Detection of conflicting Compliance rules

Francois Hantry *,Mohand-Said Hacid †,Romuad Thion ‡
Université Claude Bernard Lyon 1,
LIRIS CNRS UMR 5205, France
Email: * francois.hantry, †mohand-said.hacid, ‡Romuald.thion@liris.cnrs.fr

***Abstract*—Web-based dynamic systems and pressured business environments need more than ever formal methods to analyze evolving compliance rules. Providing adequate tools to tackle the problem of debugging conflicting temporal compliance rules is an ongoing research topic. This problem is of paramount importance to achieve automatic support for early declarative design and to support evolution of rules in contract-based or service-based systems. In this paper we investigate the problem of extracting temporal unsatisfiable cores in order to detect the inconsistent part of a specification. We extend classical boolean conflict driven solver to provide a new temporal conflict driven solver for temporal logic.**

I. Introduction

Web-based dynamic systems and pressured business environments need more than ever tools to analyze evolving compliance rules. Methods for compliance requirements analysis have become critical in many computer science domains (e.g., business process management, service oriented computing, e-commerce, component-based software). Compliance rules stems from different sources: internal ones are driven by strategical or financial considerations [1]; external ones originate from cross organisational contracts [2], regulations or laws (e.g., Sarbanes-Oxley Act [3]). Formalization of compliance rules offers many appealing property for specification, planning, verification and monitoring. One relatively unexplored formal issue is the analysis of a conflicting set of temporal compliance rules. For instance, the figure 1 gives a toy set of compliance rules. It will be used as a running example in the paper. All these rules except the last one originates from an ongoing supply contract. Let us assume that the last one (r3.c) originates from another internal requirement from the supplier. It comes out that this new requirement entails a conflict with rules (3.a) and rules (3.b). Precisely, (r3.a) and (r3.b) state that the insurance must occur once the payment have occured. However, if a payment occurs, it is forbidden to subscript to any insurrance by rule (r3.c). Thus, the addition of (r3.c) leads to an inconsistency.

A conflict in rules means that no concrete business process can satisfy the declarative specification. This example shows the interest of automatic detection of conflicting subsets of compliance rules. This problem is critical for debugging declarative specification [4], [5], [6], [7], handling conflicting contract [8], or tackling unrealizable service composition [9].

- **Order**
  (r1) The process begins by an Order
- **Good and Payment**
  (r2) The occurrence of an order will lead to a good delivery and a payment.
- **Insurrance**
  - (r3.a) The purchaser will subscribe an insurrance for protection of goods during the transport.
  - (r3.b) The purchaser must have payed before any insurrance subscription.
  - (r3.c) An insurrance subscription is forbidden After payment (new requirement).

Figure 1: Purchaser-supplier compliances rules

The sample contract is depicted using the graphical formalism DecSerFlow [7] in Figure 2. DecSerFlow is a declarative specification of business process based on temporal logic as a trade-off between compliance and flexibility. The conflicting part of the specification is the relation

between *Insurance* and the *Payment* processes. It is highlighted in Figure 2 (in yellow).
Due to the dynamic flavor of the web-based modern economy, there is a need to express dynamic pattern or complex temporal pattern, such as reaction , number of occurrences, repetition, fairness, absence, (periodic) deadline or cyclic contract. There is need for efficient techniques able to handle hundreds of temporal compliance rules.

