# Verification from Declarative Specifications Using Logic Programming

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

[1] DEIS, University of Bologna
V.le Risorgimento 2, 40136 Bologna, Italy
{marco.montali | paolo.torroni | federico.chesani | paola.mello}@unibo.it

[2] ENDIF, University of Ferrara
V. Saragat 1, 44100 Ferrara, Italy
{marco.albeti | marco.gavanelli | evelina.lamma }@ing.unife.it

**Abstract**

In recent years, the declarative programming philosophy has had a visible impact on new emerging disciplines, such as heterogeneous multi-agent systems and flexible business processes. We address the problem of formal verification for systems specified using declarative languages, focusing in particular on the Business Process Management field. 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.

## 1 Introduction

Since its introduction, the declarative programming paradigm has been successfully adopted by IT researchers and practitioners. As in the case of logic programming, the separation of logic aspects from control aspects long advocated by Kowalski [16] 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 [23] or expectations [1] and make an extensive use of rules, business rules [18] and declarative Business Process (BP) specification languages such as ConDec [19]. In ConDec, business processes are specified following an open and declarative approach: rather than completely fix the control flow among activities, ConDec focuses on the (minimal) set of constraints that must be satisfied during the execution, providing an high degree of flexibility.

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 plays in these settings a fundamental role, a natural choice would be to model such systems

*Proceedings of the 15th International RCRA workshop (RCRA 2008):*
*Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*
*Udine, Italy, 12–13 December 2008*

using temporal logic specifications. In particular, ConDec models can be represented as a conjunction of (propositional) Linear Temporal Logic (LTL, [8]) formulae, each one formalizing a specific constraint [19]. By adopting this choice, the problem of consistency and properties verification can be cast as a satisfiability problem. This problem, in turn, is often reduced to model checking [21]. 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 declarative specifications such as the ones of ConDec, in which the system is not represented as a Kripke structure, but it is itself specified as an LTL formula; the translation of an LTL formula into an automaton is exponential in the size of the formula, and it becomes undecidable for variants of temporal logic with explicit time, such as Metric Temporal Logic (MTL) with dense time [2].

Unlike model checking, 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. Hence, we address the verification problem by using Abductive Logic Programming (ALP, [15]), and in particular the SCIFF framework [1]. SCIFF is an ALP rule-based language and family of proof procedures for the specification and verification of event-based systems. The language describes which events are expected (not) to occur when certain other events happen; it includes universally and existentially quantified variables, constraint logic programming (CLP) constraints and quantifier restrictions [3]. 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. Two different proof procedures can be then used to verify SCIFF specifications, ranging from run-time/a-posteriori compliance verification (SCIFF proof procedure) to static verification of properties (g-SCIFF proof procedure).

We focus on the last point, addressing the problem of ConDec static verification by (*i*) automatically translating ConDec models into the SCIFF framework (following the mapping proposed in [17]) and (*ii*) using g-SCIFF for reasoning. Via g-SCIFF, we can carry out a goal-directed verification task, without having to generate an intermediate format (as in model checking, where the formula specifying the system must be translated into an automaton). In this setting, abduction is used to generate (simulate) partially specified execution traces which comply with the specification and entail the goal of interest. The experiments we run to assess the performance of g-SCIFF support our claims and motivate us to pursue this line of research.

The paper is organized as follows. In Section 2 we discuss the application domains, proposing some examples of specification and verification in the context of Business Process Management (BPM). Section 3 presents the SCIFF framework and our verification method based on g-SCIFF. Section 4 evaluates it experimentally, in relation with other verification techniques. Related work is described in Section 5. Finally, Section 6 discusses advan-

tages and limits of g-SCIFF and concludes the paper.

## 2 Declarative Business Processes

If we skim through recent BPM, Web Service choreography, and Multi-Agent System literature, we will find a strong push for declarativeness. In the BPM context, van der Aalst and Pesic recently proposed a declarative flow language (ConDec, [19]) to specify, enact, and monitor business processes. Their claim is that declarative languages fit better with complex, unpredictable processes, where a good balance between support and flexibility is of key importance. 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., if A is executed B cannot be executed and vice-versa. In procedural languages, such as Petri nets, it is difficult to specify the above process without introducing additional assumptions and choice points, which lead to pointlessly complicate the model. This constraint can instead be easily expressed via a simple declarative LTL expression: $\neg(\Diamond A \wedge \Diamond B)$. This is also true for 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 intended model without introducing additional constraints.

