Argumentation in the Semantic Web

Paolo Torroni and Federico Chesani, *University of Bologna*

Marco Gavanelli, *University of Ferrara*

The ArgSciff architecture supports high-level reasoning and argumentation-driven interaction among Semantic Web services. A prototype implementation is based on a concrete operational model.

The Semantic Web vision looks toward a universal medium for data exchange. This vision has motivated significant research in classifying, packaging, and semantically enriching information to support data automation, integration, and reuse across various applications. A large share of current research on these topics addresses new logics for concept description and knowledge representation; new languages for defining ontologies, taxonomies, and behavior; and new cooperation models, interchange formats, and open standards. If these efforts succeed, Web pages will eventually include descriptions of their content that can leverage Semantic Web applications and foster their growth. Search engines will be able to respond to semantic queries, and Web services will reason from semantically rich information and act accordingly.

We believe the reasoning enabled by machine-understandable, semantically rich information is essential to the Semantic Web vision. Our work focuses on making this reasoning more visible to potential users by using dialogues for service interaction. As currently understood, interaction among Semantic Web services is essentially a point-to-point service request followed by a server response. We don't propose modifying the way Web services interact. Instead, we suggest using argumentation technology to drive the interaction at a higher level, where human users can perceive message exchanges and service-request sequences as high-level dialogues that they can understand better than current modalities.

In this article, we define ArgSciff, a prototype operational argumentation framework to support dialogic argument exchange between Semantic Web services. ArgSciff is based on the Sciff abductive-logic programming (ALP) framework.[1] (Sciff is an abbreviation for "*IFF* with *c*onstraints for agent *s*ocieties," referring to the "if and only if" proof procedure developed by Tze Ho Fung and Robert Kowalski.[2]) In ArgSciff, an intelligent agent can interact with a Web service and reason from the interaction result. The reasoning semantics is an argumentation semantics that views the interaction as a dialogue. The dialogue lets two parties exchange arguments and attack, challenge, and justify them on the basis of their knowledge. This format has the potential to overcome a well-known barrier to human users' adoption of IT solutions because it permits interaction that includes justified answers that can be reasoned about and rebutted.

Semantic Web interaction

We begin with a scenario to use as a running example of a Semantic Web interaction and a general discussion of how argumentation can support it.

A scenario: Scientists, departments, and trips

In our scenario, Sarah is a Computer Science Department research scientist who often travels to conferences. Before traveling, CSD researchers must have formal approval for their trip. To this end, they send CSD a request. CSD checks its rules and regulations and answers accordingly. The CSD regulations are often cryptic and subject to change. CSD researchers

1541-1672/07/$25.00 © 2007 IEEE
Published by the IEEE Computer Society

often have obsolete information about administration regulations and little interest in them generally. However, they must abide by them. How can we help this situation?

Solution 1. CSD publishes all regulations and relevant forms on its Web site. Every time Sarah needs to travel, she reads the most recent regulations, downloads the relevant forms from the Web, does the necessary paperwork, and delivers the filled-in forms to CSD administration. She solves any problems by direct interaction with the administration staff.

This solution has many drawbacks. Employees don't necessarily want to keep up-to-date with new regulations, nor is it their job to do so. Furthermore, technology is helping provide information in this solution, but it isn't helping use it properly. All the paperwork and interaction with the administration are still there and represent potential sources of mistakes and misunderstandings. Moreover, the requirement for direct interaction to solve problems means Sarah has to coordinate her time with administration office hours. Solving problems can become a lengthy and frustrating process on both sides.

Solution 2. The administration feeds all department regulations into a server, which publishes them on the CSD Web site using a semantically rich, machine-understandable, Web-friendly format. Sarah has a PDA, which runs an intelligent agent that can automatically download information from the CSD Web service, parse it, and reason from it. Whenever Sarah needs to travel, she queries her PDA to know if her trip is approved.

This solution solves some of solution 1's problems. When Sarah needs to obtain a CSD service, she doesn't have to read the department's regulations but can instead let her PDA do the job on her behalf. Sarah interacts with her PDA by simply assigning goals to it—for example, "attend conference." Because the rules are published in a machine-understandable format and a semantically rich language, the intelligent PDA agent can understand their meaning, reason from them, and determine whether Sarah's goal can be accomplished given the current regulations. Her PDA can access the CSD Web service even when she's away, so this is an "anywhere-anytime" solution that eliminates interaction problems due to misunderstandings and limited office hours.

However, this solution presents a serious drawback: the exchange between Sarah and CSD has lost its interactive, dialogical character. Why is this a problem? Consider solution 1 again. If Sarah's request is rejected, Sarah can interact with the administration staff and find out why. Solution 2 doesn't permit this interaction. Indeed, a well-known barrier to human adoption of IT solutions is that IT tends to provide definitional answers rather than informed justifications that users could argue with and, possibly, eventually understand and accept. Even an elegant and efficient solution such as this one, based on a Semantic Web service, would hardly prevent Sarah from going and talking directly to the administration to challenge every negative answer her PDA obtains.

Solution 3. The CSD's service and Sarah's PDA agent interact by exchanging arguments in a dialogical fashion. Sarah's PDA not only posts requests to the CSD service and obtains replies but also reasons from such replies. When the replies are negative, the agent challenges them and tries to understand ways to obtain alternative, positive replies. If necessary, the agent can provide fresh information that could inhibit some regulations and activate others.

