Università degli Studi di Bologna
DEIS

Verification from declarative specifications using Logic Programming

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

June 25, 2008

DEIS Technical Report no. DEIS-LIA-08-004 *LIA Series no. 90*

Verification from declarative specifications using Logic Programming

Marco Montali [1] Marco Alberti [2] Federico Chesani [1]
Marco Gavanelli [2] Evelina Lamma [2] Paola Mello [1]
Paolo Torroni [1]

[1] *DEIS, University of Bologna*
V.le Risorgimento 2
40136 Bologna, Italy
name.surname@unibo.it

[2] *ENDIF, University of Ferrara*
V. Saragat 1
44100 Ferrara, Italy
name.surname@unife.it

June 25, 2008

Abstract. We address the problem of formal verification for systems specified using declarative languages. We propose a verification method based on the g-SCIFF abductive logic programming proof-procedure and evaluate our method empirically, by comparing its performance with that of other verification frameworks.

Keywords: *Abductive Logic Programming, SCIFF, g-SCIFF, declarative specifications, formal verification, DecSerFlow, business process management*

Contents

1 Introduction

Since its introduction, the declarative programming paradigm has been successfully adopted by a vast community of IT researchers and practitioners. As in the case of logic programming, the separation of logic aspects from control aspects long advocated by Robert Kowalski [20] enables the programmer to more easily write correct programs, improve and modify them. In recent years, the declarative programming philosophy has had a visible impact on new emerging disciplines. Examples are multi-agent interaction protocol specification languages, which rely on declarative concepts such as commitments [29] or expectations [3] and make an extensive use of rules, and declarative service flow specification languages such as DecSerFlow [30].

Although declarative technologies improve readability and modifiability, and help reducing programming errors, what makes systems trustworthy and reliable is formal verification. Since the temporal dimension in these settings plays a fundamental role, a natural choice would be to check temporal logic specifications using mainstream program verification technology such as model checking. However, it is well known that the construction of the input for model checking algorithms takes a considerable amount of resources. This is especially true if we consider that the translation of a linear time temporal logic (LTL, [14]) formula into an automaton is a very hard problem, even with small sized formulae, and it becomes undecidable for variants of temporal logic with explicit representation of time, such as metric temporal logic (MTL, [5]) and timed propositional temporal logic (TPTL, [4]) with dense time [31]. By adopting an approach based on logic programming (LP), a system's specifications can be directly represented as a logic formula, handled by a proof system with no need for a translation. Therefore, we address the problem of verifying such specifications using abductive logic programming (ALP), and in particular the SCIFF framework. SCIFF is an ALP language and proof system for the specification and run-time verification of interaction protocols. From a practical viewpoint, SCIFF has some advantages with respect to LTL, in that it enables reasoning with universally and existentially quantified variables, constraint logic programming (CLP) constraints and quantifier restrictions [10]. It has an explicit representation of time, which can be modelled as a discrete or as a dense variable, depending on the constraint solver of choice. We address the verification problem using a "generative" extension to the SCIFF abductive proof-procedure, called g-SCIFF, which can be used to prove system properties at design time, or to generate counterexamples of properties that do not hold. Via g-SCIFF, we can carry out a goal-directed verification task, starting from declarative specifications and without having to generate an automaton. The experiments we run to compare the performance of g-SCIFF and that of other model checkers support our claims and motivate us to pursue this line of research.

We continue this article by giving the necessary technical background on SCIFF.

In Section 3, we discuss the application domains and we propose some examples of specification and verification in the context of business process management (BPM). Section 4 presents our verification method based on g-SCIFF. Sections 5 and 6 discuss and evaluate it, also experimentally, in relation with other verification techniques. Section 7 concludes.

2 The SCIFF language and proof-procedure

SCIFF was initially proposed to specify agent interaction protocols [3], and it has also been successfully applied to business process [9, 24], electronic contract [2], and Web service choreography specification [1]. SCIFF specifications consist of an abductive logic program, i.e., a triplet $\langle \mathcal{P}, \mathcal{IC}, \mathcal{A} \rangle$ where $\mathcal{P}$ is a logic program (a collection of clauses), $\mathcal{IC}$ is a set of integrity constraints (IC) and $\mathcal{A}$ is a set of abducible predicates. SCIFF considers events as first class objects. Events can be, for example, sending a message, or starting an action, and they are associated with a time point. Events are identified by a special functor, **H**, and are described by an arbitrary term. SCIFF uses ICs to model relations among events and expectations about events. Expectations are abducibles identified by functors **E** and **EN**. **E** are "positive" expectations, and indicate events to be expected. **EN** are "negative" expectations and model events that are expected not to occur. Event and time variables can be constrained by means of Prolog predicates or CLP constraints [19]; the latter are especially useful to specify orderings between events and quantitative time constraints (such as delays and deadlines). An IC is a forward $body \Rightarrow head$ reactive rule which links happened events and expectations. Typically, the *body* contains a conjunction of happened events, whereas the *head* is a disjunction of conjunctions of positive and negative expectations. The intuition is that when the body of a rule becomes true (i.e., the involved events occur), then the rule fires, and the expectations in the head are generated by abduction. For example, $\mathbf{H}(a, T) \Rightarrow \mathbf{EN}(b, T')$ defines a relation between events a and b, saying that if a occurs at time T, b should not occur at any time. Instead, $\mathbf{H}(a, T) \Rightarrow \mathbf{E}(b, T') \wedge T' \leq T + 300$ says that if a occurs, then an event b should occur no later than 300 time units after a.

