or belief base configuration or simulation templates etc. Those files are described in Appendix A. At the end of the bootstraping phase the plug-ins given in the ANGERONA core configuration file are loaded too. After that ANGERONA is ready to switch to the initialize simulation phase, is in a state where:

- All plug-ins which are listed in the core configuration file and are valid JAR packages containing *JSPF* plug-in annotations are loaded.
- All syntactically correct configuration files and simulation templates are pre-loaded and shown in the UI.

- The log contains information about the circumstances of errors preventing the successful load of configuration files, simulation templates or plug-ins.

The initialize simulation phase is invoked by the *user* when he/she has loaded and then initializes a simulation. During this phase the framework instantiates the *Environment Behavior* defined in the simulation configuration file. The *Environment Behavior* is explained in the next paragraph. Then the agents with their functional and data components are instantiated. As last step the initial version of the agent's data components are stored internally. This gives ANGERONA a starting point for tracking changes of those data components. When the initialize simulation phase is done then ANGERONA is in a state where:

- The *Environment Behavior* and all agent instances defined in the simulation template and their functional and data components are created. Thus the simulation is ready to run. This statement assumes that the plug-ins loaded during the bootstrap phase provide all the needed implementations of functional and data component and also of the *Environment Behavior*.

- The report system which is explained in more detail in Section 4.2.3, is initialized and holds a report for every initial data component of the agents containing the initial version of the data component as an attachment.

During the simulation phase the initalized simulation is processed. This phase highly depends on the *Environment Behavior* which defines how an environment is simulated. It can be used to communicate with external software for example. During the simulation phase the *user* can proceed the simulation, by one tick or until it ends, by using the GUI of ANGERONA.

The implementation of a custom *Environment Behavior* allows to integrate different environments like a poker game application allowing a human agent to play against AI agents or a grid world implementation allowing the agent to move in a world. Therefore the *Environment Behavior* might communicate with an external program. Another option is to implement the complete simulation in a plug-in. Whatever option is used the *Environment Behavior* is responsible to handle the communication between the ANGERONA framework and the environment simulation.

Example 14 (The Default Behavior)**.** *simulates no environment, but this is appropriate to simulate the communication between agents. It's main purpose is the delegation of speech-acts between the agents. Such a speech-act is an action of the sending agent and a perception for the receiving agent. The simulation invokes every agent's cycle method and immediately forwards the actions to their respective receivers. So it is possible that Agent A sends an action in the first simulation* tick *and agent B receives it in the same* tick *because B's cycle runs later.*

This is an adequate implementation for the experiments so far but more complex scenarios with environmental constraints or that need fair timings for the receive of perceptions have to use another *Environment Behavior* implementation. Therefore we plan to replace

the default behavior in the near future with a behavior that delegates the speech-acts in the next simulation tick. That means the agents receive the speech-acts one tick later than they are send so that an agent B cannot send a speech-act that is received by agent C in the same tick as the speech-act has been send and by agent A a tick later.

4.2.2 Entities and Components

The terms *entity* and *component* are essential in the ANGERONA framework. ANGERONA uses the component based software paradigm [3] to provide flexibility and extensibility for its agents. An entity is a uniquely identifiable object in the simulation, like an agent or a belief base. It contains a set of components which define its behavior and data structure. An agent is an entity which contains operators, belief bases and agent components. All these are components in the sense of component driven software development. Belief bases are entities which contain components defining their behavior. Those components are the belief base's operators.

Figure 10 shows an UML class diagram of the base classes implementing the entity and component concept. An entity is any object of a simulation and has an object id of itself, the object id of its parent object and a list of object ids of all of its child objects. An *agent component* is an entity too. Therefore, *agent components* and belief bases store the object id of the agent they are part of as their parent object. The agents do not have a parent object but they store the object ids of the *agent components* and belief bases in the list of their child objects.

Every data component has to support deep copying to keep track of its changes.

Implementation Detail. *In* ANGERONA *this is implemented using the* JAVA *Cloneable interface.*

Such a copy of the data component represents the data component at a specific point in time. Therefore the object ids of the source and the copied component are the same. Although multiple JAVA objects share the same object id the object id remains unique for the simulation. From the perspective of the simulation one data component represented by multiple JAVA objects show the state of the data component at different points in time.

To inform listeners about changes the base class of the *agent components* provides methods to fire property change events which can be received by assigned listeners. This allows a loosely coupled communication between different components.

Example 15 (Component Communication)**.** *A component implementing the agent's secrets informs assigned listeners about changes using the property change events. Every other component is able to react to changes of the secrets of the agent without knowing the implementation of the secret component.*

4.2.3 Report-System

ANGERONA uses a report mechanism which allows the user to inspect the inner workings

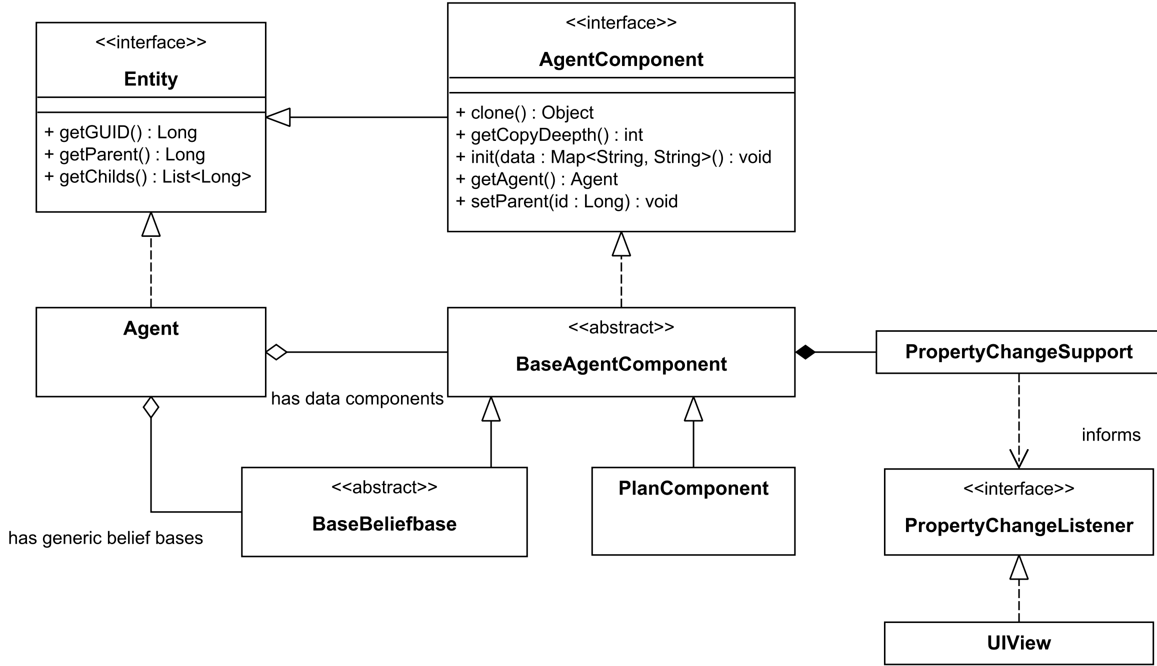

Figure 10: UML Diagram: Entities and Agent Components

of an agent. Although every change could be saved by operators when they invoke the report system everytime a change occurs, the operators use the report mechanism to inform the user about the important changes. Which changes are important is a decision the *plug-in developer* has to make when he implements an operator. But in general those changes are important that help the *user* to understand what happenend in the operator internally.

