Learning Specifications of Interaction Protocols and Business Processes and Proving their Properties

Apprendimento di specifiche di protocolli di interazione e processi di business e verifica delle loro proprietà

Marco Alberti, Marco Gavanelli, Evelina Lamma, Fabrizio Riguzzi, and Sergio Storari

SOMMARIO/*ABSTRACT*

Questo articolo descrive le nostre recenti attività di ricerca per apprendere (con tecniche di Programmazione Logica Induttiva) specifiche modellate in programmazione logica e per verificare (attraverso una procedura di dimostrazione abduttiva) le proprietà di sistemi così specificati. I sistemi realizzati qui descritti sono stati applicati rispettivamente per l'apprendimento e la verifica di proprietà di protocolli di interazione in sistemi multi-agente, servizi Web, protocolli di screening e processi di business.

In this paper, we overview our recent research activity concerning the induction of Logic Programming specifications, and the proof of their properties via Abductive Logic Programming. Both the inductive and abductive tools here briefly described have been applied to respectively learn and verify (properties of) interaction protocols in multi-agent systems, Web service choreographies, careflows and business processes.

Keywords: Computational logic, Induction, Abduction, Interaction protocols, Careflows, Business processes.

1 Introduction

Thanks to its declarative semantics and its underlying proof theory, Logic Programming, and Computational Logic (CL, for short) in a broader sense, have been proved high-level formal languages for specification and verification. The adoption of logic for computer programming was promoted and improved in the late seventies also in Italy by a clever community. Logic Programming is grounded on a purely declarative representation language, and a theorem-prover or model-generator (like in Answer Set Programming) as the problem-solver. The main task of the problem-solver is the verification that an (existential) query holds in the given specification. Variants of the problem-solver can also be exploited to enrich the representation language and empower the reasoning with new features, such as hypothetical and non-monotonic reasoning, or to prove properties arising from the specification itself. Induction techniques can also be applied, to learn (general and formal) specification from logs and extensional databases or to further abstract specifications.

In this paper, we describe the recent activity carried out at ENDIF, University of Ferrara concerning the induction of CL-based specifications, and the proof of their properties. To this purpose, in the former activity we exploit Inductive Logic Programming techniques (ILP for short), and the DPML algorithm [13] in particular. This algorithm learns a specification expressed in a CL-based language from labeled traces (a database of events recording happened interactions or activities). The target language, named $\mathcal{S}$CIFF, was originally defined for the specification of interaction protocols in the context of the UE IST-2001-32530 Project (named SOCS), and has been later adopted to specify web service choreographies [1], careflows [12] and business processes [5]. A system is specified in the $\mathcal{S}$CIFF language by a knowledge base (a logic program) and a set of $\mathcal{S}$CIFF forward rules, called *integrity constraints*. Each integrity constraint relates occurring events (in the body) with an expected behaviour (typically in the head) in terms of expectations about events. Expectations can be positive (for mandatory events) or negative (forbidden events). Given a $\mathcal{S}$CIFF specification, the compliance of the system to the specifications can be checked on-the-fly through the $\mathcal{S}$CIFF proof-procedure [3], that abduces the expected behaviour and verifies its matching with the actual one.

The adoption of a CL-based language in specifying a system paved also the way to follow a proof-theoretic approach for proving or disproving properties of the given $\mathcal{S}$CIFF specification. To this purpose, we exploit abduction, and in particular an extension of the $\mathcal{S}$CIFF proof-procedure called g-$\mathcal{S}$CIFF [2]. g-$\mathcal{S}$CIFF is an abductive proof-procedure which, starting from a goal, verifies, in a generative manner by abduction, whether there exists

a scenario (i.e., a set of generated events) supporting the goal, consistent with the given integrity constraints, and not self-contradictory (e.g., an event does not unify with any forbidden one). In this case, this scenario represents a witness for the goal, and also corresponds to extensions identified by the declarative semantics.