This solution combines the benefits of the previous two. It delegates most of the reasoning and interaction to the machine by relying on Semantic Web service technology, and it gives Sarah and CSD understandable, justified answers and decisions. The whole process is a machine-supported, collaborative problem-solving activity rather than a flat client-server, query-answer interaction.

We can consider Semantic Web services as a concrete instantiation of multiagent systems, in which the type of messages is restricted to request-response patterns.

Reasoning and argumentation

Reasoning occurs at different levels. For example, Loredana Laera and her colleagues propose an argumentation framework for reaching agreements over ontology alignments.[3] Agreement over ontologies and, more generally, over service descriptions and semantics is one part of Semantic Web service interactions. Another part is the high-level reasoning that supports message exchanges among services. To the best of our knowledge, the only research addressing the latter is in the context of argumentation-based dialogues for multiagent systems. In the multiagent literature, we typically find rich interaction protocols aimed at supporting persuasion, negotiation, and so on, as well as dedicated architectural components, such as commitment stores. We can nevertheless consider Semantic Web services as a concrete instantiation of multiagent systems, in which the type of messages exchanged is generally restricted to request-response patterns.

Argumentation is a natural way of conceptualizing nonmonotonic reasoning, appropriately reflecting its defeasible nature. The Semantic Web is a source of defeasible knowledge: it's open by nature and subject to inconsistencies deriving from multiple sources and incompleteness. So, the Semantic Web appears to be an extremely suitable domain for applying argumentation theories, especially when the services interact with each other on the basis of different and possibly inconsistent knowledge.

Interaction can occur to request services and to coordinate and exchange information. In the Semantic Web, such information will also include rules and logical constructs. The exchange requires suitable reasoning tools that can consider logical constructs as first-class entities and suitable interaction models that can provide the means to exchange rules, implications, conclusions, assumptions, and so on. Argumentation theories suit this task perfectly at both the reasoning and the interaction levels.

Abductive-logic programming

Our research builds on Phan Minh Dung's work on the acceptability of arguments.[4] SCIFF is both an ALP language and a proof procedure for generating grounded sets of arguments starting from a knowledge base.[1] Using the SCIFF ALP framework to construct arguments, we can map the arguments onto ALP *abducibles*—that is, unknown facts that SCIFF can hypothesize and reason about as if they were true.

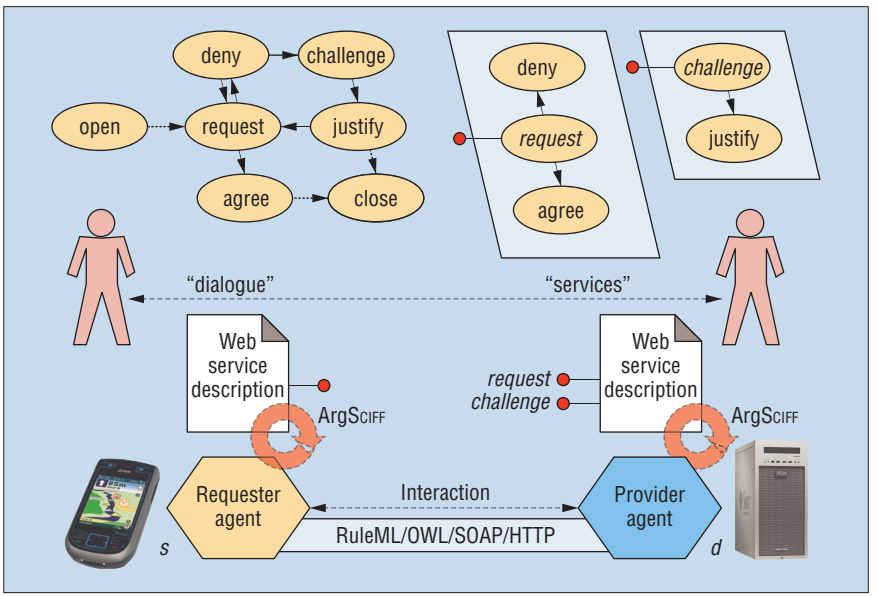

Figure 1. The ArgSciff architecture extends the Semantic Web service architecture with argumentation technology implemented through request and challenge methods. The ArgSciff argumentation protocol is asymmetric: the requester agent sees a dialogue, and the provider agent sees service requests.

Using Sciff to construct arguments has some important advantages. Sciff programs consist of rules, definitions, known facts, and events that can occur dynamically. Such elements can contain variables, quantified in various ways and possibly subject to constraints and quantifier restrictions. It's therefore an expressive language. To the best of our knowledge, most existing frameworks implementing argumentation are propositional and presume static knowledge. With Sciff, however, you can use terms to encode data structures whose size isn't known a priori. You can also represent events in a parametric way ("Employee *X* has been authorized by *Y* at time *T*") and reason from such parameters.

Moreover, Sciff distinguishes between events that are known to have happened and events that are expected either to happen or not to happen. All these elements together make it easy to represent interaction protocols,[1] norms and regulations,[5] Web service specifications,[6] and situations such as those we described in our scenario. Expected events represent future actions (Sarah traveling), whereas happened events represent facts that can become known to Web services as a dialogue develops (for example, when Sarah notifies CSD that she holds a trip authorization from the CSD head, the CSD knowledge increases by one fact).

We have developed ArgSciff as an integrated suite of extensions and tools developed on top of Sciff. The extensions address Web service discovery and contracting,[6] and the tools address formal verification—both a priori (for example, abductive-logic Web service specification and verification) and at runtime (for example, monitoring interaction-protocol execution). So, after an argumentation-based dialogue leads to an agreement ("Sarah can travel and is entitled to ask for reimbursement"), ArgSciff and the Sciff procedure can easily verify that the actual behavior of the parties involved conforms to such an agreement.

