Fundamenta Informaticae 102 (2010) 1–37
DOI 10.3233/FI-2010-395
IOS Press

Abductive Logic Programming as an Effective Technology for the Static Verification of Declarative Business Processes*

Marco Montali†, Paolo Torroni, Federico Chesani, Paola Mello
DEIS, University of Bologna – V.le Risorgimento 2, 40136 Bologna, Italy
{marco.montali,paolo.torroni,federico.chesani,paola.mello}@unibo.it

Marco Alberti
CENTRIA, Universidade Nova de Lisboa – Quinta da Torre, 2829-516 Caparica, Portugal
m.alberti@fct.unl.pt

Evelina Lamma
ENDIF, University of Ferrara – V. Saragat 1, 44100 Ferrara, Italy
evelina.lamma@unife.it

Abstract. We discuss the static verification of declarative Business Processes. We identify four desiderata about verifiers, and propose a concrete framework which satisfies them. The framework is based on the ConDec graphical notation for modeling Business Processes, and on Abductive Logic Programming technology for verification of properties. Empirical evidence shows that our verification method seems to perform and scale better, in most cases, than other state of the art techniques (model checkers, in particular). A detailed study of our framework's theoretical properties proves that our approach is sound and complete when applied to ConDec models that do not contain loops, and it is guaranteed to terminate when applied to models that contain loops.

Keywords: static verification, business process management, declarative business process modeling, abductive logic programming, model checking.

*This is a revised and extended version of: M. Montali, P. Torroni, M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello. **Verification From Declarative Specifications Using Logic Programming**. In M. G. de la Banda and E. Pontelli, eds., *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*. LNCS 5366, pp. 440–454, Springer-Verlag, 2008.

†Address for correspondence DEIS, University of Bologna – V.le Risorgimento 2, 40136 Bologna, Italy

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 [29] 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 or expectations [47] and make an extensive use of rules, business rules [37] and declarative Business Process (BP) specification languages such as ConDec [40]. In ConDec, BPs are specified following an open and declarative approach: rather than completely fixing the control flow among activities, ConDec focuses on the (minimal) set of constraints that must be satisfied during the execution, providing a high degree of flexibility.

Although declarative languages improve readability and modifiability, and help reducing programming errors, what makes systems trustworthy and reliable is formal verification. In this work, we focus on *static verification*, which aims at verifying the model during the design phase, *before* the execution. Static verification provides support for guaranteeing *a priori* that the model will behave, during the execution, in a consistent manner, enabling the early identification of errors and undesired situations which, if encountered at run-time, could be costly to repair or could even compromise the entire system. In static verification, the model to be verified is considered *per se*, without taking into account the external environment in which it will be embedded and executed[1].

In this paper, we focus on the static verification of declarative BPs specified using the ConDec notation. We identify interesting classes of properties that can be checked on these specifications, such as existential and universal properties, and then exploit the possibility of mapping ConDec to different underlying formal frameworks for concretely carrying out the verification process.

ConDec models can be formalized as a conjunction of (propositional) Linear Temporal Logic (LTL, [18]) formulae [40]. By adopting this choice, the issue of consistency and properties verification can be formalized as a satisfiability problem. This problem, in turn, is often reduced to model checking [44], enabling the possibility of exploiting state-of-the art model checkers, such as SPIN [26] and NuSMv [14], for satisfiability checking. However, it is well known that the construction of the input for model checking algorithms is computationally costly. 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 model checking becomes undecidable for variants of temporal logic with explicit time, such as Metric Temporal Logic (MTL) with dense time [6].

In this article, we rely on a different formalization of ConDec, based not on LTL, but on the mapping presented in [36]. We propose the adoption of the $\mathcal{S}$CIFF framework [4], based on Abductive Logic Programming (ALP, [28]), as an effective tool for accomplishing the static verification task. $\mathcal{S}$CIFF 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 [27]) 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.

[1] This issue belongs to what is commonly called *validation*.

It belongs to the computational logic framework, and is therefore associated to a clear declarative semantics as well as to different operational counterparts for reasoning upon the specifications. In particular, two different proof procedures can be used to verify $\mathcal{S}$CIFF specifications, spanning from run-time/*a posteriori* compliance verification ($\mathcal{S}$CIFF proof procedure) to static verification of properties (g-$\mathcal{S}$CIFF proof procedure). With g-$\mathcal{S}$CIFF, abduction is used to generate partially specified (counter-)examples of execution traces which comply with the specification and (do not) entail the property of interest; .

To demonstrate the feasibility of the approach, we show how g-$\mathcal{S}$CIFF is able to deal with both existential and universal properties. We then perform a quantitative evaluation studying how the performance of g-$\mathcal{S}$CIFF scales along different dimensions (such as the size of the model), and compares with that of explicit and symbolic model checkers.

After an empirical discussion about advantages and drawbacks of the two approaches, we sketch a method for dealing with termination issues of the verification process. The method consists in a pre-processing phase in which ConDec models are mapped to AND/OR graphs, followed by a loop detection phase. The outcome of this latter phase is then used to early identify inconsistent parts of a model and to tune the search strategy of g-$\mathcal{S}$CIFF accordingly.

The paper is organized as follows. In Section 2 we briefly sketch the ConDec notation, introducing a running example which will be used throughout the paper. Section 3 discusses the problem of static verification in the context of declarative BPs, with particular regard to the ConDec setting. Section 4 presents the $\mathcal{S}$CIFF framework and our verification method based on g-$\mathcal{S}$CIFF. Section 5 evaluates it experimentally, in relation with other verification techniques, pointing out advantages and limitations of our approach. Section 6 discusses formal properties of the framework, focusing in particular on termination issues, and Section 7 proposes a pre-processing method to overcome such termination issues. Related work and conclusions follow.

2. Declarative Business Process Modeling with ConDec

If we skim through recent Business Process Management (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, [40]) to specify, enact, and monitor Business Processes. They claim that the adoption of a declarative approach fits better with complex, unpredictable processes, where a good balance between support and flexibility must be found.

2.1. The ConDec Language

A ConDec model mainly consists of two parts: a set of activities, representing atomic units of work, and a set of relationships which constrain the way activities can be executed, and are therefore referred to as *constraints*. Constraints can be interpreted as policies/business rules, and may reflect different kinds of business knowledge: external regulations and norms, internal policies and best practices, business objectives, and so on. Differently from procedural specifications, in which activities can be inter-connected only by means of control-flow relationships (sequence patterns, mixed with constructs supporting the splitting/merging of control flows), the ConDec language provides a number of control-independent abstractions to constrain activities, alongside the more traditional ones. In ConDec it is possible to insert past-oriented constraints, as well as constraints that do not impose any ordering among activities. Furthermore, while procedural specifications are closed, i.e., all what is not explicitly modeled is forbidden,

ConDec models are open: activities can be freely executed, unless they are subject to constraints. This choice has two implications. First, a ConDec model accommodates many different possible executions, improving flexibility. Second, the language provides abstractions to explicitly capture not only what is mandatory, but also what is forbidden. In this way, the set of possible executions does not need to be expressed extensionally and models remain compact.

In [40], the authors show that the adoption of a declarative approach is fundamental for capturing this wider range of constraints. 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 (this is a typical example of a negative constraint, expressing a prohibition). 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 pointless complications of the model. This constraint can instead be easily expressed in ConDec, which represents it with a dedicated graphical relationship, which makes the power of logical formalisms accessible also to non IT-savvy. ConDec models do not impose a rigid scheduling of activities; intstead, they leave the users free to execute activities in a flexible way, but respecting at the same time the imposed constraints. An execution trace is *supported* by a ConDec model if and only if it complies with all its constraints. Finally, it is important to note that well-defined ConDec models support only *finite* execution traces, because it must always be guaranteed that a BP will eventually terminate.

Another fundamental feature of ConDec is that, thanks to its declarative nature, it can be easily mapped to different underlying logic-base frameworks. For example, the mutual exclusion between a and b can be captured via a simple declarative LTL expression: $\neg(\Diamond a \wedge \Diamond b)$. This is also true for computational logic-based rules. For example, in $\mathcal{S}$CIFF 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]. ConDec can therefore be seen as an intuitive language to model rigorous declarative specifications without directly manipulating logical formulae.

2.2. A ConDec Example

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 termination status of the payment action, and send

[2]The two rules state that if a **H**appens, then b is **E**xpected **N**ot to happen, and viceversa.

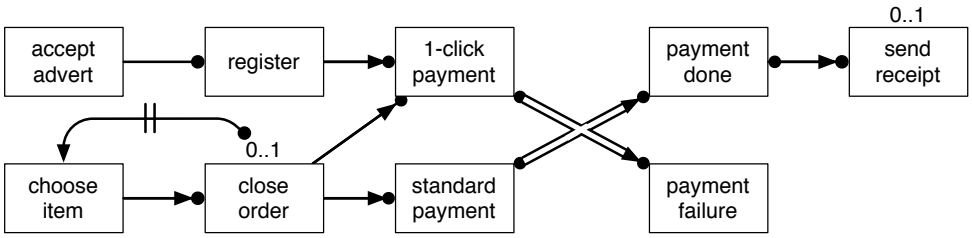


Figure 1: A ConDec model.

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.

3. Static Verification of Declarative Business Processes

In its most general meaning, static verification aims at verifying whether the model under development meets certain *properties*. Properties formally capture requirements that the user poses on the system under development. Requirements may differ from one another in nature and purpose, and can be classified along different axes. For example, requirements may formalize external regulations that must be ensured in every execution of the system, or they can capture hidden dependencies among activities, which should be entailed by the model. Here we focus on two important dimensions: *existential vs universal* and *general vs particular* properties.

3.1. Existential vs Universal Properties

When the modeler is interested in verifying whether the BP under development does not violate a mandatory external regulation, the intended meaning is that the property must be respected in any possible execution of the system. Conversely, the reachability of a certain situation is guaranteed if there exists at least an execution which reaches that situation. Generalizing, properties quantify over execution traces in two different ways:

1. A property is $\exists$-entailed by the model if at least one execution trace compliant with the model entails the property. If that is the case, then one of these execution traces can be interpreted as an example which proves the entailment.

2. A property is $\forall$-entailed by the model if all the possible execution traces compliant with the model entail the property. If that is *not* the case, an execution trace which follows the constraints of the model but breaks the property can be interpreted as a counter-example which disproves the entailment.

3.2. General vs Particular Properties

Properties can be *general*, if they describe basic/fundamental requirements which must be satisfied by an arbitrary interaction model, independently from its specific nature, or *particular*, if they target a specific model and domain. Examples of particular, domain-dependent properties are that a customer will obtain the ordered goods when executing an e-commerce protocol, or that a physician executing a clinical guideline will not administer a certain drug twice in the same day. Examples of general properties in a procedural setting, such as Petri Nets, are liveness and deadlock freedom, whereas examples of general properties in a declarative setting are *conflict freedom* and *absence of dead activities*.

Definition 3.1. (Conflict-free model[41])
A ConDec model is *conflict-free* iff it supports at least one possible execution trace. Otherwise, it is a *conflicting* model.

In the verification terminology, a ConDec model is conflict-free iff it $\exists$-entails the property "true". A conflicting model cannot be executed at all, and is therefore globally-inconsistent. It must be revised by removing a subset of its constraints.

Definition 3.2. (Dead Activity[41])
A ConDec activity is *dead* iff it can never be executed in any execution trace.

In the terminology above, a ConDec activity is dead iff the model it belongs to $\forall$-entails the *absence* of that activity.

Dead activities represent local inconsistencies in a model: there does not exist a compliant execution which contains them, and they are therefore over-constrained activities. Since ConDec constraints must be satisfied in all the possible execution of the systems, verifying if a certain activity is dead can be interpreted as checking if the original model and the model augmented with an absence constraint on that activity support exactly the same set of compliant traces.

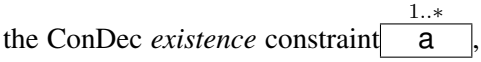
3.3. On the Safety-Liveness Classification

In model checking literature, properties are usually classified along the *safety* vs *liveness* dimension. We discuss why this classification is not suitable in the context of ConDec.

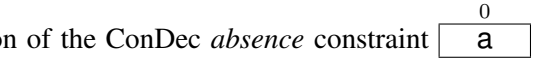
Intuitively, a safety property asserts that something bad will not happen during any execution of the system, whereas a liveness property states that something good will happen. However, as pointed out in [38], this intuitive meaning is misleading: it is often difficult to evaluate if a given property is a safety or liveness property. Formally, a property is a safety property iff every execution trace which violates

the property contains a finite prefix to which the violation can be referred to. In other words, there is no way to extend such a prefix with an infinite suffix, s.t. the property becomes satisfied: violation can be detected in a finite initial part of the trace, and its occurrence is irremediable. Conversely, a property is a liveness property if and only if, given an arbitrary partial execution trace of the system, it is possible to find an infinite suffix so that the resulting infinite execution trace satisfies the property.