The paper is organized as follows. In Section 2 we briefly introduce the $\mathcal{S}$CIFF language. In Section 3, we show how learning from interpretations can be exploited to learn a $\mathcal{S}$CIFF theory, and also discuss some experimental results. In Section 4, we present g-$\mathcal{S}$CIFF and discuss its application to the learned specification of the previous section. Related work is mentioned throughout the paper. Finally, we conclude in Section 5.

This work has been carried out in strict collaboration with the DEIS group. This paper is complementary to [7] contained in this same issue, where they focus on interaction specification and verification in several domains.

2 The $\mathcal{S}$CIFF Language

The $\mathcal{S}$CIFF proof-procedure is an abductive framework, extension of the well-known IFF abductive language and proof procedure defined by Fung and Kowalski [11]. $\mathcal{S}$CIFF is able to reason about dynamically happening events, and to generate corresponding expectations. To represent that an event ev happened (i.e., an atomic activity has been executed) at a certain time T, $\mathcal{S}$CIFF uses the symbol $\mathbf{H}(ev, T)$, where ev is a term and T is a number indicating the time. Hence, an execution trace is modeled as a set of happened events, also called *scenario* or *history* (**HAP**). For example, we could formalize that bob has performed activity a at time 5 as follows: $\mathbf{H}(a(bob), 5)$. Furthermore, $\mathcal{S}$CIFF introduces the concept of expectation, which plays a key role when defining global interaction protocols, choreographies, and more in general event-driven processes. It is quite natural, in fact, to think of a process in terms of rules of the form: "if A happened, then B is expected to happen". Positive (resp. negative) expectations are denoted by $\mathbf{E}(ev, T)$ (resp. $\mathbf{EN}(ev, T)$), meaning that ev is expected (resp. expected not) to happen at time T. To satisfy a positive (resp. negative) expectation an execution trace must contain (resp. not contain) a matching happened event.

$\mathcal{S}$CIFF Integrity Constraints (ICs for short) are forward rules of the form $Body \rightarrow Head$:

$$Body \rightarrow Disj_1 \vee \ldots \vee Disj_n \tag{1}$$

where $Body$ is a conjunction of happened events and literals of predicates defined in a $\mathcal{S}$CIFF knowledge base, and $Disj_j$ is a conjunction of expectations (positive and negative) and literals from the knowledge base.

Variables in common to $Body$ and $Head$ are universally quantified ($\forall$) with scope the whole IC. Variables occurring in positive (negative) expectations in $Head(C)$ are existentially (universally) quantified with scope the disjunct where they appear.

An example of an IC is

$$\begin{array}{ll} & \mathbf{H}(a(bob), T) \wedge T < 10 \\ \rightarrow & \mathbf{E}(b(alice), T_1) \wedge T < T_1 \\ \vee & \mathbf{EN}(c(mary), T_1) \wedge T < T_1 \wedge T_1 < T + 10 \end{array} \tag{2}$$

The meaning of the IC (2) is the following: if bob has executed action a at a time $T < 10$, then we either expect $alice$ to execute action b at some time ($\exists T_1$) later than T or we expect that $mary$ does not execute action c at any time ($\forall T_1$) within 9 time units after T.

The interpretation of an IC is the following: if there exists a substitution of variables such that the body is true in an interpretation representing a trace, then one of the disjuncts in the head must be true.

Roughly speaking, $\mathcal{S}$CIFF combines occurred events with the specified rules, to suitably generate the corresponding expectations; then, expectations are verified against the execution trace: a positive expectation must have a corresponding matching event, whereas a negative expectation forbids the presence of a matching event into the trace. If such conditions are not met (i.e., a positive/negative expectation is not/is matched by a corresponding event), then the expectations are violated, and the execution trace is evaluated as non-compliant.

The main and original application of the $\mathcal{S}$CIFF proof-procedure is to verify whether an execution of the process concretely adheres to the specification, i.e., to perform *compliance checking*. $\mathcal{S}$CIFF is seamlessly able to check compliance both at run-time, by dynamically collecting and reasoning upon occurring events, or a-posteriori, by analyzing the log of an observed execution trace.