Implementation Detail. *At the beginning of the simulation the report system is also used to save the initial versions of the data components of an agent.*

A Report Entry is the basic part of the report system. Every entry contains a message, the *tick* when it was posted, a reference to its poster, the real time when it occurred and an optional attachment. The poster might be an agent or a component like an operator or a belief base. Because the scope in which the operator is called is also interesting the report entry saves the callstack of the operators, whereby the agent is stored on the top-level of the call-stack. The attachment is a data component, reports with attachment are normally made when the attachment, for example a belief base, has changed.

The Report defines an interface to add report entries, access older report entries and access only the report entries which contain a specific type of attachment. This allows us to show a timeline of changes for a specific data component.

The report is the reason why every data component must support deep copying. Saving references would change the report of the past if the agent changes its belief base after the report.

Implementation Detail. *To keep track of copies of copies every data component saves its copy depth as an integer. The mapping between original data components and their copies is given by the unique ID of entities which is explained in Section 4.2.2.*

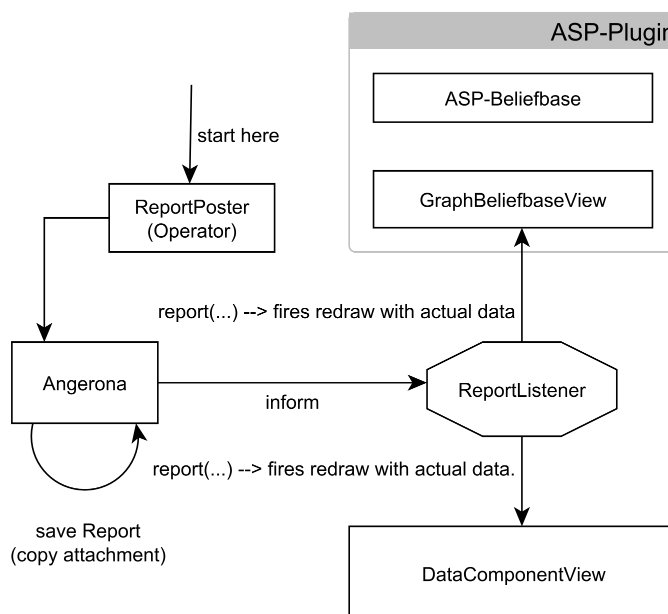

Figure 11: Flow: From Report to UI.

UI-Integration Every UI View that shows the content of an agent data component uses the report system to provide a timeline. Figure 11 shows the control flow from the report methods of the operators through the ANGERONA framework to the UI. If a report poster posts a report-entry the *Angerona* class gets informed first and stores the report. After that the *Angerona* class informs every *ReportListener* about the new report entry. Possible report listeners are UI views that show the content of agent's data components. Those report listners filter those entries that have attachments they are interested in and if they receive an entry with such an attachment the redraw their content.

4.2.4 ANGERONA's Graphical User Interface

ANGERONA uses a graphical user interface and it's default perspective is shown in Figure 12. The shown perspective is separated into three docking panes. On the left side is the resource pane. At the center is the workspace pane and at the bottom is the simulation status pane. In ANGERONA docking panes are used to allow the *user* to customize her workspace as in Eclipse or Visual Studio for example.

In Figure 12 the resource pane has one tab showing the different resources loaded by ANGERONA so far. Therefore it uses a tree view. Every configuration file like the agent and the belief base configurations or the simulation templates are displayed at the top of the tree. Nearly at the bottom of the Figure a tree node for the simulation is displayed.

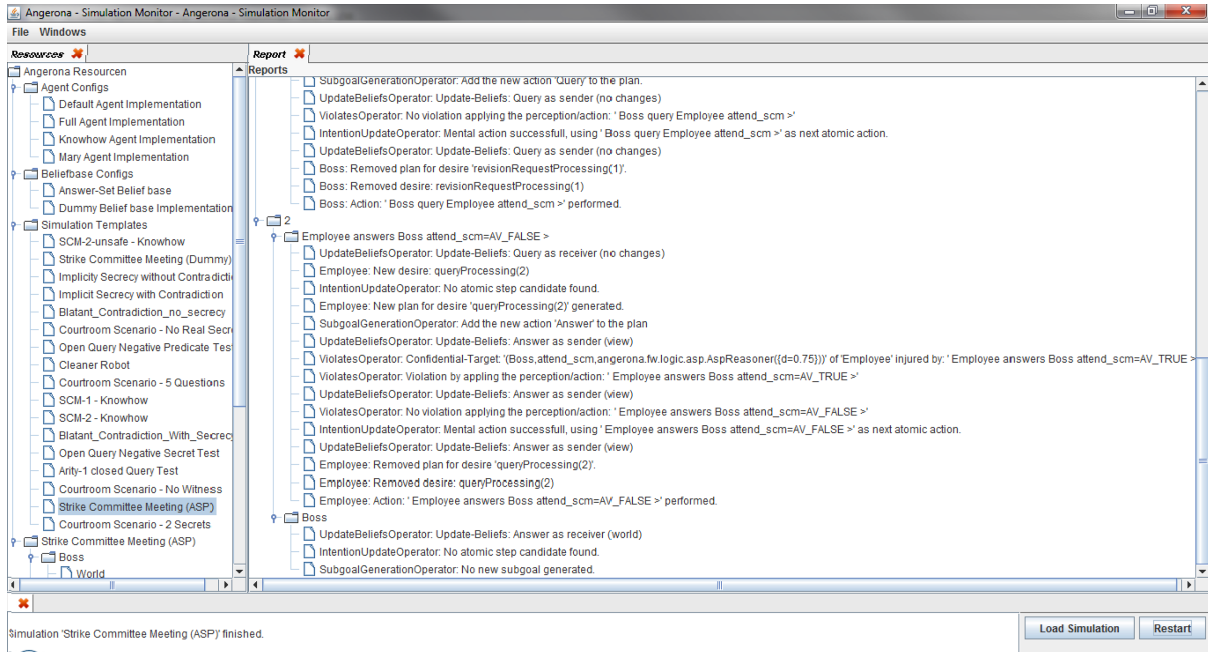

Figure 12: Angerona GUI - Overview

It contains an entry for every agent of the simulation. Every agent contains its world belief base, its belief bases representing the view on other agents and more specialized data components like *secrets*. The workspace pane shows one tab which displays a report view. It shows all important events which are reported by ANGERONA like the change of a belief base or the output of an operator. The workspace pane can also contain tabs which are opened by the *user* by double clicking on a resource tree node. The simulation status pane shows one tab which allows the *user* to run/restart a simulation and to load other simulations.