Let us consider some typical properties in a temporal logic setting. According to these definitions:

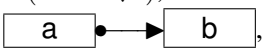
1. $\Diamond a$, which corresponds to the LTL formalization of the ConDec *existence* constraint 

, is a liveness property;

2. $\Box(\neg a)$, which corresponds to the LTL formalization of the ConDec *absence* constraint 

(invariant property), is a safety property;

3. $\Box a$ is a safety property;

4. $\Box(a \Rightarrow \Diamond b)$, which corresponds to the LTL formalization of the ConDec *response* constraint 

, is a liveness property;

5. A variation of the response property, stating for example that b must occur no later than 10 time units after a (promptness property[30]), is instead a safety property;

6. $\Box\Diamond a$ (fairness property), is a liveness property.

Let us now interpret these properties in the ConDec setting. Here, LTL propositional symbols represent activities, and a is $true$ at a certain time iff the activity a is *executed* at that time. However, it is important to remember that a ConDec model must always eventually terminate, i.e., ConDec execution traces are finite (see Section 2.1). In the light of this remark, it becomes apparent that only certain safety and liveness properties have a meaning in this context. Since there is always the underlying assumption that the process must terminate, we should imagine that the property to be verified is implicitly put in conjunction with a *termination* property [1][3]. Let us for example consider the two safety properties $\Box(\neg a)$ and $\Box a$. Despite their structural similarity, while the first is legitimate in the ConDec setting (it is in fact a ConDec constraint), the second one is not, because it states that activity a is must be continuously and infinitely executed. Similarly, while the liveness property $\Diamond a$ is acceptable and can be interpreted as "activity a must be performed at least once before the termination of the execution", fairness is never guaranteed by a ConDec model, because it is impossible to execute a certain activity infinitely often.

In conclusion, the implicit assumption that the execution of a ConDec model must eventually terminate (i.e., that execution traces are always finite) makes some typical safety/liveness properties senseless. Therefore, we will not distinguish, in the following, between safety and liveness properties, but we will instead rely on the existential vs universal dimension.

[3]Intuitively, the termination property states that a termination event e, incompatible with all other activities, must eventually occur, and then is executed infinitely often in the future.

3.4. Desiderata for Declarative Business Processes Verification Technologies

We argue that, in order to effectively help a BP developer in the process of ensuring the correctness and safety of the model and in the related debugging tasks, a static verification framework should satisfy four main desiderata:

Soundness and completeness if the verifier states that some state of affairs is reachable, we must be *sure* that the system being discussed can reach that state of affairs. In other words, our verifier should not compute wrong answers, i.e., it must be *sound*. On the other hand, it must be able to provide a positive answer if such a positive answer exists, i.e., it must be *complete*. If soundness/completeness are guaranteed only under certain assumptions, the verifier must be accompanied by techniques able to identify if these assumptions are respected by the model, taking specific countermeasures and alerting the user if that is not the case.

Generation of (counter-)examples When the verifier returns a negative answer, which shows that the model must be revised, the verifier must help isolating the relevant part of the model which violates the requirement. In this respect, it would be desirable that a verification framework does not only provide correct answers, but is also able to generate (counter-)examples showing in which cases the system supports executions which violate the requirement.

Push-button style After having produced the model and the property to be verified, the process of proving the property must not be performed manually by the user, nor should be carried out in a semi-automatic way, requiring a constant user interaction; it should operate, as much as possible, in an automatic way.

Scalability Since static verification is carried out at design time, the verifier's performance is not time-critical. However, in a typical usage scenario, static verification is repeatedly exploited by the user during the design phase, to constantly check the model under development and trigger new design cycles when it must be revised. In this respect, the design phase involves a constant interaction between the verifier and the modeler, and therefore scalability and good performances of the verification framework matter.

3.5. Verification of Properties on the Running Example

Let us now consider some examples of verification on the model presented in Section 2.2. We present four different queries which reflect the dimensions presented so far (general vs particular properties, existential vs universal verification).

Query 1. Is send receipt a dead activity?

This is a general property, verified in an universal way. A positive answer to Query 1 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 negative 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.

Query 2. Is it possible to complete a transaction under the hypotheses that the customer does not accept to receive ads, and the merchant does not offer standard payment?

This is a particular domain-dependent property that must be verified in an existential way. A verifier should return a negative answer. In fact, to complete a transaction the customer must pay, and the only accepted payment modality is 1-click payment; however, this payment modality is supported only if the customer has previously registered, which in turn requires to accept ads (contradicting customer's requirement).

Query 3. Is it true of all transactions that with 1-click payment a receipt is always sent *after* the customer has accepted the ads?

This is a domain-dependent property, which must be verified in an universal way. A verifier should discover that the property does not hold: since there is no temporally-ordered constraint associated with accept advert, accept advert does not have to be executed *before* send receipt in all valid models. The existence of an execution trace in which the customer chooses 1-click payment but accepts ads only after having received the receipt amounts to a counterexample against the universal query. More specifically, a verifier should produce, for example, the following 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 explicit *precedence* constraint from send receipt to accept advert.

Query 4. 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?

This query, that mast be verified in an existential way, contains explicit times, used to express minimum and maximum latency constraints on the activities execution. It turns out that the specification is unfeasible, because refusing ads rules out the 1-click payment path, and the standard payment path takes more than 12 time units. A verifier should therefore return failure.

4. Static Verification with $\mathcal{S}$CIFF

We briefly introduce the $\mathcal{S}$CIFF framework, its specification language and declarative semantics and the g-$\mathcal{S}$CIFF proof procedure, which specifically target the static verification task. We finally show how verification of existential and universal properties can be easily accommodated into the framework.

4.1. The $\mathcal{S}$CIFF Framework

$\mathcal{S}$CIFF was initially proposed to specify and verify agent interaction protocols [4], and it has been successfully applied also in the context of service choreographies, electronic contracts and declarative BPs [36]. $\mathcal{S}$CIFF specifications consist of an abductive logic program, i.e., a triplet $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ where $\mathcal{KB}$ 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. $\mathcal{S}$CIFF considers events as first class objects. Events can be, for example, the sending of a message, or the start of an action, and they are associated with a time point. Events which occur

during an execution of the system are identified by a special functor, **H**, and are described by an arbitrary term (possibly containing variables). $\mathcal{S}$CIFF uses ICs to model relations among events and expectations about events. Expectations are abducibles identified by functors **E** and **EN**. Therefore, for each $\mathcal{S}$CIFF specification, $\mathcal{A} \supseteq \{\mathbf{E}, \mathbf{EN}\}$. **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 [27]; 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 atoms, including happened events, whereas the $head$ is a disjunction of conjunctions of CLP constraints and atoms, including 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 γ and a triplet $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$, a set of abduced expectations must be *fulfilled* by the execution trace which characterizes a specific execution of the system. An execution trace is represented, as a set of completely ground happened events. The concept of fulfillment is formally captured by the $\mathcal{S}$CIFF declarative semantics [4], which intuitively states that $\mathcal{P}$, together with the abduced literals, must entail $\gamma \wedge \mathcal{IC}$, positive expectations must have a corresponding event in the execution trace, and negative expectations must not have a corresponding event. Note that in the remainder of the paper we will not consider an explicit goal but only integrity constraints[4].

The following definition of *compliance* refers to the declarative semantics of $\mathcal{S}$CIFF[4].

Definition 4.1. (Compliance)
An execution trace $\mathcal{T}$ is *compliant* with a $\mathcal{S}$CIFF specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ (written COMPLIANT($\mathcal{S}_{\mathcal{T}}$)) iff there exists an abducible set (i.e., a set of expectations) Δ s.t.

- Δ is an abductive explanation for $\mathcal{S}$ w.r.t. $\mathcal{T}$[4]; intuitively, an abductive explanation is an abducibles set $\subseteq \mathcal{A}$ that, together with $\mathcal{T}$ and $\mathcal{KB}$, entails $\mathcal{IC}$.

- Δ is **E**-consistent, i.e., it does not contain contradictory expectations:

$$\{\mathbf{E}(E, T), \mathbf{EN}(E, T)\} \not\subseteq \Delta$$

- Δ is fulfilled by $\mathcal{T}$, i.e., each positive expectation has a corresponding matching event in $\mathcal{T}$, and negative expectations do not have a corresponding event in $\mathcal{T}$:

$$\mathbf{E}(E, T) \in \Delta \Rightarrow \mathbf{H}(E, T) \in \mathcal{T}$$
$$\mathbf{EN}(E, T) \in \Delta \Rightarrow \mathbf{H}(E, T) \notin \mathcal{T}$$

[4]If we are not interested in the computed answer substitution, this choice does not lead to the loss of generality: the goal γ can be simply transformed into the IC $true \rightarrow \gamma$ and joined with the other integrity constraints.

As proven in [34], in the context of ConDec only a sub-set of the $\mathcal{S}$CIFF language is effectively used to represent models and their properties. This sub-set meets an important property called *compositionality* w.r.t. compliance, which is reported in the following.

Definition 4.2. (Compositionality (w.r.t. compliance))
A $\mathcal{S}$CIFF specification $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ is *compositional* iff $\forall \mathcal{T}, \forall \mathcal{IC}_1$ and $\forall \mathcal{IC}_2$ s.t. $\mathcal{IC} = \mathcal{IC}_1 \cup \mathcal{IC}_2$:

$$\begin{aligned} &\textsc{compliant}\left(\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC}_1 \rangle_{\mathcal{T}}\right) \\ \wedge &\textsc{compliant}\left(\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC}_2 \rangle_{\mathcal{T}}\right) \Leftrightarrow \textsc{compliant}\left(\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle_{\mathcal{T}}\right) \end{aligned}$$

4.2. The g-$\mathcal{S}$CIFF Proof Procedure

The $\mathcal{S}$CIFF framework includes two different proof procedures that perform verification. The $\mathcal{S}$CIFF proof procedure checks the compliance of a narrative of events with the specification, by matching events with expectations during or after the execution. The g-$\mathcal{S}$CIFF proof procedure is a "generative" extension of the $\mathcal{S}$CIFF proof procedure targeting static verification tasks. Its purpose is to prove system properties at design time, or to generate counterexamples of properties that do not hold.

The proof procedures are implemented in SICStus 4 and its Constraint Handling Rules library (CHR, [20])[5], and they are freely available[6]. The implementation relies on constraint solvers, and it can use both 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. Moreover, some domains require discrete time, other dense (continuous) time. 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.

In the style of [45], g-$\mathcal{S}$CIFF does verification by abduction: event occurrences are abduced as well as expectations, in order to simulate the possible evolutions of the system being verified. In particular, the g-$\mathcal{S}$CIFF proof procedure is a transition system which inherits all the transitions of the $\mathcal{S}$CIFF proof procedure [4], adding a new transition called *fulfiller*. Such a transition states that if an expectation $\mathbf{E}(p, t)$ is not fulfilled, an event $\mathbf{H}(p, t)$ is abduced. g-$\mathcal{S}$CIFF uses *fulfiller* to generate execution traces by abduction, starting from the specification and 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. $\mathcal{S}$CIFF and g-$\mathcal{S}$CIFF also exploit an implementation of reified unification (a solver on equality/disequality of terms) which takes into consideration quantifier restrictions [10] and variable quantification. Traces are thus generated intensionally, and hypothetical events can contain variables, possibly subject to CLP constraints.

Given a $\mathcal{S}$CIFF specification $\mathcal{S}$ and a goal γ, when g-$\mathcal{S}$CIFF is able to find a successful derivation for γ starting from $\mathcal{S}$ we write $\mathcal{S} \vdash_{\Delta}^{g-SCIFF} \gamma$, where Δ is the computed abductive answer. When $\gamma \equiv true$, g-$\mathcal{S}$CIFF checks if a specification is conflict-free. It is worth noting that Δ contains the abduced expectations as well as the intensional happened events generated by applying the *Fulfiller* transition on

[5] Other proof procedures [13] 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.
[6] See `http://lia.deis.unibo.it/sciff/`

these expectations. The following definition gives an explicit characterization to the generated happened events, i.e., to the intentional simulated trace.

Definition 4.3. (Generated trace)
Given a $\mathcal{S}$CIFF specification $\mathcal{S}$, a goal γ and an abductive answer Δ s.t. $\mathcal{S} \vdash_{\Delta}^{g-SCIFF} \gamma$, the generated trace w.r.t. Δ (written $\mathcal{T}_{gen}(\Delta)$) is the set of abduced happened events:

$$\mathcal{T}_{gen}(\Delta) \equiv \{\mathbf{H}(E,T) \mid \mathbf{H}(E,T) \in \Delta\}$$

Finally, we briefly recall soundness and (weak) completeness results (as stated before, we abstract away from the goal, which can be incorporated in a dedicated IC), using the terminology introduced so far. At the moment, we do not discuss the *acyclicity* property. In Section 6 we will introduce the syntactic restrictions needed by g-$\mathcal{S}$CIFF in order to meet this property, and we will show how these restrictions impact on the static verification of ConDec models.

Theorem 4.1. (Soundness of g-$\mathcal{S}$CIFF)
Given an acyclic $\mathcal{S}$CIFF specification $\mathcal{S}$:

$$\forall\ \Delta \text{ s.t. } \mathcal{S} \vdash_{\Delta}^{g-SCIFF} true \Rightarrow \text{COMPLIANT}\left(\mathcal{S}_{\mathcal{T}(\Delta)}\right)$$

Proof:
See [12, 4]. □

In the "generative" context of g-$\mathcal{S}$CIFF, completeness would state that, given a $\mathcal{S}$CIFF specification, g-$\mathcal{S}$CIFF is able to generate *all* the possible execution traces which are compliant with the specification. However, it does not meet such a completeness property: since its generative approach is driven by the ICs of the specification, only a sub-set of these traces will be found.