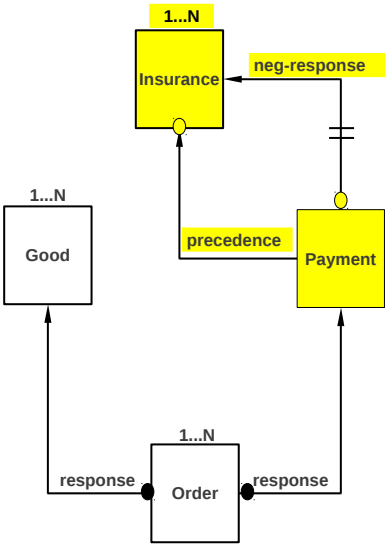

Figure 2: Conflicting DecSerFlow specification

There exist several formalisms to deal with time such as LTL, MSO [10], TLTL , MTL [11]. These logics underpine many of modern compliance languages and their associated theories and tools are used to address problems related to verification [12], [13], [14], service composition [9], goal oriented requirement analysis [4], graphical design of property patterns [15], [14], [7], [16].
We investigate the problem of efficiently extracting temporal logic unsatisfiable core for debugging compliance rules. Intuitively, an unsatisfiable core is a conflicting subset of rules. We restrict ourselves to LTL for which many results and efficient model checking methods exist. However, the problem of accurately detecting the conflicting LTL formulas has not been solved yet [17]. Conflict driven methods exist for SAT-solver algorithms. They provide quite efficient extraction of conflicting rules written in propositional logic but have not been extended to deal with the more expressive LTL.
In this paper, we propose a new temporal conflict driven solver inspired by SAT-based ones. The proof of unsatisfiability computed by the solver leads to small unsatisfiable cores, enabling debugging. Section 2 introduces some preliminaries and discusses the gap in model checking methods. Section 3 describes our design of a solver. Section 4 is devoted to the extraction of unsatisfiable cores. Section 5 provides a richer example with deadline. We conclude in Section 6.

II. TECHNICAL BACKGROUND

A. Preliminaries

Definition (Syntax of LTL)
Let P be a non empty finite set of propositional variables, and $p \in P$. A temporal logic formula is built by means of the following rules:

$$p \mid A \wedge B \mid A \vee B \mid \neg A \mid X(A)$$
$$\mid G(A) \mid F(A) \mid AUB \mid AWB$$

Definition (Semantics of LTL) A linear time structure is an element M in $(2^P)^{\mathbb{N}}$. $\forall i \in \mathbb{N}$,

- $(M,i) \vDash A$ with A in P iff $A \in M(i)$
- $(M,i) \vDash \neg A$ iff $(M,i) \nvDash A$
- $(M,i) \vDash A \wedge B$ iff $(M,i) \vDash A$ and $(M,i) \vDash B$
- $(M,i) \vDash A \vee B$ iff $(M,i) \vDash A$ or $(M,i) \vDash B$
- $(M,i) \vDash X(A)$ iff $(M,i+1) \vDash A$
- $(M,i) \vDash G(A)$ iff $\forall j \geq i, (M,j) \vDash A$
- $(M,i) \vDash F(A)$ iff $\exists j \geq i, (M,j) \vDash A$
- $(M,i) \vDash AUB$ iff $\exists j \geq i, (M,j) \vDash B$ and $\forall k, i \leq k < j, (M,k) \vDash A$
- $(M,i) \vDash AWB$ iff $\forall j \geq i, (M,j) \vDash A$ or ($\exists j \geq i, (M,j) \vDash B$ and $\forall k, i \leq k < j, (M,k) \vDash A$)

The result of the translation of our running example given figure 1 into LTL formulas is shown figure 3. It rests on the DecSerFlow translation. This language is similar to the well known property patterns [15]. For instance $\neg iWp$ (rule r3.b) means

Rules	LTL
r1	$F(o)$
r2	$G(o \Rightarrow (F(p) \wedge F(g)))$
r3.a	$F(i)$
r3.b	$(\neg i)Wp$
r3.c	$G(p \Rightarrow G(\neg i))$

Figure 3: From rules to LTL

that either the payment will not happen and the insurance subscription will never happen, either the payment happens but the insurance subscription does not occur until it occurs.

Definition (LTL SAT problem) A LTL formula ϕ is satisfiable iff there exists a linear model M such that $(M, 0) \vDash \phi$. Conversely, a LTL formula ϕ is unsatisfiable iff there is no linear model M such that $(M, 0) \vDash \phi$.