To exhibit a correct behavior, given a goal $\mathcal{G}$ and a triplet $\langle \mathcal{P}, \mathcal{IC}, \mathcal{A} \rangle$, a set of abduced expectations must be *fulfilled* by corresponding events. The concept of fulfillment is formally captured by the SCIFF declarative semantics [3], which intuitively states that $\mathcal{P}$, together with the abduced literals, must entail $\mathcal{G} \wedge \mathcal{IC}$, positive expectations must have a corresponding matching happened event, and negative expectations must not have a corresponding matching event.

The language is associated with a proof-procedure which checks the compliance of a narrative of events with the protocol specifications, by matching events with expectations. Such a unification can be checked by the SCIFF proof-procedure dynamically, as occurring events are detected (run-time monitoring and checking) or

after execution, based on a given narrative of events (log-based, a posteriori checking). The SCIFF proof-procedure enjoys termination, soundness, and completeness properties with respect to its declarative semantics [3]. It is implemented in SICStus 4, and its implementation features a unique design, that has not been used before in other abductive proof-procedures. First, the various transitions in the operational semantics are implemented as constraint handling rules (CHR, [16]).[1] The second important feature is its ability to interface with constraint solvers: both with the CLP(FD) solver and with the CLP($\mathcal{R}$) solver embedded in SICStus. The user can thus choose the most suitable solver for the application at hand, which is an important issue in practice. It is well known, in fact, that no solver dominates the other, and we measured, in different applications, orders of magnitude of improvements by switching solver. In this paper, we report the results obtained with the CLP($\mathcal{R}$) solver, which is based on the simplex algorithm, and features a complete propagation of linear constraints.

3 Business processes: specification and verification

If we take some time to skim through recent BPM, Web service choreography, and multi-agent system literature, we will find a strong push for declarativeness. In the BPM context, Wil van der Aalst and Maja Pesic recently proposed a declarative service flow language (DecSerFlow, [30]) to specify, enact, and monitor service flows. Their claim is that declarative languages fit better than procedural ones with the autonomous nature of services. To motivate their claim, the authors show a simple example with two activities, A and B, which can be executed multiple times but exclude each other, i.e., after the first occurrence of A it is not permitted to do B anymore and after the first occurrence of B it is not permitted to do A. This could be expressed declaratively via a simple LTL expression: $\neg(\Diamond A \wedge \Diamond B)$. But in a procedural language, such as the most commonly used business process execution language for web services, BPEL4WS [6], and the like, it is difficult to specify the above process without implicitly introducing additional assumptions and choice points. LTL does not need to introduce such additional objects. This is also true of LP rules. For example, in SCIFF we could use two ICs, $\mathbf{H}(a, T) \Rightarrow \mathbf{EN}(b, T')$ and $\mathbf{H}(b, T) \Rightarrow \mathbf{EN}(a, T')$, to define precisely the same model without introducing additional constraints.

The case for declarative approaches is equally strong in other domains. In [32], Pınar Yolum and Munindar Singh motivate very convincingly the case for the adoption of a "social" semantics for agent interaction protocols grounded on the declarative notion of commitment. They take the position that protocols should not only

[1] Other proof-procedures [11] have been implemented on top of CHR, but with a different design: they map integrity constraints (instead of transitions) into constraint handling rules. This choice gives more efficiency, but less flexibility.

constrain the actions of the participants, but also recognize the open and dynamic nature of interaction. In particular, to promote autonomy, protocol specifications should not be over-constrained, and they should be flexible enough to support heterogeneity, i.e., they should enable agents to adopt different strategies to carry out their interactions. Such a flexibility is achieved by using declarative formalisms to specify interaction, as opposed to rigid flow charts or, e.g., Petri nets.

Another area in which declarative approaches are becoming increasingly popular is contract specification and negotiation in virtual enterprises. In [17], Guido Governatori points out that the need to formalize business rules explicitly has become increasingly essential, and the use of logic modeling techniques is beneficial for reasoning about contracts, because it helps anomaly detection, hypothetical reasoning to investigate the effects of changes to contract clauses, and debugging.

In this article, we focus on the BPM domain. We use DecSerFlow [30] as a declarative flow specification language. Fig. 1 shows the DecSerFlow specification of a payment protocol. Boxes represent instances of activities. Numbers (e.g., 0; N..M) above the boxes are cardinality constraints that tell how many instances of the activity have to be done (e.g., never; between N and M). Edges and arrows represent relations and temporal relations between activities. Double line arrows indicate alternate execution (after A, B must be done before A can be done again), while barred arrows and lines indicate negative relations (doing A disallows doing B). Finally, a solid circle on one end of an edge indicates which activity activates the relation associated with the edge. For instance, the execution of accept advert in Fig. 1 does not activate any relation, because there is no circle on its end (a valid model could contain an instance of accept advert and nothing else), register instead activates a relation with accept advert (a model is not valid if it contains only register). If there is more than one circle, the relation is activated by each one of the activities that have a circle. Arrows with multiple source and/or destinations indicate temporal relations activated by either of the sources and satisfied by either of the destination actions. The parties involved—a merchant, a customer, and a banking service to handle the payment—are left implicit.