Nevertheless, it is guaranteed that, although g-$\mathcal{S}$CIFF is not able to generate all the (possibly infinite) execution traces compliant with a specification, it has however the capability to generate a compliant sub-set (by Theorem 4.1) of each of them. This forma property is referred to as *weak completeness*.

Theorem 4.2. (Weak Completeness of g-$\mathcal{S}$CIFF)
Given an acyclic $\mathcal{S}$CIFF specification $\mathcal{S}$:

$$\forall\ \mathcal{T}\ \text{COMPLIANT}(\mathcal{S}_{\mathcal{T}}) \Rightarrow \exists\ \Delta\ \mathcal{S} \vdash_{\Delta}^{g-SCIFF} true \wedge \mathcal{T}_{gen}(\Delta) \subseteq \mathcal{T}$$

Proof:
See [34]. □

By combining soundness and weak completeness it is guaranteed that *at least one execution trace compliant with a $\mathcal{S}$CIFF specification exists iff g-$\mathcal{S}$CIFF has a successful derivation, and therefore g-$\mathcal{S}$CIFF deals with the problem of checking conflict freedom (see Definition 3.1) in a correct way*.

Corollary 4.1. (Conflict-freedom checking with g-$\mathcal{S}$CIFF)
A $\mathcal{S}$CIFF specification $\mathcal{S}$ has at least one compliant execution trace iff g-$\mathcal{S}$CIFF has a successful derivation:

$\mathcal{S} \vdash_{\Delta}^{g-SCIFF} true \Leftrightarrow \exists \mathcal{T}\, \text{COMPLIANT}(\mathcal{S}_{\mathcal{T}})$

Proof:
Straightforward from the soundness and completeness results (Theorems 4.1 and 4.2). □

4.3. Verification of Existential and Universal Properties in g-$\mathcal{S}$CIFF

In the $\mathcal{S}$CIFF framework, properties are represented with the same language used for specifying models, i.e. by way of ICs. Verification of existential and universal properties is then achieved by suitably quantifying on execution traces and combining the concept of compliance with the specification and with the property. We first formalize these two concepts, then we show a reduction of the verification problem to conflict-freedom, which is suitably tackled by g-$\mathcal{S}$CIFF as shown in Corollary 4.1.

Definition 4.4. ($\exists$-entailment)
A $\mathcal{S}$CIFF specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ $\exists$-entails a set of ICs Ψ ($\mathcal{S} \models_{\exists} \Psi$) iff:

$$\exists\, \mathcal{T} \text{ COMPLIANT}\,(\mathcal{S}_{\mathcal{T}}) \wedge \text{COMPLIANT}\,(\langle \mathcal{KB}, \mathcal{A}, \Psi \rangle_{\mathcal{T}})$$

$\mathcal{S}$ $\exists$-violates Ψ ($\mathcal{S} \not\models_{\exists} \Psi$) iff:

$$\neg\exists\, \mathcal{T} \text{ COMPLIANT}\,(\mathcal{S}_{\mathcal{T}}) \wedge \text{COMPLIANT}\,(\langle \mathcal{KB}, \mathcal{A}, \Psi \rangle_{\mathcal{T}})$$

Definition 4.5. ($\forall$-entailment)
A $\mathcal{S}$CIFF specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ $\forall$-entails a set of ICs Ψ ($\mathcal{S} \models_{\forall} \Psi$) iff:

$$\forall\, \mathcal{T} \text{ COMPLIANT}\,(\mathcal{S}_{\mathcal{T}}) \Rightarrow \text{COMPLIANT}\,(\langle \mathcal{KB}, \mathcal{A}, \Psi \rangle_{\mathcal{T}})$$

$\mathcal{S}$ $\forall$-violates Ψ ($\mathcal{S} \not\models_{\forall} \Psi$) iff:

$$\exists\, \mathcal{T} \text{ COMPLIANT}\,(\mathcal{S}_{\mathcal{T}}) \wedge \neg\text{COMPLIANT}\,(\langle \mathcal{KB}, \mathcal{A}, \Psi \rangle_{\mathcal{T}})$$

As described in Section 4.2, the g-$\mathcal{S}$CIFF proof procedure works by generating intensional execution traces which comply with a given specification. Therefore, two questions concerning $\exists$ and $\forall$-entailment arise: how to combine the specification of the system and the property and obtain the final specification used by g-$\mathcal{S}$CIFF for verification? Is it possible to reduce $\forall$-entailment to $\exists$-entailment? We care about the last point because there are potentially infinite execution traces compliant with a specification, and it is therefore impossible to address universal verification directly. The same approach is used in model checking-based verification.

Theorem 4.3. ($\exists$-entailment with g-$\mathcal{S}$CIFF)
Given a $\mathcal{S}$CIFF specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ and a set Ψ of integrity constraints,

$$\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \cup \Psi \rangle \vdash_{\Delta}^{g-SCIFF} true \Leftrightarrow \mathcal{S} \models_{\exists} \Psi$$

Proof:
Let us introduce the following symbols: $\mathcal{S}^{\Psi} \equiv \langle \mathcal{KB}, \mathcal{A}, \Psi \rangle$, $\mathcal{S}^{\cup} \equiv \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \cup \Psi \rangle$. We have:

$$\mathcal{S}^{\cup} \vdash_{\Delta}^{g-SCIFF} true \Leftrightarrow \quad \text{(Cor. 4.1)}$$

$$\exists \mathcal{T} \text{COMPLIANT}(\mathcal{S}_{\mathcal{T}}^{\cup}) \Leftrightarrow \quad \text{(Def. 4.2)}$$

$$\exists \mathcal{T} \text{COMPLIANT}(\mathcal{S}_{\mathcal{T}}) \wedge \text{COMPLIANT}(\mathcal{S}_{\mathcal{T}}^{\Psi}) \Leftrightarrow \mathcal{S} \models_{\exists} \Psi \quad \text{(Def. 4.4)}$$

□

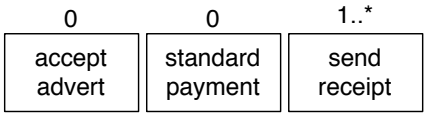


Figure 2: ConDec representation of Query 2. The 1..* cardinality constraint is used to state that the transaction must be completed, whereas the two 0 cardinality constraints are used to express that the customer refuses ads and that standard payment is not available.

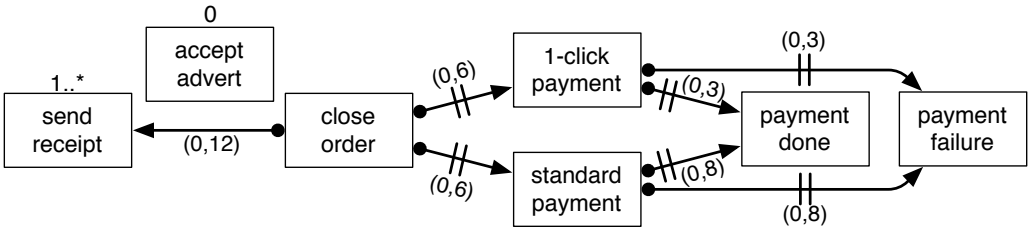


Figure 3: Representation of Query 4 in extended ConDec.

If we imagine that $\mathcal{S}$ and Ψ respectively represent the formalization of a ConDec specification and of an existential property, which can be also specified using the ConDec notation, Theorem 4.3 states that verification can be accomplished by joining the ICs of the two specification, and checking if the resulting global model is conflict-free. Moreover, the $\mathcal{T}_{gen}$ computed by g-$\mathcal{S}$CIFF is an example of execution trace which complies with the specification as well as with the property.

For example, Query 2 can be represented in ConDec as shown in Figure 2.

Query 4 can be represented instead using an extended ConDec notation (see Figure 3), proposed in [36]. 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 $A \dots B$).

We now show how universal verification can be reduced to existential verification. The reduction uses complementary Integrity Constraints, which are defined as follows.

Definition 4.6. (Complementary integrity constraints)
Given a knowledge base $\mathcal{KB}$ and a set of abducible predicates $\mathcal{A}$, two integrity constraints IC^1 and IC^2 are complementary w.r.t. $\mathcal{KB}$ and $\mathcal{A}$(written $IC_1 = \overline{IC_2}^{\mathcal{KB},\mathcal{A}}$, $IC_2 = \overline{IC_1}^{\mathcal{KB},\mathcal{A}}$) iff:

$$\forall \mathcal{T} \text{ COMPLIANT}\left(\langle \mathcal{KB}, \mathcal{A}, \{IC_1\}\rangle_{\mathcal{T}}\right) \Leftrightarrow \neg \text{COMPLIANT}\left(\langle \mathcal{KB}, \mathcal{A}, \{IC_2\}\rangle_{\mathcal{T}}\right)$$

We will use the simplified notation $IC_1 = \overline{IC_2}$, $IC_2 = \overline{IC_1}$ where the context (i.e., $\mathcal{KB}$ and $\mathcal{A}$) is apparent.

Given an IC, many different ICs complementary to it exists. In the general case, the synthesis of a complementary integrity constraint is a task that must be accomplished manually, but a simple syntactic

complementation procedure can be easily defined for the sub-set of the $\mathcal{S}$CIFF language used to formalize ConDec models[7].

The following Theorem states that an IC IC_1 is $\forall$-entailed by a $\mathcal{S}$CIFF specification iff the specification does not $\exists$-entail an IC IC_2 which is complementary to IC_1. If this latter complementary IC is $\exists$-entailed, the execution trace generated by g-$\mathcal{S}$CIFF amounts to a counter-example, which demonstrates that the original property is not guaranteed in every possible execution. In the case of a complex property, composed by more ICs, it is sufficient to prove that at least one complemented integrity constraint is not $\exists$-entailed to disconfirm the property.

Theorem 4.4. (Reduction of $\forall$-entailment to $\exists$-entailment)
Given a $\mathcal{S}$CIFF specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ and a property $\Psi \equiv \bigcup_{i=1}^{n} IC_i$

$$\forall\, i = 1, \ldots, n \; \mathcal{S} \not\models_{\exists} \overline{IC_i} \Rightarrow \mathcal{S} \models_{\forall} \Psi$$

Proof:
Let us introduce the following symbols: $\mathcal{S}^{\Psi} = \langle \mathcal{KB}, \mathcal{A}, \Psi \rangle$, $\mathcal{S}_i^{\Psi} = \langle \mathcal{KB}, \mathcal{A}, \{IC_i\} \rangle$, $\overline{\mathcal{S}_i^{\Psi}} = \langle \mathcal{KB}, \mathcal{A}, \{\overline{IC_i}\} \rangle$.

For each $i = 1, \ldots, n$, we have:

$$\begin{aligned}
\mathcal{S} \not\models_{\exists} \overline{IC_i} \Leftrightarrow & \qquad \text{(Def. 4.4)} \\
\neg\exists \mathcal{T}\, \text{COMPLIANT}\,(\mathcal{S}_{\mathcal{T}}) \wedge \text{COMPLIANT}\left(\overline{\mathcal{S}_i^{\Psi}}_{\mathcal{T}}\right) \Leftrightarrow & \qquad \text{(Def. 4.6)} \\
\forall \mathcal{T} \neg \left(\text{COMPLIANT}\,(\mathcal{S}_{\mathcal{T}}) \wedge \neg \text{COMPLIANT}\left(\mathcal{S}_i^{\Psi}{}_{\mathcal{T}}\right)\right) \Leftrightarrow & \\
\forall \mathcal{T}\, \text{COMPLIANT}\,(\mathcal{S}_{\mathcal{T}}) \Rightarrow \text{COMPLIANT}\left(\mathcal{S}_i^{\Psi}{}_{\mathcal{T}}\right) \Leftrightarrow & \qquad \text{(Def. 4.5)} \\
\mathcal{S} \models_{\forall} IC_i \Leftrightarrow \mathcal{S} \models_{\forall} \bigcup_{i=1}^{n} IC_i & \qquad \text{(Def. 4.2)}
\end{aligned}$$

□

When g-$\mathcal{S}$CIFF succesfully computes a Δ showing that there exists a k for which $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \cup \overline{IC_k} \rangle$ is conflict-free, $\mathcal{T}_{gen}(\Delta)$ can be considered as a counter-example which amounts to a proof that Ψ is not $\forall$-entailed. Therefore, verification can be carried out by considering each single IC belonging to the property separately. The IC is first complemented, and the complemented version is then checked for $\exists$-entailment. The first complemented IC (if there exists one) which is $\exists$-entailed by the specification proves that the original property is not $\forall$-entailed, and thus the verification procedure terminates without needing to check any other IC.

Query 1 can be verified in an existential way by complementing the concept of dead activity, i.e., by looking for an execution trace in which send receipt is executed at least once [8]. If such an execution trace exists, then the activity is not dead. Similarly, the following IC represents a possible way to complement the requirement specified in Query 3:

$$\begin{aligned}
true \rightarrow & \mathbf{E}(1 click_payment, T_p) \\
& \wedge \mathbf{E}(send_receipt, T_r) \wedge \mathbf{E}(accept_advert, T_a) \wedge T_a > T_r.
\end{aligned}$$

[7] Such a complementation procedure is beyond the scope of this paper.
[8] The ICs $true \rightarrow \mathbf{EN}(a, T)$ and $true \rightarrow \mathbf{E}(a, T)$ are complementary to each other.