All material, including tools, is available from the Sciff Web site, http://lia.deis.unibo.it/sciff.

ArgSciff: Extending the Semantic Web architecture

We can view the Semantic Web as a layered architecture. At the bottom are standards for unique resource identification, text encoding, message and resource descriptions, and ontologies. On the top layer, the Semantic Web emerges as a trusted convergence point of core technologies based on semantic descriptions, security technologies, logical models, and automated reasoning procedures. The central layers mediate between the bottom (ontology) layer and the top (logic and proof) layers. This is where reasoning about Semantic Web resources takes place.

Machine-to-machine interaction over the network occurs via Semantic Web services. Web services are an instance of the service-oriented-computing paradigm. They are stateless servers, implemented by software agents, interacting with each other through simple request-response message exchanges. Service providers use WSDL to specify Web service descriptions—specifically, XML descriptions of the service's methods and the concrete network protocols and message formats needed to access them.

The World Wide Web Consortium has a recommendation that supports semantic descriptions of Web services (http://w3.org/2002/ws/sawsdl). Such descriptions could take the form of rules. More specifically, the Semantic Web Services Language is a general-purpose language to formally characterize service concepts and descriptions. Its sponsors have submitted SWSL to the W3C for consideration as a recommendation (http://w3.org/Submission/SWSF). SWSL contains several sublanguages, including SWSL-Rules, which is based on RuleML-serialized logic programming and aims to support the use of the service ontology in reasoning and execution environments.

Figure 1 shows the ArgSciff architecture as an extension of the Web service architecture. On the left side of the figure is an agent, *s*, which could be the intelligent agent running on Sarah's PDA. On the right side, we have a Semantic Web service, *d*, which could be the CSD's Web service. *s* and *d* interact with each other using Semantic Web technologies. From the Semantic Web's ontology layer downward, *s* and *d* will adopt some agreed-upon standard. In the current prototype, they exchange SOAP messages, which can contain Sciff rules. Messages are passed using an Internet transfer protocol such as HTTP. *s* and *d* will adopt some common domain-related ontology, such as one provided by the CSD for inquiries about regulations. At the logic level, *s* and *d* use knowledge expressed by Sciff programs. At the proof level, they use the ArgSciff proof procedure to evaluate queries and replies, according to the abductive semantics we define in the next section.

The exchanged messages follow a simple request-reply protocol, but at a high level, we can view the way *d* interacts with *s* as a dialogue, in which *s* argues for its case against *d*. From *d*'s standpoint, no dialogue occurs. *d* simply provides two methods: *request* and *challenge*. The two different views of the ongoing interaction nevertheless generate a decoupling, and this decoupling makes it possible to marry stateless Web services with argumentation dialogues.

SCIFF abductive semantics

"Argument" is a semantically overloaded term.[7] We'll define it formally later, but in the general context of argumentation frameworks, we use "argument" and "argumentation" in the sense that Dung uses them in his seminal work.[4] Specifically, an argument is an abstract entity whose role is solely determined by its relation to other arguments. We pay no special attention to an argument's internal structure. An argumentation framework is defined as a pair $AF = \langle AR, attacks \rangle$, where AR is a set of arguments and *attacks* is a binary relation on AR—that is, $attacks \subseteq AR \times AR$. Related to this notion of attacks is that of *defense*: an argument can defend itself from an attacking argument by having a set S of arguments that attack the attacking argument in turn. Accordingly, S supports the first argument.

Dung gives a model-theoretic semantics to abstract argumentation frameworks via the notion of *admissibility*. In particular, an AF's admissible models are sets of arguments that don't attack each other and can defend each other from attacks originating from the outside. We can map Dung's AF onto the SCIFF ALP framework and show that the sets of arguments SCIFF produces are admissible in Dung's sense.

The SCIFF ALP proof procedure

ALP is a computational paradigm aimed at introducing hypothetical reasoning in the context of logic programming.[8] A logic program $\mathcal{P}$ is a collection of clauses with an associated notion of entailment indicated by $\models$. In ALP, the abductive reasoner can assume some predicates—namely, *abducibles*, belonging to a special set $\mathcal{A}$—to be true, if need be. To prevent unconstrained hypothesis-making, ALP programs typically contain expressions that must be true at all times, called *integrity constraints* (IC). $\mathcal{IC}$ indicates a set of such ICs, whereas *ic* indicates a singleton integrity constraint (an *ic* in SCIFF is an implication written as $Body \rightarrow Head$). An abductive-logic program is the triplet $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$, with an associated notion of abductive entailment.

SCIFF provides the reference-logic framework for ArgSCIFF. A distinguishing feature of SCIFF is its notion of expectations about events. Events are denoted as H atoms. Expectations are abducibles denoted as $E(X)$ (positive expectations) and $EN(X)$ (negative expectations), where $E(X)/EN(X)$ stands for "X is expected/expected not to happen." For example, we can express the expectation that Sarah won't attend a conference by the atom $EN(action(attend(sarah, conf)))$. Variables in events, expectations, and other atoms can be subject to constraint-logic programming (CLP) constraints and quantifier restrictions (intuitively, quantifier restrictions are constraints on universally quantified variables). The following example IC

$$not\ H(tell(csd, X, authorization)) \rightarrow EN(action(attend(X, C))) \quad (1)$$

means, "If an (individual) X does not hold authorization from *csd*, X is expected not to attend (any conference) C." We use the functor *tell* to represent communicative actions, and the functor *action* to represent all other actions.

