Expressing and Verifying Business Contracts with Abductive Logic Programming

Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, Marco Montali, and Paolo Torroni

ABSTRACT: *S*CIFF is a declarative language, based on abductive logic programming, that accommodates forward rules, predicate definitions, and constraints over finite domain variables. Its abductive declarative semantics can be related to that of deontic operators; its operational specification is the sound and complete *S*CIFF proof procedure, defined as a set of transition rules implemented and integrated into a reasoning and verification tool. A variation of the *S*CIFF proof procedure (g-*S*CIFF) can be used for static verification of contract properties. The use of *S*CIFF for business contract specification and verification is demonstrated in a concrete scenario. Encoding of *S*CIFF contract rules in RuleML accommodates integration of *S*CIFF with architectures for business contracts.

KEY WORDS AND PHRASES: Abductive logic programming, business contracts, declarative specifications, g-*S*CIFF, *S*CIFF, runtime verification, static verification.

Business contracts are an important conceptual abstraction and a practical guiding and governance mechanism for cross-organizational collaboration. Contracts can, in fact, be considered as the main coordination mechanisms for extended enterprises [40]. A business contract architecture is therefore an important part of the extended enterprise that aims to provide such functionalities as contract management and monitoring [41]. The natural requirements for a contract management framework are (1) a language with clear semantics for specifying contracts and (2) operational procedures that can verify contract properties at design time and, as well, the compliance of the parties to the contract provisions at runtime.

From a high-level, functional viewpoint, a contract management system is a component that is fed the "what" of the problem by domain expert users and takes care of the "how" through a suitable execution model. Computational logics offer a broad range of languages and mechanisms that couple declarative ("what is") specification languages with sound operational ("how to") execution models that need not be disclosed to the user of the specification language. For this reason, frameworks based on computational logic, adequately extended to support event-based monitoring of business activities associated with contracts, should play a key role in contract management systems.

Among the most influential computational logic frameworks for business contract representation and reasoning are courteous logic programming and

This work has been supported in part by the MIUR PRIN 2005 projects Specification and Verification of Agent Interaction Protocols and Vincoli e preferenze come formalismo unificante per l'analisi di sistemi informatici e la soluzione di problemi reali, and by the MIUR FIRB project Tecnologie Orientate alla Conoscenza per Aggregazioni di Imprese in Internet. The authors thank the anonymous reviewers for their useful comments on previous versions of this paper.

International Journal of Electronic Commerce / Summer 2008, Vol. 12, No. 4, pp. 9–38.
Copyright © 2008 M.E. Sharpe, Inc. All rights reserved.
1086-4415/2008 $9.50 + 0.00.
DOI 10.2753/JEC1086-4415120401

defeasible logic (DL) [30, 35]. The former, in fact, is a variant of the latter [13]. These are languages for nonmonotonic reasoning, mainly used in the context of business contracts to enable normative reasoning and to identify and resolve conflicts arising from events and contract rules, reason about violations, specify and enforce reparation obligations, and so on. This article, in the context of contract management systems, is mainly concerned with runtime monitoring and verification of contracts rather than with the ontological and semantic aspects of contract specification. It focuses primarily on the problem of *runtime evaluation* of contract policies—expressions consisting of behavior constraints, event patterns, and states—to determine whether the obligations of the parties have been satisfied or there are violations of the contract [41]. The work is based on *S*CIFF, the language and framework based on computational logic that was conceived in the context of the Societies Of ComputeeS (SOCS) EU Project to specify agent interaction protocols [46]. *S*CIFF consists of a logic language based on abductive logic programming, a sound and complete proof procedure [3, 8], and a software tool that implements it, based on an efficient inference engine and constraint-solving technology [5]. First-class entities in the *S*CIFF language are *events* that represent entities (e.g., actions taken), timeouts associated with deadlines, external events (e.g., messages sent, services requested), and *expectations*, which describe a desired behavior in terms of events. Expectations are related to each other and to events by logical expressions called integrity constraints (ICs). ICs express behavior constraints and are the main building blocks in the specification of policies. Expectations are modeled in *S*CIFF as abducible predicates, since they model events that may happen or that must not happen (but we do not know whether that will be the case). They are assumptions about future events, and reasoning on them means reasoning on hypotheses, as in abductive reasoning. Expectations are related to the deontic concepts commonly used to model normative systems, such as obligation, prohibition, and permission, and this permits a deontic reading of *S*CIFF specifications [9].

This paper proposes *S*CIFF as a language and operational framework with which to specify and reason on business contracts. The deontic reading of *S*CIFF specifications is one of the elements that make the *S*CIFF language a good candidate for a contract specification and reasoning language. Reasoning on contract specifications (and events) can be done at two different stages of contract design and enactment—runtime (as proposed in this paper) and design time (as with DL and courteous logic programming). It is important to enable these two kinds of verification within the same framework and, if possible, use the same specification language so as to minimize translation errors and the unavoidable inaccuracy resulting from the use of different languages. To this end, an extension of *S*CIFF, called g-*S*CIFF, has been defined to verify protocol properties at design time [7]. This paper shows how it can be used to enable design-time reasoning on contracts and the verification of contract properties.

Figure 1 summarizes the components of the *S*CIFF framework that can be used in contract specification and verification. The specification is given through the knowledge base and a set of integrity constraints. It can then be

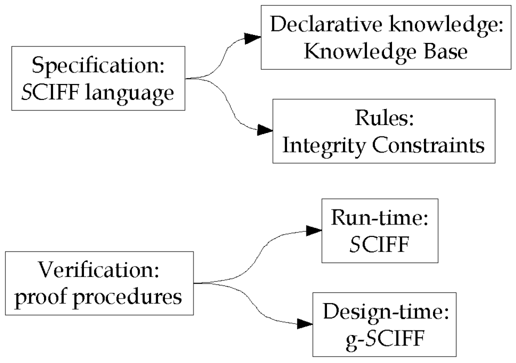
Specification: *S*CIFF language
Declarative knowledge: Knowledge Base
Rules: Integrity Constraints
Verification: proof procedures
Run-time: *S*CIFF
Design-time: g-*S*CIFF

Figure 1. Contract Specification and Verification in the *S*CIFF Framework

used for runtime verification of compliance through the *S*CIFF proof procedure, or for design-time verification by using the g-*S*CIFF proof procedure.

Contract Specification

A contract in the *S*CIFF language is specified by means of two components: a knowledge base, which declaratively defines domain-specific knowledge (e.g., deadlines), and a set of integrity constraints, which describe contract clauses and can be seen as forward rules that generate expectations about the behavior of the parties to the contract (*see Figure 1*). A declarative semantics based on abductive logic programming determines whether the parties have complied with the contract. The use of constraint logic programming (CLP) technology makes deadlines easy to specify and efficient to verify [37].

Syntax of the SCIFF Language

The *S*CIFF language is composed of entities for expressing events and expectations about events, and relationships between events and expectations.

Representation of the Behavior of Parties

Events are the abstractions used to represent actual behavior.

Definition 1: An *event* is an atom:

- with predicate symbol **H**;
- whose first argument is a ground term; and
- whose second argument is a number.

Intuitively, the first argument is meant to represent the description of the happened event, according to application-specific conventions, and the second argument is meant to represent the time when the event happened.

In this paper, all events are mapped to communicative events, identified by the functor *tell*. In particular, the description of happened events is of the format *tell*(*Sender*,*Receiver*,*Content*[,*Dialog*]), where the optional *Dialog* parameter is an identifier of the interaction being described and the other arguments have the obvious meaning.

Example 1:

H*(tell(telco,c,phonebill(39-051-209-3086,145886,205)),19).* (1)

says that *telco* sent to *c* a *phonebill* (whose identifier is *145886* and whose amount is *205*, for the phone number *39-051-209-3086*) at time *19*.

A *negated event* is a negative literal **not H**(...,...), where **not** represents negation as failure.

For the purposes of this discussion, *history* is defined as a set of happened events and is denoted with the symbol **HAP**.

Expectations are the abstractions used to represent the desired events from an external viewpoint. They represent the ideal behavior of the system—the actions that, once performed, will make the system compliant to its specifications. The choice of the term "expectation" is intended to stress that events cannot be forced to be as we would like them to be, but can only be expected.

Expectations are of two types:

- *positive:* representing some event that is expected to happen
- *negative:* representing some event that is expected not to happen

Definition 2: A *positive expectation* is an atom

- with predicate symbol **E**,
- whose first argument is a term, and
- whose second argument is a variable or a number.

Intuitively, the first argument is meant to represent an event description, and the second argument is meant to tell at what time the event is expected (not to be confused with the time when the expectation is generated, which is not modeled by *S*CIFF's declarative semantics). Expectations may contain variables that leave the expected event not completely specified. Variables in positive expectations are always existentially quantified: If the time argument is a variable, for example, this means that the event is expected to happen at *any* time. A specific semantics is not associated to time, but instead is treated an expectation's time argument like any other variable. This choice simplifies the *S*CIFF language's declarative and operational semantics.

Example 2 *The atom*

E*(tell(telco, c, phonebill(39-051-209-3086,Id,Amount)),T).* (2)

says that *telco* is expected to send to *c* a *phonebill* (for the number *39-051-209-3086*, with some identifier *Id*, for some *Amount* of money) at time *T*.

A *negated positive expectation* is a positive expectation with the explicit negation operator $\neg$ applied to it. Variables in negated positive expectations are quantified in the same way as those in positive expectations.

Definition 3: A *negative expectation* is an atom

- with predicate symbol **EN**,
- whose first argument is a term, and
- whose second argument is a variable or a number.

Intuitively, the first argument is meant to represent an event description, and the second argument is meant to tell at which points in time the event is expected not to happen. Like positive expectations, negative expectations may contain variables that typically are universally quantified.[1] For example, if the time argument is a variable, then the event is expected not to happen at *all* times.