Intuitively, the IC states that 1-click payment must be chosen, and that advertising is expected to be accepted after the receipt is sent.

All the described queries are correctly handled by g-$\mathcal{S}$CIFF. Query 1 completes in 10ms[9], Query 2 in 20ms, Query 3 in 420ms, and Query 4 in 80ms.

5. Experimental Evaluation

We ran an extensive experimental evaluation to compare g-$\mathcal{S}$CIFF with model checking techniques, in the context of static verification of declarative BPs specified in ConDec. To the best of our knowledge, there are no benchmarks on the verification of declarative BP specifications (yet). We created our own, starting from the sample model introduced in Section 2.2, Figure 1.

We first briefly recall how ConDec models can be represented by means of $\mathcal{S}$CIFF specifications and Linear Temporal Logic (LTL) formulae; then, we describe the verification procedures for handling $\exists$ and $\forall$-entailment in the two settings, respectively exploiting g-$\mathcal{S}$CIFF and model checking techniques. Finally, we introduce the benchmarks and present the obtained results.

5.1. Mapping ConDec Models to $\mathcal{S}$CIFF and Linear Temporal Logic

The semantics of ConDec can be given both in terms of Linear Temporal Logic formulae [40, 36, 39] and of $\mathcal{S}$CIFF programs [36, 34]. 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 $\mathcal{S}$CIFF the approach is similar: each ConDec constraint is mapped to an IC, and the entire model is represented by the set of all ICs.

For example, the relation between accept advert and register in Figure 1 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 define two functions which incapsulate the translation of a ConDec model to a $\mathcal{S}$CIFF specification and an LTL formula respectively.

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

function: g-$\mathcal{S}$CIFF($\mathcal{S}$*CIFF specification* $\mathcal{S}$*, goal* γ)
returns : $true$, together with a sample execution trace $\mathcal{T}_{gen}$ if there is a g-$\mathcal{S}$CIFF successful derivation for γ starting from $\mathcal{S}$, $false$ otherwise

1.1 **begin**
1.2 **if** $\mathcal{S} \vdash_{\Delta}^{g-SCIFF} \gamma$ **then**
1.3 $\mathcal{T}_{gen} \leftarrow \{\mathbf{H}(E,T) \in \Delta\}$;
1.4 **return** *[true,* $\mathcal{T}_{gen}$*]*;
1.5 **end**
1.6 **return** *[false, -]*;
1.7 **end**

Function g-$\mathcal{S}$CIFF($\mathcal{S}$,γ)

function: $\exists$-entailment$_{g-\mathcal{S}\text{CIFF}}$(*ConDec model* $\mathcal{M}$*,* $\mathcal{S}$*CIFF property* Ψ)
returns : $true$, together with a sample execution trace if sciff_transform($\mathcal{M}$) $\models_{\exists} \Psi$, $false$ otherwise

2.1 **begin**
2.2 $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle \leftarrow$ sciff_transform($\mathcal{M}$);
2.3 $\mathcal{S} \leftarrow \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \cup \Psi \rangle$;
2.4 **return** g-$\mathcal{S}$CIFF($\mathcal{S}$, *true*);
2.5 **end**

Function $\exists$-entailment$_{g-\mathcal{S}\text{CIFF}}$($\mathcal{M}$,$\Psi$)

Definition 5.1. ($\mathcal{S}$CIFF transformation)
sciff_transform is a function which transforms an arbitrary ConDec model to a corresponding $\mathcal{S}$CIFF specification, following the mapping presented in [34].

Definition 5.2. (LTL transformation)
LTL_transform is a function which transforms an arbitrary ConDec model to a corresponding LTL formula, following the mapping presented in [39].

Both the transformations require an amount of time which linearly depends on the "size" of the ConDec model, i.e., on the number of constraints it contains (indeed, each ConDec constraints is transformed to an LTL formula or $\mathcal{S}$CIFF IC). Furthermore, it is worth noting that, when the same model must be checked against many different properties, it can be translated only once and then put in conjunction/joined with the translation of each property.

5.2. Verification by g-$\mathcal{S}$CIFF

Functions $\exists$-entailment$_{g-\mathcal{S}\text{CIFF}}$($\mathcal{M}$,$\Psi$) and $\forall$-entailment$_{g-\mathcal{S}\text{CIFF}}$($\mathcal{M}$,$\Psi$) below describe a procedure for verifying $\exists$ and $\forall$-entailment of a property Ψ by the ConDec model $\mathcal{M}$, exploiting Theorems 4.3 and 4.4 for verification. The two listings make use of a function g-$\mathcal{S}$CIFF($\mathcal{S}$,γ), which encapsulates the g-$\mathcal{S}$CIFF computation.

function: $\forall$-entailment$_{g-\mathcal{S}\texttt{CIFF}}$(*ConDec model* $\mathcal{M}$*,* $\mathcal{SCIFF}$ *property* Ψ)
returns : $true$ if sciff_transform($\mathcal{M}$) $\models_\forall \Psi$, $false$, together with a counter-example execution trace $\mathcal{T}$, otherwise

```
3.1  begin
3.2    foreach IC_i ∈ Ψ do
3.3      Ψ_neg ← complement(IC_i);
3.4      [Success, T] ← ∃-entailment_{g-SCIFF}(M, Ψ_neg);
3.5      if Success then
3.6        return [false, T];
3.7      end
3.8    end
3.9    return [true, -];
3.10 end
```

Function $\forall$-entailment$_{g-\mathcal{S}\texttt{CIFF}}(\mathcal{M},\Psi)$

5.3. Verification by Model Checking

Static verification of LTL specifications can be addressed by means of *satisfiability checking*. An LTL formula is satisfiabile iff it admits at least one model. As already pointed out, when ConDec models are mapped to LTL formulae, propositional symbols represent activities, and, as a consequence, LTL models represent execution traces. Therefore, given the LTL formula μ which formalizes the ConDec model under study and a property γ:

- $\mu \models_\exists \gamma$ iff $\mu \wedge \gamma$ is satisfiable;
- $\mu \models_\forall \gamma$ iff $\mu \wedge \neg\gamma$ is unsatisfiable.

As shown by Rozier and Vardi in [44], LTL satisfiability checking can be in turn reduced to model checking. The reduction process starts by building a *universal model* $\mathcal{U}$ which, in our context, is able to generate all the (infinite) possible execution traces composed by the activities contained in the ConDec diagram. Given an LTL formula ϕ, satisfiability checking is then realized by model checking $\neg\phi$ against $\mathcal{U}(\phi)$ (the universal model able to generate all models composed with the propositional symbols of ϕ). If the model checker finds a counter-example, satisfiability of ϕ is successfully proven, and the generated counter-example is actually a positive example of a model which satisfies ϕ.

Functions $\exists$-entailment$_{MC}(\mathcal{M},\gamma)$ and $\forall$-entailment$_{MC}(\mathcal{M},\gamma)$ implement these ideas. Besides the function which maps a ConDec model to an LTL conjunction formula, two others support functions are used:

- universal_model($\mathcal{M}$) takes the activities of the ConDec model $\mathcal{M}$, building an universal model able to generate all the execution traces which involve these activities. The complexity of the construction of an universal model depends on the chosen tool. For example, for SPIN it is exponential in the number of activities, because one has to explicitly enumerate all their possible combinations [44]; conversely, for NuSMV [14] it is linear in the number of activities, because it is sufficient to list them in the model, and then NuSMV will take care of computing all the possible combinations.

```
function: ∃-entailment_MC(ConDec model M, LTL property γ)
returns : true, together with a sample execution trace T if
          LTL_transform(M) ⊨∃ γ, false otherwise
4.1  begin
4.2    μ ← LTL_transform(M) ∧ γ;
4.3    U ← universal_model(M);
4.4    [Success, T] ← model_checking(U, ¬μ);
4.5    if ¬Success then
4.6      return [true, T];
4.7    else
4.8      return [false, -];
4.9    end
4.10 end
```

Function $\exists$-entailment$_{MC}(\mathcal{M},\gamma)$

```
function: ∀-entailment_MC(ConDec model M, LTL property γ)
returns : true if LTL_transform(M) ⊨∀ γ, false, together with a counter-example
          execution trace T, otherwise
5.1  begin
5.2    return ∃-entailment_MC(M, ¬γ);
5.3  end
```

Function $\forall$-entailment$_{MC}(\mathcal{M},\gamma)$

- model_checking($\mathcal{S}$,ϕ) model checks $\mathcal{S}$ againts ϕ, returning $true$ if $\mathcal{S}$ meets ϕ in every possible execution, $false$, together with a counter-example, otherwise.

In order to choose a suitable model checker and run a comparative evaluation of g-$\mathcal{S}$CIFF with the state of the art, we referred to the results of an experimental investigation conducted by Rozier and Vardi on LTL satisfiability checking [44]. The authors found that the symbolic approach is clearly superior to the explicit approach, and that NuSMV is the best performing model checker for the benchmarks they considered. We thus chose NuSMV and ran our benchmarks to compare it with g-$\mathcal{S}$CIFF[10].

5.4. Verification Benchmark

To obtain our benchmarks, we made the model shown in Section 2.2 (Figure 1) more complex, so as to stress g-$\mathcal{S}$CIFF and emphasize its performance results in both favorable and unfavorable cases. In particular, verification techniques are stressed along two axes: size of the model (i.e., number of constraints and disjunctive ramifications) and required executions of a certain activity. The standard payment ac-

[10]We ruled out explicit model checkers, such as SPIN, because in our experiments they could not handle in reasonable time even a ConDec chart and properties as simple as the ones we described in Section 3.5.

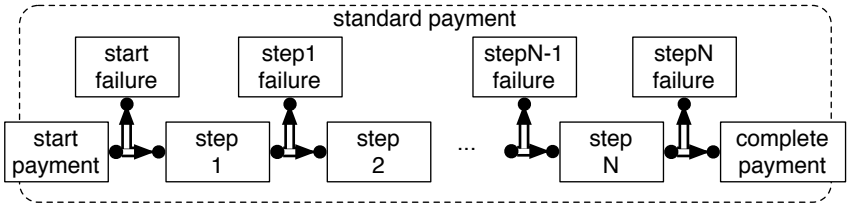


Figure 4: Parametric extension to the model presented in Fig.1

tivity has been extended 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.

5.5. Experimental Results

We compared g-$\mathcal{S}$CIFF with NuSMV on two sets of benchmarks:

1. Query 2, presented in Section 2.2 and shown in Fig. 2;
2. a simple existential query stating that the send receipt activity must be executed at least once.

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. The input files are available on a Web site[11]. The runtime resulting from the benchmarks has been presented in [35] and is reported in Table 1. It shows the amount of time required for generating a (counter-)example or for detecting unsatisfiability. Figure 5 shows the ratio NuSMV/g-$\mathcal{S}$CIFF runtime, in Log scale.

It turns out that g-$\mathcal{S}$CIFF outperforms NuSMV in most cases, up to several orders of magnitude. This is especially true for the first benchmark, for which g-$\mathcal{S}$CIFF is able to complete the verification task always in less than 0.15s, while NuSMV takes up to 136s. For the second benchmark, g-$\mathcal{S}$CIFF does comparatively better as N increases, for a given K, whereas NuSMV improves w.r.t. g-$\mathcal{S}$CIFF and eventually outperforms it, for a given N, as K increases.

5.6. Empirical Evaluation of the Benchmarks Results

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

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

Table 1: Results of the benchmarks ($\mathcal{S}$CIFF/NuSMV), in seconds [35].

$K \setminus N$	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

as ConDec, then the specification is mapped automatically to a $\mathcal{S}$CIFF program. Using g-$\mathcal{S}$CIFF, it is possible to verify the specification's properties. Using the $\mathcal{S}$CIFF 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 expectations and using abduction as a mechanism to simulate events. No intermediate format 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 $\mathcal{S}$CIFF programs are written w.r.t. the property. Due to the left-most, depth-first search tree exploration strategy that $\mathcal{S}$CIFF inherits from Prolog, the order of clauses influences performance, and so does the ordering of atoms inside the clauses. However, this does not impact on soundness.