### 2.1 A ConDec Example

In this article, we focus on the BPM domain. We use ConDec [19] as a declarative process specification language. Fig. 1 shows the ConDec 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 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 sources and/or destinations indicate temporal relations activated/satisfied by either of the source/destination activities. 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 termi-

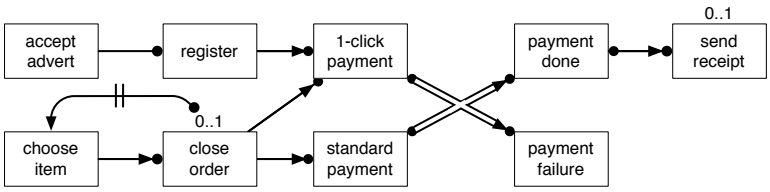


Figure 1: A ConDec model.

nation status of the payment action, and send receipt is a merchant action. The ConDec 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 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 activity executed), 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.

## 2.2 Static Verification of ConDec Models

Static verification helps to ensure safety and consistency of the model under study at design time. If the outcome of the verification process is "bad", a new design cycle can be triggered, to the aim of properly revising the model. In this respect, even if the verification process is not time-critical, it is anyway an important aspect: the development of "correct" models involves a constant interaction between static verification and the modeler.

Let us consider some examples of verification on the model. A first, simple type of verification is known as checking for dead activities [20]. We want to check whether a given activity, say send receipt, can be executed. To verify the query, we add a 1..* cardinality constraint on the activity. If the extended specification is unfeasible, it means that send receipt cannot 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

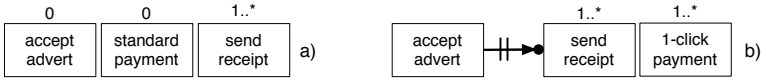


Figure 2: Two sample queries: (a) existential and (b) universal properties.

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 the query, 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))[1]. Such an extended specification is unsatisfiable: a verifier should return a negative answer.

Let us now consider another complex 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. Since the query is, in this case, universal, to verify we have 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 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, e.g., 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 ConDec notation, proposed in [17]. 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, the query depicted in Fig. 3 contains a $(0, 12)$ labelled arrow, expressing that $B$ must occur after $A$ and at most 12 time units after $A$ (maximum latency constraint on the sequence $A \dots B$). The query also contains a 0

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

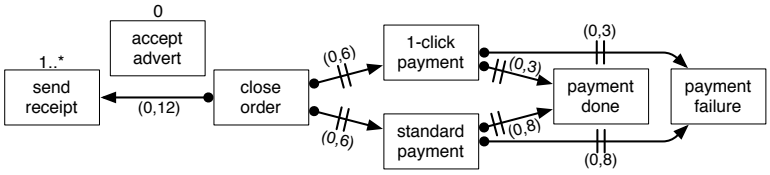


Figure 3: Sample query concerning verification of properties on models with explicit time.

cardinality constraint on accept advert (the customer does not accept ads). The intuition behind the whole 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.

# 3 The SCIFF Framework

SCIFF was initially proposed to specify and verify agent interaction protocols [1], but it has also been successfully applied in the context of service choreographies, electronic contracts and declarative business processes [17].

## 3.1 The SCIFF Language

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 (possibly containing variables). 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. Happened events and expectations explicitly contain a time variable, to represent when the event occurred/is expected (not) to occur. Event and time variables can be constrained by means of Prolog predicates or CLP constraints [14]; the latter are especially useful to specify orderings between events and quantitative time constraints (such as delays and deadlines). An IC is a forward $body \Rightarrow head$ 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. ICs are interpreted in a reactive manner; 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 [1], 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.