Example 3: *The atom*

$$\mathbf{EN}(tell(telco, c, phonebill(39\text{-}051\text{-}209\text{-}3086, Id, Amount)), T). \tag{3}$$

means that *telco* is expected not to send *c* a *phonebill* (for the number *39-051-209-3086*, with any *Id* and for any *Amount*) at any time *T*.

A *negated negative expectation* is a negative expectation with the explicit negation operator $\neg$ applied to it. Variables in negated negative expectations are quantified in the same way as those in negative expectations.

The syntax of events and expectations is summarized in Table 1 and will be used as such in Tables 2 and 3.

Contract Specifications

A contract specification *S* is composed of two elements: a knowledge base and set of integrity constraints.

The *knowledge base* (KB_S) is a set of *Clause*s in which the body can contain (besides defined literals) expectation literals and restrictions.[2] Intuitively, the KB_S is used to express declarative knowledge about the specific application domain.

The syntax of the knowledge base is given in Table 2 and will also be used in Table 3.

A *goal* in the *S*CIFF framework has the same role as in the logic programming literature—as a predicate to be entailed. Therefore, the term "goal" does not necessarily have the typical connotation (of "common" or "social" goal) found in the literature on multiagent systems, although it can be used for such a purpose.

EventLiteral::=[**not**]*Event*

Event::=**H**(*GroundTerm,Number*)

ExpLiteral::=*PosExpLiteral* | *NegExpLiteral*

PosExpLiteral::=[¬]*PosExp*

NegExpLiteral::=[¬]*NegExp*

PosExp::=**E**(*Term,Variable* | *Number*)

NegExp::=**EN**(*Term,Variable* | *Number*)

ExistLiteral::=*PosExpLiteral* | *Literal*

Literal::=[**not**]*Atom*

Table 1. Syntax of Events and Expectations.

Clause::=KBHead←KBBody

KBHead::=*Atom*

KBBody::=*ExtLiteral* [∧ *ExtLiteral*]*[:*Restriction* [,*Restriction*]*]

|*true*

ExtLiteral::=*Literal* | *ExpLiteral*

Table 2. Syntax of Knowledge Base.

The syntax of the goal is the same as the *KBBody* of a clause (*see Table 2*).
Integrity constraints are implications that are used operationally as forward rules, as will be explained further on. Declaratively, they relate the various entities in the *S*CIFF framework (i.e., expectations, events, constraints/restrictions) to the predicates in the knowledge base.

The syntax of ICs is given in Table 3. The *Body* of an IC can contain conjunctions of all the elements in the language (namely, **H**, **E**, and **EN** literals, defined literals, and restrictions). The *Head* contains a disjunction of conjunctions of any of the literals in the language, except for **H** literals.

Contract specification. Given a knowledge base KB_S and a set IC_S of integrity constraints, the pair $<KB_S, IC_S>$ is called a *Contract Specification*. Intuitively, a contract specification is a description of the acceptable, or desirable, histories, as defined by its declarative semantics, given formally in the next section.

Declarative Semantics and Mapping into Deontic Logic

Declarative Semantics

The (abductive) declarative semantics of the *S*CIFF framework is inspired by other abductive frameworks, such as the IFF by Fung and Kowalski [29], but introduces the concept of fulfillment to express a correspondence between expected and actual events. The declarative semantics of a contract specification is given for each specific history. A specification grounded in a history is called an *instance* of the contract.

$$IC_s ::= [\mathrm{IC}]^*$$
$$IC ::= Body \rightarrow Head$$
$$Body ::= (EventLiteral \mid ExpLiteral)[\wedge BodyLiteral]^*[:Restriction\ [,Restriction]^*\ ]$$
$$BodyLiteral ::= EventLiteral \mid ExtLiteral$$
$$Head ::= HeadDisjunct\ [\ \vee HeadDisjunct\ ]^*\ \mid false$$
$$HeadDisjunct ::= HeadLiteral[\wedge HeadLiteral]^*[:Restriction\ [,Restriction]^*\ ]$$
$$HeadLiteral ::= Literal \mid ExpLiteral$$

Table 3. Syntax of Integrity Constraints (ICs).

Definition 4: Contract instance Given a contract specification S and a history **HAP**, $S_{\mathbf{HAP}}$ represents the pair $\langle S,\mathbf{HAP}\rangle$, called the **HAP**-*instance* of S (or simply an *instance of S*).

In this way, $S_{\mathbf{HAP}^i}$, $S_{\mathbf{HAP}^f}$ denotes different instances of the same contract specification S, based on two different histories: $\mathbf{HAP}^i$ and $\mathbf{HAP}^f$, respectively.

An abductive semantics is adopted for the contract instance. Declaratively, a ground set **EXP** of hypotheses should entail the goal and satisfy the integrity constraints. In our case, the set **EXP** of hypotheses is, in particular, a set of ground expectations, positive and negative, possibly negated by explicit negation. Note that, by virtue of explicit negation, all such expectations are positive abducible literals in abductive logic programming terminology.

Definition 5: Abductive explanation Given a contract specification S, an instance $S_{\mathbf{HAP}}$ of S, and a goal G, **EXP** is an abductive explanation of $S_{\mathbf{HAP}}$ for goal G if:

$$Comp(KB_S \cup \mathbf{HAP} \cup \mathbf{EXP}) \cup CET \cup T_X \vDash IC_S \tag{4}$$

$$Comp(KB_S \cup \mathbf{EXP}) \cup CET \cup T_X \vDash G \tag{5}$$

where

- CET is Clark's Equality Theory [24], where equality (=) is considered a special two-valued predicate and the following axioms hold:

  1. $f(X_1,\ldots,X_n) = f(Y_1,\ldots,Y_n) \rightarrow (X1 = Y1) \wedge \ldots \wedge \quad (\forall f)$
  2. $f(X_1,\ldots,X_n) \neq g(Y_1,\ldots,Y_m)$ *(whenever f and g are distinct or n≠m)*
  3. $X \neq T$ *(∀ X and T where X is a proper subterm of T)*

- Comp represents the three-valued *completion* of a theory [38], that is, the set of the completed definitions of its predicates (intuitively, a predicate is true if and only if there exists a clause for it whose body is true) interpreted in a three-valued setting (where truth values are true, false, and unknown)
- X is the constraint theory [37], that is, the theory defined by the declarative semantics of CLP constraints.

The symbol $|=$ is interpreted in three-valued logics. In particular, if expectations are interpreted as abducible predicates, we can rely upon a three-valued model-theoretic semantics as intended meaning, as done, for instance, in a different context, by Fung and Kowalski [29] and by Denecker and De Schreye [25].

The following definition implements explicit negation for expectations [15]:

Definition 6: ¬-consistency A set **EXP** of expectations is ¬-consistent if and only if for each (ground) term p and integer t:

$$\neg(\{\mathbf{E}(p,t),\neg\mathbf{E}(p,t)\}\subseteq\mathbf{EXP}) \tag{6}$$

$$\neg(\{\mathbf{EN}(p,t),\neg\mathbf{EN}(p,t)\}\subseteq\mathbf{EXP}) \tag{7}$$

The following two definitions require consistency between positive and negative expectations—that is, they prevent an event from being expected to happen and expected not to happen in the same set of expectations.

Definition 7: E-consistency A set **EXP** of expectations is **E**-consistent if and only if for each (ground) term p and integer t:

$$\neg(\{\mathbf{E}(p,t),\mathbf{EN}(p,t)\}\subseteq\mathbf{EXP}) \tag{8}$$

The following definition establishes a link between happened events and expectations by requiring positive expectations to be matched by events, and negative expectations not to be matched by events.

Definition 8: Fulfillment Given a history **HAP**, a set **EXP** of expectations is **HAP**-fulfilled if and only if ∀p and ∀t

$$\mathbf{E}(p,t)\in\mathbf{EXP}\Rightarrow\mathbf{H}(p,t)\in\mathbf{HAP} \tag{9}$$

$$\mathbf{EN}(p,t)\in\mathbf{EXP}\Rightarrow\mathbf{H}(p,t)\notin\mathbf{HAP} \tag{10}$$

Otherwise, **EXP** is **HAP**-violated.

When all the given conditions (4–10) are met for at least one set of expectations **EXP**, the goal is said to be *achieved* and **HAP** is *compliant* to S with respect to G and **EXP**; this is written as $S_{\mathbf{HAP}}\vDash_{\mathbf{EXP}} G$. In particular:

Definition 9: Goal achievement Given an instance $S_{\mathbf{HAP}}$ of a contract specification S and a goal G, if there exists an **EXP** that is an abductive explanation of $S_{\mathbf{HAP}}$ for G, and is *¬-consistent*, **E**-*consistent,* and **HAP**-*fulfilled*, then G is said to be achieved w.r.t. **EXP** (and this is written $S_{\mathbf{HAP}}\vDash_{\mathbf{EXP}} G$). Given an instance $S_{\mathbf{HAP}}$ and a goal G, it is said that G is achieved if ∃**EXP** such that G is achieved w.r.t. **EXP**.

Operator	Abducible
Forb *A*	EN(*A*)
Obl *A*	E(*A*)
Perm *A*	¬EN(*A*)
Perm *NON A*	¬E(*A*)

Table 4. Deontic Notions as Expectations.

In the remainder of this article, when the text says that a history **HAP** is compliant to a contract specification *S*, it will mean that **HAP** is compliant to *S* with respect to the goal *true*. A statement that **HAP** *violates* a specification *S* will mean that **HAP** is not compliant to *S*. When **HAP** is apparent from the context, it will often not be mentioned.

Expectations and Deontic Operators

Mapping from deontic operators (obligation, permission, prohibition) to the expectations of the *S*CIFF framework was proposed in [9]. Such a mapping can be used to attribute a deontic meaning to *S*CIFF-based contract specifications.