Figure 13 shows the ASP belief base view of ANGERONA. It is opened as a tab for the workspace pane. The user can open UI components to show data components by double clicking on the tree nodes in the resource view which represent data components of an agent. At the top of the ASP belief base panel is a navigation control which allows the navigation through the different versions of the belief base during the simulation.

Implementation Detail. *The navigation control internally uses a custom control called counter control which consists of a text label in the middle and buttons on the left and right side. The text label contains text of the form '$actual/$max'. $actual means the actual value of the counter variable and $max is the maximum counter variable. Whereby the left button decreases and the right button increases the version counter variable if possible.*

The first counter control allows us to iterate over all report entries containing the belief base as an attachment. The second counter control allows to iterate over the *tick*s. It selects the first version of the belief base in the selected *tick*. The third counter control allows the selection of the several report entries containing the belief base as an

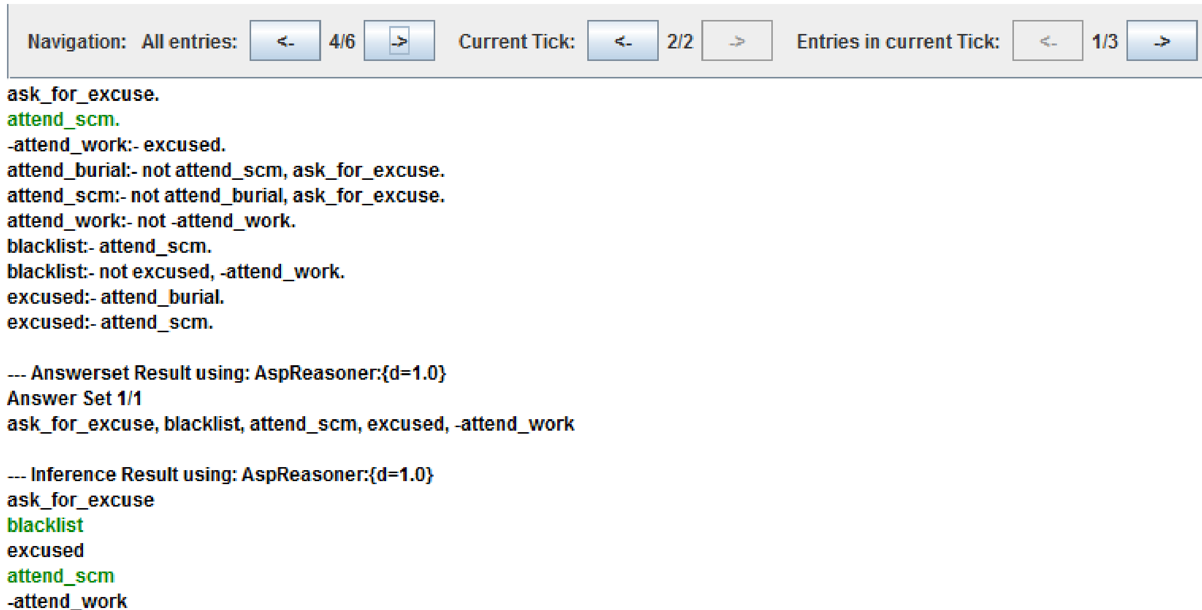

Figure 13: Angerona GUI - ASP Belief Base UI Component in Workspace Pane

attachment in the currently selected *tick*. So the third slider is useful to investigate what happens in one specific *tick*.

Beneath the navigation panel is a list which shows the content of the belief base and the literals which can be inferred using the belief base. The ASP belief base view also shows the answer sets generated during the inference. The changes between the currently shown belief base and it's predecessor are expressed by colors. The color green means that the literal is new in this version of the belief base and is not part of the previous version. The color red means that the literal is not part of the belief base anymore but is part of the previous version. The color black means no change to the previous belief base version.

4.3 Implementation of Secrecy Preserving Agents

The integration of the agent model described in Section 3 into the Angerona framework consists of three major steps. The first step is the definition of the data components that are needed by the agent model. Section 4.3.1 explains how desires, intentions, plans and secrets are stored in the agent. This Section does not explains how a belief base is integrated because this has been explained in Section 4.1.3. The second step is the definition of the agent's functional component by defining the *operation types* used by the agent model. Section 4.3.2 describes the *operation types* purpose, their input parameters, their return types and explains how the default implementation in Angerona behaves. The last step is the definition of the agent cylce. This is done by defining an agent cylce using the *operation types* and ASML script and is explained in Section 4.3.3.

4.3.1 Data Components

This section describes the different data components implemented for the secrecy agent model. The components described here only store data and send messages if they are changed. How the data components are filled by the functional components is explained in Section 4.3.2.

The (B)DI Part As mentioned earlier the beliefs of an agent are defined by its world belief base, it's views on other agents and it's set of secrets. A desire contains a $\mathcal{L}_{Base}$ fact describing the desire. The *agent component* 'Desires' is a collection of the described desires.

Implementation Detail. *An desire might also store the perception that lead the agent to the desire.*

A plan element represents a step in a plan. It is either an action or a subgoal. An action can directly executed by the agent and corresponds to an atomic intention. A subgoal in general is a complex, i.e. non atomic, intention in ANGERONA. The subgoal contains a list of stacks of plan elements. The operators that change the plan data define if the several stacks represent alternative or parallel plans. In the base implementation the stacks represent alternative plans. A plan element also contains a user defined object which can contain further information. Which information is stored in this object can be decided by the *plug-in developer*.

Example 16 (User defined Object in Plan Element)**.** *One possible use-case of the user defined object is to save a boolean flag that indicates if this plan element represents a dishonest action. Such information allows to easily rate the plans after they are generated.*

As the above example illustrates the user defined object can be used by the *plug-in developer* to provide additional information that allows the rating of the plan. But one can also think about more complex user defined objects that could help the planner to identify steps of the plan that need re-planning, etc.

The *plan component* is an *agent component* which represents a simplified form of the plan defined in Definition 11. It does not implement the level of detail δ. It contains a collection of subgoals. There might be agents which are not qualified to perform a specific type of action. ANGERONA uses the term *capability* to define which type of actions an agent can perform.

Secrets The *secrets* are an *agent component* which implements the secrets $\mathcal{S}(\mathcal{D})$ of an agent $\mathcal{D}$. Like the desires the secrets component is implemented as a collection of secrets. Every secret is defined as a triple $(\Phi, Bel, \mathcal{D})$. Bel is a *beliefs operator*, The agent who has the secret does not want the agent $\mathcal{D}$ to reveal the secret piece of information Φ using the operator Bel.