### 3.2 Static Verification Using g-SCIFF

The SCIFF framework includes two different proof procedures to perform verification. The SCIFF proof procedure checks the compliance of a narrative of events with the specification, by matching events with expectations during the execution (run-time monitoring) or a-posteriori. The g-SCIFF proof procedure is a "generative" extension of the SCIFF proof procedure whose purpose is to prove system properties at design time (static verification), or to generate counterexamples of properties that do not hold.

The proof procedures are implemented in SICStus 4 and are freely available[2]. Their 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, [10])[3]. The second important feature is their 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 discuss static verification, reporting the results obtained with the CLP($\mathcal{R}$) solver, which is based on the simplex algorithm, and features a complete propagation of linear constraints.

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 cardi-

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

[3] Other proof procedures [5] 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.

nality 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 [13]. g-SCIFF presents interesting features from both approaches. Like theorem proving, its performance is not heavily affected by domain cardinality, and it accommodates domains with infinite elements, such as dense time. Similarly to model checking, it works in a push-button style, thus offering a high level of automation.

In the style of [22], we do verification by abduction: in g-SCIFF, event occurrences are abduced as well as expectations, in order to model all the possible evolutions of the system being verified. In particular, the g-SCIFF proof procedure is a transition system which inherits all the transitions of the SCIFF proof procedure [1], adding a new transition called *fulfiller*. Fulfiller states that if an expectation $\mathbf{E}(p,t)$ is not fulfilled, an event $\mathbf{H}(p,t)$ is abduced. g-SCIFF uses *fulfiller* to generate narratives of events ("histories") starting from the specification (and the query of interest): abduction is used to simulate executions of the system which comply with the specification and entail the query. 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. *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 [3] 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. An existential 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 universal property $\mathcal{Q}$ can be negated (as in model checking), and then passed to g-SCIFF. If the g-SCIFF proof procedure succeeds in finding a history which satisfies the negated property, such a history is a counterexample against $\mathcal{Q}$. The examples shown in Section 2.1 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.

# 4 Experimental Evaluation

A ConDec chart is a good starting point to compare two verification methods: satisfiability checking LTL formulas via model checking, and g-SCIFF.

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

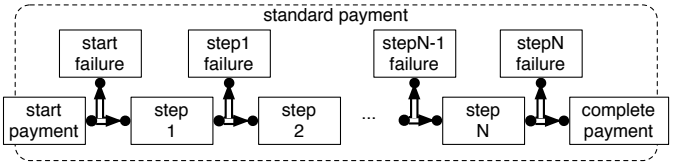


Figure 4: Parametric extension to the model presented in Fig.1

Indeed, the semantics of ConDec can be given both in terms of LTL formulae [19, 17] and of SCIFF programs [17]. By adopting LTL, each ConDec constraint is associated with a formula; the conjunction of all formulae ("conjunction formula") gives the semantics of the entire chart. In SCIFF the approach is similar: each ConDec constraint is mapped to a set of ICs, and the entire model is represented by the union of all ICs.

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:

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

We run an extensive experimental evaluation to compare g-SCIFF with model checking techniques. To the best of our knowledge, there are no benchmarks on the verification of declarative business process specifications. We created our own, starting from the sample model introduced in Section 2.1, 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 $\Rightarrow$ step 1 $\Rightarrow$ step 2 $\Rightarrow$ ... $\Rightarrow$ step N $\Rightarrow$ 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.

### 4.1 Verifying ConDec Models with g-SCIFF and Model Checking Techniques

To verify ConDec models with g-SCIFF, we adopted the following methodology. Given a ConDec specification $\mathcal{S}$ and a query (negated, if the query is universal) $\mathcal{Q}$: *(i)* build a SCIFF specification which formalizes $\mathcal{S}$, following the translation described in [17], and do the same with $\mathcal{Q}$; *(ii)* run g-SCIFF with the translation of $\mathcal{Q}$ as goal. If the query is entailed by $\mathcal{Q}$, then g-SCIFF generates an execution trace which complies with $\mathcal{S}$ and satisfies $\mathcal{Q}$.