The mapping is shown in Table 4. The first line of the table proposes a correspondence between the deontic notion of prohibition (which requires an action not to be performed) and our notion of negative expectation (which requires an event not to occur). In fact, the correspondence becomes more apparent if one considers Definition 8, which requires, for a set of expectations to be fulfilled, the absence from the history of events of any event matching a negative expectation. This definition closely resembles the reduction of the prohibition operator proposed by Meyer, where "it is forbidden to perform (an action) α in (a state) σ if one performs α in σ one gets into trouble" [39]. (In Meyer's paper, "trouble" means an "undesirable state of affairs," which is a good description of our state of violation).

Reasoning in a similar way, one notes a correspondence between the deontic notion of obligation (which requires an action to be performed) and the notion of positive expectation (which requires an event to occur), as shown in the second line in Table 4. Moreover, since a negative expectation **EN**(*A*) has to be read as *it is expected not A* (i.e., as a shorthand for **E**(*not A*)), its (explicit) negation, ¬**EN**(*A*), corresponds to permission of *A*. Finally, due to the logical relations among obligation, prohibition, and permission discussed by Sartor [43], the fourth line of Table 4 shows how to map permission of a negative action.

A formal support of this mapping is provided in [9], based on the correspondence between the Kripke semantics of deontic operators and the declarative semantics of the *S*CIFF framework.

The correspondence shown in Table 4 illustrates more intuitively the difference between ¬**E**(*tell*(*telco*,*c*,*phonebill*(39-051-209-3086,*Id*,*Amount*)),*T*) and

EN(*tell*(*telco*,*c*,*phonebill*(39-051-209-3086,*Id*,*Amount*)),*T*). The intuitive meaning of the former is that no *phonebill* is expected from *telco* (if this happens, it simply was not expected), which corresponds to the negation of the obligation for *telco*. The latter has a different, stronger meaning—it is expected that *telco* will not produce a *phonebill* (doing so would violate the expectation), corresponding to a prohibition for *telco*.

Sample Contract Specification

A sample specification of a contract in the *S*CIFF language will now be presented. The example is a simplified version of a real-life situation, describing the activation of a telephone line (carrier) by a customer. The discussion considers the clauses of the contract a user must sign as the building blocks of a contract that makes use of expressive combinations of **E**, **EN**, and **H** predicates, CLP constraints, and predicates defined in the $\mathcal{S}$. With *S*CIFF one can give a faithful representation of such a contract that is understandable, modular, and verifiable. Despite the efforts of the telephone company to make things as obscure as possible, we (as customers) will at any time be able to detect, via *S*CIFF, whether the telephone company (*telco* in the example) has the right to interrupt the service or to request a payment from us and whether we have the right to complain to *telco* and not to pay part of the bill. Similarly, *telco* will receive indications about when to send requests for payment and when (not) to activate or (not) to deactivate the carrier.

Description of the Contract

The procedures that regulate the concession of a carrier to a customer are contained in a contract agreed upon by the parties (*telco* and the customer). The contract states what to do when the customer requests a new carrier, the procedures for paying bills and for handling complaints, what obligations/penalties apply in case of late payments, and how to delegate authority to the relevant agent, when necessary, to determine whether the parties have complied with all the requirements set forth in the contract. A set of clauses is nucleated in the contract, and their specifications are given in the *S*CIFF framework. ICs are reported in Table 5, and the KB_S is reported in Table 6. A set of clauses about bill and complaint handling was chosen.

After sending a phone bill to a customer, *telco* cannot send requests for payment before a predefined period of time (*TWait*) has passed.

1. After *TWait*, either the customer has paid the bill or filed a complaint, or *telco* is allowed to send a request for payment.
2. After receiving a legitimate request for payment, either the customer pays the bill or *telco* is allowed to deactivate the carrier after a further *TWait*.
3. If, upon receiving a request for payment, the customer pays by *TWait*, *telco* is not allowed to deactivate the carrier.

[IC1]

$$\mathbf{H}(tell(T,C,phonebill(PhoneNo,BillId,BillAmnt),D),T1) \wedge$$
$$defaultwait(TWait) \rightarrow$$
$$\mathbf{EN}(tell(T,C,requestpayment(PhoneNo,BillId,AnyAmnt),D),T2)$$
$$:T2 > T1,\ \ T2 < T1 + TWait.$$

[IC2]

$$\mathbf{H}(tell(T,C,phonebill(PhoneNo,BillId,BillAmnt),D),T1) \wedge$$
$$defaultwait(TWait) \rightarrow$$
$$\mathbf{E}(tell(C,T,pay(PhoneNo,BillId,BillAmnt,PaymtRcpt),D),T2)$$
$$:T2 < T1 + TWait$$
$$\vee\ \mathbf{E}(tell(C,T,complain(PhoneNo,BillId,PartlAmnt),D),T3):$$
$$T3 < T1 + TWait$$
$$\vee\ \neg\mathbf{EN}(tell(T,C,requestpayment(PhoneNo,BillId,BillAmnt),D),T4):$$
$$T4 > T1 + TWait.$$

[IC3]

$$\mathbf{H}(tell(T,C,phonebill(PhoneNo,BillId,BillAmnt),D),T1) \wedge$$
$$\mathbf{H}(tell(T,C,requestpayment(PhoneNo,BillId,BillAmnt),D),T2) \wedge$$
$$\neg\mathbf{EN}(tell(T,C,requestpayment(PhoneNo,BillId,BillAmnt),D),T2) \wedge$$
$$defaultwait(TWait) \rightarrow$$
$$\neg\mathbf{EN}(tell(T,C,deactivate(PhoneNo,reason(BillId)),D),T3)$$
$$:T3 > T2 + TWait$$
$$\vee\ \mathbf{E}(tell(C,T,pay(PhoneNo,BillId,BillAmnt,PaymtRcpt),D),T4):$$
$$T4 < T2 + TWait.$$

[IC4]

$$\mathbf{H}(tell(T,C,requestpayment(PhoneNo,BillId,BillAmnt),D),T1) \wedge$$
$$\mathbf{H}(tell(C,T,pay(PhoneNo,BillId,BillAmnt,PaymtRcpt),D),T2) \wedge$$
$$defaultwait(TWait) \wedge T2 < T1 + TWait \rightarrow$$
$$\mathbf{EN}(tell(T,C,deactivate(PhoneNo,reason(BillId)),D),T3).$$

[IC5]

$$\mathbf{H}(tell(T,C,phonebill(PhoneNo,BillId,BillAmnt),D),T1) \wedge$$
$$\mathbf{H}(tell(C,T,complain(PhoneNo,BillId,PartlAmnt),D),T2) \wedge$$
$$defaultwait(TWait) \wedge T2 < T1 + TWait \wedge$$
$$isadmissiblecomplaint(BillId,PartlAmnt) \rightarrow$$
$$\neg\mathbf{E}(tell(C,T,pay(PhoneNo,BillId,PartlAmnt,PaymtRcpt),D),T3) \wedge$$
$$\mathbf{EN}(tell(T,C,requestpayment(PhoneNo,BillId,BillAmnt),D),T4):T3 > T1$$

Table 5. *IC*$_S$ in the Contract Between *telco* (*T*) and a Customer (*C*).

KB_S:
societygoal.
defaultwait(10).
isadmissiblecomplaint(*BillId,PartlAmnt*)← *listofbills*(*L*1),
    member((*BillId,TotalAmnt*),*L*1),
    PartlAmnt<*TotalAmnt*.
listofbills([(145886,205),(114477,407),(168945,126)]).

Table 6. *KB*$_S$ in the Contract Between *telco* and a Customer.

4. If by *TWait*, the customer files an admissible complaint about a received bill, the customer is no longer expected to pay for it, and *telco* is not allowed to request a payment.

*S*CIFF Specification of the Contract

Table 5 contains five ICs. Roughly speaking, the first three describe the expected behavior of *telco* regarding bill handling, whereas the last two are about the rights of the customer (*C*).

The ICs state the following:

- By [*IC*1], after sending a bill at time *T*1, *telco* may not send requests for payments before time *T*1 + *TWait*, where *TWait* is the amount of time defined by the *defaultwait* predicate in the KB_S.
- By [*IC*2], after *telco* sends a bill at time *T*1, one of the following expectations holds: Either *C* pays the bill in full by *T*1 + *TWait*, or *C* complains about (part of) the bill by *T*1 + *TWait*, or *telco* obtains the right to send a request for payment at some time *T*4 later than *T*1 + *TWait*. Note that any complaints *C* sends after the deadline (*T*1 + *TWait*) will have no impact on the state of affairs in these procedures, since they will not match with any expectation.
- By [*IC*3], if *telco* sent a bill, and later a request for payment at a time when doing so was not prohibited, and if the request for payment concerns the bill in full, then either *C* pays the bill or *telco* gains the right to deactivate the carrier (although *telco* is not obliged to do so).
- By [*IC*4], if *C* has paid the bill by the deadline, then *telco* cannot deactivate the carrier. Note that [*IC*4] takes effect independently of whether *telco* actually has the right to send a request for payments.
- By [*IC*5], after complaining about some part of the bill (*PartlAmnt*), *C* is no longer expected to pay the full *BillAmnt*.

The KB_S part of the *S*CIFF program, shown in Table 6, specifies deadlines, as in the previous example, and defines what an "admissible complaint" is. To this end, a predicate *isadmissiblecomplaint*/2 is defined that relies upon a database of bills ("list of bills"). In this simplified example, the database is mimicked by a predicate named *listofbills*/1. The predicate *member*/2 used by *isadmissiblecomplaint*/2 is predefined in most Prolog distributions. This example in particular uses the implementation that comes together with [45].