In particular, since verification is performed by g-$\mathcal{S}$CIFF starting from expectations, its performance is heavily influenced by the presence of existence/choice constraints in the ConDec model/query. Indeed, existence and choice constraints are the ones that, translated to $\mathcal{S}$CIFF, impose expectations about the execution of a certain activity independently of the other activities. At the beginning of the computation, these expectations are transformed by g-$\mathcal{S}$CIFF to happened events via the *fulfiller* transition. These happened events, in turn, trigger new parts of the model under study, leading to the generation of new expectations and happened events. This is why the performance of g-$\mathcal{S}$CIFF decrease as the K value

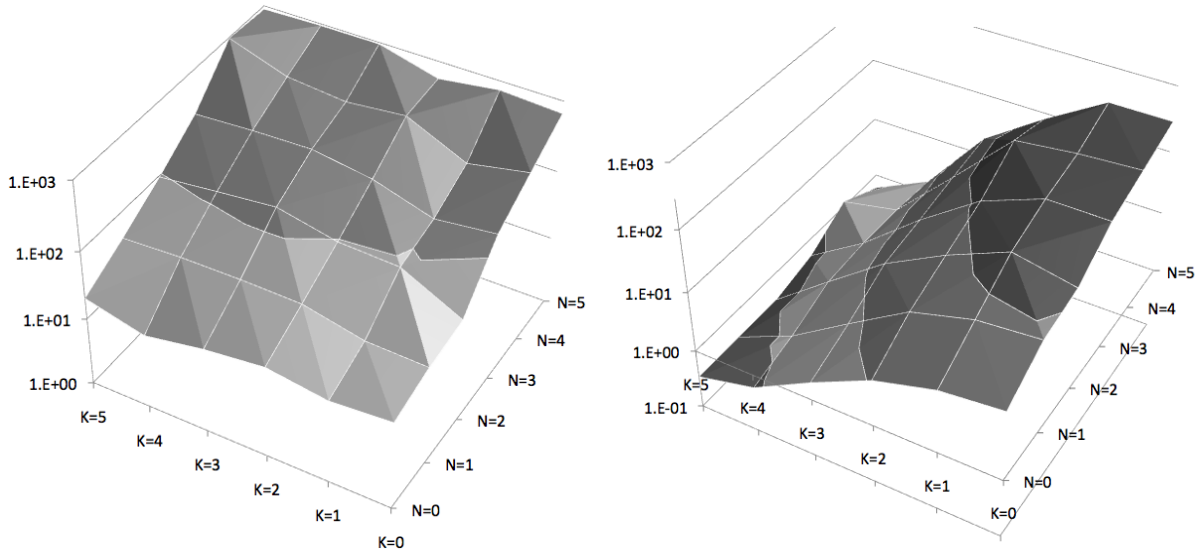


Figure 5: Charts showing the ratio NuSMV/g-$\mathcal{S}$CIFF runtime, in Log scale.

of the benchmarks increases: all the K expected executions are simulated, triggering the outgoing relationships K times, and so on. On the other side, if a portion of the ConDec model is not affected by this propagation, i.e., its execution is not mandatory, then its constraints do not affect verification at all. The extreme case is the one in which no activity is expected to be executed by the ConDec model nor by the query: in this situation, independently of the size of the model, g-$\mathcal{S}$CIFF answers immediately by returning the void execution trace as an example. This is the case, for example, when presence of conflict is checked on the running example shown in Figure 1. This smart exploration of the search space motivates why when N (i.e., the number of constraints) increases, the performance of g-$\mathcal{S}$CIFF degrades gracefully. Furthermore, it suggests that suitable heuristics that choose how to explore the search tree could help to improve the g-$\mathcal{S}$CIFF performance. This is subject for future research.

Differently from $\mathcal{S}$CIFF, model checking techniques first translate the formula representing the ConDec model and the query to an intermediate structure (an automaton in the case of explicit model checking, a Binary Decision Diagram - BDD - in the case of symbolic model checking). It is well known that this translation is exponential in the size of the formula [15, 17], and that model checking suffers from the state explosion problem. When model checking is adopted, as in our case study, to face the satisfiability problem, state explosion is a central issue, because both the model of the system and the property are represented by means of LTL formulae. Such a problem is partially mitigated by symbolic model checking, because it relies on BDDs which are an efficient way to organize data. In any case, it comes as no surprise that NuSMV experiences an exponential degradation as N increases: N reflects the number of constraints contained in the ConDec model, and thus also the size of the underlying LTL formula. When K increases, the degradation is more graceful; indeed, an existence constraint stating that activity a must be executed at least K times is modeled in LTL with the formula $existence_K = \Diamond(a \wedge \Diamond existence_{K-1})$, and the the size of the formula does not dramatically increase from $K-1$ to K.

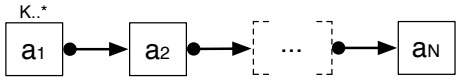
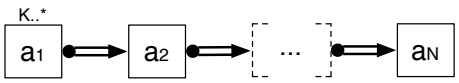
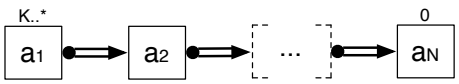
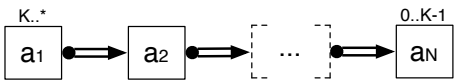
ID	Sat/Unsat	Model
`resp_sat`	satisfiable	
`alt_resp_sat`	satisfiable	
`alt_resp_unsat-0`	unsatisfiable	
`alt_resp_unsat-K`	unsatisfiable	

Table 2: Four benchmarks thought to stress verifiers along different dimensions.

5.7. Further Investigation

In order to sustain our empirical evaluation, we identified a new set of four "artificial" benchmarks, thought to emphasize the performance of verifiers along the three fundamental dimensions considered so far:

- number of constraints in the model (N value);
- minimum number of executions required for a certain activity (K value);
- satisfiable vs unsatisfiable models.

The four benchmarks are shown in Table 2. The first two benchmarks deal with satisfiable models. The difference is that alternate response constraints are more complex than basic response constraints, and therefore benchmark `alt_resp_sat` is more difficult to handle than `resp_sat`. The third and last benchmark deal instead with unsatisfiable (conflicting) models. The two benchmarks are unsatisfiable because, for a sequence of activities interconnected by means of alternate response constraints, it holds that each activity of the sequence must be executed a number of times which is greater or equal than the number of times the first activity is executed; in particular, activity a_N must be executed at least K times. Roughly speaking, in `alt_resp_unsat-0` unsatisfiability can be detected "sooner" than in `alt_resp_unsat-K`, because it is sufficient to discover that a_N must be executed at least once to detect the conflict.

To broaden the scope of our comparison, we also included a third verifier, namely the Zot model checker [42]. Differently from NuSMV, Zot is a bounded model checker which employs SAT-based techniques to carry out the verification. Zot is able to deal also with metric specifications (such as MTL [6]), and it addresses the satisfiability checking of MTL/LTL formulae directly, without requiring a reduction to model checking. Being the bound a critical parameter for what concerns timings of the verifier, we ran our benchmarks by setting different bounds for Zot (50, 100 and, sometimes, 200 time units).

	g-SCIFF (◆)			NuSMV(■)			Zot(50)(▲)			Zot(100)(×)		
	K=1	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**
N=**1**	0.0	0.0	0.0	0.03	0.02	0.02	0.69	1.15	1.44	0.89	1.71	2.66
2	0.0	0.0	0.03	0.02	0.02	0.03	1.10	1.51	1.88	1.70	2.55	3.57
3	0.0	0.01	0.07	0.02	0.03	0.02	1.58	1.97	2.39	2.71	3.53	4.32
4	0.0	0.02	0.14	0.02	0.03	0.03	2.13	2.46	2.81	3.79	4.57	5.38
5	0.0	0.04	0.23	0.03	0.03	0.03	2.50	2.84	3.17	4.88	5.67	7.24
6	0.01	0.05	0.34	0.03	0.03	0.04	2.93	3.45	4.75	6.86	6.94	7.87
7	0.01	0.07	0.48	0.04	0.06	0.05	3.56	4.01	4.43	8.51	9.54	10.40
8	0.01	0.09	0.64	0.05	0.06	0.07	4.47	4.70	4.53	9.39	13.58	13.34
9	0.01	0.11	0.85	0.07	0.06	0.07	4.53	4.74	5.30	16.07	13.09	11.81
10	0.01	0.13	1.09	0.06	0.09	0.09	4.82	5.56	6.07	10.67	17.65	14.73
11	0.01	0.15	1.38	0.08	0.16	0.12	5.76	6.22	6.34	13.09	14.47	15.84
12	0.01	0.18	1.73	0.20	0.21	0.24	5.87	6.22	7.03	15.98	16.98	16.26
13	0.02	0.22	2.17	0.30	0.42	0.53	6.44	7.08	8.31	15.10	17.39	20.13
14	0.02	0.25	2.89	0.55	0.64	0.79	7.27	7.75	9.07	18.31	18.97	23.41
15	0.02	0.3	3.7	0.73	1.33	1.35	7.75	8.33	9.01	22.03	22.48	23.81
16	0.02	0.34	4.73	2.19	2.80	2.79	8.63	9.46	10.93	23.84	22.30	29.32
17	0.03	0.37	5.82	3.75	4.55	5.56	12.38	10.46	9.73	25.31	25.10	31.14
18	0.02	0.42	7.23	6.79	8.44	13.85	9.84	9.86	12.49	25.97	29.36	32.76
19	0.03	0.46	8.77	13.53	13.97	19.48	11.23	12.45	12.10	33.04	32.57	28.33
20	0.04	0.53	10.68	22.85	27.82	54.18	11.38	12.62	13.38	32.17	30.34	32.59

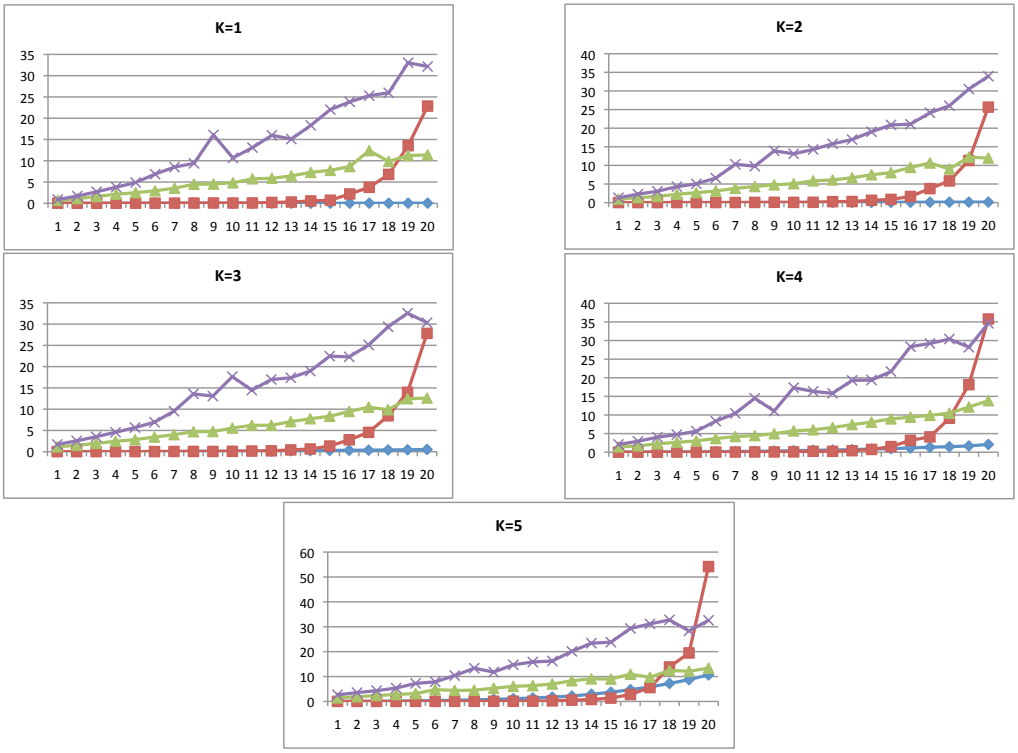


Table 3: Results for the `resp_sat` benchmark.

	g-SCIFF (◆)			NuSMV(■)			Zot(50)(▲)			Zot(100)(✕)			Zot(200)(✳)		
	K=1	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**
N=1	0.0	0.0	0.0	0.0	0.0	0.0	1.7	1.1	1.4	0.9	1.7	2.8	2.4	3.0	5.0
2	0.0	0.0	0.1	0.0	0.0	0.0	1.4	1.2	2.9	1.8	2.7	3.7	3.3	2.8	6.4
3	0.0	0.0	0.2	0.0	0.0	0.1	1.6	2.4	3.7	2.9	3.8	5.2	5.3	7.6	9.9
4	0.0	0.1	0.4	0.0	0.0	0.0	2.2	2.7	3.0	4.2	5.4	6.2	8.1	10.1	13.9
5	0.0	0.1	0.8	0.0	0.1	0.1	3.0	3.1	3.4	4.7	5.9	7.1	9.0	14.2	16.9
6	0.0	0.1	1.3	0.0	0.1	0.1	3.7	3.5	4.1	7.2	9.9	8.6	14.8	17.3	18.8
7	0.0	0.1	1.8	0.0	0.1	0.1	3.6	4.1	4.4	7.6	8.0	11.4	16.8	20.2	24.7
8	0.0	0.2	2.8	0.1	0.1	0.2	4.1	4.7	5.0	9.7	11.5	11.8	18.1	23.1	24.2
9	0.0	0.2	4.3	0.1	0.2	0.4	4.8	5.3	6.0	10.1	10.6	14.1	26.7	28.5	33.2
10	0.0	0.3	7.2	0.2	0.4	0.7	5.4	6.1	6.6	16.8	15.6	16.0	29.2	33.8	34.2
11	0.0	0.4	12.5	0.3	1.0	2.5	5.7	6.6	6.9	14.1	16.1	19.7	30.5	39.5	45.7
12	0.0	0.4	16.5	0.5	1.9	4.0	6.6	6.9	7.4	17.2	15.5	20.2	43.1	59.0	65.3
13	0.0	0.5	21.8	1.1	4.7	8.7	8.4	9.4	10.5	18.5	20.0	20.5	48.5	71.8	57.9
14	0.0	0.6	29.2	2.0	8.3	18.9	9.3	9.7	8.9	16.6	21.5	23.0	63.5	66.1	65.0
15	0.0	0.7	34.1	5.3	20.7	43.4	9.3	10.7	11.2	20.7	22.6	23.2	60.7	62.2	70.0
16	0.0	0.8	43.4	12.9	49.1	99.5	10.3	10.4	11.4	24.8	22.1	28.2	77.0	80.6	89.6
17	0.0	1.0	53.4	23.5	102.3	233.3	12.1	11.3	10.3	25.1	27.9	28.7	69.3	88.2	102.7
18	0.0	1.1	65.1	67.2	236.9	467.9	10.4	12.2	14.6	28.3	26.7	25.8	95.3	91.1	114.3
19	0.0	1.2	72.6	175.5	603.2	¿10′	12.8	13.7	12.8	30.8	31.7	32.4	105.0	113.6	114.0
20	0.1	1.7	98.9	463.3	¿10′	¿10′	13.2	16.7	14.2	31.2	36.2	39.7	129.0	125.4	154.6