In the LTL setting, the problem of static verification is cast as a satisfiability problem, which in turn can be reduced to model checking [21]: *(i)* map activities to boolean variables (1=execution); *(ii)* build a "conjunction-formula" $\phi$ of $\mathcal{S}$ and $\mathcal{Q}$, following the translation described in [19]; *(iii)* build a universal model $\mathcal{M}$, capable to generate all the activity execution traces; *(iv)* model check $\neg\phi$ against $\mathcal{M}$. If the model checker finds a counterexample, $\phi$ is satisfiable and the counterexample is in fact an execution trace satisfying both $\mathcal{S}$ and $\mathcal{Q}$.

In order to choose a suitable model checker, we followed on the results of an experimental investigation conducted by Rozier and Vardi on LTL satisfiability checking [21], by which it emerges that the symbolic approach is clearly superior to the explicit approach, and that NuSMV [6] is the best performing model checker for the benchmarks they considered. We thus chose to run our benchmarks to compare g-SCIFF with NuSMV[5].

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 [21] 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 [2]. 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.

### 4.2 Experimental Results

We compared g-SCIFF with NuSMV on two sets of benchmarks: *(i)* the existential query presented in Section 2.1, Fig. 2(a)[6]; *(ii)* 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[7]. The

[5]It is worth noticing that explicit model checkers, such as SPIN, in our experiments could not handle in reasonable time a ConDec chart such as the one we described earlier.

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

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

| $K \setminus N$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **First benchmark** | | | | | | |
| 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 |
| **Second benchmark** | | | | | | |
| 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 1: Results of the benchmarks (SCIFF/NuSMV), in seconds.

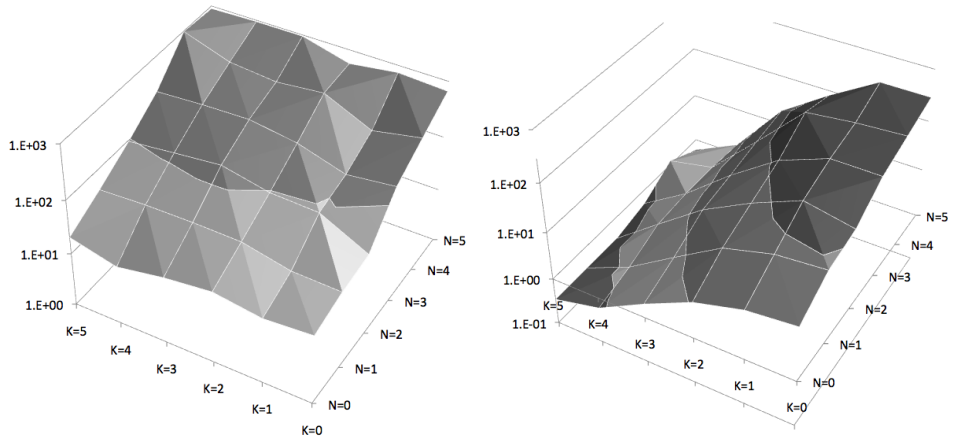


Figure 5: Charts showing the ratio NuSMV/g-SCIFF runtime, in Log scale.

runtime resulting from the benchmarks is reported in Table 4.2. Fig. 5 shows the ratio NuSMV/g-SCIFF runtime, in Log scale.

It turns out that g-SCIFF outperforms NuSMV in most cases, up to several orders of magnitude. This is especially true for the first benchmark, for which g-SCIFF is able to complete the verification task always in less than 0.15s, while NuSMV takes up to 136s. For the second benchmark, g-SCIFF does comparatively better as N increases, for a given K, whereas NuSMV improves w.r.t. 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 the runtime of g-SCIFF, which follows a "simulation by abduction" approach, heavily depends on the 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 used in the SCIFF program. 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.

## 5 Related Work

We discuss other related approaches to verification, starting by those using ALP. Alessandra Russo et al. [22] 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 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.