Contract Verification

Two types of verification supported by the *S*CIFF framework will now be described. The first is a verification that the parties involved in a contract are interacting in accordance with the contract terms. The second is a formal verification of whether a contract has certain properties.

Runtime Verification

The runtime verification of contracts specified in the *S*CIFF language is performed by means of an abductive proof procedure that is itself called *S*CIFF [8]. After a review of the *S*CIFF proof procedure, its behavior will be shown on sample interactions regulated by the contract described in the preceding section.

The SCIFF Proof Procedure

Since the *S*CIFF language and its declarative semantics are closely related to those of the IFF abductive framework, the *S*CIFF proof procedure is also inspired by the IFF proof procedure [29]. *S*CIFF is a substantial extension of IFF. In a nutshell, the main differences between the frameworks are as follows:

- *S*CIFF supports the dynamic happening of events—that is, the insertion of new facts in the knowledge base during the computation.
- *S*CIFF supports universally quantified variables in abducibles.
- *S*CIFF supports quantifier restrictions.
- *S*CIFF supports the concepts of fulfillment and violation (see Definition 8).

The *S*CIFF proof procedure is based on a rewriting system that transforms one node to another (or to others). In this way, starting from an initial node, it defines a proof tree. A node can either be the special node *false* or can be defined by the tuple

$$\mathrm{T} \equiv \langle \mathrm{R}, \mathrm{CS}, \mathrm{PSIC}, \mathbf{PEND}, \mathbf{HAP}, \mathbf{FULF}, \mathbf{VIOL} \rangle. \quad (11)$$

The set of expectations **EXP** is partitioned into the fulfilled (**FULF**), violated (**VIOL**), and pending (**PEND**) expectations. The other elements are:

- *R* is the resolvent: a conjunction whose conjuncts can be literals or disjunctions of conjunctions of literals.
- *CS* is the constraint store: It contains CLP constraints and quantifier restrictions.
- *PSIC* is a set of implications, called partially solved integrity constraints
- **HAP** is the history of happened events, represented by a set of events, plus a *closed*(**HAP**) Boolean attribute.

If one of the elements of the tuple is *false*, then the tuple is the special node *false*, without successors.

Initial Node and Success

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 IC_S, and an initial history $\mathbf{HAP}^i$, the first node is built in the following way:

$$T_0 \equiv \langle \{G\}, \emptyset, IC_S, \emptyset, \mathbf{HAP}^i, \emptyset, \emptyset \rangle$$

with *closed() = false*. The other nodes are obtained by applying the transitions described in the next subsection until no further transition can be applied.

> **Definition 10: Successful derivation** Given an instance $S_{\mathbf{HAP}^i}$ of a contract specification S, and a set $\mathbf{HAP}^f \supseteq \mathbf{HAP}^i$, there exists a successful derivation for a goal G if the proof tree with root node T_0 has at least one leaf node

$$\langle \emptyset, CS, PSIC, \mathbf{PEND}, \mathbf{HAP}^f, \mathbf{FULF}, \emptyset \rangle$$

where CS is consistent, and **PEND** contains only negations of expectations $\neg\mathbf{E}$ and $\neg\mathbf{EN}$. In such a case, we write:

$$S_{\mathbf{HAP}^i} \vdash_{\mathbf{EXP}}^{\mathbf{HAP}^f} G.$$

From a nonfailure leaf node $N \equiv \langle R_N, CS_N, PSIC_N, \mathbf{PEND}_N, \mathbf{HAP}_N, \mathbf{FULF}_N, \mathbf{VIOL}_N \rangle$, answers (called *expectation answers* and including, in particular, the actual set of expectations required by the declarative semantics according to Definition 9) can be extracted in a similar way to the IFF proof procedure. To compute an expectation answer, a substitution σ' is computed such that

- σ' replaces all variables in N that are not universally quantified by a ground term
- σ' satisfies all the constraints in the store CS_N

If the constraint solver is (theory) complete (i.e., for each set of constraints c, the solver always returns *true* or *false*, and never *unknown*), then there will always exist a substitution σ' for each nonfailure leaf node N [37]. If the solver is incomplete, σ' may not exist. The nonexistence of σ' is discovered during the answer-extraction phase. In such a case, the node N will be marked as a failure node, and another nonfailure node can be selected (if there is one).

> **Definition 11: Expectation answer** Let $\sigma = \sigma'|_{vars(G)}$ be the restriction of σ' to the variables occurring in the initial goal G. Let $\Delta_N = (\mathbf{FULF}_N \cup \mathbf{PEND}_N)\sigma'$. The pair $\langle \Delta_N, \sigma \rangle$ is the expectation answer obtained from the node N.

Transitions

The transitions are based on those of the IFF proof procedure [29], enlarged with those of CLP and with specific transitions accommodating the concepts of dynamically growing history and consistency of the set of expectations. The inference rules derived from IFF are:

Unfolding substitutes an atom p with its definitions in KB_S:

$$p^1 = l_{i^1} \wedge \ldots \wedge l_{i^{m_1}}$$
$$\ldots$$
$$p^n = l_{i^n} \wedge \ldots \wedge l_{i^{m_n}}$$

If the literal p occurs in the resolvent R, then n new nodes are generated. If p occurs in the body of an IC $\equiv p \wedge B \rightarrow H$, then one node with n ICs is generated.

$$p^1 = l_{i^1} \wedge \ldots \wedge l_{i^{m_1}} \wedge B \rightarrow H$$
$$\ldots$$
$$p^n = l_{i^n} \wedge \ldots \wedge l_{i^{m_n}} \wedge B \rightarrow H$$

Propagation propagates ICs: if a literal $p \in \Delta$ and

$$IC_i \equiv p_1 \wedge B \rightarrow H$$

generates a new node with the additional IC

$$(p = p_1) \wedge B \rightarrow H$$

Splitting distributes conjunctions and disjunctions, making the final formula in a sum-of-products form.

Case analysis: If $IC_i \equiv (X = t) \wedge B \rightarrow H$, case analysis 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 reuses previous hypotheses: If $p_1, p_2 \in \Delta$, factoring generates two nodes, one with $p_1 = p_2$ and the other with $p_1 \neq p_2$

Rewrite rules for equality: Use the inferences in the Clark equality theory to perform unification (i.e., $p(t_1, \ldots, t_n) = p(s_1, \ldots, s_n)$ is replaced with $\forall_{i=1}^{n} t_i = s_i$)

Logical simplifications: Try to simplify a formula through equivalences like $A \wedge false \leftrightarrow false$, $[A \leftarrow true] \leftrightarrow A$, ….

Additionally, SCIFF-specific inference rules are:

Happening: A new happened event $\mathbf{H}(t)$ is added to the set **HAP**.

Closure: Assumes that no more events can happen (sets the *closure* flag to *true*). Useful for reasoning under the Closed World Assumption.

Nonhappening: If $ICs_i \equiv \neg\mathbf{H}(X) \wedge B \rightarrow H$, and *closure*(**HAP**) = *true*, performs constructive negation to derive that $\forall X$ such that $\forall_{\mathbf{H}(t)\in\mathbf{HAP}} X \neq t, B \rightarrow H$ (i.e., for each possible instance of $\mathbf{H}(X)$ that does not unify with any element of **HAP**, $B \rightarrow H$ holds).

Consistency: If $\{\mathbf{E}(X),\mathbf{EN}(Y)\}\subseteq\Delta$ (or $\{\mathbf{E}(X),\neg\mathbf{E}(Y)\}\subseteq\Delta$ or $\{\mathbf{EN}(X),\neg\mathbf{EN}(Y)\}\subseteq\Delta$), imposes $X\neq Y$.

Fulfillment: If $\mathbf{H}(X)\in\mathbf{HAP}$ and $\mathbf{E}(Y)\in\Delta$ generates two nodes, one is with $X = Y$ and the expectation $\mathbf{E}(Y)$ fulfilled, and the other is with $X \neq Y$.

Violation: If $\mathbf{H}(X)\in\mathbf{HAP}$ and $\mathbf{EN}(Y)\in\Delta$ imposes $X\neq Y$.

CLP: Constraint logic programming reasoning.

SCIFF Properties

The most significant formal properties of the *S*CIFF proof procedure are stated and proven in [8]. They are briefly restated here.

Termination is proven, as for SLD resolution (Linear resolution with a Selection function for Definite clauses [14]), for *acyclic* knowledge bases and *bounded* goals and implications. The notion of acyclicity of an abductive logic program is an extension of the corresponding notion given for SLD resolution. Intuitively, for SLD resolution a level mapping must be defined such that the head of each clause has a higher level than the body. For the IFF, since it contains integrity constraints that are propagated forward, the level mapping should also map atoms in the body of an IC to higher levels than the atoms in the head. This should also hold for possible unfoldings of literals in the body of an IC [48]. Similar considerations hold for *S*CIFF. The level mapping was extended for considering also CLP constraints. For definitions of boundedness and acyclicity for the contract specification, the reader can refer to [48].

Theorem 1 (Termination of *S*CIFF): *Let G be a query to a contract* $S = \langle KB_s, IC_s\rangle$*, where* KB_S*,* IC_S*, and G are acyclic w.r.t. some level mapping, and G and all implications in* IC_S *are bounded w.r.t. the level mapping. Then, every SCIFF derivation for G for each instance of G is finite, assuming that happening is not applied.*

Moreover, under the following conditions:

- *the number of happened events is finite,*
- *happening is applied only when no other transitions can be applied, and*
- *nonhappening has higher priority than other transitions,*

SCIFF also terminates with dynamically incoming events.

The *S*CIFF proof procedure uses a constraint solver, so its *soundness* depends on the solver. Soundness was proved for a limited solver containing only the rules for equality and disequality of terms.