In our example, the six left-most boxes are customer actions, payment done/ payment failure model a banking service notification about the termination status of the payment action, and send receipt is a merchant action. The DecSerFlow chart specifies relations and constraints among such actions. If register is done (once or more than once), then also accept advert must be done (before or after register) at least once. No temporal ordering is implied by such a relation. Conversely, the arrow from choose item to close order indicates that, if close order is done, choose item must be done at least once before close order. However, due to the barred arrow, close order cannot be followed by (any instance of) choose item. The 0..1 cardinality constraints say that close order and send receipt can be done at most once. 1-click payment must be preceded by register and by close order, whereas standard payment

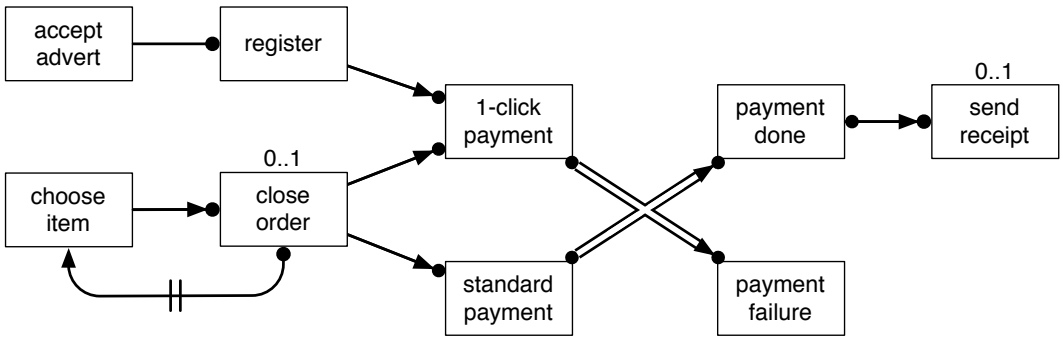

Figure 1. *A sample DecSerFlow model*

needs to be preceded only by close order (registration is not required). After 1-click or standard payment, either payment done or payment failure must follow, and no other payment can be done, before either of payment done/failure is done. After payment done there must be at most one instance of send receipt and before send receipt there must be at least a payment done. Sample valid models are: the empty model (no action done), a model containing one instance of accept advert and nothing else, and a model containing 5 instances of choose item followed by a close order. A model containing only one instance of 1-click payment instead is not valid.

The semantics of DecSerFlow is given in [30] in terms of LTL formulae, each one associated with a relation or constraint in the chart. The conjunction of all these formulae ("conjunction formula") gives the semantics of the entire chart. Conversely, in [24] Montali and colleagues have shown how to automatically translate a DecSerFlow chart into a SCIFF program.

For example, the relation between accept advert and register corresponds to the LTL formula ($\Diamond$register) $\Rightarrow$ ($\Diamond$accept advert) and to the following IC:

$$\mathbf{H}(register, T) \Rightarrow \mathbf{E}(acceptAdvert, T').$$

The barred arrow from close order to choose item corresponds to the LTL formula $\Box$(close order $\Rightarrow \neg(\Diamond$choose item)) and to the following IC:

$$\mathbf{H}(closeOrder, T) \Rightarrow \mathbf{EN}(chooseItem, T') \wedge T' > T.$$

Finally, the relation between payment done and send receipt corresponds to the LTL formula ($\Box$(payment done $\Rightarrow \Diamond$send receipt))$\wedge$(($\Diamond$send receipt) $\Rightarrow$ (($\neg$send receipt)$\mathcal{U}$payment done)) and to the following two ICs:

$$\mathbf{H}(paymentDone, T) \Rightarrow \mathbf{E}(receipt, T') \wedge T' > T$$
$$\mathbf{H}(receipt, T) \Rightarrow \mathbf{E}(paymentDone, T') \wedge T' < T.$$

A DecSerFlow chart is thus a good starting point to compare two verification methods: model checking LTL formulas, and our method, based on SCIFF.

Let us consider some examples of verification on the model. From now on, we only use the DecSerFlow notation for both specifications and properties to verify. 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 specifications are respected). A liveness property is, instead, existential: it ensures that something good will eventually happen.

A first, simple type of verification is known in the BPM domain as checking for dead activities. This is a liveness property. We want to check whether a given activity, say send receipt, will ever be executed at all. To verify, we add a 1..* cardinality constraint on the activity. If the extended specification becomes unfeasible, it means that send receipt will never be executed in any possible valid model, indicating that probably there is a mistake in the design. In our example, a verifier should return a positive answer, together with a sample valid execution, such as: choose item → close order → standard payment → payment done → send receipt, which amounts to a proof that send receipt is not a dead activity.

Let us consider a more elaborated example. We want to check whether it is still possible to have a complete transaction, if we add some constraints such as: the customer does not accept to receive ads, and the merchant does not offer standard payment. To verify, we add a 0 cardinality constraint on accept advert and on standard payment, and a 1..* cardinality constraint on send receipt, expressing that we want to obtain a complete transaction (see Fig. 2(a)).[2] Such an extended specifications is unsatisfiable. A verifier should return a negative answer.