Definition (unsatisfiable core) An unsatisfiable core[1] of an unsatisfiable formula ϕ is a formula ϕ' resulting from the substitution of some subformulae of ϕ by TRUE and such that ϕ' still remains unsatisfiable.

Intuitively, a specification is satisfiable if a concrete business process can realize the specification. For instance the formula of our running example $r1 \wedge r2 \wedge r3.a \wedge r3.b \wedge r3.c$ is not satisfiable. One possible unsatisfiable core is $TRUE \wedge TRUE \wedge r3.a \wedge r3.b \wedge r3.c$. The remaining rules highlight a conflict. It is critical to find a small (or ideally a minimal) unsatisfiable core in order to detect the cause of a conflict.

Traditional techniques for satisfiability of temporal logic (e.g., [18], [19]) use tableau or Büchi automata (eg. Figure 4). A state (fullstate) is built from a prestate. A prestate is either the starting state containing only the starting formula to study either a state containing only formulas derived from the precedent state. On figure 4, the top state is a prestate, the others are fullstates. A fullstate is computed by unwinding a formula and making a choice for the disjunctive one. For instance, the formula $F(i)$ standing for the future occurrence of the insurance subscription is unwound by $i \vee XF(i)$. Similarly the occurrence of $G(\neg i)$ implies the occurrence of $\neg i$. Moreover, each disjunction leads to a choice. In Figure 4, the second state corresponds to the choice of p for the disjunction $p \vee (\neg i \wedge X(\neg iWp))$.

[1]We assume that any formula is in Negative Normal Form.

Theorem 1: A formula is satisfied iff there exists a path (finite or infinite) such that any occurrence of Future and Until modal operator fullfils its corresponding promise later (in the future) in the path.

In Figure 4, the part of the automata shows only unsatisfiable paths (infinite in this case) since each possible path contains a Future $F(i)$ but does not realize the promise i. The argument is that a path will reach a Strongly Connected Component (SCC). When reaching a SCC a path will remain in forever.

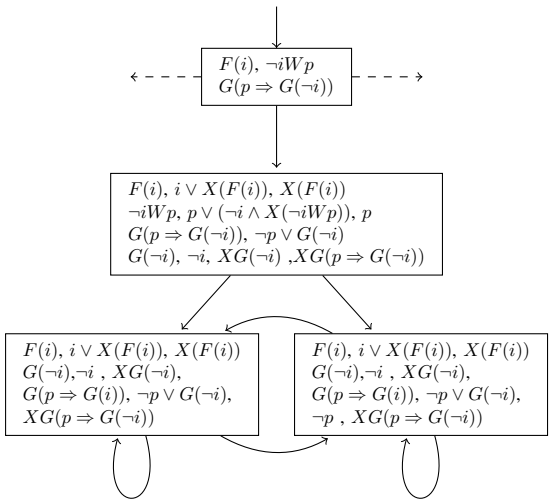

Figure 4: Deep first search model checking

B. A gap in current model checking tools

Basic local techniques use on the fly deep first search of fair loop or of strongly connected component checking the promise fulfillment. Basic global techniques use fixpoint computation over all the states of the automata [20]. Recently the use of BDD [21] and SAT-solver for Bounded Model Checking (BMC) [22] as implicit method have improved performance.

Modern model checking (eg. (BMC) [22], parallel model checking [23]) techniques efficiently provide shortest counter-examples for debugging. However, they fail to debug hundreds compliance rules containing inconsistent ones [17]. For instance, if a counter-example is found while a new rule is introduced, the business process may be modified, but there is no warranty that remaining rules would be satisfied. This is even impossible in the case of a set of inconsistent rules.