Theorem 2 (Soundness of *S*CIFF): *Given a contract instance* $S_{\mathbf{HAP}^f}$, *if*

$$S_{\mathbf{HAP}^i} \vdash^{\mathbf{HAP}^f}_{\mathbf{EXP}} G.$$

for some $\mathbf{HAP}^i \subseteq \mathbf{HAP}^f$, *with expectation answer* ($\mathbf{EXP}$,σ), *then*

$$S_{\mathbf{HAP}^f} \models_{\mathbf{EXP}\sigma} G\sigma.$$

Completeness states that if goal G is achieved under the expectation set **EXP**, then a successful derivation can be obtained for G, possibly computing a set **EXP**′ of the expectations whose grounding (according to the expectation answer) is a subset of **EXP**.

Theorem 3: *Given a contract instance* $S_{\mathbf{HAP}}$, *a (ground) goal G, for any ground set* **EXP** *such that* $S_{\mathbf{HAP}} \models_{\mathbf{EXP}} G$, *then* $\exists \mathbf{EXP}'$ *such that* $S_{\Delta} \vdash^{\mathbf{HAP}}_{\mathbf{EXP}'} G$ *with an expectation answer* ($\mathbf{EXP}'$,σ) *such that* $\mathbf{EXP}'\sigma \subseteq \mathbf{EXP}$.

Runtime Verification Examples

The following case will be considered: *telco* sends the bill, and *C* does not pay. After *TWait* time units, *telco* sends *C* a request for payment.

$$\begin{gathered}\mathbf{H}(tell(telco, c, phonebill(39\text{-}051\text{-}209\text{-}3086,145886,205),d_1), 19).\\ \mathbf{H}(tell(telco, c, requestpayment(39\text{-}051\text{-}209\text{-}3086,145886,205),d_1), 33).\\ \mathbf{H}(tell(c, telco, pay(39\text{-}051\text{-}209\text{-}3086,145886,205,1674521),d_1), 37).\end{gathered} \quad (12)$$

This sequence of events (12) generates a set of fulfilled expectations. After the first message at time 19 (the notification of the *phonebill*), [*IC*2] generates three alternative and equally plausible sets of expectations: Either *C* is expected to pay before time 29, or *C* is expected to complain before time 29, or *telco* has the right ($\neg$**EN**) to issue a request for payment after time 29. In all cases, because of [*IC*1], *telco* does not have the right to send a request for payment before time 29. The first two alternatives become invalid at time 29 due to the expired deadline. The message *requestpayment* at time 33 is acceptable according to the contract and gives *telco* explicit right to deactivate the carrier any time later than 43. In particular, by [*IC*3] an alternative is generated: In one case *telco* has the right to deactivate the carrier after time 43, in the other case *C* is expected to pay. Because of [*IC*4], the last message, in which *C* notifies the payment to *telco*, has as a side effect that *telco* loses its right to deactivate the carrier at any time in connection to the bill No. 145886.

As the second example shows (13), a violation can be generated if *telco* deactivates the carrier. In that case, *S*CIFF detects a violation because the fourth message violates the contract, and in particular [*IC*4], by which *telco* is expected not to deactivate the carrier if *C* pays within 10 time units after receipt of *telco*'s request for payment.

$$\begin{array}{c}\mathbf{H}(\mathit{tell}(\mathit{telco}, c, \mathit{phonebill}(39\text{-}051\text{-}209\text{-}3086,145886,205),d_1), 19).\\ \mathbf{H}(\mathit{tell}(\mathit{telco}, c, \mathit{requestpayment}(39\text{-}051\text{-}209\text{-}3086,145886,205),d_1), 33).\\ \mathbf{H}(\mathit{tell}(c, \mathit{telco}, \mathit{pay}(39\text{-}051\text{-}209\text{-}3086,145886,205,1674521),d_1), 37).\\ \mathbf{H}(\mathit{tell}(\mathit{telco}, c, \mathit{deactivate}(39\text{-}051\text{-}209\text{-}3086,\mathit{reason}(145886)),d_1), 38).\end{array} \qquad (13)$$

A third example will now be considered. Like the other examples, it starts with *telco* sending *C* a bill. *C* complains at time 33, which unfortunately is past the deadline of 10 time units after the bill. The complaint, although not specifically disallowed by the contract, does not change the state of expectations in the system, since no IC fires. In particular, [*IC*5] says that if *C* complains before the deadline, *C* is no longer expected to pay the amount complained about, and *telco* loses the right to send requests for payment concerning either the amount *C* complained about or the full amount of the bill. But [*IC*5] (as well as the other ICs) does not say what happens in case of a late complaint, so *telco* exercises its right to send *C* a request for payment. The only option for *C* is either to pay or to have the carrier deactivated. *C* pays, and *telco* no longer has a right to deactivate the line, which incidentally makes the second option (have the carrier deactivated) inconsistent, besides fulfilling all the expectations of the first branch (14).

$$\begin{array}{c}\mathbf{H}(\mathit{tell}(\mathit{telco}, c, \mathit{phonebill}(39\text{-}051\text{-}209\text{-}3086,145886,205),d_1), 19).\\ \mathbf{H}(\mathit{tell}(c, \mathit{telco}, \mathit{complain}(39\text{-}051\text{-}209\text{-}3086,145886,150),d_1), 33).\\ \mathbf{H}(\mathit{tell}(\mathit{telco}, c, \mathit{requestpayment}(39\text{-}051\text{-}209\text{-}3086,145886,205),d_1), 34).\\ \mathbf{H}(\mathit{tell}(c, \mathit{telco}, \mathit{pay}(39\text{-}051\text{-}209\text{-}3086,145886,205,1674521),d_1), 37).\end{array} \qquad (14)$$

In the last example, *telco* as usual sends *C* a bill. However, this time *C* sends a complaint before the deadline. *C* complains about the amount of €150 out of €205. The complaint is judged admissible (as shown in the example with the *isadmissiblecomplaint* predicate). In consequence, if *telco* sends *C* a request for payment (14), it violates the contract. Due to [*IC*5], *telco* can no longer issue a request for payment. Unfortunately, *telco* does so at time 34, and consequently *S*CIFF detects the violation of [*IC*5].

$$\begin{array}{c}\mathbf{H}(\mathit{tell}(\mathit{telco}, c, \mathit{phonebill}(39\text{-}051\text{-}209\text{-}3086,145886,205),d_1), 19).\\ \mathbf{H}(\mathit{tell}(c, \mathit{telco}, \mathit{complain}(39\text{-}051\text{-}209\text{-}3086,145886,150),d_1), 24).\\ \mathbf{H}(\mathit{tell}(\mathit{telco}, c, \mathit{requestpayment}(39\text{-}051\text{-}209\text{-}3086,145886,205),d_1), 34).\end{array} \qquad (15)$$

Design-Time Property Verification

An extension of the *S*CIFF proof procedure, called g-*S*CIFF, has been developed to verify contract properties [7]. g-*S*CIFF is briefly reviewed below, followed by a demonstration of its use to refute a formal property that is not possible with the contract described in the second section.

The g-SCIFF Proof Procedure

Besides verifying whether a history is in compliance with a contract, g-*S*CIFF is able, given a contract, to generate a compliant history. This is achieved by (1) considering **H** events as abducibles and allowing variables in them, and (2) adding a new transition to those of *S*CIFF, which, when a positive expectation $\mathbf{E}(p,t)$ is added to the set of expectations, generates an event $\mathbf{H}(p,t)$ that fulfills it. g-*S*CIFF has been proved sound [6], which means that the histories it generates (in case of success) are guaranteed to be compliant to the interaction contracts while entailing the goal. Note that the histories generated by g-*S*CIFF are not, in general, a collection only of ground events, like the **HAP** sets given as an input to *S*CIFF. They can, in fact, contain variables, which means that they represent *classes* of event histories.

In order to use g-*S*CIFF for verification, the property to be verified is expressed as a conjunction of literals. Thus, to verify whether a formula f is a property of a contract P, the contract is expressed in our language and $\neg f$ as a g-*S*CIFF goal. Then, either

- g-*S*CIFF returns success, generating a history **HAP**. Thanks to the soundness of g-*S*CIFF, **HAP** entails $\neg f$ while being compliant to P: f is not a property of P, **HAP** (and its groundings) being a counterexample; or
- g-*S*CIFF returns failure, suggesting that f is a property of P.[3]

Design-Time Property Verification Example

This section shows the refutation, by means of g-*S*CIFF, of a simple property of the contract described earlier. For simplicity, details related to the management of restrictions and defined predicates will not be shown.

The property is: *"if a phone bill is sent, then the customer will pay for it."* Using our formalism for events, the property can be written as follows:

$$\begin{aligned}&\mathbf{H}(tell(T,C,phonebill(N,I,A),D),\,T_b)\\ &\rightarrow \mathbf{H}(tell(C,T,pay(N,I,A,R),D),\,T_p)\end{aligned} \tag{16}$$

The negation of the property is:

$$\begin{aligned}&\mathbf{H}(\,tell(\,T,C,phonebill(\mathrm{N,I,A}),\mathrm{D}),\,T_b)\\ &\wedge\neg\mathbf{H}(\,tell(\,C,T,pay(N,I,A,R),D),\,T_p)\end{aligned} \tag{17}$$

Therefore, a history that entails Equation (17) is a counterexample of the property to be verified. To try and find such a history, one writes the following g-*S*CIFF goal:

$$\begin{aligned}G=\;&\mathbf{E}(\,tell(\,T,C,phonebill(N,I,A),D),\,T_b)\\ &\wedge\mathbf{EN}(\,tell(\,C,T,pay(N,I,A,R),D),\,T_p)\end{aligned} \tag{18}$$

In general, a history that achieves a goal (see Definition 9) will necessarily include events that are expected to happen, and not include events that are expected not to happen, in the goal. Thus, in this case, a history that achieves G will entail Equation (17).