Example 17 (The *emp*'s secret)**.** *The* emp *has the secret that he does not want his boss to know that he plans to attend a strike committee meeting and this secret is represented*

by the following triple as in Definition 5 on Page 11:
(attend_scm, Bel, boss*)*

As mentioned in Section 4.1.4 when the belief base of an agent changes, then every component of the agent are informed. The *secrets* component checks if the changed belief base is a view. If the world belief base is changed then the *secrets* needs no update but otherwise the *secrets* might contain a secret that has to be kept safe from the agent that's view has been changed. If there are such secrets they have to be checked otherwise they might become inconsistent and therefore must be dropped. Inconsistent means the secret piece of information is already known by the agent.

4.3.2 Functional Components

Multiple *operation types* are defined to implement the secrecy agent model explained in Section 3.3.3 into the ANGERONA framework. We start by defining *operation types* for the functions belief update, opt, intention update, subgoal generation and violates. The following paragraphs explain the task of each *operation type*, its input and output parameters and the behavior of the base implementation. So this Section gives a *plug-in developer* an overview on the secrecy agent model in the ANGERONA framework. For a full reference of the parameters and settings used by the several operators see Appendix B.1.

***change beliefs operator*:** It implements the managing part of the belief update operator. Together with the *translator* and *change belief base operator* it defines the complete functionality of belief update. It is responsible to select the affected belief bases which are updated. It is also responsible to select the used operators to translate and perform the update. Further details of the *change beliefs operator* are given in Section 4.1.4. This operator is part of the general ANGERONA framework. The *plug-in developer* can write implementations that also inform other data components, like the secrets, but it is highly recommend to loosely couple the data components and let them react to the change event of the belief base.

***generate options operator*:** It generates new desires for the agent. Its input is the actual perception of the agent. Its output is the number of newly created desires. The base implementation creates desires to react on different perceptions. If the agent receives a Query for example the default operator creates the desire 'queryProcessing' which stores a reference to the query. This allows the agent to respond reactively to queries from other parties.

***intention update operator*:** It implements the deliberation and the selection of the next atomic intention. Its input is the agent's plan component and it's desires, its output the plan element containing the next atomic intention. Together with *subgoal generation operator* it implements the *intsel* operator of the BDI model. The base implementation iterates through every plan in the agent's *plan component* and selects the first intention

in the plan sequence. If this intention is atomic and it does not violate secrecy, then it is selected by the *intention update operator*; otherwise the iteration continues.

***subgoal generation operator*:** It is the planner which is responsible to break complex intentions down into atomic intentions and generates elements for the agents' plans. Its input is the agent's *plan component*, its output is a boolean indicating if the agent's plan has changed. As long as no data driven concept, like know-how, is used for planning then every scenario needs an implementation of the *subgoal generation operator* to perform the scenario dependent planning. Together with *intention update operator* it implements *intsel* of the BDI model. The base implementation has a general behavior to react to `queryProcessing` desires. It creates two plans one giving an answer which confirms the query (true) and the second plan is giving an answer denying the query (false).

***violates operator*:** It checks if the execution of a single action or an entire plan violates secrecy. Its input is an action or a plan and the used version of the beliefs of the agent. Its output is a boolean flag. The flag indicates if a violation occurred. The base implementation uses the *beliefs operator* defined in the secrets to check if a secret is revealed after an action is performed.

After the formal description of the different *operation types* and their base implementation as operator the following example gives a clue on how the operators can interact during the agent cycle.

Example 18 (Operator Sequence *strike committee meeting*)**.** *In the* strike committee meeting *scenario the* emp *receives a query from the* boss*. The* boss *asks the* emp *if he attends the strike committee meeting. Using the default operators of the secrecy agent model in* ANGERONA *gives the following sequences of operator calls:*
generate options operator *creates a desire 'queryProcessing'*
intention update operator *finds no atomic intention yet*
subgoal generation operator *creates two plans (confirm,deny)*
intention update operator *selects the first action: confirm*
violates operator *returns a violation for confirm*
intention update operator *selects the next action: deny*
violates operator *returns no violation for deny*
The action deny is performed by the agent emp

The secrecy agent model is an extension of the BDI model and contains all of its parts as explained in Section 3.1. The *change beliefs operator* handles the change of the beliefs and therefore implements the *brf* function in the BDI model. The *generate options operator* maps to the *desgen* function of the BDI model. The *intention update operator* and *subgoal generation operator* operators together map to the BDI *intsel* function. The extensions are the data components for *secrets* and *plans* and also the *violates operator* which can be used by the other operators to check if an action causes a violation before selecting the action.

4.3.3 Secrecy Agent Cycle

This Section explains the secrecy agent cylce that defines the behavior of an agent in the secrecy agent model. Figure 7 on page 23 shows the agent cycle for the secrecy agent model. The following listing represents the agent cycle in ASML and is used as default agent cycle for the secrecy agent model in the Angerona framework:

```
<asml-script name="secrecyCycle">
    <operation type="UpdateBeliefs">
        <param name="information" value="$perception" />
        <param name="beliefs" value="$beliefs" />
    </operation>

        <operation type="GenerateOptions">
            <param name="perception" value="$perception" />
            <output>newDesireCount</output>
        </operation>

        <assign name="running" value="TRUE" />
        <while condition="$running==TRUE">

                <operation type="IntentionUpdate">
                    <param name="plan" value="$plan" />
                <output>action</output>
                </operation>

                <conditional>
                        <if condition="$action==null">
                                <operation type="SubgoalGeneration">
                                    <param name="plan" value="$plan" />
                                    <output>running</output>
                                </operation>
                        </if>
                        <else>
                            <assign name="running" value="FALSE" />
                        </else>
                </conditional>
        </while>

        <execute action="$action" />
</asml-script>
```

At the beginning of each cycle the agent receives a perception from the environment and changes its current beliefs using this perception. In the next step it updates its desires using the *generate options operator*. After that the planning loop, which is represented by the while tag in the ASML code listing, is executed trying to find the next atomic action. If the *intention update operator* does not find an atomic intention the *subgoal generation operator* is invoked to create new plan elements. The cycle invokes this loop until an action is found or no new subgoals are created. The found action is invoked by using the execute operation in ASML. Both, the *subgoal generation* and the *intention update* operators might use the *violates operator*, which checks if a secret piece of information would be revealed after the agent performs the action. The *violates operator* works with copies of the beliefs. The data components are only informed about changes on the

original belief bases and not on their copies. If an action is executed it is send to the environment.

The behavior of an agent in the BDI secrecy agent model can be adapted by implementing new operators. The *plug-in developer* can implement different implementations of the belief update, opt, intention update, subgoal generation and violates functions by providing other operator implementations. This allows the *plug-in developer* to implement several concepts for a secrecy preserving agent model. Providing different agent cylce scripts can further adapt the secrecy agent model. The ASML and the *operation types* give the ANGERONA framework the flexibility to support several agent models that are completely different to each other. It provides a library to perform experiments that measure the performance of the different concepts like knowledge representation mechanism or even entire agent models.