Another system aimed at proving properties of graphical specifications translated to logic programming formalisms is West2East [4], where interaction protocols modeled in Agent UML are translated to a Prolog program representing the corresponding finite state machine, whose properties can be verified exploiting the Prolog meta-programming facilities. However, the focus of that work is more on agent oriented software engineering, rather than verification: the system allows (conjunctions of) existential or universal queries about the exchanged messages or guard conditions, and it is not obvious how to express and verify more complex properties.

In [9], Fisher and 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 into 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.

Differently from the approach here presented, in other works LP and CLP have been exploited to implement model checking techniques. Of course, since they mimic model checking, they inherit the same drawbacks of classical model checkers when applied for the static verification of ConDec models.

For example, Delzanno and Podelski [7] propose to translate a procedural system specification into a CLP program. Safety and liveness properties, expressed in Computation Tree Logic, are checked by composing them with

the translated program, and by calculating the least and the greatest fix-point sets. In [11], Gupta and Pontelli model the observed system through an automaton, and convert it into CLP. As in our approach, they cannot handle infinite sequences without the intervention of the user.

## 6 Discussion and Conclusion

A most prominent feature and, in our opinion, a major advantage of the approach we present,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 ConDec, then the specification is mapped automatically to a SCIFF program. Using g-SCIFF, It is possible to verify the specification's properties. Using the SCIFF proof procedure it is possible to monitor and verify at run-time that the execution of an implemented system 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.

Apart from the language, the main difference with model checking is that queries are evaluated top-down, i.e., starting from a goal. 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 w.r.t. the property. 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 termination even when checking formulae producing models of infinite length, such as $\Box(a \rightarrow \Diamond a)$. 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 ter-

minate. A typical ConDec model does not contain infinite loops—at least, not intentionally. In particular, all ConDec 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 ConDec relations can indeed produce infinite loops, we can consider them to be uncommon cases which can be identified through a pre-processing phase and verified by using g-SCIFF with iterative deepening. A promising approach to deal with infinite computations during verification seems to be Coinductive Logic Programming [12], which extends the usual operational semantics of logic programming to allow reasoning over infinite and cyclic structures and properties. It might be, therefore, a useful approach to deal with models which lead to infinite g-SCIFF computations. This issue, together with a more extensive theoretical and experimental evaluation, will be our next research direction.

**Acknowledgments**

This work has been partially supported by the FIRB project *TOCAI.IT*.

# References

[1] M. Alberti et al. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logic*, 9(4):1–43, 2008.

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

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

[4] G. Casella and V. Mascardi. West2east: exploiting web service technologies to engineer agent-based software. *IJAOSE*, 1:396–434, 2007.

[5] H. Christiansen and V. Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In *Proceedings of ICLP 2005*, pages 159–173, 2005.

[6] A. Cimatti et al. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[7] G. Delzanno and A. Podelski. Model checking in CLP. In *Proocedings of TACAS 1999*, LNCS, pages 223–239. Springer Verlag.

[8] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B*, pages 995–1072. MIT Press, 1990.

[9] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Transactions on Computational Logic*, 2(1):12–56, 2001.

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

[11] G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *Proceedings of RTSS 1997*, pages 230–239. IEEE Computer Society, 1997.

[12] G. Gupta et al. Coinductive logic programming and its applications. In *Proceedings of ICLP 2007*, LNCS, pages 27–44, 2007.

[13] 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*, pages 151–176, 1991.

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

[15] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

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

[17] M. Montali et al. Declarative specification and verification of service choreographies. *Submitted to ACM Transaction on the Web*, 2008.

[18] G. Nalepa. Proposal of business process and rules modeling with the xtt method. In *Proceedings of SYNASC 2007*, pages 500–506, 2007.

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

[20] M. Pesic et al. Declare: Full support for loosely-structured processes. In *Proceedings of EDOC 2007*, pages 287–300. IEEE Computer Society, 2007.

[21] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Model Checking Software. Proceedings of the 14th International SPIN Workshop*, volume 4595 of *LNCS*, pages 149–167. Springer Verlag, 2007.

[22] A. Russo et al. An abductive approach for analysing event-based requirements specifications. In P. Stuckey, editor, *Proceedings of ICLP 2002*, volume 2401 of *LNCS*, pages 22–37. Springer Verlag, 2002.

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