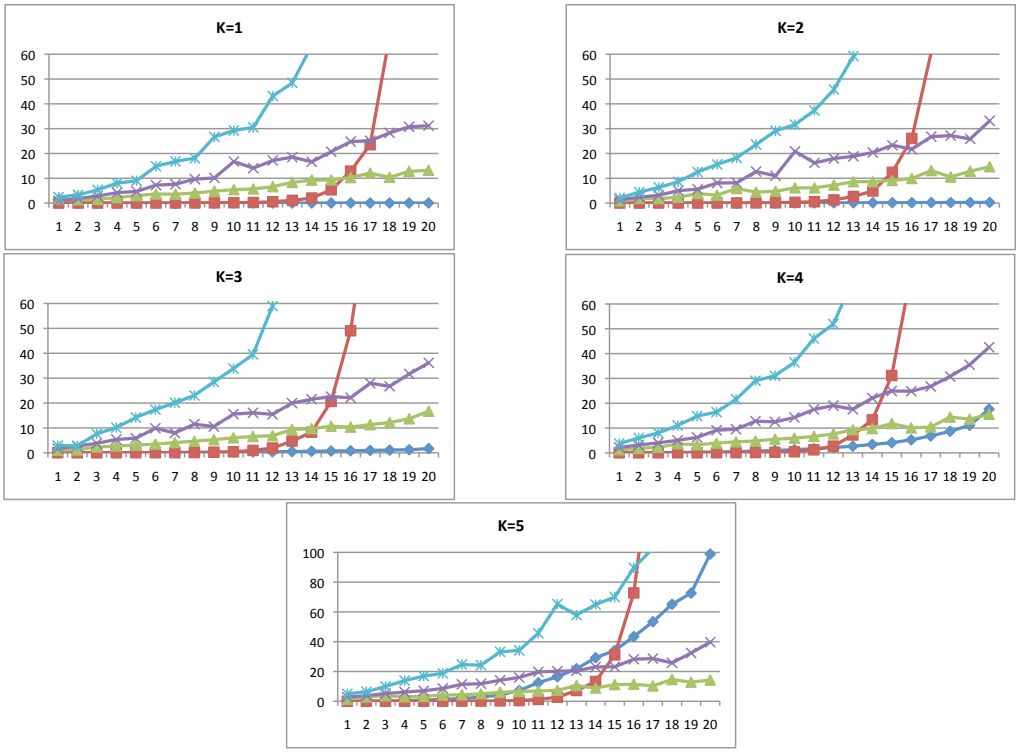


Table 4: Results for the `alt_resp_sat` benchmark.

Tables 3, 4, 5 and 6 report the results obtained by checking conflict-freedom of the four benchmarks depicted in Table 2, by varying the N parameter between 1 and 20 and the K parameter between 1 and 5.

For what concerns g-$\mathcal{S}$CIFF and NuSMV, the results confirm the empirical evaluation discussed in Section 5.6. g-$\mathcal{S}$CIFF's timings for benchmark `alt_resp_unsat_0` are particularly good because as soon as a positive expectation concerning a_N is generated, an $\mathbf{E}$-inconsistency between such an expectation and the 0 cardinality constraint on a_N is encountered, leading to immediately detect the conflict.

Zot exhibits a quite different behavior. First of all, its performance obviously strictly depends on the chosen bound. Finding a suitable bound is a very difficult task in the context of ConDec, due to the interplay between the different constraints contained in the model; for example, although they share the same number of activities and constraints, benchmark `resp_sat` requires the execution of at least $N+K-1$ activities, while benchmark `alt_resp_sat` of at least $N \times K$ activities. Since the performance of Zot decreases as the value of the bound increases, Zot cannot efficiently deal with metric specifications where different time granularities (such as minutes and days) are mixed together.

Second, while the performance of Zot is very competitive when the model is conflict-free (surprisingly, it handles the resp_sat and alt_resp_sat benchmarks with the same timings), its behavior quickly degrades when the model contains conflicts. This becomes apparent by looking at Table 6: as already pointed out, the conflict detection requires, for benchmark `alt_resp_unsat-K`, a high number of inference steps and variables assignments, causing Zot to explode even with small values of N and K.

6. Formal Properties

In Section 4.2 we have introduced soundness and termination of g-$\mathcal{S}$CIFF on *acyclic* $\mathcal{S}$CIFF specifications. *Acyclic* specifications are specifications which respect stratification, i.e., for which a suitable level mapping can be found [12, 36, 48]. *Acyclicity* is a critical point for g-$\mathcal{S}$CIFF, because g-$\mathcal{S}$CIFF implicitly adds to the specifications the generative constraint $\mathbf{E}(E,T) \rightarrow \mathbf{H}(E,T)$. While $\mathcal{S}$CIFF specifications obtained by translating an arbitrary ConDec model are always acyclic w.r.t. the $\mathcal{S}$CIFF proof procedure, this is not the case for g-$\mathcal{S}$CIFF: it is possible to design ConDec diagrams whose underlying $\mathcal{S}$CIFF formalization, augmented with the rule $\mathbf{E}(E,T) \rightarrow \mathbf{H}(E,T)$, is not stratified.

When stratification is not guaranteed, g-$\mathcal{S}$CIFF may experience non-termination: instead of providing an answer, the proof procedure loops. This comes as no surprise, since the $\mathcal{S}$CIFF language is a first-order language with function symbols, and it is therefore semi-decidable. Model checking propositional LTL formulae with unbounded model checkers, instead, is decidable and has the ability to reason upon infinite specifications; therefore, it always provides a correct answer in finite time. Furthermore, it can easily accommodate the assumption that executions of ConDec models must eventually terminate, thus making possible to use unbounded model checkers for verification of finite-trace systems.

Figure 6 shows three ConDec models for which g-$\mathcal{S}$CIFF may not terminate when performing static verification. Let us for example consider the problem of checking if d is a dead activity in the three models:

- g-$\mathcal{S}$CIFF always loops when checking if d is a dead activity in Figure 6a; since it follows a simulation approach, when trying to check if d can be executed in at least one trace, it generates the execution trace $d \rightarrow c \rightarrow a \rightarrow b \rightarrow c \rightarrow \ldots$, which is infinite. Note that activity d is, in this model, dead, because each execution of a BP must eventually terminate, while the execution of d leads to enter within an infinite loop.

	g-SCIFF (◆)			NuSMV(■)			Zot(50)(▲)			Zot(100)(×)		
	K=1	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**
N=1	0.00	0.00	0.00	0.01	0.03	0.02	0.85	1.28	1.72	1.33	2.13	2.76
2	0.00	0.00	0.01	0.02	0.02	0.02	1.30	1.65	2.12	2.08	2.81	3.61
3	0.01	0.01	0.01	0.02	0.03	0.02	1.51	2.02	2.61	2.72	3.89	4.42
4	0.01	0.01	0.01	0.02	0.03	0.09	2.20	2.84	2.69	3.98	4.61	5.30
5	0.01	0.01	0.01	0.03	0.03	0.05	2.61	2.93	3.56	4.75	5.26	6.44
6	0.00	0.01	0.01	0.03	0.05	0.05	3.49	3.36	3.65	5.51	6.55	8.14
7	0.01	0.01	0.00	0.04	0.05	0.06	3.37	3.73	4.18	7.52	9.40	9.33
8	0.01	0.01	0.02	0.06	0.08	0.10	2.05	4.27	4.81	8.38	10.76	10.26
9	0.01	0.02	0.02	0.11	0.16	0.11	4.41	4.90	5.29	9.90	10.31	13.68
10	0.02	0.02	0.02	0.09	0.11	0.13	4.95	5.58	6.66	13.32	13.65	12.30
11	0.02	0.02	0.02	0.11	0.23	0.59	6.76	6.90	6.83	12.12	13.86	15.00
12	0.02	0.02	0.02	0.46	0.29	0.41	6.20	6.91	7.58	13.42	14.35	17.12
13	0.02	0.03	0.03	0.78	0.53	0.67	6.90	7.08	7.54	15.59	16.65	17.50
14	0.02	0.03	0.04	1.20	1.00	1.19	7.40	8.61	10.47	16.79	21.61	17.81
15	0.03	0.03	0.04	1.80	2.11	2.42	9.59	9.61	9.36	16.05	19.24	21.70
16	0.03	0.04	0.04	4.25	5.00	5.70	8.75	8.88	9.27	18.75	21.86	23.17
17	0.04	0.04	0.04	2.44	4.13	15.19	9.78	10.25	11.05	24.60	22.04	24.90
18	0.04	0.05	0.05	3.29	4.35	4.60	9.63	11.40	11.71	22.52	24.96	28.21
19	0.05	0.05	0.05	7.33	6.35	7.21	10.93	13.43	12.56	29.17	25.60	28.66
20	0.05	0.05	0.05	15.86	17.08	12.46	11.42	12.67	13.57	27.35	31.98	31.09

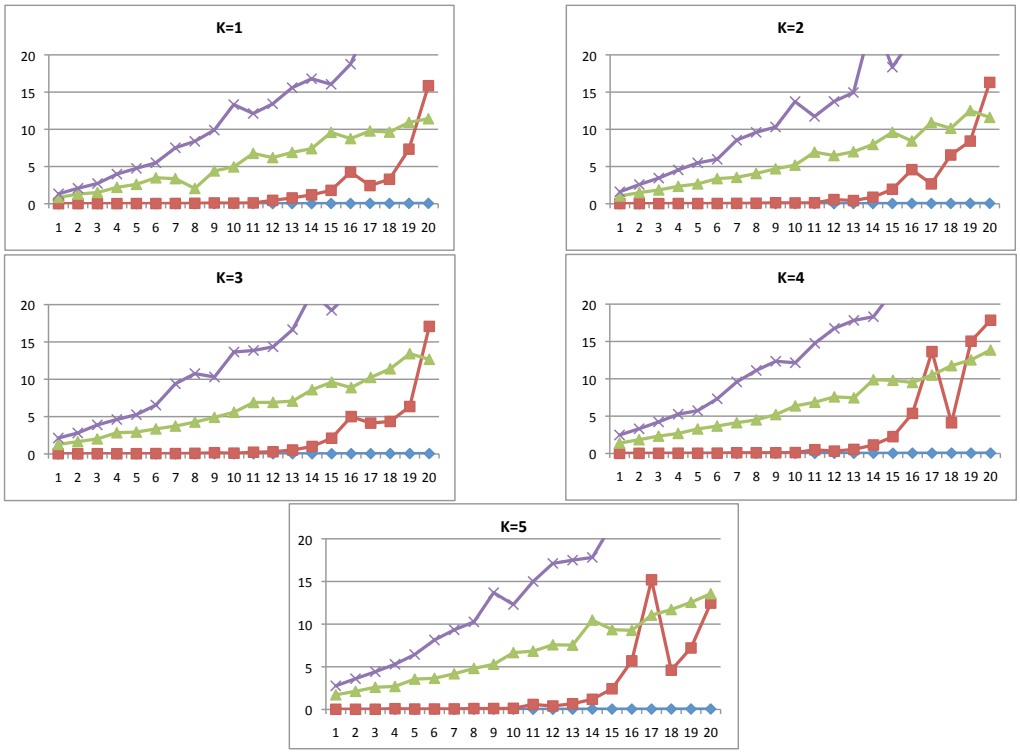


Table 5: Results for the `alt_resp_unsat-0` benchmark.