4.4 The Tweety Library

The tweety library [31] is a collection of sub libraries for artifical intelligence and knowledge representation. It is open-source and available at *Sourceforge*. ANGERONA uses the tweety core library, several tweety language libraries and several tweety translation libraries for knowledge representation. The following list shows the tweety libraries that are used by the ANGERONA framework:

- net.sf.tweety
  core library providing base classes and interfaces
- net.sf.tweety.logics.commons
  core library providing base classes and interfaces for logic languages
- net.sf.tweety.beliefdynamic
  revision / change library for logic languages, provides base classes for revision and change operations on belief bases
- net.sf.tweety.lp.nlp
  language library implementing nested logic programs [20]
- net.sf.tweety.lp.asp
  language library implementing disjunctive logic programs under the answer set semantics[12]
- net.sf.tweety.logics.cl
  language library implementing conditional logic and OCF reasoning [28]
- net.sf.tweety.logics.pl
  language library implementing propositional logic
- net.sf.tweety.logics.fol
  language library implementing first order logic

- net.sf.tweety.logics.translators
  translation library providing translation between:
  - disjunctive logic programs and first order logic
  - disjunctive logic programs and nested logic program
  - first order and propositional logic

Tweety core provides base classes for knowledge representation, artificial intelligence and reasoning and is a dependency of the Angerona core.

The tweety language libraries implement a representation for a logical language like first order logic or DLPs. They provide a parser to parse files to belief bases. The language libraries share the same interfaces for atoms, literals negations, conjunctions and disjunctions allowing to read and change the constructs in the same way. Some of the languages allow typed constants and typed predicate arguments. The libraries for first order logics and for nested logic programs are dependencies of the Angerona core.

The tweety ASP library can represent disjunctive logic programs. It provides an interface to use answer set solvers for those programs. The interface is implemented for CLASP, Clingo, DLV and DLV Complex. This tweety language library is a dependency of the ASP plug-in of the Angerona framework.

The tweety conditional logic library provides ordinal conditional function and reasoners for them. It is used by the OCF plug-in of the Angerona framework.

5 Conclusion

In this work we approached the topic of secrecy from the perspective of an autonomous epistemic agent. From this perspective information is uncertain and incomplete, secrets are specific to other agents and vary in strength. In the first part we presented a general epistemic agent model in which we define secrets and what it means to an agent to preserve secrecy. To this end we introduced action functions characterizing the behavior of an agent and a change operator which adapts the beliefs of the agent, its secrets and views on other agents. To our knowledge no similar approach to secrecy from a subjective, epistemic perspective of an agent has been put forward so far.

In the second part we elaborated an agent model for secrecy preservation. The model incorporates secrecy preservation into the agent cycle such that secrecy violation can be determined as early as possible. In [7] it has been discussed how techniques from the database setting can be used for preserving secrecy in multiagent negotiation. There a censor component is used to control the agents actions prior to execution which is external to the agent cycle. Secrecy is checked in the last moment and the agent cannot reason about secrecy. In said work the typical query-answer of databases setting has been generalized to negotiation acts of agents. Here we allow for arbitrary actions. Our approach of lifting secrecy evaluation to the deliberation process of an agent is close to the consideration of secrecy preservation as a maintenance goal [36]. This indicates that mechanisms for the satisfaction of maintenance goals, e. g., [14], might be used for

preserving secrecy. Our notions of secrecy is formulated in a general epistemic model of agents allowing for a variety of logical formalisms. It includes the modeling of the beliefs of other agents and their reasoning behavior. This is not present in the agent programming languages. Secrecy preservation is a very specific maintenance goal which needs specific considerations, operators and models.

We see our framework as a good starting point for the development of secrecy preserving agents and implementations of those. We have a running prototypical implementation of our model based on the presented BDI model and an ASP reasoning and deliberation component. In current and future work we will investigate the properties of secrecy in our considered setting further. Especially with respect to handling uncertainty, non-monotonic logics and planning under these circumstances. Evaluations of the approach using the implementation are in progress.

5.1 Future Work

In this Section planned enhancements for future releases of Angerona are described.

5.1.1 Version aware Data Components

Currently the data components use a deep copy to save different version at specific point in time. This leads to memory drawbacks and there exists better solutions from a conceptual perspective.

The data component might be a container of different versions of its content. Concepts of a version control system are useable like saving changesets instead the entire content. The mental action of an agent can be represented as branches of the main sequence of the data component changes. The responsibility of informing Angerona about changes in the data components would move from the operators invoking the changes to the data components which is a better solution.

5.1.2 *ASML* based Operation Definition for Belief Bases

The *ASML* based operation definition can be used to define the three belief base operations which are directly mapped to the *beliefs operator*, *translator* and *change belief base operator*. Allowing multiple operators for those operations gives the belief base functional component more flexibility.

Example 19. *There a several ways to define a change operation for a belief base. Way one might be an easy expansion using only one operator. Way two might use two steps starting with a selection step to filter the new information beforehand and than merge the information with the belief base. This can be accomplished using two operators with different* operation types*.*

5.1.3 *ASML* for the Selection of the Operator Implementation

In the current version of ANGERONA mostly the default operator is used to perform a specific *operation type*. The *plug-in developer* can use the ANGERONA core interface to change the default operator but this behavior is plug-in dependent. This approach is not so flexible because there might be plug-in A and plug-in B which do not depend on each other. At the moment it is not possible to select between an operator implementation I_1 of plugin A and I_2 of plugin B because the *plug-in developer* of A does not know I_2 and the *plug-in developer* of B does not know I_1, therefore a approach is needed that allows the dynamic selection of the preferred operator during runtime.

The ANGERONA *Script Markup Language* could provide operator selector scripts to dynamically select the used operator. This allows the *user* to have more control over the used operators of the agent.

Furthermore a new *ASML* command shall be introduces that allows the global change of settings of an operator.

5.1.4 Annotation based Context Providers

Currently the generation of the context provider is fully automated and the accessible variables are defined by all parameterless methods of the context provider that names start with a 'get'. Thus the definition of the variables of the context provider for the agent in Appendix B.2 is not optimal, there are variables accesible that are not need by *ASML* and there are also duplicated variables. Therefore the use of annotations to mark those methods that shall be accessible by *ASML* is a future task.

References

[1] Relax ng specification. `http://relaxng.org/spec-20011203.html`, 2001. [Online; accessed 11-December-2012].

[2] Relax ng compact syntax tutorial. `http://relaxng.org/compact-tutorial-20030326.html`, 2003. [Online; accessed 11-December-2012].

[3] Mikio Aoyama et al. New age of software development: How component-based software engineering changes the way of software development. In *1998 International Workshop on CBSE*. Citeseer, 1998.

[4] Trevor J. M. Bench-Capon and Paul E. Dunne. Argumentation in artificial intelligence. *Artificial Intelligence*, 171(10-15):619–641, 2007.

[5] Joachim Biskup. Usability confinement of server reactions: Maintaining inference-proof client views by controlled interaction execution. In *Databases in Networked Information Systems*, volume 5999 of *LNCS*, pages 80–106. Springer, 2010.