To begin, g-*S*CIFF is run with G as a goal. g-*S*CIFF imposes the first expectation of the goal,

$$\mathbf{E}(tell(\ T,\ C,\ phonebill(N,I,A),D),\ T_b),$$

which generates the following event:

$$\mathbf{H}(tell(\ T,\ C,\ phonebill(N,I,A),D),\ T_b)$$

which in turn, due to the first IC in Table 5, generates the expectation

$$\mathbf{EN}(tell(T,C,requestpayment(N,I,A),D),T2)$$

and, due to the second, one of

$$\mathbf{E}(tell(C,T,pay(N,I,A,PR),D),T2):\ T2<T_b+10$$

$$\mathbf{E}(tell(C,T,complain(N,I,PA),D),T3)\ :\ T3<T_b+10$$

$$\neg\mathbf{EN}(tell(T,C,requestpayment(N,I,A),D),T4)\ :\ T4<T_b+10.$$

The **E**-consistency requirement (Definition 7) rules out the first alternative, because of the negative (**EN**) expectation imposed by the goal (see Equation (18)); so the second branch is explored, and the event

$$\mathbf{H}(tell(C,T,complain(N,I,PA),D),T3)$$

is generated.

Due to the fifth IC in Table 5, the following expectations are generated:

$$\neg\mathbf{E}(tell(C,T,pay(N,I,PA,PR),D),T3)$$

and

$$\mathbf{EN}(tell(T,C,requestpayment(N,I,BillAmnt),D),T4)$$

and finally g-*S*CIFF terminates and returns success, with the history

$$\mathbf{HAP}=\{\mathbf{H}(tell(T,C,phonebill(N,I,A),D),\ T_b),$$

$$\mathbf{H}(tell(C,T,complain(N,I,PA),D),T3)\}$$

Thanks to the soundness of g-*S*CIFF, any grounding of **HAP** is a counterexample of the property that was to be proved, and it is also compliant to the contract. Thus, it shows that the contract does not enjoy the property. In particular, it shows that a customer can avoid being expected to pay by filing a complaint.

Rule Mark-Up

An architecture and a formal framework that enable Web services to reason on publicly available *S*CIFF-based specifications is proposed in [11]. In particular, it is possible for a Web service to verify whether it can interact with another and achieve a goal. An interested party could fruitfully perform such a step before agreeing on a contract with another party. Obviously, this requires a formalism that makes it practical to exchange *S*CIFF-based specifications.

RuleML is a suitable mark-up language for exchanging rules on the Web [1]. RuleML 0.9 contains mark-ups for expressing important concepts of the *S*CIFF proof procedure. In particular, *S*CIFF is a rule engine able to distinguish and use both backward and forward rules. Backward rules are used to plan, reason on events, and perform proactive reasoning. Forward rules are used for reactive reasoning and to quickly perform actions in response to occurred events. Both are seamlessly integrated in *S*CIFF. RuleML 0.9 contains a *direction* attribute that can be attached to rules. Because it is based on abduction, *S*CIFF can deal both with explicit negation and with negation by default that have appropriate tagging in RuleML. The present work only uses standard RuleML syntax. In future work, it might be interesting to distinguish between defined and abducible predicates, or between expectations and events.

*S*CIFF was implemented in SICStus Prolog. SICStus contains an implementation of the PiLLoW library [23], which makes it easy to perform http requests, as well as to implement services on the Web. SICStus also contains an XML parser that made it possible to easily implement the RuleML parser. The RuleML parser is freely available on the *S*CIFF Web site [44].

Related Work

The reduction of deontic concepts such as obligations and prohibitions has been the subject of extensive research. Among the most influential approaches are Anderson's, by which *A* is obligatory if its absence produces a state of violation [12], and Meyer's, by which an action *A* is prohibited if performed it produces a state of violation [39]. These two reductions strongly resemble our definition of fulfillment (Definition 8), which requires positive (resp. negative) expectations to have (resp. not to have) a corresponding event.

Several authors have studied "sub-ideal" situations—namely, how to manage situations in which some of the norms are not respected.

For instance, van der Torre and Tan show the relation between diagnostic reasoning and deontic logic, importing the *principle of parsimony* from diagnostic reasoning into their deontic system, in the form of a requirement to minimize the number of violations [47]. In particular, given the specification of a normative system (as a set of formulae that tell when a norm is violated) and a state of affairs, they define a minimal (with respect to inclusion) set of norms such that the violation of those norms is consistent with the specification and the state of affairs. The SOCS social framework currently distinguishes only between empty and nonempty sets of violations, and does not define minimal sets. However, it would be possible to do so by taking the minimal, with respect to inclusions, among the sets of expectations that are consistent with a social specification and a history, but possibly not fulfilled by the history. This will probably be our approach when we tackle the management of violations (by means of sanctions and recovery procedures) in future work.

Prakken and Sergot propose a solution to the problem and paradoxes stemming from earlier logical representations of *contrary-to-duty* obligations (CTDs), meaning obligations that become active when other obligations are violated [42]. They do so by introducing a new operator $O_B(A)$, meaning that A is obligatory given the subideal context B. The semantics of this operator is of the Kripke type but differs from the standard modal logic because of the accessibility relation: In that work, the accessible worlds are the best alternatives, given the truth of B. In the "mainstream" of our research, we do not support CTDs. However, a modified version of our framework provides a simplified language and does support alternative obligations at different levels of priority [10]. A further step could be to integrate priority levels in the main SOCS social framework.

Deontic operators have not only been used to model normative concepts related to agent interaction in institutional contexts, but they are also part of agent programming languages. Notably, in IMPACT, agent programs make use of permission, obligation, and prohibition operators, with a semantics intuitively similar to that used in deontic logics, but with the purpose of determining possible courses of action that an agent may take in a given situation [16, 27]. In this respect, the IMPACT and *S*CIFF models have similarities even if their purposes and expressivity are different. The main difference is that agent programs in IMPACT express and determine the behavior of a single agent, whereas the goal of the *S*CIFF framework is to express rules of interaction and norms that cannot really determine and constrain the behavior of the single agents participating in a society, since agents are autonomous.

Governatori uses defeasible logics with deontic operators of obligation and permission to define contracts [30]. He proposes the introduction in RuleML of new tags for identifying obligations and permission, and creates graded violations and corresponding ideal and subideal states. In *S*CIFF, explicit permission is generally not used, because everything is allowed by default. Typically, when an action is expected not to happen, **EN** is stated explicitly. There are connections between **EN** and $\neg P$ of deontic logics (studied in [9]), so it might be possible to use the same tags proposed by Governatori (e.g., `<neg><Permission>` to represent **EN**).

Governatori also introduces an operator $\otimes$ to address recovery from violation [30]. For example, $A \Rightarrow OB \otimes OC$ means that A implies that B is obligatory; but if OB is violated, C becomes obligatory. In *S*CIFF, recovery expectations can be inserted as an alternative in each of the rules: $A \Rightarrow OB \otimes OC$ could be written in *S*CIFF as $\mathbf{H}(A) \rightarrow \mathbf{E}(B) \vee \mathbf{E}(C)$. Interestingly, Governatori also proposes an inference rule that derives recovery rules from the other rules of the contract (from $A \rightarrow OB$ and $\neg B \rightarrow OC$ derives $A \rightarrow OB \otimes OC$) [30]. This is an interesting line of research that in future work will also be applied to *S*CIFF.

Governatori and Milosevic discuss the need for contract verification and contract monitoring to check how parties fulfill their policies [31, 32]. Both these issues are addressed by the adoption of a formal specification language for contracts. The system they propose, and their Business Contract Language (BCL) in particular, is based on the formalism for the representation of CTDs. The formal representation they adopt for contracts is based upon a propositional logic language, with the deontic operators of obligation, permission, and contrary-to-duty. Each condition or policy of a contract is represented by a rule where the *antecedent* is a literal or a modal literal (built with the deontic operators of permission and obligation, possibly negated), and the *conclusion* of the rule is a CTD expression. Contract analysis then reduces the contract to a normal form that makes explicit all the contract conditions that can be generated/derived from the given specification. The procedure for generating normal forms is expressed in terms of inference rules that merge two rules in a new clause through the violations of conditions (e.g., when the former rule mentions an obligation $O\,A$ in its conclusion and the latter rule has the negation $\neg A$ in its antecedent, then their conclusions are composed in order to build a CTD formula for A). Normal forms are then a sort of *partial evaluation* of specification rules, in the logic of violation, aiming at producing rules with CTD formulas in their conclusions that summarize all the possible violations and recovery actions implicitly specified by the original (logic) representation of a contract. On generated normal forms, they can therefore detect conflicts arising from, for example, obligation of A and $\neg A$, or occurrence of A and $\neg A$ in conclusions without any CTD for A neither $\neg A$.

Governatori et al. also consider the problem of checking the compliance of a business process expressed in the Business Process Modeling Notation to a business contract expressed in the aforementioned language [22, 33]. They define ideal, subideal, and nonideal situations to reflect decreasing degrees of compliance, and use these terms to characterize a business process with respect to a contract. Business processes are outside the focus of this paper, but we have proposed an approach to the definition of compliance of agents to interaction protocols and Web services to choreographies that is similar to that of Governatori et al. and also provides an automatic verification procedure [2, 4].

Although our proposed language does not support CTDs, it is first-order and supports the deontic operators of permission and obligation (and their negation, as discussed in [9]). The proposed approach exploits *S*CIFF at runtime for contract monitoring (e.g., conflicts and contradictions are detected at runtime by the notions of *E*-consistency and $\neg$-consistency). More general contract properties (beside the absence of conflicts) can be also statically verified by g-*S*CIFF. In particular, g-*S*CIFF generates every possible *compliant* history that satisfies a given goal and a contract specified in the *S*CIFF language. Each generated history can be considered as a set of obligations in the approach of Governatori and Milosevic [31, 32], since g-*S*CIFF turns obligations into events.