In equation 1, failure to hold authorization is mapped onto a negative H literal. In the SCIFF language, H denotes events or facts that can become known in a dynamic fashion, and it supports SCIFF's ability to reason about them.

Two fundamental SCIFF concepts are hypothesis consistency and goal entailment, where a goal G reflects a logic-programming conjunction of literals and possibly constraints.

DEFINITION 1. *A set of hypotheses* Δ *is* consistent *if and only if* $\forall$ (ground atom) p, $p \in \Delta \rightarrow not\ p \notin \Delta$ *and* $\forall$ (ground term) t, $E(t) \in \Delta \rightarrow EN(t) \notin \Delta$.

Definition 2 summarizes SCIFF's abductive semantics. It's based on Kenneth Kunen's 3-valued completion semantics;[9] as such, it relies on Clark's Equality Theory (CET).

DEFINITION 2. *A SCIFF ALP* $S = \langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ entails *a goal* G (*written* $S \models_{\Delta} G$), *if and only if* $\exists\ \Delta \subseteq A$ *such that* Δ *is consistent and*

$$\begin{cases} Comp(\mathcal{P} \cup HAP \cup \Delta) \cup CET \cup T_x \models G \\ Comp(\mathcal{P} \cup HAP \cup \Delta) \cup CET \cup T_x \models \mathcal{IC} \end{cases}$$

where Comp stands for completion, T_x *is the constraints theory, and HAP is the set of known events.*

To exemplify, consider the following ALP:

$$\mathcal{P} = \{ p \leftarrow E(t), E(s) \}$$
$$\mathcal{IC} = \{ E(t) \rightarrow EN(s) \vee E(r) \} \quad (2)$$

The abductive program $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ entails the goal p by a set $\Delta = \{E(t), E(s), E(r)\}$.

SCIFF operates by considering G together with $\mathcal{IC}$ and by calculating a *frontier* as a disjunction of formula conjunctions. Each step in this process uses one of the inference rules defined in the SCIFF framework.[1] Given the frontier, a selection function can pick one among the equally true disjuncts at any step; we call this selection an *abductive answer* to G. When no inference rule applies (*termination*), if there exists one disjunct that isn't false, then SCIFF succeeds and the frontier contains at least one abductive answer (Δ) to G.

To exemplify, let's consider the following IC, which could belong to Sarah:

$$H(tell(csd, sarah, deny(E(action(A))))) \rightarrow EN(action(A)) \vee challenge(csd, EN(action(A)))$$

Let its head elements be abducible predicates, and let *HAP* contain

$$H(tell(csd, sarah, deny(E(action(attend(conf))))))$$

The frontier will then eventually contain at least two disjuncts:

- Δ_1, holding $EN(action(attend(conf)))$, and
- Δ_2, holding $challenge(csd, EN(action(attend(conf))))$.

The intuitive reading is that once CSD has told Sarah that it denies her request to attend the conference, the world can evolve in two possible ways: Δ_1, by which Sarah accepts that she can't attend the conference (and possibly tries to satisfy her goal in other ways), or Δ_2, by which she challenges CSD's denial of authorization.

ArgSCIFF argumentation

Following the work of Antonis Kakas and Francesca Toni,[10] ArgSCIFF maps arguments to abducibles. In particular, arguments can

be generic abducibles or expectations. As we noted earlier, expectations about events are particularly suited to representing actions, which leads to smooth modeling of regulations and norms.[5] So, as definition 3 specifies, ArgSciff lets the involved parties consider actions and other normative elements as arguments that they can propose and the system can reason about.

Definition 3. *An* Argument *is a literal p or not p of an abducible predicate p, where p could be any element of $\mathcal{A}$, including expectations in the form E(t)/EN(t), where t is a term.*

From now on, if not explicitly stated otherwise, we'll refer to an arbitrary but fixed instance $S = \langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ of a Sciff program. We also use the terms "hypotheses" and "arguments" interchangeably.

We can now recast Dung's notion of attacks as a binary relation so that it fits the ALP semantics of the Sciff framework.

Definition 4. *A set of arguments A attacks another set Δ if and only if at least one of the following expressions is true:*

$S \models_A not\ p$, *for some* $p \in \Delta$,
$S \models_A E(t)$, *for some* $EN(t) \in \Delta$, or
$S \models_A EN(t)$, *for some* $E(t) \in \Delta$.

In the example of equation 2, $A = \{E(t), E(s), E(r)\}$ *attacks* $\Delta_1 = \{EN(s)\}$ and $\Delta_2 = \{not\ p\}$.

We can prove that ArgSciff has the properties that Kakas and Toni considered fundamental of an attacking relation:[10]

- No set of arguments attacks the empty set of arguments ∅.
- *attacks* is monotonic—that is, for all (consistent) A, A', Δ, and Δ' $\subseteq \mathcal{A}$, if A *attacks* Δ, then
  (i) if $A \subseteq A'$ then A' *attacks* Δ, and
  (ii) if $\Delta \subseteq \Delta'$ then A *attacks* Δ'.
- *attacks* is compact—that is, $\forall A, \Delta \subseteq \mathcal{A}$, if A *attacks* Δ then there exists a finite $A' \subseteq A$ such that A' *attacks* Δ.

The notion of *attacks* in definition 4 is symmetric, which makes ArgSciff a symmetric argumentation framework. Moreover, *attacks* is irreflexive and, in all nontrivial cases, nonempty. This leads to the agreement of several semantics and makes ArgSciff a coherent, grounded framework.[11] However, we'll focus on the admissible sets semantics, which suffices for our purposes here.