	g-SCIFF (◆)			NuSMV(■)			Zot(50)(▲)			Zot(100)(✕)		
	K=1	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**	**1**	**3**	**5**
N=**1**	0.0	0.0	0.2	0.0	0.0	0.0	0.9	1.7	2.9	1.3	3.2	9.2
2	0.0	0.0	0.4	0.0	0.0	0.0	1.3	2.8	7.1	2.0	9.0	79.2
3	0.0	0.0	0.7	0.0	0.0	0.1	1.7	5.2	38.2	3.4	19.0	393.0
4	0.0	0.1	1.3	0.1	0.0	0.0	2.2	10.6	131.9	5.3	159.2	2824.4
5	0.0	0.1	2.0	0.0	0.0	0.1	2.7	56.6	459.0	6.7	697.7	¿10′
6	0.0	0.1	2.8	0.0	0.1	0.2	3.0	74.7	¿10′	5.8	¿10′	¿10′
7	0.0	0.2	4.3	0.0	0.1	0.4	3.7	289.9	¿10′	7.2	¿10′	¿10′
8	0.0	0.2	5.9	0.1	0.3	1.2	4.2	447.0	¿10′	9.6	¿10′	¿10′
9	0.0	0.3	10.5	0.1	0.2	0.8	4.7	350.6	¿10′	121	¿10′	¿10′
10	0.0	0.3	13.9	0.1	0.5	1.5	4.8	938.6	¿10′	122	¿10′	¿10′
11	0.0	0.4	20.0	0.1	0.5	1.5	5.6	¿10′	¿10′	139	¿10′	¿10′
12	0.0	0.5	30.3	0.5	1.0	3.4	5.8	¿10′	¿10′	146	¿10′	¿10′
13	0.0	0.6	36.6	0.9	2.4	11.9	6.5	¿10′	¿10′	142	¿10′	¿10′
14	0.0	0.6	54.4	1.4	4.6	15.1	7.5	¿10′	¿10′	173	¿10′	¿10′
15	0.0	0.8	63.3	2.0	113	18.7	7.4	¿10′	¿10′	192	¿10′	¿10′
16	0.0	0.9	71.8	5.1	9.6	50.5	8.7	¿10′	¿10′	224	¿10′	¿10′
17	0.0	1.1	100.2	3.1	101	79.5	102	¿10′	¿10′	200	¿10′	¿10′
18	0.0	1.2	110.5	4.9	160	33.9	9.2	¿10′	¿10′	209	¿10′	¿10′
19	0.1	1.3	124.5	117	243	233.2	122	¿10′	¿10′	233	¿10′	¿10′
20	0.1	1.5	143.0	198	382	145.8	107	¿10′	¿10′	241	¿10′	¿10′

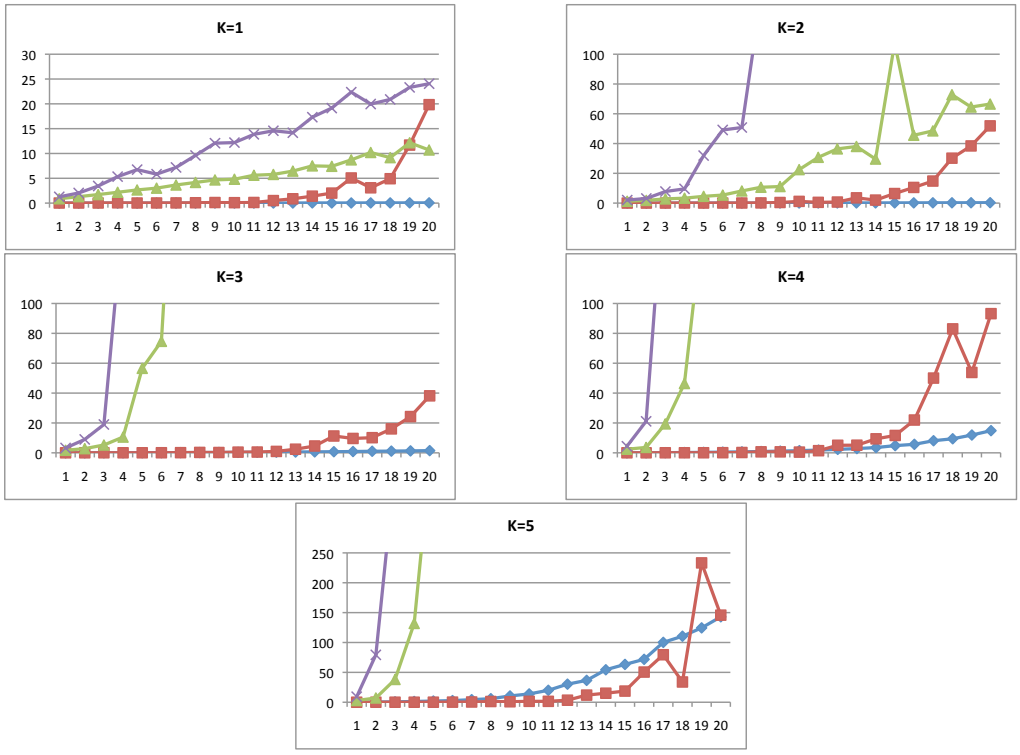
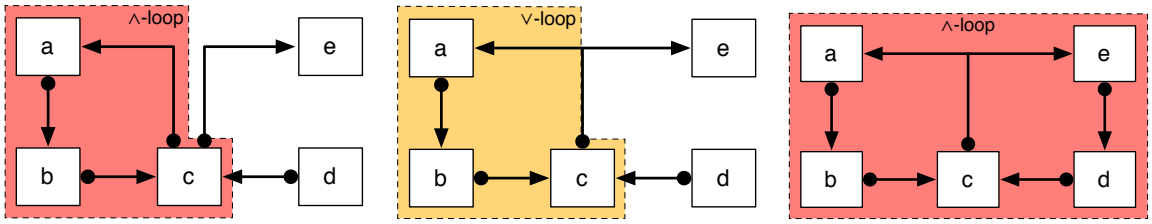


Table 6: Results for the `alt_resp_unsat-K` benchmark.



(a) All activities but e are dead. (b) No activity is dead. (c) All activities are dead.

Figure 6: Three ConDec models containing different kinds of loops.

- In Figure 6b, d is not dead; indeed, even if a loop is contained in the model, it is possible to exit the loop, by choosing to execute e after c instead of a. Termination of g-$\mathcal{S}$CIFF when checking if d is dead is determined, in this case, by the syntactic structure of the integrity constraint which represents the disjunctive response spanning from activity c. Indeed, g-$\mathcal{S}$CIFF adopts, by default, a depth-first search strategy; therefore, if a is the first choice, g-$\mathcal{S}$CIFF loops as in the case of Figure 6a, whereas if e is the first choice, g-$\mathcal{S}$CIFF correctly terminates by stating that d is not dead and providing the execution trace $d \rightarrow b \rightarrow e$ as an example. Adopting an iterative deepening strategy (where deepening is referred to the maximum number of generated happened events) would help in this specific case, but not in the general one.
- Figure 6c extends Figure 6b, introducing a further response constraint between e and d. In this situation, both choices spanning from b cause a loop. Therefore, d is a dead activity, but g-$\mathcal{S}$CIFF is unable to prove it.

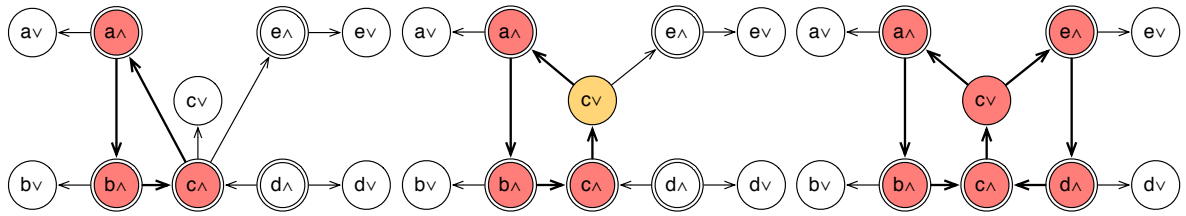
7. Pre-processing of ConDec Models and Loops Detection

In [34], it has been shown that only ConDec models that contain loops, constituted by constraints which are temporally-ordered towards the same "direction" (past vs forward-orientation in time), produce non-stratified specifications. Therefore, a pre-processing analysis can be carried out by reasoning upon the structure of the graphical model, identifying possible loops and taking proper countermeasures. Two different kinds of loop can be identified in a ConDec model:

- the execution of an activity belonging to an $\wedge$-loop forces the user to re-execute the same activity afterwards (or before);
- the execution of an activity belonging to an $\vee$-loop may force the user to re-execute the same activity afterwards (or before), depending on the choice made by the user.

In Figure 6a, activities a, b, c belong to an $\wedge$-loop. In Figure 6b, they belong to an $\vee$-loop, because only if the user chooses a after c the loop is entered. Finally, in Figure 6c all activities belong to an $\wedge$-loop: no matter what choice is taken after having performed b, the user is forced to re-execute the same activity again. Instead, the ConDec model shown in Figure 1 is loop-free.

To discover the presence of $\wedge$- and $\vee$-loops, it is possible to map a ConDec model to two AND/OR graphs, taking into account backward-oriented and forward-oriented constraints separately, and then find



(a) All activities but e are dead. (b) No activity is dead. (c) All activities are dead.

Figure 7: The three AND/OR forward-oriented graphs corresponding to the ConDec models shown in Figure 6. Single-lined circles represent AND nodes, whereas double-lined circles represent OR nodes.

loops in these graphs. The intuitive idea of the mapping, in the case of forward-oriented constraints, is the following[12]. For each activity a in the model:

1. map each a to two nodes, one AND-node ($a\wedge$) and one OR-node ($a\vee$), connected by an edge;

2. for each forward-oriented disjunctive constraint spanning from a and pointing to $b_1, \ldots, b_n$, add an edge from $a\vee$ to each $b_i\wedge$;

3. for each forward-oriented non-disjunctive constraint spanning from a and pointing to b, add an edge from $a\wedge$ to $b\wedge$.

The application of this translation procedure to the three diagrams shown in Figure 6 is depicted in Figure 7[13]. As the three graphs of Figure 7 suggest, an $\vee$- and an $\wedge$-loop differ from each other in that an $\vee$-loop is a loop which contains at least one OR-node, and this OR-node is connected with at least one node which does not belong to a $\wedge$-loop (the presence of this node reflects the possibility of exiting from the loop in the corresponding ConDec model). The outcome of this pre-processing analysis can be exploited as follows:

1. If the model contains an $\wedge$-loop s.t. none of its activities has an explicit 0 cardinality constraint attached to it, ConDec semantics states that all the activities belonging to the loop are dead, because their execution would undermine the condition that the process will eventually terminate. An explicit 0 cardinality constraint can be therefore attached to each of these activities, obtaining an more constrained model.

2. If the model is $\vee$-loops free, then g-$\mathcal{S}$CIFF will correctly reason upon the augmented model without experiencing non-termination issues; indeed, as soon as g-$\mathcal{S}$CIFF tries to enter a loop by generating the execution of one of its activities, the presence of an explicit 0 cardinality constraint, which is formalized in g-$\mathcal{S}$CIFF with a negative expectation, leads immediately to a failure, because fulfillment is not respected.

[12] A more complete description can be found in [34].

[13] Note that the models of Figure 6 contain only response constraints, which are forward-oriented; the application of the translation method on past-oriented constraints would therefore produce graphs with isolated nodes.

3. If instead the model contains a $\vee$-loop, then termination cannot be guaranteed anymore. The outcome of this pre-processing phase can be used to impose a maximum bound on the length of the possible execution traces produced by g-$\mathcal{S}$CIFF (i.e., to choose a bounded depth-first search strategy, similarly to bounded model checking[14]), alerting the user that termination is preserved but completeness is not. Indeed, by adopting a bounded search strategy, a *no* answer does not prove that no execution trace compliant with the specification exists, but only that there is no compliant execution trace whose length is up to the bound.

8. Related Work

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 range on finite domains, and that the cardinality of domains impacts heavily on their performance, especially in relation to the production of an automaton starting from a logical formula. 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 [25]. g-$\mathcal{S}$CIFF 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.

A vast literature addresses the problem of formally specify business processes, service choreographies and multi-agent systems by exploiting procedural languages. Such specifications can be then verified using standard model checking techniques. For example, in [7] Awad et al. adapt the BPMN-Q graphical notation (based on the BPMN elements), to express business rules and queries, and exploit BPMN as a modeling notation for BP. The process model is then translated to a Petri Net via a multi-step methodology able to isolate its relevant sub-parts, while the BPMN-Q query is expressed as a past-LTL formula. Model checking is then employed to verify if the formula is satisfied by the Petri Net.

In [43], Raimondi and Lomuscio present a methodology for verifying multi-agent systems. Properties representing the correct (desired) behavior of agents are specified by means of a modal logic which includes temporal and epistemic operators, and are checked by means of symbolic model checking.

Differently from all these works, the approach presented in this paper relies on a more flexible way to model the system under study: with ConDec, the system itself is modeled by means of declarative constraints which are then translated to the $\mathcal{S}$CIFF computational logic-based language.

In [45], Russo et al. exploit abduction for verification of declarative specifications expressed in terms of required reactions to events. In particular, their approach is grounded on the Event Calculus, and includes an explicit time structure. Global systems invariants, and in particular safety properties, are proved in a refutation mode (i.e., negated and mapped into a goal, starting from which the uderlying proof procedure possibly generates a counter example via abduction). The adopted proof procedure always terminates, in contrast to most conventional theorem proving techniques. It does not rely on a complete description of the initial state (in contrast to model checking approaches). It also support reasoning about specification whose state-spaces may be infinite; this last feature is mainly because the procedure is property-driven.

[14] Bounded model checking addresses the problem of verifying the validity of a formula within a predefinite number of transitions of a system.