The problem of representing violations and CTD formulas using first-order logic has also been studied by Herrestad, who discusses several solutions and their limits [36]. Our concept of positive and negative expectations, together with an explicit time representation, supports quite well the representation of deontic operators. However, our representation of CTD formulas does not fully support the idea of "suboptimal worlds" as discussed by Herrestad [36], since recovery actions are actually represented as plain alternatives to duties. Future research will also address this issue. A solution might consist in extending our declarative semantics with the concept of preference.

An interesting extension would be to equip the *S*CIFF language with CTD expressions to occur in the head of ICs. In particular, when dealing with CTD expressions, one needs to select preferred models, such that the expectations in the recovery branch are imposed only if the normal branch is not fulfilled. One way to state such preferential reasoning is through qualitative choice logic [19]. This issue will be investigated in future work.

Boella and van der Torre discuss how a normative system can be seen as a normative agent, equipped with mental attitudes about which other agents can reason, choosing either to fulfill their obligations or to face the possible sanctions [18]. Conceptually, the social infrastructure in the SOCS model could be viewed as an agent whose knowledge base is the society specification, whose mental attitude is a set of expectations and whose reasoning process is the *S*CIFF proof procedure.

The ability to reason with time and deadlines is a distinguishing feature of the *S*CIFF language. Temporal aspects in normative positions have been the focus of previous work, such as [34] and [20]. In [34], Governatori et al. show how the analysis of normative conditionality and normative positions has to include temporal aspects in order to capture a number of important concepts. They propose a framework with temporalized normative positions in which literals may be labeled by time instants like *S*CIFF events, and assuming linear and discrete time. In this way it is possible to model deadlines and timeouts. Following a different approach from ours, Governatori et al. use the event calculus to deal with time. Broerson et al. investigate the deontic logic of deadlines by introducing an operator $O(\rho \leq \delta)$, which means, intuitively, that the action ρ ought to be brought about before (or at the same time) another event δ happens [20]. They model time by means of Computation Tree Logic temporal logic. We can express a similar concept by means of an integrity constraint $\mathbf{H}(\delta, T_\delta) \rightarrow \mathbf{E}(\rho, T_\rho) \wedge T\rho \leq T_\delta$, which says that if δ has happened, than ρ is expected to have happened before (or at the same time).

The *S*CIFF framework can capture, in a computational setting, the concept of (conditional) obligation with deadline presented by Dignum et al., with an explicit mapping of time [26]. Dignum et al. write: $Oa(r < d - p)$ to state that if the precondition p becomes valid, the obligation becomes active. The obligation expresses the fact that a is expected to bring about the truth of r before a certain condition d holds.

For instance, if

$$P = \mathbf{H}(tell(S,a,request(G),D,T))$$

$$R = \mathbf{H}(tell(a,S,answer(G),D,T')),T' > T$$

$$D = T' > T + 2$$

then $Oa(r < d - p)$ can be mapped into a IC:

$$\mathbf{H}(tell(S,a,request(G),D),T) \rightarrow \mathbf{E}(tell(a,S,answer(G),D),T'),\ T'{>}T,\ T'{\leq}T{+}2.$$

Many of the works that have used the event calculus (EC) for the purpose of reasoning over the effects of events are very close to this paper. In particular, it is especially related to the work by Farrell et al. [28]. They are principally concerned with the representation of contracts and particularly their normative state, in terms of obligation, power, and permission. The effects of contract events on the normative state of a contract are specified using an XML formalization of the event calculus. This representation may be used to track the state of the agreement, according to a narrative of contract events similar to our concept of history.

The present work is similar to the work of Farrell et al. in that *S*CIFF can be seen as a generic language for expressing backward and forward rules and reasoning about (conformance) properties of a specific where the representation of contracts is just one application.

The work differs from Farrell et al. in that it shows that being able to describe contracts as logical theories is extremely useful not only for tracking, but also for proving general or specific properties of the contracts by using the same formalism. Artikis, Sergot, and Pitt adopt a similar approach by using a formalization in terms of transition systems and model checking techniques [17].

Conclusions

This paper proposes the use of the *S*CIFF framework, originally developed for agent interaction protocols, to specify and verify business contracts. The proposal was supported intuitively by showing a deontic reading of *S*CIFF specifications. The specification of sample business contract clauses was given in the *S*CIFF language.

The paper demonstrates how verification is performed in the *S*CIFF framework, in particular, runtime verification by means of the *S*CIFF proof procedure, and design-time property verification with the g-*S*CIFF proof procedure. It also shows how *S*CIFF rules can be encoded in RuleML in order to enable potential contract parties to reason on contracts in advance.

Future work will be devoted to experimentation with the *S*CIFF framework on real-world contracts, testing both the expressiveness of the *S*CIFF language and the effectiveness of the proof procedures used for verification. We are also working on a formal completeness result (possibly for restricted cases) for g-*S*CIFF. On the language side, it would be interesting to explore recovery from violations, possibly with a mechanism similar to contrary-to-duty obligations, and to extend the *S*CIFF language to accommodate other legal reasoning concepts, such as power and immunity (e.g., such as Hohfeldian power and immunity).

NOTES

1. For a complete treatment of quantification in the *S*CIFF language, the interested reader is referred to [8].

2. In the *S*CIFF language, restrictions can be considered as CLP constraints that can also be applied to universally quantified variables with the semantics defined by Bürckert [21].

3. If we had a completeness result for g-*S*CIFF, this would indeed be a proof and not only a suggestion.

REFERENCES

1. Adi, A.; Stoutenburg, S.; and Tabet, S. (eds.), *Rules and Rule Markup Languages for the Semantic Web: First International Conference.* LNCS, vol. 3791. Berlin: Springer-Verlag, 2005.

2. Alberti, M.; Chesani, F.; Gavanelli, M.; Lamma, E.; and Mello, P. A verifiable logic-based agent architecture. In F. Esposito, Z.W. Rás, D. Malerba, and G. Semeraro (eds.), *Foundations of Intelligent Systems: 16th International Symposium.* LNAI, vol. 4203. Berlin: Springer-Verlag, 2006, pp. 188–197.

3. Alberti, M.; Gavanelli, M.; Lamma, E.; Mello, P.; and Torroni, P. The *S*CIFF abductive proof procedure. In S. Bandini and S. Manzoni (eds.), *Proceedings of the 9th Congress of the Italian Association for Artificial Intelligence*. LNAI, vol. 3673. Berlin: Springer-Verlag, 2005, pp. 135–147.

4. Alberti, M.; Chesani, F.; Gavanelli, M.; Lamma, E.; Mello, P.; and Montali, M. An abductive framework for a-priori verification of Web services. In A. Bossi and M. Maher (eds.), *Proceedings of the Eighth Symposium on Principles and Practice of Declarative Programming*. New York: ACM Press, 2006, pp. 39–50.

5. Alberti, M.; Chesani, F.; Gavanelli, M.; Lamma, E.; Mello, P.; and Torroni, P. Compliance verification of agent interaction. A logic-based tool. *Applied Artificial Intelligence, 20,* 2–4 (February/April 2006), 133–157.

6. Alberti, M.; Chesani, F.; Gavanelli, M.; Lamma, E.; Mello, P.; and Torroni, P. On the automatic verification of interaction protocols using *g-S*CIFF. Technical Report DEIS-LIA-04-004. LIA Series, no. 72. University of Bologna, 2005.

7. Alberti, M.; Chesani, F.; Gavanelli, M.; Lamma, E.; Mello, P.; and Torroni, P. Security protocols verification in abductive logic programming. A case study. In O. Dikenelli, M-P Gleizes, and A. Ricci (eds.), *ESAW 2005 Post-Proceedings.* LNAI, vol. 3963. Berlin: Springer-Verlag, 2006, pp. 106–124.

8. Alberti, M.; Chesani, F.; Gavanelli, M.; Lamma, E.; Mello, P.; and Torroni, P. Verifiable agent interaction in abductive logic programming: The *S*CIFF framework. *ACM Transactions on Computational Logic*, *9,* 4 (2008), forthcoming.

9. Alberti, M.; Gavanelli, M.; Lamma, E.; Mello, P.; Sartor, G.; and Torroni, P. Mapping deontic operators to abductive expectations. *Computational and Mathematical Organization Theory, 12*, 2–3 (October 2006), 205–225.

10. Alberti, M.; Chesani, F.; Daolio, D.; Gavanelli, M.; Lamma, E.; Mello, P.; and Torroni, P. Specification and verification of agent interaction protocols

in a logic-based system. *Scalable Computing: Practice and Experience, 8*, 1 (2007), 1–13.

11. Alberti, M.; Chesani, F.; Gavanelli, M.; Lamma, E.; Mello, P.; Montali, M.; and Torroni, P. Policy-based reasoning for smart Web service interaction. In A. Polleres, S. Decker, G. Gupta, and J. de Bruijn (eds.), *Proceedings of the 1st International Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services.* CEUR Workshop Proceedings, vol. 196. Aachen: RWHT, 2006, pp. 87–102.

12. Anderson, A. A reduction of deontic logic to alethic modal logic. *Mind*, 67 (1958), 100–103.

13. Antoniou, G.; Maher, M.J.; and Billington, D. Defeasible logic versus logic programming without negation as failure. *Journal of Logic Programming, 42*, 1 (2000), 47–57.

14. Apt, K.R., and Bezem, M. Acyclic programs. *New Generation Computing, 9*, 3/4 (1991), 335–364.

15. Apt, K.R., and Bol, R.N. Logic programming and negation: A survey. *Journal of Logic Programming, 19/20* (1994), 9–71.