For a set of arguments A such that $S \models_A p$ for some p, it follows from Sciff's declarative semantics that A is consistent and that if a set of arguments Δ is attacked by A, $A \cup \Delta$ isn't consistent in the Sciff sense.

In the following definition, we extend Dung's abstract argumentation framework:[4]

Definition 5. *A set Δ of arguments is said to be* conflict-free *if there are no sets of arguments A and B ⊆ Δ such that A attacks B.*

For example, the set $\Delta = \{E(t), E(s), E(r), EN(s)\}$ isn't conflict-free because it contains $A = \{E(s)\}$ and $B = \{EN(s)\}$ and A *attacks* B.

It follows from definition 5 that all consistent argument sets in the Sciff sense are conflict-free and therefore that all arguments A such that $S \models_A p$ are conflict-free.

Finally, we define admissible sets of arguments according to the work of Dung[4] and Kakas and Toni.[10]

Definition 6. *A (conflict-free) set of arguments Δ is* admissible *if and only if for all sets of arguments A, if A attacks Δ, then Δ attacks A \ Δ.*

Dung's Fundamental Lemma,[4] together with the fact that the empty set is always admissible, implies that all arguments A such that $S \models_A p$ are admissible sets of arguments for S. This result determines an ArgSciff semantics based on admissible sets. In other words, Web services using ArgSciff will produce requests and responses that contain only consistent argument sets and that can therefore defend each other against attacks of external defeaters.

Dung defines preferred extensions as maximal sets of admissible sets of arguments,[4] but we focus here instead on admissible sets of arguments. In fact, as Kakas and Toni stress,[10] because every admissible set of arguments is contained in some preferred extension, determining that a given query holds with respect to the semantics of admissible sets is sufficient for determining that the query holds with respect to the preferred extension and partial stable-model semantics.

ArgSciff lets the involved parties consider actions and other normative elements as arguments that they can propose and the system can reason about.

The *attacks* relation can apply to all arguments, including elements of an *IC*'s body. For example, if we consider $ic = Body \rightarrow Head$ and $p \in Body$, then *not p* represents an attack to *ic*'s body. The reasoning Web service agent can use such an attack to inhibit *ic*. This corresponds to the concept of *undercut*, which appears in the argumentation literature. We can now show how ArgSciff implements this feature for use inside dialogues.

ArgSciff proof theory

ArgSciff's proof-theoretic semantics is based on the Sciff proof procedure.[2] The Sciff procedure is a rewriting system that transforms one node into other nodes and, starting from an initial node, defines a proof tree. A node can be either the special node *false* or a node defined by the tuple

$$T \equiv \langle R, CS, PSIC, HAP, \Delta \rangle \quad (3)$$

where R is the *resolvent*—that is, a conjunction of literals; *CS* is the constraint store, containing CLP constraints and quantifier restrictions; *PSIC* is a set of implications; *HAP* is the history of happened events, represented by a set of events; and Δ is the set of hypotheses generated by Sciff (corresponding to a set of arguments in ArgSciff).

If one element of the tuple is *false*, then the tuple is the special node *false*, without successors. A derivation D is a sequence of nodes $T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow T_n$.

Given a goal G, a set of integrity constraints $\mathcal{IC}$, and an initial history HAP^i, the first node is $T_0 \equiv \langle \{G\}, \emptyset, \mathcal{IC}, HAP^i, \emptyset \rangle$. We obtain the other nodes by applying transitions until no further transition can be applied.

DEFINITION 7. *Given an initial history HAP^i, a SCIFF program $S = \langle \mathcal{P}, \mathcal{A}, IC \rangle$ and a set $HAP^f \supseteq HAP^i$, there exists a* successful derivation *for a goal G if and only if the proof tree with root node T_0 has at least one leaf node $\langle \emptyset, CS, PSIC, HAP^f, \Delta \rangle$, where CS is consistent. In such a case, we write*

$$S_{HAP^i} \vdash_{\Delta}^{HAP^f} G$$

The transitions are based on those of Fung and Kowalski's IFF proof procedure,[2] enlarged with those of CLP and with specific transitions accommodating the concepts of dynamically growing history and consistency of the set of expectations. The transition inference rules are

- *Unfolding*: Substitutes an atom p with its definitions in P:

$$p^1 \leftarrow l_{i1} \wedge \ldots \wedge l_{im_1}$$
$$\vdots$$
$$p^n \leftarrow l_{in} \wedge \ldots \wedge l_{im_n}$$

  If p occurs in R, this rule generates n new nodes. If p occurs in the body of an $ic \equiv p \wedge B \rightarrow H$, then it generates one node with n *IC*s:

$$p = p^1 \wedge l_{i1} \wedge \ldots \wedge l_{im_1} \wedge B \rightarrow H$$
$$\vdots$$
$$p = p^n \wedge l_{in} \wedge \ldots \wedge l_{im_n} \wedge B \rightarrow H$$

- *Propagation*: If a literal $p \in \Delta$ and $ic_i \equiv p_1 \wedge B \rightarrow H$, generates a new node with the additional $ic \equiv (p = p_1) \wedge B \rightarrow H$.
- *Splitting*: Distributes conjunctions and disjunctions so that the final formula takes a sum-of-products form.
- *Case analysis*: If $ic_i \equiv (X = t) \wedge B \rightarrow H$, generates two nodes: one with $X = t$ and $ic_i \equiv B \rightarrow H$, and the other with $X \neq t$ and ic_i substituted with *true*.
- *Factoring*: If $p_1, p_2 \in \Delta$, generates two nodes: one with $p_1 = p_2$ and the other with $p_1 \neq p_2$.
- *Rewrite rules for equality*: Uses CET inference rules to perform unification (thus $p(t_1, \ldots, t_n) = p(s_1, \ldots, s_n)$ is replaced by $\wedge_{i=1}^{n} t_i = s_i, \ldots$).
- *Logical simplifications*: Simplify a formula through equivalences such as $A \wedge false \leftrightarrow false$, $[A \leftarrow true] \leftrightarrow A$, ….