3 Learning $\mathcal{S}$CIFF specifications

Since ICs can be seen as an extension of logical clauses, we can apply the techniques developed in the learning from interpretations setting of Inductive Logic Programming [14] to the problem of inducing ICs. In particular, in [13] we modified the Inductive Constraint Logic (ICL) algorithm [9] that takes as input a set of interpretations labeled as positive or negative and returns a clausal theory that is true in as many positive interpretations as possible and false in as many negative interpretations as possible. We called the resulting system DPML [13], for Declarative Process Model Learner.

DPML modifies ICL by replacing the procedure for testing the truth of a clause in an interpretation with a $\mathcal{S}$CIFF-like procedure, by defining a generality order among ICs and, on the basis of this order, by defining a refinement operator. In this way, we can perform search in the space of ICs and evaluate each candidate against the training set.

In DPML the θ-subsumption generality order among clauses is modified in order to take into account the fact that the head is a disjunction of conjunctions. With the

new generality relation, we can obtain a generalization D of an IC C by adding a literal to the body, adding a disjunct to the head, removing a literal from a disjunct in the head or adding a literal to a disjunct in the head. This generalization operator is used by DPML to search the space of ICs from specific to general.

The literals to be added are defined by the *language bias*, an intensional definition of the search space. In DPML the language bias is a set of assertions in the form of pairs (BS, HS), where BS is a set that contains the literals that can be added to the body and HS is a set that contains the disjuncts that can be added to the head.

Inducing $\mathcal{S}$CIFF theories is also interesting because it has been shown [6] that other declarative process languages such as DecSerFlow [17] or ConDec [16] can be mapped to $\mathcal{S}$CIFF. Therefore, if we can ensure that the form of the learned ICs corresponds to one of the constraints of these languages, we could learn such constraints by first learning ICs and then translating them into DecSerFlow or ConDec. By providing DPML with a language bias that suitably restricts the search space of ICs, DPML returns a theory with ICs in the desired form, that can be automatically translated into one of the above declarative process languages (see also [12]).

We implemented the whole process of induction plus translation in the DecMiner [12] plug-in of ProM. DecMiner assists the user in all the phases of the learning process, from the definition of the language bias, to the labeling of traces, to the translation of the mined ICs into ConDec constraints.

In particular, the language bias is automatically generated by DecMiner considering a subset of activities A and a subset of ConDec constraints T choosen by the user.

DPML and DecMiner have been tested on artificial and real datasets. The artificial datasets were randomly generated from two process models, namely the NetBill protocol [8] and an electronic auction protocol [4]. The real dataset regards the healthcare process of cervical cancer screening in the Emilia-Romagna Italian region. DPML and DecMiner results were compared with those of the α-algorithm [18] and of the Multi-Phase Miner (MPM) [19] that learn procedural process models. The comparison shows that DPML and DecMiner were able to induce more accurate models, often by a large amount.

We now briefly discuss the methodology followed by illustrating the application of DecMiner to the hotel and spa case: the model, inspired by the example presented in [15], describes a simple process of renting rooms and services in a hotel and spa. After registering at the front desk, the client can request one or more rooms, laundry and massage services. Each service, identified by a code, is followed by the registration of the service costs into the client bill. Moreover, if the client chooses a "Shiatzu" massage, the spa presents her/him a special offer. The cost related to the number of nights can be billed before check-out, during check-out or even after check-out.

The $\mathcal{S}$CIFF representation of the hotel model is composed of eight ICs. One of them:

$$\begin{aligned}&\mathbf{H}(massage\_service(Type, ma\_id(ID_{ls})), T_{ls})\\ \rightarrow\ &\mathbf{E}(bill\_massage\_service(ma\_id(ID_{bls})), T_{bls})\\ &\wedge ID_{ls} = ID_{bls} \wedge T_{bls} > T_{ls}.\end{aligned}$$

specifies that a massage service must be followed by the registration of the cost into the client bill.

Five training sets have been generated by randomly building a trace and then classifying it with the ICs of the correct model. The trace is then assigned to the set of positive or negative traces depending on the result of the test. The process is repeated until 2000 positive traces and 2000 negative traces have been generated.