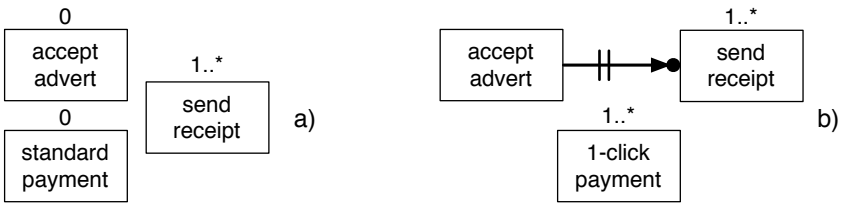

Figure 2. *Two sample queries: checking (a) existential and (b) universal properties.*

Let us now consider a safety property. A merchant wants to make sure that during a transaction with 1-click payment a receipt is always sent *after* the customer has accepted the ads. To verify, we extend the specifications with the query's negation, which is an existential query (*"does there exist a transaction executing 1-click payment in which accept advert is* not *executed before send receipt?"*). The negated query corresponds to the relations shown in Fig. 2(b). Given the model, this query

[2]This technique is also used to avoid vacuous answers, in which the model is trivially satisfied if nothing happens.

should succeed, since there is no temporal constraint associated with accept advert, thus accept advert does not have to be executed *before* send receipt in all valid models. The success of the existential negated query amounts to a counterexample against the initial (universal) query. A verifier should produce such a counterexample: choose item → close order → register → 1-click payment → payment done → send receipt → accept advert. That could lead a system designer to decide to improve the model, for example, by introducing an arrow from accept advert to send receipt.

Let us finally consider an example of a query with explicit time. We adopt an extended DecSerFlow notation with explicit time, proposed in [24]. In such a notation, arrows can be labeled with (*start time*, *end time*) pairs. The meaning of an arrow labelled (T_s, T_e) linking two activities A and B is: B must be done between T_s and T_e time units after A. A labeled barred arrow instead indicates that B cannot be executed between T_s and T_e time units after A. In this way we can express minimum and maximum latency constraints. For instance, express that B must occur after A and at most 12 time units after A, which amounts to a maximum latency constraint on the sequence $A \ldots B$, we connect A to B using a $(0, 12)$ labelled arrow.

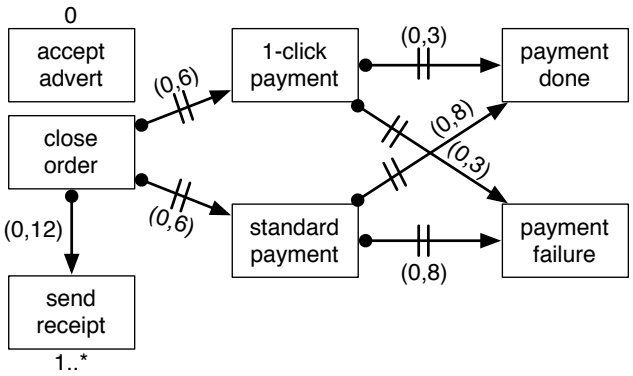

Figure 3. *Sample query concerning verification of properties on models with explicit time.*

The query depicted in Fig. 3 shows several such constraints, plus a 0 cardinality constraint on accept advert (the customer does not accept ads). The intuition behind the query is: *"is there a transaction with no accept advert, terminating with a send receipt within 12 time units as of close order, given that close order, 1-click payment, and standard payment cause a latency of 6, 3, and 8 time units?"* It turns out that the specification is unfeasible, because the 0 cardinality constraint on accept advert rules out the 1-click payment path, and the standard payment path takes more than 12 time units. A verifier should return failure.

4 Verification using g-SCIFF

The problems of run-time and a-posteriori verification of compliance are already effectively addressed by SCIFF [3]. In this section we discuss static verification.

Existing formal verification tools rely on model checking or theorem proving. However, a drawback of most model checking tools is that they typically only accommodate discrete time and finite domains. Moreover, the cardinality of domains impacts heavily on the performance of the verification process, especially in relation to the production of a model consisting of a state automaton. On the other hand, theorem proving in general has a low level of automation, and it may be hard to use, because it heavily relies on the user's expertise [18].

We propose a verification method which starts from SCIFF specifications and presents interesting features from both approaches. Like theorem proving, its performance is not heavily affected by domain cardinality, and it accommodates domains with uncountable elements, such as dense time. Similarly to model checking, it works in a push-button style, thus offering a high level of automation. Our method uses the g-SCIFF proof-procedure, which also insists on the SCIFF language. In fact, the implementation of SCIFF and g-SCIFF are a part of the same distribution package, which is freely available.[3]

In the style of [27], we do verification by abduction: in g-SCIFF, event literals are abduced as well as expectations, in order to model all the possible evolutions of the system being verified. SCIFF and g-SCIFF work by applying the transitions sketched in the following, until a fix-point is reached:

Unfolding substitutes an atom with its definitions in $\mathcal{P}$;

Propagation given an implication $(\mathbf{a}(X) \wedge R) \Rightarrow H$ and an abduced literal $\mathbf{a}(Y)$, generates the implication $(X = Y \wedge R) \Rightarrow H$;

Case Analysis Given an implication $(c(X) \wedge R) \Rightarrow H$ in which c is a constraint (possibly the equality constraint '='), generate two children: $c(X) \wedge (R \Rightarrow H)$ and $\neg c(X)$;

Splitting distributes conjunctions and disjunctions;

Logical Equivalences performs usual replacements: $true \Rightarrow A$ with A, etc.;

Constraint Solving posts constraints to the constraint solver of choice;

Fulfilment declares fulfilled;

- an expectation $\mathbf{E}(p, t)$ if there is a corresponding literal $\mathbf{H}(p, t)$, or

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

- an expectation $\mathbf{EN}(p,t)$ if there is no matching literal $\mathbf{H}(p,t)$ and it cannot happen in the sequel (e.g., because a deadline has expired);

Violation declares violated an expectation: symmetrical to fulfilment;

Fulfiller if an expectation $\mathbf{E}(p,t)$ is not fulfilled, abduces an event $\mathbf{H}(p,t)$;

Consistency imposes consistency of the set of expectations, by which $\mathbf{E}(p,t)$ and $\mathbf{EN}(p,t)$ cause failure.