[6] Joachim Biskup. Usability confinement of server reactions: Maintaining inference-proof client views by controlled interaction execution. In Shinji Kikuchi, Shelly Sachdeva, and Subhash Bhalla, editors, *Databases in Networked Information Systems*, volume 5999 of *Lecture Notes in Computer Science*, pages 80–106. Springer Berlin / Heidelberg, 2010.

[7] Joachim Biskup, Gabriele Kern-Isberner, and Matthias Thimm. Towards enforcement of confidentiality in agent interactions. In Maurice Pagnucco and Michael Thielscher, editors, *Proceedings of the 12th International Workshop on Non-Monotonic Reasoning (NMR'08)*, pages 104–112, Sydney, Australia, September 2008. University of New South Wales, Technical Report No. UNSW-CSE-TR-0819.

[8] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge J. Gomez-Sanz, João Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multiagent systems. *Informatica*, 30:33–44, 2006.

[9] Ronald J. Brachman and Hector J. Levesque. *Knowledge Representation and Reasoning.* Elsevier and Morgan Kaufmann Publishers, 2004.

[10] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355, 1988.

[11] C. Farkas and S. Jajodia. The inference problem: a survey. *ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Explorations Newsletter*, 4:6–11, 2002.

[12] Michael Gelfond and Nicola Leone. Logic programming and knowledge representation: the A-Prolog perspective. *Artificial Intelligence*, 138, 2002.

[13] Joseph Y. Halpern and Kevin R. O'Neill. Secrecy in multiagent systems. *ACM Transactions on Information and System Security*, 12:5:1–5:47, October 2008.

[14] Koen Hindriks and M. van Riemsdijk. Satisfying maintenance goals. In Matteo Baldoni, Tran Son, M. van Riemsdijk, and Michael Winikoff, editors, *Declarative Agent Languages and Technologies V*, volume 4897 of *Lecture Notes in Computer Science*, pages 86–103. Springer Berlin / Heidelberg, 2008.

[15] Hirofumi Katsuno and Alberto Mendelzon. On the difference between updating a knowledge base and revising it. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 387–394. Morgan Kaufmann, San Mateo, California, 1994.

[16] V. Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *Software, IEEE*, 21(4):70–77, 2004.

[17] Patrick Krümpelmann and Matthias Thimm. A logic programming framework for reasoning about know-how. In *Proceedings of the 13th International Workshop on Non-Monotonic Reasoning (NMR'10)*, 2010.

[18] Patrick Krümpelmann, Matthias Thimm, Gabriele Kern-Isberner, and Regina Fritsch. Motivating agents in unreliable environments: A computational model. In Franziska Klügl and Sascha Ossowski, editors, *Multiagent System Technologies - 9th German Conference, (MATES 2011), Berlin, Germany, October 6-7, 2011. Proceedings*, volume 6973 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2011.

[19] Jerome Lang. About time, revision and update. In *Proceedings of the 11th Workshop on Nonmonotonic Reasoning (NMR'06)*, 2006.

[20] Vladimir Lifschitz, Lappoon R Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.

[21] Paolo Mancarella, Fariba Sadri, Giacomo Terreni, and Francesca Toni. Planning partially for situated agents. In *Proceedings of the 5th international conference on Computational Logic in Multi-Agent Systems (CLIMA'04)*, CLIMA'04, pages 230–248, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] Oracle. The java tutorials: Annotations. `http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html`. [Online; accessed 10-December-2012].

[23] Oracle. Trail:: The reflection api. `http://docs.oracle.com/javase/tutorial/reflect/index.html`. [Online; accessed 10-December-2012].

[24] T. Lottermann R. Biedert, N. Delsaux. Java simple plugin framework. http://code.google.com/p/jspf/. [Online; accessed 10-December-2012].

[25] Anand S. Rao and Michael P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the 1st International Conference on Multiagent Systems (ICMAS'05)*, San Francisco, 1995.

[26] Martijn Schut and Michael Wooldridge. Principles of intention reconsideration. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS '01, pages 340–347, New York, NY, USA, 2001. ACM.

[27] G. L. Sichermann, W. de Jonge, and R. P. van de Riet. Answering queries without revealing secrets. *ACM Transactions on Database Systems*, 8:41–59, 1983.

[28] W. Spohn. Ordinal conditional functions: a dynamic theory of epistemic states. In W.L. Harper and B. Skyrms, editors, *Causation in Decision, Belief Change, and Statistics*, volume 2, pages 105–134. Kluwer Academic Publishers, 1988.

[29] John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI*, pages 721–726. Academic Press, 2003.

[30] Matthias Thimm. Tweety - a comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation.

[31] Matthias Thimm. Tweety - a comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, July 2014.

[32] Matthias Thimm and Patrick Krümpelmann. Know-how for motivated BDI agents (extended abstract). In Decker, Sichman, Sierra, and Castelfranchi, editors, *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*, May, 10–15 2009.

[33] Matthias Thimm and Patrick Krümpelmann. Know-how for motivated BDI agents (extended version). Technical Report 822, Technische Universität Dortmund, Department of Computer Science, February 2009.

[34] Leendert van der Torre. Logics for security and privacy. In Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquin Garcia-Alfaro, editors, *Data and Applications Security and Privacy XXVI*, volume 7371 of *Lecture Notes in Computer Science*, pages 1–7. Springer Berlin / Heidelberg, 2012.

[35] Frank van Harmelen, Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of Knowledge Representation.* Elsevier Science, San Diego, USA, 2007.

[36] M. Birna van Riemsdijk, Mehdi Dastani, and Michael Winikoff. Goals in agent systems: A unifying framework. In Müller Padgham, Parkes and Parsons, editors, *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 713–720, May 2008.

[37] G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1999.

[38] Michael J. Wooldridge. *An Introduction to MultiAgent Systems.* Wiley, 2 edition, 2009.

[39] Shung-Bin Yan, Zu-Nien Lin, Hsun-Jen Hsu, and Feng-Jian Wang. Intention scheduling for BDI agent systems. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 1, pages 133–140, 2005.

A File Formats

This appendix describes every file format used by the ANGERONA framework by providing its XML schema. To describe the schema the compact *Relax NG* schema syntax is used. *Relax NG* allows more flexible definitions for the data formats. It strongly supports unordered content. *Relax NG* also allows child elements to depend on the definition of parent elements and it has a more compact syntax than W3C XML Schema. For further details on *Relax NG* see [1, 2].

This appendix gives a snapshot on the *Relax NG* schema files at a specific point in time. The most recent schema files are in the source control system of ANGERONA on the paths:
'software/fw/src/main/resources/angerona/fw/serialize/schema'
'software/fw/src/main/resources/angerona/fw/asml/schema'.

date of schema snapshots: June 24, 2014

A.1 Core Configuration

The core configuration file is used to configure the ANGERONA framework. The following *Relax NG* schema describes the configuration file.

```
1 element config {
2         element plugin {text}* &
3         element parameter {
4                 attribute name {text},
5                 attribute value {text}
6         }*
7 }
```

The 'plugin' elements contain text to describe the path to a plugin *jar* which shall be loaded by the ANGERONA test application. The 'parameter' elements contain name value pairs. These pairs are accessible through the ANGERONA configuration facility. They can be used by the *user* to change ANGERONA's application and plugin behavior.