16. Arisha, K.A.; Ozcan, F.; Ross, R.; Subrahmanian, V.S.; Eiter, T.; and Kraus, S. IMPACT: A platform for collaborating agents. *IEEE Intelligent Systems, 14*, 2 (March/April 1999), 64–72.

17. Artikis, A.; Sergot, M.J.; and Pitt, J. An executable specification of an argumentation protocol. In G. Sartor (ed.), *Proceedings of the 9th International Conference on Artificial Intelligence and Law*. New York: ACM, 2003, pp. 1–11.

18. Boella, G., and van der Torre, L.W.N. Attributing mental attitudes to normative systems. In J.S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo (eds.), *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*. New York**:** ACM Press, 2003, pp. 942–943.

19. Brewka, G.; Benferhat, S.; and Le Berre, D. Qualitative choice logic. *Artificial Intelligence, 157*, 1–2 (2004), 203–237.

20. Broersen, J.; Dignum, F.; Dignum, V.; and Meyer, J-J. Designing a deontic logic of deadlines. In A. Lomuscio and D. Nute (eds.), *DEON*. LNCS, vol. 3065. Berlin: Springer-Verlag, 2004, pp. 43–56.

21. Bürckert, H.J. A resolution principle for constrained logics. *Artificial Intelligence, 66,* 2 (1994), 235–271.

22. Business Process Modeling Notation Web site. www.bpmn.org.

23. Cabeza Gras, D., and Hermenegildo, M.V. Distributed WWW programming using (Ciao-)Prolog and the PiLLoW library. *Theory and Practice of Logic Programming, 1*, 3 (2001), 251–282.

24. Clark, K.L. Negation as failure. In H. Gallaire and J. Minker (eds.), *Logic and Data Bases*. New York: Plenum Press, 1978, pp. 293–322.

25. Denecker M., and De Schreye, D. SLDNFA: an abductive procedure for abductive logic programs. *Journal of Logic Programming, 34*, 2 (1998), 111–167.

26. Dignum, V.; Meyer, J.J.; Dignum, F.; and Weigand, H. Formal specification of interaction in agent societies. In M.G. Hinchey, J.L. Rash, W.F. Truzkowski, C. Rouff, and D. Gordon-Spears (eds.), *Proceedings of the Second Goddard Workshop on Formal Approaches to Agent-Based Systems.* LCNS, vol. 2699. Berlin: Springer, 2002, pp. 37–52.

27. Eiter, T.; Subrahmanian, V.S.; and Pick, G. Heterogeneous active agents, I: Semantics. *Artificial Intelligence, 108*, 1–2 (March 1999), 179–255.
28. Farrell, A.D.H.; Sergot, M.J.; Sallé, M.; and Bartolini, C. Using the event calculus for tracking the normative state of contracts. *International Journal of Cooperative Information Systems, 14*, 2–3 (2005), 99–129.
29. Fung, T.H., and Kowalski, R.A. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming, 33*, 2 (November 1997), 151–165.
30. Governatori, G. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems, 14*, 2–3 (2005), 181–216.
31. Governatori, G., and Milosevic, Z. Dealing with contract violations: Formalism and domain specific language. In M.J. van Sinderen, M.W.A. Steen, M.M. Lankhorst, M. Alesky, and P.C.K. Hung (eds.), *Proceedings of the Enterprise Distributed Object Computing Conference.* Los Alamitos, CA: IEEE Computer Society, 2005, pp. 46–57.
32. Governatori, G., and Milosevic, Z. A formal analysis of a business contract language. *International Journal of Cooperative Information Systems, 15*, 4 (2006), 659–685.
33. Governatori, G.; Milosevic, Z.; and Sadiq, S. Compliance checking between business processes and business contracts. In P.C.K. Hung, Q. Li, and D. Sparrow (eds.), *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference.* Los Alamitos, CA: IEEE Computer Society, 2006, pp. 221–232.
34. Governatori, G.; Rotolo, A.; and Sartor, G. Temporalised normative positions in defeasible logic. In G. Sartor and A. Gardner (eds.), *Proceedings of the 10th International Conference on Artificial Intelligence and Law.* New York: ACM Press, 2005, pp. 25–34.
35. Grosof, B.N.; Labrou, Y.; and Chan, H.Y. A declarative approach to business rules in contracts: Courteous logic programs in XML. In S. Feldman and M. Wellman (eds.), *ACM Conference on Electronic Commerce*. New York: ACM Press, 1999, pp. 68–77.
36. Herrestad, H. Norms and formalization. In A.R. Susskind (ed.), *Proceedings of the 3rd International Conference on Artificial Intelligence and Law*. New York: ACM Press, 1991, pp. 175–184.
37. Jaffar, J., and Maher, M.J. Constraint logic programming: A survey. *Journal of Logic Programming, 19-20* (1994), 503–582.
38. Kunen, K. Negation in logic programming. *Journal of Logic Programming, 4* (1987), 289–308.
39. Meyer, J.J. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. *Notre Dame Journal of Formal Logic, 29*, 1 (1988), 109–136.
40. Milosevic, Z. Enterprise aspects of open distributed systems. Ph.D. dissertation, University of Queensland, Computer Science Department, October 1995.
41. Milosevic, Z.; Gibson, S.; Linington, P.F.; Cole, J.; and Kulkarni, S. On design and implementation of a contract monitoring facility. In B. Benatallah, C. Godart, and M.-C. Shan (eds.), *Proceedings of the First International Work-*

shop on Electronic Contracting. Los Alamitos, CA: IEEE Computer Society, 2004, pp. 62–70.
42. Prakken, H.; and Sergot, M. Contrary-to-duty obligations. *Studia Logica, 57*, 1 (1996), 91–115.
43. Sartor, G. *Legal Reasoning: A Cognitive Approach to the Law*. Berlin: Springer, 2005.
44. *S*CIFF abductive proof procedure, 2005. http://lia.deis.unibo.it/research/sciff/.
45. SICStus prolog user manual, release 3.12.7, October 2006. www.sics.se/isl/sicstus/.
46. Societies Of ComputeeS (SOCS): A computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. IST2001-32530, 2002-2005. http://lia.deis.unibo.it/research/socs/.
47. van der Torre, L.W.N., and Tan, Y.-H. Diagnosis and decision making in normative reasoning. *Artificial Intelligence and Law*, 7, 1 (1999), 51–67.
48. Xanthakos, I. Semantic integration of information by abduction. Ph.D. dissertation, Imperial College London, 2003. www.doc.ic.ac.uk/~ix98/PhD.zip).

MARCO ALBERTI (marco.alberti@unife.it) received his M.Eng. in electronic engineering (2001) and Ph.D. in information engineering (2005) from the University of Ferrara, where he is currently a research fellow in the Department of Engineering. His research interests are abductive logic programming, constraint logic programming, multi-agent systems, and normative systems.

FEDERICO CHESANI (fchesani@deis.unibo.it) received his Ph.D. in computer science from the University of Bologna, where he is currently a research assistant in the Department of Electronics, Informatics, and Systems. His research interests include abduction and computational logic, verification techniques and specification languages, applied to multi-agent systems and service-oriented computing. He is a member of the Italian Interest Group on Logic Programming (GULP).

MARCO GAVANELLI (marco.gavanelli@unife.it) is a researcher in the Department of Engineering, Ferrara University. He received his laurea degree in computer science engineering from the University of Bologna, and his Ph.D. in information engineering from the University of Modena and Reggio Emilia. His interests are in constraint logic programming languages, abductive reasoning, multi-criteria optimization, and reformulation of combinatorial problems. He is member of the Italian Association for Artificial Intelligence (AI*IA) and the Association of Logic Programming, and is coordinator of the former's interest group on Knowledge Representation and Automated Reasoning.

EVELINA LAMMA (evelina.lamma@unife.it) received her degree in electronic engineering and her Ph.D. in computer science (1990) from the University of Bologna. She is a full professor in the Faculty of Engineering of the University of Ferrara, where she teaches artificial intelligence and foundations of computer science. Her research focuses on programming languages (logic languages, modular and object-oriented programming), artificial intelligence, knowledge representation, multi-agent systems, machine learning, and data mining. She has participated in several national and international research projects, and was responsible for the research group of the Department of Engineering of the University of Ferrara in the context of the UE V Framework

Program—Global Computing Action. She also coordinates her department's Ph.D. program in engineering science.

PAOLA MELLO (pmello@deis.unibo.it) received her degree in electronic engineering (1982) and her Ph.D. in computer science (1989) from the University of Bologna, where she is currently a full professor in the Faculty of Engineering, teaching artificial intelligence and foundations of computer science. Her research focuses on programming languages (logic languages, modular and object-oriented programming), artificial intelligence, knowledge representation and knowledge-based systems, intelligent agents and multi-agent systems, and machine learning. She has participated in several national and international research projects, and was responsible for the research group of the Department of Engineering, Information and Systems (DEIS) of the University of Bologna in the context of the UE V Framework Program—Global Computing Action.

MARCO MONTALI (mmontali@deis.unibo.it) studied informatics engineering in the Engineering Faculty of the University of Bologna, and received his master's degree in 2005, with a thesis about a graphical language for the specification and formalization of business processes. Since 2006 he is a Ph.D. candidate in the Department of Electronics, Informatics, and Systems at the University of Bologna. His research focuses on the application of computational logic approaches to the declarative specification and verification of business processes and services.

PAOLO TORRONI (paolo.torroni@unibo.it) is an assistant professor in computer engineering in the University of Bologna's Department of Electronic Engineering. His research interests include the use of logic in computer science and AI, particularly declarative and logic programming, hypothetical reasoning, argumentation, and agent-based systems. He received his Ph.D. in computer science from the University of Bologna in 2002. He is a member of the DALT and CLIMA steering committees and secretary of the Italian Interest Group on Logic Programming (GULP).