Most of the transitions above are common to the two proof-procedures, but only g-SCIFF uses the *fulfiller* transition to generate narratives of events ("histories"). To do so, it applies the rule $\mathbf{E}(P,T) \rightarrow \mathbf{H}(P,T)$, which fulfills an expectation by abducing a matching event (possibly with variables). *Fulfiller* is applied only at the fix-point of the other transitions. SCIFF and g-SCIFF also exploit an implementation of reified unification (a solver on equality/disequality of terms) which takes into consideration quantifier restrictions [10] and variable quantification. Histories are thus generated intensionally, and hypothetical events can contain variables, possibly subject to CLP constraints.

Verification of properties is conducted as follows. A liveness property can be passed to g-SCIFF as a goal containing positive expectations: if the g-SCIFF proof-procedure succeeds in proving the goal, the generated history proves that there exists a way to obtain the goal via a valid execution of the activities. A safety property ϕ can be negated (as in model checking), and then passed to g-SCIFF as a goal $\mathcal{G} \equiv \neg\phi$. If the g-SCIFF proof-procedure succeeds in finding a history which satisfies $\mathcal{G}$, such a history is a counterexample against ϕ.

The examples shown in Section 3 are correctly handled by g-SCIFF. The first one (check for dead activity) completes in 10ms,[4] the second one (Fig. 2(a)), in 20ms, the third one (Fig. 2(b)) in 420ms, and the last one (Fig. 3) in 80ms.

5 Some considerations

A most prominent feature and, in our opinion, a major advantage of the approach we present, with respect to other approaches to verification in the same application domains, is the language, as we have discussed earlier. It is declarative and it accommodates explicit time and dense domains. A software engineer can specify the system using a compact, intuitive graphical language such as DecSerFlow, then the specification is mapped automatically on to a SCIFF program. Using g-SCIFF, It is possible to verify the specification's properties. Using SCIFF it is possible to monitor and verify at run-time that the execution of an implemented system

[4] Experiments have been performed on a MacBook Intel CoreDuo 2 GHz machine.

complies with the specifications. This eliminates the problem of having to produce two sets of specifications (one for static and one for run-time verification) and of verifying that they are equivalent.

Let the language aside, the main difference with model checking is that queries are evaluated top-down, i.e., starting from a goal. Thus, no model needs to be generated, which eliminates a computationally expensive step. By going top-down, the verification algorithm only considers relevant portions of the search space, which can boost performance. On the downside, the performance strongly depends on the way SCIFF programs are written, with respect to the property we are verifying. Due to the left-most, depth-first search tree exploration strategy inherited from Prolog by SCIFF, the order of clauses influences the performance, and so does the ordering of atoms inside the clauses. However, this does not impact on soundness.

A major drawback of our approach is that it does not always guarantee termination, as opposed to unbounded model checkers, which typically guarantee that the verification algorithm terminates even when checking formulae producing models of infinite length, such as, for instance, $\Box(a \rightarrow \Diamond a)$. In general, g-SCIFF would not terminate in such a case - although it does terminate if it is used with finite domains, such as discrete time and limited time span. However, g-SCIFF implements a work-around to address this deficiency, similar to the one used in bounded model checking. In particular, g-SCIFF can be invoked in bounded mode, which restricts the number of actions generated by g-SCIFF. In this way, g-SCIFF does not guarantee completeness in the general case, but it is still able to say that, for example, a query fails with models consisting of at most N actions. Another technique implemented by SCIFF is iterative deepening, which can be used to address similar cases at the cost of a worse performance. However, we emphasize that we are proposing g-SCIFF for use in application domains in which interactions are expected to eventually terminate. A typical DecSerFlow model does not contain infinite loops—at least, not intentionally. In particular, all DecSerFlow relations individually produce loop-free SCIFF programs, and specifications such as the one we presented earlier do not have this problem. Thus, although a combination of DecSerFlow relations can indeed produce infinite loops, we can consider them to be uncommon cases which can be verified by using g-SCIFF with iterative deepening.

6 Experimental evaluation

These considerations led us to believe that in the application domain under consideration g-SCIFF could be a valid alternative to other verification methods. We run an extensive experimental evaluation to substantiate our guess. We followed on the results of an experimental investigation conducted by Kristin Rozer and Moshe Vardi on LTL satisfiability checking [26], by which it emerges that the symbolic approach is clearly superior to the explicit approach, and that NuSMV [12] is the best

performing model checker in the state of the art for the benchmarks they considered. We thus chose to run our benchmarks to compare g-SCIFF with NuSMV. Rozer and Vardi also describe a reduction of LTL satisfiability to model checking, which we adopted for experimenting with NuSMV:

1. Map activities on to boolean variables (1=execution);
2. Build a universal model $\mathcal{M}$, capable to generate all activity execution traces;
3. Build a "conjunction-formula" ϕ of the DecSerFlow specifications together with the query to verify (see Section 3), following the translation described in [30];
4. Model check $\neg\phi$ against $\mathcal{M}$: if the model checker finds a counterexample, ϕ is satisfiable and the counterexample is in fact an execution trace satisfying the DecSerFlow specifications and query.