Example 20. *The ASP* beliefs operator *of the ASP plugin uses the* ANGERONA *configuration facility to decide which ASP solver it uses. The facility is also used to get the path to the several ASP solvers.*

A.2 Operator Sets and XML References

ANGERONA uses two *Relax NG* schemas as references in more complex schemas. The operator set schema and the xml reference schema.

The following listing describes the content of an element which describes an operator set.

```
1 start = (
2     element default-operator {text}&
3     element operator-cls {text}*
```

```
4 )
```

An operator set is a set of operators with the same *operation type*. It has to define at least one default operator using the element 'default-operator'. The *user* can also define an arbitrary count of alternative operators using the 'operator-cls' elements. The inner text of the elements has to be the full java class name of the operator.

The following *Relax NG* schema is used to describe the XML format used by simplexml to reference other XML files within an XML file. It describes the content of an XML element which links to another XML configuration file.

```
1 start = (
2     attribute source {text}
3 )
```

The attribute 'source' contains the path to the external XML configuration file. Because the JAVA class used to read the external files is given as an annotation in the code of the JAVA class which links to the external file the full java class name of the classes responsible for including the external files must not be known. But for completeness Table 2 maps the classes responsible for loading external XML configuration files to their types.

De/Serialize Class	**Configuration Type**
angerona.fw.serialize.AgentConfigImport	agent configuration
angerona.fw.serialize.BeliefbaseConfigImport	belief base configuration
angerona.fw.serialize.CommandSequenceSerializeImport	agent cycle

Table 2: Mapping: Java XML De/Serialization Classes to Configuration Types

A.3 Simulation Template

The simulation template files are used to generate simulations. They give ANGERONA the needed information to initialize a simulation. The following *Relax NG* schema describes the content of a simulation template file.

```
1 grammar {
2         start = sc
3         sc = element simulation-configuration {
4                 element name {text} &
5                 element behavior {text} &
6                 element category {text} ?
7                 element description {text} ?
8                 ai+
9         }
10
11        ai = element agent-instance {
12                element name {text} &
13                element beliefbase-name {text} ?
```

```
14                     element agent-config {
15                             external "xml_ref.rnc"
16                     } &
17                     element beliefbase-config {
18                             external "xml_ref.rnc"
19                     } &
20                     element view-config {
21                             attribute name {text} &
22                             external "xml_ref.rnc"
23                     }*
24                     element data {
25                       attribute key {text},
26                       text
27                 }*
28             }
29 }
```

A simulation template consist of a name for the simulation, a behavior for the simulation and at least one agent instance. The name can be given by the *user* and it has to be unique because the ANGERONA UI does not show more than one simulation template when several simulation templates exist which share the same name. The text in the 'behavior' element of the simulation has to be the full java class name of a class containing an implementation for an environment behavior. The elements 'category' and 'description' are optional. The description shortly describes the scenario. The category is used to group similar scenarios together. A category can be hierarchically by using the '/' character, such that a category 'asp/scm' means the scenario belongs to the subgroup scm of the group asp.

In every 'agent-instance' element a name is defined. The name of the agent instance has to be unique in the scope of the simulation. An element 'beliefbase-name' is used to define another prefix of the belief base file name. This element is optional and in the default case the agent's name is used as prefix for the belief base file. The belief base implementation used by the agent defines the postfix of the belief base file. The belief base content is read from multiple files located in the same directory as the simulation template file. Those belief base files are explained in more detail in Appendix A.6.

The element 'config' links to an agent configuration file which is explained in Appendix A.4. The element 'beliefbase-config' links to a belief base configuration file which is explained in Appendix A.5. The belief base config given in the 'beliefbase-config' element is used as base implementation for all belief bases of the agents, that means for its world belief base and all of it views. Nevertheless the 'view-config' element can be used to use another belief base implementation for a specific view. The element's attribute 'name' maps the config to the view of the agent with the given name, such that an agent can use different implementations of belief bases for its views on the several agents in the simulation.

An agent instance can contain an arbitrary count of data elements. Every data element has a key attribute and an inner element text. This allows the definition of key value

pairs which are saved as a map and delegated to the initialization methods of the *agent components*.

Example 21 (Know-how initialization)**.** *The inner text of a data element with the key 'know-how' is parsed by the know-how component and uses as initial data for the know-how statements.*

A.4 Agent Configuration

The agent configuration files is used to define the agent model of an agent instance using the following *Relax NG* schema.

```
1  element agent-configuration {
2          element name {text} &
3          element cycle-script {
4             external "xml_ref.rnc"
5          } &
6          element operation-set {
7             attribute operation-type {text},
8             external "op-set.rnc"
9          }* &
10         element component {text}*
11 }
```

The given name has to be unique and is used as reference in the ANGERONA UI. The element 'cycle-script' links to an XML file defining the agent's cycle using *ASML*. The agent configuration has an arbitrary count of 'operation-set' elements. They use the content of an operator set element defined earlier and an extra attribute which gives the *operation type* of the operators in the operator set. This allows the definition of different operator sets for *operation types* like intention update for example. The 'component' elements add *agent components* to the agent. The given text has to be the full java class name of an *agent component*.

A.5 Belief Base Configuration

Similar to the agent configuration file the belief base configuration file describes the used operators of a belief base and the type of the belief base implementation. The following *Relax NG* schema is used to define a belief base configuration file.

```
1  element beliefbase-configuration {
2          element name {text} &
3          element reasoners {external "op-set.rnc"} &
4          element change-operators {external "op-set.rnc"} &
5          element translators {external "op-set.rnc"} &
6          element beliefbase-class {text}
7  }
```

Like the agent configuration the belief base configuration contains a name which has to be unique. It also defines the 'reasoners', 'change-operators' and 'translators' elements which are operator sets. But unlike the agent configuration's operator set the operation type is determined by the element name instead an extra attribute. This is possible because the operation types for belief bases are definitely defined to this three types. The belief configuration also contains an element 'beliefbase-class' which has to be the full java class name of the class implementing the belief base concept.

A.6 Belief Base Files

Every agent in the simulation uses N belief base files, where N is the number of agents in the simulation. Those belief base files are used to generate the initial content of the belief bases of the agents in the simulation initialization phase. The belief base files have the following name schemata:

- World: '$belief-base-prefix.$suffix'
- Views: '$belief-base-preifx_$viewed-agent-name.$suffix'

Where '$belief-base-prefix' is either the name of the agent or the given name in the 'beliefbase-name' element of the agent-instance configuration which is explained in Appendix A.3. '$suffix' is given by the used belief base implementation and is 'asp' for ASP and 'ocf' for OCF. The '$viewed-agent-name' is the name of the agent that beliefs are represented by the view.