DecMiner, the α-algorithm and MPM were applied to each training set and the learned model was tested on a randomly generated testing set. DecMiner achieved an average accuracy of 99.96%, higher than those of the α-algorithm and MPM.

The sets of ICs returned by DPML/DecMiner can also be used to check (intensional) properties. This can be done by exploiting the g-$\mathcal{S}$CIFF proof-procedure described in the following.

4 Proving properties by g-$\mathcal{S}$CIFF

The $\mathcal{S}$CIFF proof-procedure addresses the important software engineering task of checking compliance during runtime (or a-posteriori using an *event log*), i.e., whether the agents behave in a compliant manner with respect to a given interaction protocol or specification. However, this does not exhaust the possible uses of abductive reasoning: the event literals composing the history can be assumed as well, in order to foresee all the possible evolutions of the system under test. Knowing the specification (in terms of an abductive program), one could (in principle) generate all the histories that the system can support and then study them for common patterns or to formally prove properties of the system.

Of course, explicitly generating all the histories is not feasible, since the number of histories compliant to a protocol are typically infinite for protocols of practical use. However, we can generate compliant histories in intensional way, and then reason upon them: the hypothetical events can contain variables, possibly subject to CLP constraints. In order to generate compliant histories, $\mathcal{S}$CIFF has been improved and extended to a generative version, called g-$\mathcal{S}$CIFF. g-$\mathcal{S}$CIFF considers $\mathbf{H}$ literals as abducibles, and contains a new transition, called *fulfillment*, that fulfils the expectations by abducing matching events:

$$\mathbf{E}(X, T) \rightarrow \mathbf{H}(X, T).$$

g-$\mathcal{S}$CIFF is provably sound: all generated histories fulfil the given specifications.

In the literature, properties are often classified as safety or liveness properties. A *safety* property is a universal property: intuitively, it ensures that nothing bad will ever

happen (whenever the protocol/specification is respected). A *liveness* property is, instead, existential: it ensures that something good will eventually happen. A liveness property can be passed to g-$\mathcal{S}$CIFF as a goal containing positive expectations: if the g-$\mathcal{S}$CIFF proof-procedure succeeds in proving the goal, the generated history witnesses that there exists a way to obtain the goal while being conformant to the protocol. A safety property ϕ can be negated (as in model checking), and then passed to g-$\mathcal{S}$CIFF as a goal $\mathcal{G} \equiv \neg\phi$. If the g-$\mathcal{S}$CIFF proof-procedure succeeds in finding a history $\mathbf{HAP}$ (i.e., $\models_{\mathbf{HAP}} \neg\phi$), we have a counterexample: the history $\mathbf{HAP}$ satisfies the protocol and does not enjoy the safety property ϕ.

The g-$\mathcal{S}$CIFF proof-procedure is implemented in SICStus 4, making extensive use of Constraint Handling Rules [10] to implement its transitions. $\mathcal{S}$CIFF and g-$\mathcal{S}$CIFF come in a same package, that can be freely downloaded from the web[1]: the g-$\mathcal{S}$CIFF behaviour is activated by simply setting an option.

The g-$\mathcal{S}$CIFF proof-procedure has been applied to the formal verification of various systems and protocols. g-$\mathcal{S}$CIFF was able to derive the flawedness of the Needham-Schroeder security protocol [2], and the good atomicity property of the NetBill protocol [2]. It is also a basic component of the A^{l}LoWS framework [1], for the proof of interoperability between Web services.

The g-$\mathcal{S}$CIFF proof-procedure operates top-down in a deductive and abductive manner, by manipulating the specification driven by the goal, as usual in Logic Programming, and also generating expectations as $\mathcal{S}$CIFF does and, by *fulfillment* an (intensional) set of events needed to support the goal. This way, g-$\mathcal{S}$CIFF can be used to prove properties of any $\mathcal{S}$CIFF protocol. For example, one may wonder if the protocol allows a massage service not to be followed by a shiatzu package offer. By expressing this combination as a g-$\mathcal{S}$CIFF query, the user can ask g-$\mathcal{S}$CIFF to generate an intensional history that satisfies the query while fulfilling the protocol. In fact, g-$\mathcal{S}$CIFF generates such a history, with the constraint that the massage type must not be shiatzu.