It is worth noticing that explicit model checkers, such as SPIN, in our experiments could not handle in reasonable time a DecSerFlow chart such as the one we described earlier. We thus focus on the comparison with symbolic model checkers only. Unfortunately, the comparison could not cover all relevant aspects of the language, such as some temporal aspect, because neither NuSMV nor any other model checker cited in [26] offers all of the features offered by SCIFF. As a future work, we plan to compare the performance of g-SCIFF against that of other model checkers for MTL or TPTL [5, 4]. However, since existing MTL tools seem to use classical model checking and not symbolic model checking, our feeling is that g-SCIFF would largely outperform them on these instances.

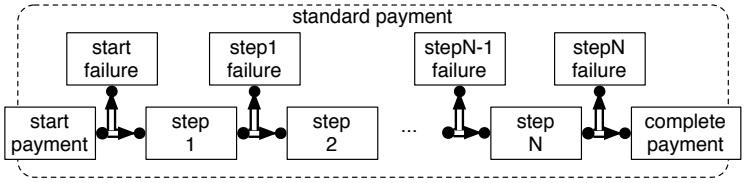

Figure 4. *Parametric extension to the model presented in Fig.1*

To the best of our knowledge, there are no benchmarks on the verification of declarative protocol specifications. We created our own, starting from the sample model introduced in Section 3, Fig. 1, and extending the standard payment activity as follows. Instead of a single activity, standard payment consists of a chain of N activities in alternate succession: start payment ⇻ step 1 ⇻ step 2 ⇻ ... ⇻ step N ⇻ complete payment, in which every two consecutive steps are linked by an alternate succession relation. Moreover, we model a possible failure at each of

these steps (start failure, step 1 failure, ...). This extension to the model is depicted in Fig. 4. Additionally, we add a K..* cardinality constraint on action payment failure, meaning that payment failure must occur at least K times. The new model is thus parametric on N and K. We complicated the model in such a way to stress g-SCIFF and emphasize its performance results in both favorable and unfavorable cases. We compared g-SCIFF with NuSMV on two sets of benchmarks:

1. the existential query presented in Section 3, Fig. 2(a);[5]
2. a variation of the above, without the 0 cardinality constraint on std payment.

Of the two benchmarks, the first one concerns verification of unsatisfiable specifications and the second one verification of satisfiable specifications. The latter requires producing an example demonstrating satisfiability, which generally increases the runtime. The input files are available on a Web site.[6] The runtime resulting from the benchmarks is reported in Appendix A. Fig. 5 shows the ratio NuSMV/g-SCIFF runtime, in Log scale. It turns out that g-SCIFF outperforms NuSMV in

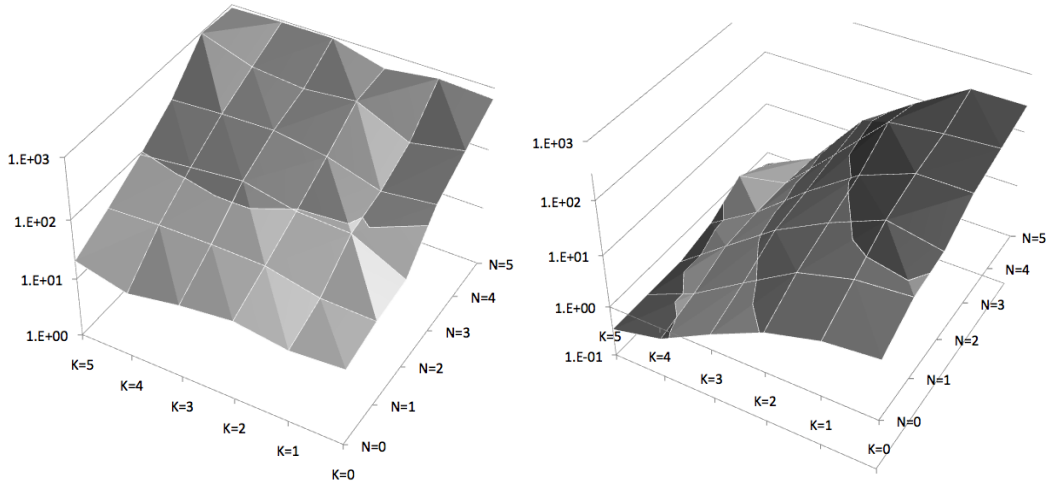

Figure 5. *Chart showing the ratio NuSMV/g-SCIFF runtime, in Log scale.*

most cases, up to several orders of magnitude: g-SCIFF does comparatively better as N increases, for a given K, whereas NuSMV improves with respect to g-SCIFF and eventually outperforms it, for a given N, as K increases. This is the case, because NuSMV's runtime is somehow proportional to the size of the LTL formula to be checked, whereas g-SCIFF's runtime heavily depends type of query it has to answer to, rather than on its length, and on the order of clauses and on the type of functors

[5] The 0 cardinality constraint is set on the start payment activity.

[6] See `http://www.lia.deis.unibo.it/research/climb/iclp08benchmarks.zip`.

used in the SCIFF program, rather than on the program size. This suggests that suitable heuristics that choose how to explore the search tree could help improve the g-SCIFF performance. This is subject for future research.