Additional, SCIFF-specific inference rules are

- *Happening*: Adds a new happened event $H(t)$ to the set *HAP*.
- *Closure*: Assumes that no more events can happen (sets a *closure* flag to *true*). Used to reason under the Closed World Assumption.
- *Nonhappening*. If $IC_i \equiv not\ H(X) \wedge B \rightarrow H$, and *closure* = *true*, performs constructive negation to derive that for each possible instance of $H(X)$ that doesn't unify with any *HAP* element, $B \rightarrow H$ holds.
- *Consistency*. If $\{E(X), EN(Y)\} \subseteq \Delta$ (or $\{p(X), not\ p(Y)\} \subseteq \Delta$), imposes $X \neq Y$.
- CLP: Performs CLP reasoning.

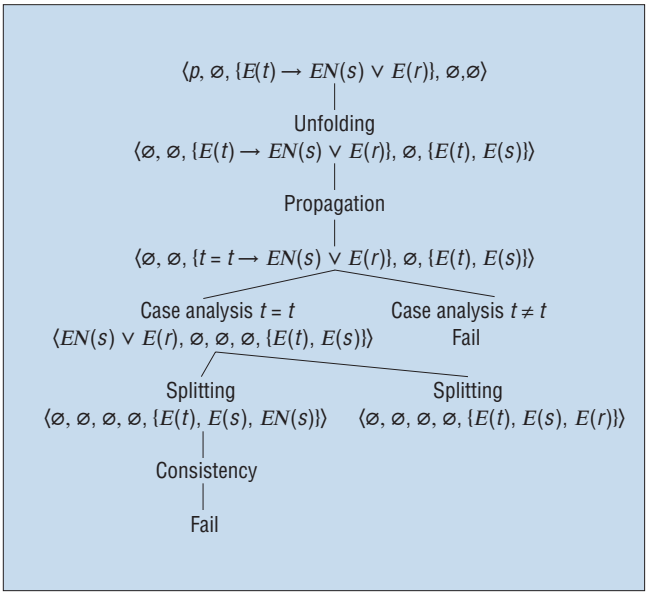

Figure 2. Derivation tree for the goal *p* in the example of equation 2.

Figure 2 presents a (simplified) derivation tree for the equation 2 example.

ArgSCIFF extends SCIFF in two ways. First, it introduces a notion of *attacks* that plays a counterpart to the admissible-sets semantics introduced earlier. Second, it accommodates the acquisition of and reasoning on new knowledge (happened events or integrity constraints received from the counterpart). In particular, new ICs could be part of the content of a *justify* message, by which an agent maintains that it can't accept a request because of a specific IC.

Reasoning upon ICs taken from the outside is accommodated by a new transition:

- *Deny Body*. Upon receipt of an argument of the type $ic = [Body \rightarrow Head]$, transition Deny Body attacks it by inserting in the resolvent R a literal L such that $\{L\}$ *attacks* (in the sense of definition 4) *Body*.

Essentially, in ArgSCIFF, an agent tries to attack the incoming arguments that it can't accept. So, if these arguments are based on ICs, the agent tries to undercut the implication by attacking its preconditions.

ArgSCIFF implementation

We developed a prototype ArgSCIFF implementation using a suite of logic-programming and Semantic-Web technologies. These included SICStus Prolog and constraint-handling rules for implementing the transitions, RuleML 0.9 as a mark-up language for exchanging rules such as those in *IC*, and Axis2 to realize Web services. A Web form user interface enables calls to CSD's Web services.

We followed the SCIFF operational semantics to implement ArgSCIFF. All exchanged messages are of the kind

H(*tell*(*Sender*, *Receiver*, *Content*))

in which *Content* can be any term, including expectations or rules. Messages are communicated through the activation of a goal:

sendMessage(*Receiver*, *Content*)

which schedules a message for delivery. *sendMessage* is abducible. The implementation doesn't actually deliver the message until the derivation procedure terminates because the set of expectations in intermediate nodes could be inconsistent. Only after the derivation terminates are *sendMessage* abducibles selected from the frontier to generate corresponding messages and package them for delivery.

Attacks and undercuts

Two facts encode the *attacks* relations of definition 4:

$$attacks(E(X), EN(X)) \quad (4)$$

and

$$attacks(EN(X), E(X)) \quad (5)$$

We specialize the notion of attack further to events. For instance, a rule whose precondition contains a *not H* literal applies only if such an event doesn't happen. Such a rule's consequences are then undercut by a possible matching event happening. Hence, the parties could attack each other's arguments by event generation. Operationally,

$$attacks(not\ H(tell(sarah, X, Content)), sendMessage(sarah, X, Content)) \quad (6)$$

defines Sarah's "active" attack against an argument of type *not H*. The attack is precisely the message being sent.