The main difference between the work of Russo et al. and ours concerns the underlying specification language: while they rely on a general purpose ALP proof procedure which handles Event Calculus specifications and requirements, we adopt a language which directly captures the notion of occurred events and expectations, and whose temporal relationships are mapped on CLP constraints. In this way, we can rely on a variety of CLP domains (e.g., integers, reals, just to mention the two most relevant ones), exploiting the CLP machinery to handle metric temporal constraints such as deadlines and latencies.

Another system aimed at proving properties of graphical specifications translated to logic programming formalisms is West2East [11], 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 [19], 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 formulae, 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.

In [41], Pesic et al. describe an integrated framework in which ConDec models can be graphically specified, automatically obtaining the underlying LTL formalization, and then verified and executed. Instead of adopting standard model checking techniqes, verification is carried out by exploiting the finite-trace model checking approach proposed in [22]. The approach reviews the LTL semantics to reflect finite-trace systems, and propose a translation algorithm which produces a standard automaton. However, the approach presents the same computational drawbacks of explicit model checking: the translation phase has a cost which is exponential in the size of the formula.

As pointed out in Section 5.3, the presented quantitative evaluation does not cover all aspects of the $\mathcal{S}$CIFF language, which contains variables and supports the possibility of expressing quantitative time constraints. Different extensions of propositional LTL have been proposed to explicitly referencing time and expressing quantitative time constraints. For example, the timed requirement of Query 4, Fig. 3, stating that a receipt is expeted by 12 time units after having executed accept advert, can be expressed in MTL[6] as

$$\Box(\texttt{accept\_advert} \rightarrow \Diamond_{\leq 12}\texttt{send\_receipt})$$

which is equivalent to the TPTL[5] formula:

$$\Box x.(\texttt{accept\_advert} \rightarrow \Diamond y.(y - x \leq 12 \wedge \texttt{send\_receipt}))$$

Many tools have been developed to verify real-time systems w.r.t. timed temporal logics, relying on timed automata [8]. For example, Uppaal [31] 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.

Among the temporal logic-based languages able to specify quantitative time constraints, we cite TRIO[21], a language, based on a metric extension of first-order temporal logic, for modeling critical real-time systems. Similarly to ConDec, systems are modeled in a declarative manner, i.e., as a conjunction of TRIO formulae. Different approaches have been investigated for model checking TRIO specifications, but many important features of the initial language are lost in the effort to obtain a decidable and tractable specification language. For example, the time domain is reduced to natural numbers, there is no quantification over time variables, and the language can range only on finite domains. Such a restricted language can be translated onto a Promela alternating Büchi automaton using the Trio2Promela tool [9], or encoded as a SAT problem in Zot [42].

SAT-based technologies have been introduced to overcome the state-explosion problem of classical bounded and unbounded model checking. One represents with boolean formulae 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 $0 \dots k$ iff the formula [32]

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

is unsatisfiable. SAT-based unbounded model checking is based on analogous formulae, but it also adds formulae that verify loop freeness (as in induction-based unbounded model checking [46]) or use SAT specific features (as in interpolant-based unbounded model checking [33]). In all cases, the transition function should be unfolded for a set of possible states, making the formula quite large. Indeed, modern SAT solvers can handle millions of boolean variables, but even generating a large SAT can be very costly.

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 [16] 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 [24], 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: the user must provide a predicate that generates a finite number of event sequences, representing all the possible evolutions of the system.

Finally, in relation to our own previous work, this article is an extended version of [35]. We characterize here in a more accurate way the static verification of declarative BPs, giving a detailed account about how verification can be tackled by the $\mathcal{S}$CIFF framework and effectively carried out by the g-$\mathcal{S}$CIFF proof procedure. Furthermore, in this work we have extended the experimental evaluation of the framework, comparing it also with the SAT-based bounded model checker Zot. We also provide a study of our framework's theoretical properties, with particular regard to termination issues.

The use of generative variants of the $\mathcal{S}$CIFF proof procedure, in order to carry out static forms of verification and reasoning, has been exploited also in [2, 3]. In these works, static verification is carried out in order to respectively check whether the behavioral interface of a concrete service could play a role

inside of a choreography, or could fruitfully interact with another service. However, differently from the work here presented, they do not apply g-$\mathcal{S}$CIFF's generative rule $\mathbf{E}(E,T) \rightarrow \mathbf{H}(E,T)$, but they instead rely on specialized generative rules, used to specify under which circumstances the emission of an expected message should be effectively simulated.

9. Conclusions

We identified the problem of static verification in the context of declarative Business Processes. We observed that this is still an open issue. Specifically, state of the art verification methods for declarative BP specifications, such as model checking, borrowed from other domains, in this context are particularly prone to the state explosion problem. Other methods such as theorem proving are not sufficiently automated to become accessible to the non IT-savvy. We provided a declarative, operational and scalable approach to the problem, discussing its formal properties and presenting an extensive experimental evaluation aimed at showing the feasibility of the proposed approach.

Future work will focus on a more extensive experimentation, aimed at comparing g-$\mathcal{S}$CIFF with other verifiers, taking into account also benchmarks including metric temporal constraints. Furthermore, we will investigate the possibility of augmenting g-$\mathcal{S}$CIFF for guaranteeing termination not only for the specific case of ConDec, but also for arbitrary $\mathcal{S}$CIFF specifications. A promising approach to deal with infinite computations seems to be Coinductive Logic Programming [23], which extends the usual operational semantics of logic programming to allow reasoning over infinite and cyclic structures and properties. The accommodation of Coinductive techniques inside the $\mathcal{S}$CIFF framework could be an interesting approach to overcome the termination issues of g-$\mathcal{S}$CIFF in the general case.

9.0.1. Acknowledgements

We would like to thank Marco Gavanelli for his valuable contribution on a previous version of this work. This work has been partially supported by the FIRB project *TOCAI.IT* (RBNE05BFRK) and by the Italian MIUR PRIN 2007 project No. 20077WWCR8.

References

[1] van der Aalst, W. M. P., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language., *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings* (M. Bravetti, M. Núñez, G. Zavattaro, Eds.), 4184, Springer, 2006, ISBN 3-540-38862-1.

[2] Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M.: An Abductive Framework for A-Priori Verification of Web Services, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (A. Bossi, M. J. Maher, Eds.), ACM Press, 2006.

[3] Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Torroni, P.: Web Service contracting: Specification and Reasoning with SCIFF, *Proceedings of the 4th European Semantic Web Conference (ESWC'07)* (E. Franconi, M. Kifer, W. May, Eds.), 4519, Springer Verlag, 2007.

[4] Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable Agent Interaction in Abductive Logic Programming: the SCIFF framework, *ACM Transactions on Computational Logic*, **9**(4), 2008, 1–43.

[5] Alur, R., Henzinger, T. A.: A really temporal logic, *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, 1989.

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

[7] Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic, *6th International Conference on Business Process Management (BPM 2008)* (M. Dumas, M. Reichert, M.-C. Shan, Eds.), 5240, Springer Verlag, 2008.

[8] Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools, *Lectures on Concurrency and Petri Nets* (J. Desel, W. Reisig, G. Rozenberg, Eds.), 3098, Springer, 2003.

[9] Bianculli, D., Morzenti, A., Pradela, M., San Pietro, P., Spoletini, P.: Trio2Promela: a Model Checker for Temporal Metric Specifications, *Proceedings of the 20th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, 2007.

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

[11] Casella, G., Mascardi, V.: West2East: exploiting WEb Service Technologies to Engineer Agent-based SofTware, *IJAOSE*, **1**, 2007, 396–434.

[12] Chesani, F.: *Specification, execution and verification of interaction protocols: an approach based on computational logic*, Ph.D. Thesis, University of Bologna, 2007, Available at `http://amsdottorato.cib.unibo.it/392/`.

[13] Christiansen, H., Dahl, V.: HYPROLOG: A New Logic Programming Language with Assumptions and Abduction., *Proceedings of ICLP 2005*, 2005.

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

[15] Clarke, E., Grumberg, O., Peled, D.: *Model Checking*, The MIT Press, Cambridge, Massachusetts and London, UK, 1999, ISBN 0-262-03270-8.

[16] Delzanno, G., Podelski, A.: Model Checking in CLP, *Proocedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999)*, LNCS, Springer Verlag, 1999.

[17] Demri, S., Laroussinie, F., Schnoebelen, P.: A Parametric Analysis of the State-Explosion Problem in Model Checking, *Journal of Computer and System Sciences*, **72**(4), 2006, 547–575.

[18] Emerson, E. A.: Temporal and Modal Logic, in: *Handbook of Theoretical Computer Science, Volume B*, MIT Press, 1990, 995–1072.

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

[20] Frühwirth, T.: Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming*, **37**(1-3), October 1998, 95–138.

[21] Ghezzi, C., Mandrioli, D., Morzenti, A.: TRIO: A logic language for executable specifications of real-time systems, *Journal of Systems and Software*, **12**(2), 1990, 107–123.

[22] Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems, *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference (ESEC/FSE 2003)*, ACM, 2003.

[23] Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and Its Applications, *Proceedings of the 23rd International Conference on Logic Programming (ICLP'2007)*, LNCS, 2007.

[24] Gupta, G., Pontelli, E.: A constraint-based approach for specification and verification of real-time systems, *Proceedings of RTSS 1997*, IEEE Computer Society, 1997.

[25] Halpern, J. Y., Vardi, M. Y.: Model Checking vs. Theorem Proving: A Manifesto, *In Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, 1991, 151–176.

[26] Holzmann, G. J.: The Model Checker SPIN, *IEEE Transactions on Software Engineering*, **23**(5), 1997, 279–295.

[27] Jaffar, J., Maher, M.: Constraint Logic Programming: a Survey, *Journal of Logic Programming*, **19-20**, 1994, 503–582.

[28] Kakas, A. C., Kowalski, R. A., Toni, F.: Abductive Logic Programming, *J. Log. Comput.*, **2**(6), 1992, 719–770.

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

[30] Kupferman, O., Piterman, N., Vardi, M. Y.: From Liveness to Promptness, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)* (W. Damm, H. Hermanns, Eds.), 4590, Springer, 2007.

[31] Larsen, K. G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell, *STTT*, **1**(1-2), 1997, 134–152.

[32] Marques-Silva, J.: Model Checking with Boolean Satisfiability, *Journal of Algorithms*, **63**(1-3), 2008, 3–16.

[33] McMillan, K. L.: Interpolation and SAT-Based Model Checking, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)* (W. A. H. Jr., F. Somenzi, Eds.), 2725, 2003.

[34] Montali, M.: *Specification and Verification of Declarative Open Interaction Models: a Logic-Based Framework*, Ph.D. Thesis, University of Bologna, 2009.

[35] Montali, M., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verification from declarative specifications using Logic Programming, *24th International Conference on Logic Programming (ICLP)* (M. G. D. L. Banda, E. Pontelli, Eds.), number 5366 in Lecture Notes in Computer Science, Springer Verlag, Udine, Italy, December 2008.

[36] Montali, M., Pesic, M., van der Aalst, W. M. P., Chesani, F., Mello, P., Storari, S.: Declarative Specification and Verification of Service Choreographies, *ACM Transactions on the Web, accepted with minor revision*, 2009.

[37] Nalepa, G.: Proposal of Business Process and Rules Modeling with the XTT Method, *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*, 2007.

[38] Naumovich, G., Clarke, L. A.: Classifying properties: an alternative to the safety-liveness classification, *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: 21st Century Applications*, ACM, 2000.

[39] Pesic, M.: *Constraint-Based Workflow Management Systems: Shifting Controls to Users*, Ph.D. Thesis, Beta Research School for Operations Management and Logistics, Eindhoven, 2008.

[40] Pesic, M., van der Aalst, W. M. P.: A Declarative Approach for Flexible Business Processes Management, *Proceedings of the BPM 2006 Workshops*, 4103, Springer, 2006.

[41] Pesic, M., Schonenberg, H., van der Aalst, W. M. P.: DECLARE: Full Support for Loosely-Structured Processes, *Proceedings of EDOC 2007*, IEEE Computer Society, 2007.

[42] Pradella, M., Morzenti, A., Pietro, P. S.: Refining Real-Time System Specifications through Bounded Model- and Satisfiability-Checking, *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, IEEE, 2008.

[43] Raimondi, F., Lomuscio, A.: Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams, *Journal of Applied Logic*, **5**(2), 2007, 235–251.

[44] Rozier, K. Y., Vardi, M. Y.: LTL Satisfiability Checking, *Model Checking Software. Proceedings of the 14th International SPIN Workshop*, 4595, Springer Verlag, 2007.

[45] Russo, A., Miller, R., Nuseibeh, B., Kramer, J.: An abductive Approach for Analysing Event-Based Requirements Specifications, *Proceedings of ICLP 2002* (P. Stuckey, Ed.), 2401, Springer Verlag, 2002, ISBN 3-540-43930-7.

[46] Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver, *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)* (W. A. H. Jr., S. D. Johnson, Eds.), 1954, 2000.

[47] Torroni, P., Chesani, F., Mello, P., Yolum, P., Singh, M. P., Alberti, M., Gavanelli, M., Lamma, E.: Modeling Interactions via Commitments and Expectations, *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models* (V. Dignum, Ed.), Information Science Reference, 2009.

[48] Xanthakos, I.: *Semantic Integration of Information by Abduction*, Ph.D. Thesis, Imperial College London, 2003, Available at `http://www.doc.ic.ac.uk/~ix98/PhD.zip`.