7 Related Work and Conclusion

We conclude by discussing other related approaches to verification, starting by those using ALP. Alessandra Russo et al. [27] exploit abduction for verification of declarative specifications expressed in terms of required reactions to events. They use the event calculus (EC) and include an explicit time structure. Global systems invariants, and in particular safety properties, are proved by refutation, and adopting a goal-driven approach similar to ours. The main difference concerns the underlying specification language: while Russo et al. rely on a general purpose ALP proof procedure which handles EC specifications and requirements, we adopt a language which directly captures the notion of occurred events and expectations, whose temporal relationships are mapped on CLP constraints. In this way, for the time structure we can rely on a variety of CLP domains (e.g., integers, reals, just to mention the two most relevant ones).

The idea of using CLP to perform model checking is also exploited by Giorgio Delzanno and Andreas Podelski [13], who propose to translate a procedural system specification into a CLP program. Safety and liveness properties, expressed as CTL formulas, are then checked by composing them with the translated program, and by calculating the least and the greatest fix-point sets respectively. Their methodology greatly resembles classic model checking, and applies to classic procedural models. Our approach instead targets a domain in which system specifications are mainly declarative, and a procedural model is not elicitable or it does not exist at all. Moreover, as in standard model checking, the solution discussed in [13] does not support any reasoning with time constraints, we instead exploit CLP constraints to support dense and discrete time reasoning.

In [15], Michael Fisher and Claire Dixon propose a clausal temporal resolution method to prove satisfiability of arbitrary propositional LTL formulae. The approach is two-fold: first, the LTL formula is translated in to a standard normal form (SNF), which preserves satisfiability; then a resolution method, encompassing classical as well as temporal resolution rules, is applied until either no further resolvents can be generated or *false* is derived, in which case the formula is unsatisfiable. From a theoretical point of view, clausal temporal resolution always terminates, while avoiding the state-explosion problem; however, the translation to SNF produces large formulas, and finding suitable candidates for applying a temporal resolution step makes the resolution procedure exponential in the size of the formula. Furthermore, in case of satisfiability no example is produced.

Bounded and unbounded model checking have also been addressed by means of

SAT technologies. Bounded model checking addresses the problem of verifying the validity of a formula within k transitions of a system. One represents with boolean formulas the initial state of the system $I(Y_0)$, the transition relation between two consecutive states $T(Y_i, Y_{i+1})$, and the (denied safety) property $F(Y_i)$. Then, the property is verified in the set of states $0 \dots k$ if and only if the formula [22]

$$I(Y_0) \wedge \left(\bigwedge_{i=0}^{k} T(Y_i, Y_{i+1}) \right) \wedge \left(\bigvee_{i=0}^{k} F(Y_i) \right)$$

is unsatisfiable. SAT-based unbounded model checking is based on similar formulas, but it also adds formulas that verify loop freedom (as in induction-based unbounded model checking [28]) or use SAT specific features (as in interpolant-based unbounded model checking [23]). In all cases, the transition function should be unfolded for a set of possible states, which makes the boolean formula quite large. Indeed, modern SAT solvers can handle millions of boolean variables, but even generating a large SAT can be costly.

A different research direction aims to extend propositional LTL to accommodate quantitative time constraints. For example, the timed requirement of Fig. 3, stating that a receipt is expected by 12 time units after executing accept advert, can be expressed in timed propositional temporal logic (TPTL, [4]) as follows: $\Box x.(\texttt{accept\_advert} \rightarrow \Diamond y.(y - x \leq 12 \wedge \texttt{send\_receipt}))$ and more compactly in metric temporal logic (MTL, [5]) as: $\Box(\texttt{accept\_advert} \rightarrow \Diamond_{\leq 12} \texttt{send\_receipt})$. Several tools have been developed to verify real-time systems w.r.t. timed temporal logics. One of them uses the TRIO language, a metric extension of first-order temporal logic, for modeling critical real-time systems. Unfortunately, the approaches proposed to model check TRIO specifications often give away with important features of the initial language, in the effort to obtain a decidable and tractable specification language. In a typical setting, the time domain is reduced to natural numbers, there is no quantification over time variables, no states, no events, and the language can range only on finite domains. Such a restricted language can then be efficiently translated onto a Promela alternating Büchi automaton using the Trio2Promela tool[8], or encoded as a SAT problem in Zot [25], to perform bounded SAT checking. Other tools rely instead on timed automata [7]. For example, Uppaal [21] is an integrated environment for modeling and verifiying real-time systems as networks of timed automata; it supports a limited set of temporal logic properties to perform reachibility tests. A timed automaton is a finite-state Büchi automaton extended with a set of real-valued (constrained) variables modeling clocks. As in standard explicit model checking, building and exploring (product of) timed automata is a very time and space-consuming task, made even more complex due to presence of such clocks.

We are aware that our method needs a more extensive theoretical and experimental evaluation, and a more thorough comparison with other related approaches.

This will be our next research direction. What we have achieved so far is the definition of a method and an ALP based proof-procedure, g-SCIFF, which draws from a high-level, graphical specification language such as DecSerFlow to perform verification tasks easily and efficiently. We envisage g-SCIFF to be a part of a framework that provides a suite of specification and verification tools for the designer of BPM service flows and interaction protocols. In this sense, an important feature of g-SCIFF is that it insists on the same SCIFF language used for interaction protocol specification and run-time monitoring and verification.

References

[1] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Contracting for dynamic location of web services: specification and reasoning with SCIFF. In *Proc. ESWC*, LNCS 4519, pp. 68-83. Springer-Verlag, 2007.

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Expressing and verifying contracts with abductive logic programming. *International Journal on Electronic Commerce*, 12(4):9–39, 2008.

[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 Logic*, 9(4), 2008.