5 Conclusions

We have presented the CL-based language $\mathcal{S}$CIFF for the specifications of complex systems with interacting entities, such as multi-agent systems, business processes or web services. Moreover, we have discussed how techniques from Inductive Logic Programming were applied for inducing $\mathcal{S}$CIFF theories which can be then translated into graphical languages. Finally, the abductive g-$\mathcal{S}$CIFF proof procedure can be used for proving properties of specifications, either learned or provided by the user.

[1] `http://lia.deis.unibo.it/sciff/`

Acknowledgements

We are in debt with the Artificial Intelligence group at DEIS, University of Bologna who shared with us most of the activity here reported.

REFERENCES

[1] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and M. Montali. An abductive framework for a-priori verification of web services. In *PPDP 2006*, pages 39–50. ACM Press, 2006.

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Security protocols verification in abductive logic programming: a case study. In *ESAW 2005*, pages 283–295, 2005.

[3] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logics*, 9(4), 2008. Accepted for publication.

[4] A. Chavez and P. Maes. Kasbah: An agent marketplace for buying and selling goods. In *PAAM 1996*, pages 75–90, London, 1996.

[5] F. Chesani, P. Mello, M. Montali, and S. Storari. Testing careflow process execution conformance by translating a graphical language to computational logic. In *AIME 07*, number 4594 in LNAI, pages 479–488, 2007.

[6] F. Chesani, P. Mello, M. Montali, and S. Storari. Towards a DecSerFlow declarative semantics based on computational logic. Technical Report DEIS-LIA-07-002, University of Bologna, 2007.

[7] F. Chesani, P. Mello, M. Montali, and P. Torroni. Modeling and verification of business processes and choreographies in ALP. *Intelligenza Artificiale*. In this issue.

[8] B. Cox, J. D. Tygar, and M. Sirbu. NetBill security and transaction protocol. In 1^{st} *USENIX Workshop on Electronic Commerce*, pages 77–88, 1995.

[9] L. De Raedt and W. Van Laer. Inductive constraint logic. In *ALT 1995*, volume 997 of *LNAI*, pages 80–94, 1995.

[10] T. Frühwirth. Theory and practice of constraint handling rules. *J. of Logic Prog.*, 37(1-3):95–138, 1998.

[11] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[12] E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Inducing declarative logic-based models from labeled traces. In *BPM 2007*, volume 4714 of *LNCS*, pages 344–359, 2007.

[13] E. Lamma, P. Mello, F. Riguzzi, and S. Storari. Applying inductive logic programming to process mining. In *ILP 2007*, volume 4894 of *LNAI*, pages 132–146, 2007.

[14] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *J. Logic Prog.*, 19/20:629–679, 1994.

[15] M. Pesic, H. Schonenberg, and W.M.P. van der Aalst. Declare: Full support for loosely-structured processes. In *EDOC 2007*, pages 287–300. IEEE Computer Society, 2007.

[16] W.M.P. van der Aalst and M. Pesic. A declarative approach for flexible business processes management. In *BPM 2006*, volume 4103 of *LNCS*, pages 169–180, 2006.

[17] W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In *WS-FM 2006*, volume 4184 of *LNCS*, pages 1–23, 2006.

[18] W.M.P. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1128–1142, 2004.

[19] B. F. van Dongen and W. M. P. van der Aalst. Multi-phase process mining: Building instance graphs. In *ER 2004*, volume 3288 of *LNCS*, pages 362–376, 2004.

Contacts

Marco Alberti, marco.alberti@unife.it
Marco Gavanelli, marco.gavanelli@unife.it
Evelina Lamma, evelina.lamma@unife.it
Fabrizio Riguzzi, fabrizio.riguzzi@unife.it
Sergio Storari, sergio.storari@unife.it

tutti affiliati al

Dipartimento di Ingegneria,
Università di Ferrara
Via Saragat, 1
44100 Ferrara