Example 22 (Alice's belief bases files). *Alice uses OCF to represent her beliefs but the view on Bob is represented by ASP, such that she uses the following belief base files:*

- *Alice.ocf*
- *Alice_Bob.asp*

If one of the agents' belief base files does not exists on the file system it is assumed to be an empty belief base at initalization phase but a warning is logged.

During the initialization phase of ANGERONA the belief base instances of an agent's world beliefs and its views on other agents are created. Then the file contents are forwarded to the parse method of the belief base implementation which is responsible to generate a JAVA object from the given file.

B ANGERONA *Script Markup Language*

This Appendix contains a reference to the ASML script language used by the ANGERONA framework. It starts in Appendix B.1 by giving a language reference that explains the commands and concepts like control and iteration structures. Then Appendix B.2 explains which variables are accessible through the agent context provider.

B.1 Language Reference

An *ASML* script consists of a sequence of *ASML* statements that are either commands, control structures or iteration structures. The following *Relax NG* schema defines an *ASML* script. The sub commands, control structures and the iteration structures are explained in the sub sections.

```
grammar {
        start = element asml-script {
                        attribute name {text} &
                        cmd+
                } |
                cmd+

        cmd = {
                external "assign.rnc" |
                external "invoke-operation.rnc" |
                external "conditional.rnc" |
                external "while.rnc"
        }
}
```

The above *Relax NG* schema can either be used as starting point for a script by using the 'asml-script' element or it can be used inline by other parts of the ASML specification like control and iteration structures.

B.1.1 *ASML* Command: assign

The command *assign* assigns a value to a specified variable. The variable is identified by its name. If a variable with the given name exists in the context its value is overridden otherwise the variable is generated and added to the context. The value is can be of any type that means that an integer like 1, a floating point number like 3.14 or even a complex type like a query: <boss query employee attend_scm> can be given.

Listing 3: Assign in ASML Code

```
<assign name="counter" value="3" />
<assign name="text" value="a string text" />
```

```
element assign {
        attribute name {text} &
        attribute value {text}
}
```

B.1.2 *ASML* Command: invoke operation

The command *invoke-operation* calls the preferred operator of an specific *operation type*. The *operation type* is given as attribute 'type'. The invoke-operation has an optional

Operator	Parameter	Description
update beliefs	information	A logical expression or a perception
update beliefs	beliefs	The used beliefs to perform the update
violates	information	A logical expression, an action or a plan
violates	beliefs	The used beliefs to perform the violation check
generate options	perception	The perception that is received in this agent cylce, might be empty if no perception is received.
intention update	plan	The used plan data component
subgoal generation	plan	The used plan data component

Table 3: Parameter of the Operator of the Secrecy Agent Model

'output' element. The return value of the operation is saved in the context under the name that is given as inner text of the 'output' element. The invoke-operation command also distinguish between parameters and settings. A parameter is used as an input for the invoked operator, some of the parameters are required, some are optional. A setting is used to change the strategy of an operation. For a setting a default value exists and the ASML script has the ability to change the setting for the specific call. The default *intention update operator* uses a setting 'allowUnsafe' to determine if it shall use violation checking before selecting an atomic intention. The Tables 3 and 4 list the parameters and settings of the operators implemented so far. And the following listing shows the *Relax NG* schema of the invoke-operation command.

```
1  element operation {
2          attribute type {text} &
3          element param {
4                  attribute name {text} &
5                  attribute value {text}
6          }*
7          element setting {
8                  attribute name { text } &
9                  attribute value { text }
10         }*
11         element output { text } ?
12 }
```

B.1.3 *ASML*: Control Structures

ASML uses control structures that are defined by the following *Relax NG* schema:

```
1  grammar {
2          start = cond
3          cond = element conditional {
4                  element if { cs },
```

Operator	Setting	Description	Default
a.f.e.o Intention-Update-Operator	allowUnsafe	If set to true the operator selects atomic actions without checking for violation.	false
a.f.e.o Subgoal-Generation-Operator	generateLies	If set to true the operator generates for every closed query that can be answered honestly the invert answer (lie) as alternative plan.	true
a.f.l.a Asp-Reasoner	d	Defines in how many percent of the answer sets a literal has to be to be deducted by the reasoner. A value of one means it has to be in every answer set (sceptical reasoner), a value of zero means it has to be in at least one answer set (credulous reasoner) and a value of 0.75 means it has to be in 75 percent of the answer sets.	1.0
a.f.k Knowhow-Subgoal	allowUnsafe	If set to true the operator also generates plans that contain actions that violate secrecy.	false

Table 4: Settings of Operators

```
5                     element elseif { cs }*
6                     element else { cs }?
7           }
8
9           cs = {
10                    attribute condition {text} &
11                    external "command_sequence.rnc"
12          }
13 }
```

This allows the *ASML* script developer to use control structures that contain at least one if statement, an arbitrary count of elseif statements and an optional else statement.

Example 23. *This example maps different color values to variable in* ASML *depending on the index variable* i.

Listing 4: Control Structures in ASML Code

```
1 <conditional>
2           <if condition="i==1">
3                     <assign name="color" value="red" />
4           </if>
5           <elseif condition="i==2">
6                     <assign name="color" value="blue" />
```

```
7           </elseif>
8           <else>
9                   <assign name="color" value="black" />
10          </else>
11 </conditional>
```

B.1.4 *ASML*: Iteration Structures

The *ASML* script implements a while loop as iteration structure. The following *Relax NG* schema describes the loop.

```
1 element while {
2         attribute condition {text},
3         external "command_sequence.rnc"
4 }
```

As inner element the command-sequence is used. Therefore the while element only needs an attribute that defines the condition.

Example 24. *A loop that iterates ten times and simulates a for loop:*

Listing 5: Iteration Structures in ASML Code

```
1 <assign name="i" value="0" />
2 <while condition="$i<10">
3         <assign name="i" value="i+1" />
4 </while>
```

B.2 Context Provider Variables

ASML script can use different context provider. In the cycle script the agent is used as context provider.The agent's context provides every variable that is accessible using a getter method. This is achieved by using a ContextFactory that heavily uses reflection. The following list shows the variables that are accessible through the agent context:

- beliefs - The beliefs of the agent, this data structure contains the agents custom components, its world belief base and its views.
- capabilities - A list containing the actions an agent can perform.
- components - A list containing all data components of the agent.
- context - The context itself
- environment - The environment of the agent simulation.
- gUID - The global unique id of the agent
- name - The name of the agent.
- operators - A reference of the operator provider of the agent.
- operatorStack - The call-stack of operators
- parent - The id of the parent object (is null all the time)
- posterName - The currently active operator, might be the agent name if no operator is explicitly called.

- reporter - Reference to the agent.
- stack - Reference to the agent.
- string - The name of the agent.
- type - The type of the agent (interactive or ai).