[4] R. Alur and T. A. Henzinger. A really temporal logic. In *Proc. 30th IEEE FOCS*, pp. 164–169, Washington, DC, USA, 1989. IEEE Computer Society.

[5] R. Alur and T. A. Henzinger. Real-time logics: complexity and expressiveness. *Information and Computation*, 104:35–77, 1993.

[6] T. Andrews et al. Business Process Execution Language for Web Services (BPEL-WS), V1.1, May 2003. `http://www.ibm.com/developerworks/library/ws-bpel/`.

[7] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, LNCS 3098, pp. 87–124. Springer, 2003.

[8] D. Bianculli, A. Morzenti, M. Pradela, P. San Pietro, and P. Spoletini. Trio2Promela: a model checker for temporal metric specifications. In *Proc. 20th ICSE*, pp. 61–62, Washington, DC, USA, 2007. IEEE Computer Society.

[9] V. Bryl, P. Mello, M. Montali, P. Torroni, and N. Zannone. B-Tropos: Agent-oriented requirements engineering meets computational logic for declarative

business process modeling and verification. In K. Satoh and F. Sadri, eds., *Proc. CLIMA VIII*, LNCS. Springer-Verlag, 2008.

[10] H. Bürckert. A resolution principle for constrained logics. *Artificial Intelligence*, 66:235–271, 1994.

[11] H. Christiansen and V. Dahl. HYPROLOG: A new LP language with assumptions and abduction. In *Proc. ICLP*, LNCS 3668, pp. 159–173. Springer, 2005.

[12] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Int. J. Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[13] G. Delzanno and A. Podelski. Model checking in CLP. In *Proc. 5th TACAS*, LNCS 1579, pp. 223–239. Springer-Verlag, 1999.

[14] E.A. Emerson. Temporal and modal logic. In *Handbook of theoretical computer science, Part B*, pp. 995–1072. North-Holland, 1990.

[15] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Trans. Comput. Log.*, 2(1):12–56, 2001.

[16] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, Oct. 1998.

[17] G. Governatori and D. P. Hoang. A semantic web based architecture for e-contracts in defeasible logic. In *Proc. 1st RuleML*, LNCS 3791, pp. 145–159. Springer, 2005.

[18] J. Y. Halpern and M. Y. Vardi. Model checking vs. theorem proving: a manifesto. In *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pp. 151–176, San Diego, CA, 1991. Academic Press Prof., Inc.

[19] J. Jaffar and M. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[20] R. A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, July 1979.

[21] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Int. J. Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[22] J. Marques-Silva. Model checking with boolean satisfiability. *Journal of Algorithms*, 2008. doi:10.1016/j.jalgor.2008.02.007

[23] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pp. 1–13. Springer, 2003.

[24] M. Montali, M. Pesic, W. M. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographies. *Subm. ACM TWeb*, 2008.

[25] M. Pradella, C. A. Furia, A. Morzenti, and P. S. Pietro. Zot, a flexible bounded model and satisfiability checker. In *Proc. 15th FM*, LNCS, Springer, 2008.

[26] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Model Checking Software. Proc. 14th Int. SPIN W.*, LNCS 4595, pp. 149–167. Springer, 2007.

[27] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. In *Proc. 18th ICLP*, LNCS 2401, pp. 22–37. Springer, 2002.

[28] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. FMCAD*, LNCS 1954, pp. 108–125. Springer, 2000.

[29] M. P. Singh. Agent communication language: rethinking the principles. *IEEE Computer*, pages 40–47, Dec. 1998.

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

[31] F. Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, Aug. 2004.

[32] P. Yolum and M. Singh. Flexible protocol specification and execution: applying EC planning using commitments. In *Proc. AAMAS*, pp. 527–534. ACM Press, 2002.

A Experimental results

Results of the experiments described in Section 6 and summarized in Fig. 4.

$K \setminus N$	0	1	2	3	4	5
0	0.01/0.20	0.02/0.57	0.03/1.01	0.02/3.04	0.02/6.45	0.03/20.1
1	0.02/0.35	0.03/0.91	0.03/2.68	0.04/4.80	0.04/8.72	0.04/29.8
2	0.02/0.46	0.04/1.86	0.05/4.84	0.05/10.8	0.07/36.6	0.07/40.0
3	0.03/0.54	0.05/2.40	0.06/8.75	0.07/20.1	0.09/38.6	0.10/94.8
4	0.05/0.63	0.05/2.34	0.08/9.51	0.10/27.1	0.11/56.63	0.14/132
5	0.05/1.02	0.07/2.96	0.09/8.58	0.12/29.0	0.14/136	0.15/134

Table 1. *Results of first benchmark (SCIFF/NuSMV), in seconds*

$K \setminus N$	0	1	2	3	4	5
0	0.02/0.28	0.03/1.02	0.04/1.82	0.05/5.69	0.07/12.7	0.08/37.9
1	0.06/0.66	0.06/1.67	0.07/4.92	0.08/9.21	0.11/17.3	0.15/57.39
2	0.14/0.82	0.23/3.44	0.33/8.94	0.45/22.1	0.61/75.4	0.91/72.86
3	0.51/1.01	1.17/4.46	1.87/15.87	3.77/41.2	5.36/79.2	11.4/215
4	1.97/1.17	4.79/4.43	10.10/17.7	26.8/52.2	61.9/116	166/268
5	5.78/2.00	16.5/5.71	48.23/16.7	120/60.5	244/296	446/259

Table 2. *Results of second benchmark (SCIFF/NuSMV), in seconds*