Finally, undercuts are implemented by the definition

$$attacks(Body \rightarrow Head, Literal) \leftarrow member(Atom, Body) \wedge attacks(Atom, Literal) \quad (7)$$

Argumentation protocol

Figure 1 depicts the ArgSciff argumentation protocol. It's asymmetric because it involves entities of different natures. Services (on the right side of figure 1) are stateless, reactive entities that simply reply to known requests. In particular, services can agree to or deny a requesting agent's request and can give justifications in response to challenges.

Requesting agents (on the left side of figure 1) are proactive entities that engage in dialogues to achieve a goal. They can request services and challenge denials. The dialogue protocol's implementation relies on two kinds of knowledge:

- a domain-independent knowledge that encodes the argumentation protocol and is the same for both parties, and
- specific, private knowledge, which distinguishes one party from the other.

Here, we show the domain-independent knowledge base in relation to the argumentation protocol's messages.

> The dialogue protocol implementation relies on domain-independent knowledge to encode the argumentation protocol and private knowledge to distinguish between parties.

Request. When a peer receives a request message, it will try to accept it—that is, it will abduce the expectation carried by the message. This might succeed or it might generate a failure—for example, because of a clash with other expectations. In the case of success, the peer will agree; in the case of failure, it will deny the argument and send an attacking argument.

$$H(tell(X, Y, request(E(action(A))))) \rightarrow E(action(A)) \wedge sendMessage(X, agree(E(action(A)))) \vee sendMessage(X, deny(E(action(A)), reason(EN(action(A))))) \quad (8)$$

$$H(tell(X, Y, request(EN(action(A))))) \rightarrow EN(action(A)) \wedge sendMessage(X, agree(EN(action(A)))) \vee sendMessage(X, deny(EN(action(A)), reason(E(action(A))))) \quad (9)$$

We can use preferential reasoning to select between *agree* and *deny* by giving higher preference to accepting expectations than to avoiding them. In the current implementation, we've realized a built-in, primitive, preferential-reasoning strategy by simply applying a left-to-right selection rule in the search-space exploration.

Deny. In the case of denial, a requesting agent will try to accept the denial by abducing the attacking expectation. It might then iterate by sending a different request. However, if no alternative way to achieve its goal exists, the agent will try to challenge the attacking peer's expectation—that is, it will ask the peer why its request wasn't satisfied:

$$H(tell(Service, Agent, deny(E(action(A)), reason(EN(action(A))))) \rightarrow EN(action(A)) \vee sendMessage(Service, challenge(EN(action(A)))) \quad (10)$$

$$H(tell(Service, Agent, deny(EN(action(A)), reason(E(action(A))))) \rightarrow E(action(A)) \vee sendMessage(Service, challenge(E(action(A)))) \quad (11)$$

Challenge. The reply to a challenge is an agent or service *X*'s set of rules that can generate an attacking argument. In this way, the other peer (*Y*) can reason on new, previously unknown rules and possibly attack the arguments that *X* puts forward. An IC implements this mechanism:

$$H(tell(X, Y, challenge(Argument))) \rightarrow replyChallenge(Y, X, Argument)$$

where *replyChallenge* filters out and groups together all relevant rules—for example, a selection of those used by *X*'s reasoning engine (Sciff) to generate *Argument*—and sends them packaged in a *justify* message.

Justify. A *justify* message contains a rule that explains how the agent or service produced the *challenged* argument. An agent receiving a *justify* message will try to undercut the argument:

$$H\,(tell(X, Y, justify(Argument))) \rightarrow attacks(Argument, AttackingArgument) \wedge sendMessage(X, request(AttackingArgument))$$

The *attacks* predicate is defined in a knowledge base (equations 4–7). It relies on the semantics of attacks and will try to prove that the *AttackingArgument* is true. In other words, it will try to find an admissible set containing the *AttackingArgument*. If it does, it sends a corresponding message to the other party in the form of a new request.

Sarah versus CSD reloaded

We now show Sarah's and CSD's specific knowledge and demonstrate how we implemented the running example.

Besides the general knowledge base and ICs used to implement the argumentation framework, Sarah also has a goal G—namely, attending the conference *conf*:

$$G \leftarrow E(action(attend(sarah, conf)))$$

Moreover, she knows that whenever she wants to attend a conference, she must send a request to the administration's authorization service:

$$E(action(attend(sarah, conf))) \rightarrow sendMessage(csd, request(E(action(attend(sarah, conf)))))$$

The CSD administration service has the following rule:

$$not\ H(tell(X, csd, authorization)) \rightarrow EN(action(attend(X, C))) \quad (12)$$

which means that no member of the department can attend any conference, unless authorized.

Sarah tries to satisfy her goal and abduces $E(action(attend(sarah, conf)))$. She sends this argument to CSD in a request message. CSD tries to accept Sarah's argument but finds that it clashes with the argument stated in equation 12. It therefore doesn't accept the argument and sends a term

$$deny(E(action(attend(sarah, conf))), reason(EN(attend(sarah, conf))))$$

to Sarah (see equation 8). Sarah, on her side, tries to accept the department's argument, but it clashes with her goal. Her only choice is to attack the argument. Using the rule in equation 10, she challenges the department to explain why it denied her request. The department replies by selecting the appropriate rules from the knowledge base via the *replyChallenge* predicate and communicating them to Sarah via a *justify* message. In particular, CSD sends Sarah the rule in equation 12.

Now Sarah knows why the department didn't accept her argument and tries to undercut the received rule by negating its preconditions. Equation 12's only precondition is a missing authorization, *not* $H(tell(X, csd, authorization))$, and Sarah knows (by the rule in equation 6) that she can provide the authorization and thus attack the ground for CSD's argument.

Sarah queries the CSD service again, providing the authorization together with her goal:

$$H(tell(sarah, csd, authorization)), E(action(attend(sarah, conf)))$$

Once CSD knows that Sarah has an authorization, it no longer generates an argument against her request. Instead, according to the SCIFF semantics, it generates an argument containing a quantifier restriction:

$$\forall_{X \neq sarah}\ EN(action(attend(X, C)))$$

meaning that no one except Sarah can attend a conference. Because this argument doesn't clash with Sarah's goal, CSD agrees with her request.

> Our first simulations show that a dialogical interface facilitates users' acceptance of applied regulations because they can better understand them.

Besides identifying an important role for argumentation in the Semantic Web, the work we report here advances the state of the art by casting the semantic and operational foundations for the ArgSCIFF architecture. This architecture provides a solid base for future implementations to support argumentation-based interaction among Web services.

The prototype system implementation is currently undergoing testing. Our first simulations, which ran in a controlled environment inside our department, show that a dialogical interface facilitates users' acceptance of applied regulations because they can better understand them. Future work includes extensive empirical system evaluation and experimentation, study of the proof procedure's computational complexity, identification of tractable problem classes, and a deeper investigation of combining argumentation technology and ALP. ■

Acknowledgments

We thank Paola Mello and the anonymous reviewers for their valuable feedback. This work has been partially supported by national projects PRIN-2005-011293, PRIN-2005-015491, and TOCAI.it ("Knowledge-Oriented Technologies for Enterprise Aggregation in the Internet").

References

1. M. Alberti et al., "Verifiable Agent Interaction in Abductive Logic Programming: The SCIFF Framework," to be published in *ACM Trans. Computational Logic*; http://tocl.acm.org/accepted.html.
2. T.H. Fung and R. Kowalski, "The IFF Proof Procedure for Abductive Logic Programming," *J. Logic Programming*, vol. 33, no. 2, 1997, pp. 151–165.

The Authors

Paolo Torroni is an assistant professor of computer engineering at the University of Bologna's Department of Electronic Engineering. His research interests include using logic in computer science and AI, particularly declarative and logic programming, hypothetical reasoning, argumentation, and agent-based systems. He received his PhD in computer science from the University of Bologna. He's a member of the Italian Association for Artificial Intelligence and the secretary of the Italian Interest Group on Logic Programming. Contact him at DEIS, Univ. of Bologna, V.le Risorgimento 2, 40136 Bologna, Italy; paolo.torroni@unibo.it.

Marco Gavanelli is an assistant professor of computer engineering at the University of Ferrara's Department of Engineering. His interests are in Constraint Logic Programming languages, abductive reasoning, multicriteria optimization, and reformulation of combinatorial problems. He received his PhD in information engineering from the University of Modena and Reggio Emilia. He's a member of the Italian Association for Artificial Intelligence, coordinator of its interest group on Knowledge Representation and Automated Reasoning, and a member of the Italian Interest Group on Logic Programming. Contact him at the Dept. of Eng., Univ. of Ferrara, Via Saragat 1 – 44100 Ferrara, Italy; marco.gavanelli@unife.it.

Federico Chesani is a research assistant in the University of Bologna's Department of Electronic Engineering. His research interests include abduction and computational logic as well as verification techniques and specification languages applied to multiagent systems and service-oriented computing. He received his PhD in computer science from the University of Bologna. He's a member of the Italian Interest Group on Logic Programming. Contact him at DEIS, Univ. of Bologna, V.le Risorgimento 2, 40136 Bologna, Italy; federico.chesani@unibo.it.

3. L. Laera et al., "Reaching Agreement over Ontology Alignments," *Proc. 5th Int'l Semantic Web Conf.* (ISWC 06), LNCS 4273, Springer, 2006, pp. 371–384.
4. P. Dung, "On the Acceptability of Arguments and Its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and N-Person Games," *Artificial Intelligence*, vol. 77, no. 2, 1995, pp. 321–357.
5. M. Alberti et al., "Mapping Deontic Operators to Abductive Expectations," *Computational and Mathematical Organization Theory*, vol. 12, nos. 2–3, 2006, pp. 205–225.
6. M. Alberti et al., "Web Service Contracting: Specification and Reasoning with SCIFF," *Proc. 4th European Semantic Web Conf.* (ESWC 07), LNCS 4519, Springer, 2007, pp. 68–83.
7. A. Wyner, T. Bench-Capon, and K. Atkinson, "Three Senses of 'Argument,'" *Proc. Workshop Argumentation and Nonmonotonic Reasoning* (ArgNMR 07), 2007, pp. 1–15; http://lia.deis.unibo.it/confs/ArgNMR/proceedings/ArgNMR-proceedings.pdf.
8. A. Kakas, R. Kowalski, and F. Toni, "Abductive Logic Programming," *J. Logic and Computation*, vol. 2, no. 6, 1993, pp. 719–770.
9. K. Kunen, "Negation in Logic Programming," *J. Logic Programming*, vol. 4, 1987, pp. 289–308.
10. A. Kakas and F. Toni, "Computing Argumentation in Logic Programming," *J. Logic and Computation*, vol. 9, no. 4, 1999, pp. 515–562.
11. S. Coste-Marquis, C. Devred, and P. Marquis, "Symmetric Argumentation Frameworks," *Proc. 8th European Conf. Symbolic and Quantitative Approaches to Reasoning with Uncertainty* (ECSQUARU 05), LNCS 3571, Springer, 2005, pp. 317–328.