In some logics, it is usual to produce an unsatisfiable core for debugging (eg. [24], [25]). In fact, albeit it is possible to efficiently compute an unsatisfiable core for propositional logic [26], [27], or minimal inconsistent constraints for Constraint Solving Problem [28], for Sat Modulo Theory [29], or for Description Logic [30], it has been shown that efficient computation of small unsatisfiable core of a temporal logics formula is still an open problem [17]. Moreover, no modern method is devised to quickly decide if hundreds of clauses are satisfiable [31]. This lack of efficient method for computing small unsatisfiable core is mainly due to two points: (1) an explicit method suffers from an exponential blow up of automata size [31], and (2) the ad-hoc usage of boolean sat-solver for Unbounded Model Checking (UMC)[22],[32],[33] completely disregard reachability search algorithm in tableau or Büchi automata. The use of Büchi automata information for UMC is proposed in [34] to shrink the size of the input problem from a syntactic to a semantic form. Other authors (e.g., [35]) provided a similar idea but applied it only to finitely falsifiable formula. Note that boolean UMC solvers always enable revisiting a state until exploring all loop paths of a length to be determined using the size of the Buchi automata.

Only a few techniques (e.g., [32]) try to avoid revisiting states as using induction for safety and Craig interpolant (e.g., [33]). However, these last techniques assume the translation of a LTL formula to a safety problem, which is in general not efficient for big size formula [36]. In the case of the interpolant technique, the use of resolution induces the use of large memory space, which contradicts the cause for introducing DPLL [37] rather than DP solver [38].

A particular case of unsatisfiable core is vacuity checking [39]. Vacuity checking provides a Model-valid strengthened specification formula by substituting positive polarity subformula by FALSE (resp. negative subformula by TRUE) but usually needs several model checking tests. Some approaches (e.g., [40]) use Bounded Model Checking (BMC) and reuse a boolean SAT-solver method to extract cores but get only a partial result. Other approaches (e.g., [41]) extract from model checker (like Büchi automata like) a proof to detect vacuity result.

Finally, to the best of our knowledge, the author in [17] is the only one who investigated the extraction of unsatisfiable core without model. However, the provided algorithms use selector variable methods, BMC or expensive unwinding of tableau. Those methods suffer from the above mentioned drawbacks.

To summarize, explicit techniques are not efficient and implicit SAT-based techniques are ad-hoc and usually not efficient to prove properties or to reveal conflicts. However, works in the nineties [42], [43] about resolution for temporal logic provide quite good candidate method to prove unsatisfiability and extract cores. It is well known that resolution underpins classical SAT-solvers and it is considered as a mean to compute unsatisfiable cores. The main drawback of this resolution for temporal logic is that it is space consuming and the state search is backward style. However, it provides a temporal conflict analysis which is missing in modern methods and would provide a pruning state space method and a way to compute the core.
Thus, a first step is to design a temporal conflict driven solver. We then efficiently derive a small unsatisfiable core.

III. A TEMPORAL CONFLICT DRIVEN SOLVER

The temporal conflict driven solver is a combination of deep first search of SCC in automata [44] and of boolean SAT-solver. It uses unit rule propagation method, watcher techniques and classical conflict learning [45]. It also uses a new temporal conflict driven method.

A. Propagation

Unit rule propagation (DPLL) [37] is a enhancement of resolution method DP [38]. In order to reuse the unit rule propagation method (we describe later) of SAT-solver, we slightly change the unwinding of formulas in state of automata (see figure 5). The goal is to get a set of disjunctive formulas. For instance $G(\neg i)$ that stands for the absence of the insurance subscription is unwound by $G(\neg i) \Rightarrow \neg i$. We assume, for simplicity (but w.l.g), that any $\neg$ symbol of any subformulae has been pushed until reaching propositional variables (Negative Normal Form). Let Set be a set of formulas to unwind. The algorithm begins by putting the formulas of the prestate in Set. Furthermore, each formula ψ in Set is unwound following the rules of figure 5.

- If $\psi = \psi_1 \wedge .. \wedge \psi_s$ and ψ_j is not a conjunction, then add $\forall j$ the formulae $\psi \Rightarrow \psi_j$ to the state and add ψ_j to Set
- If $\psi = \psi_1 \vee \psi_2 ... \vee \psi_r$ and ψ_j is not a disjunction, then add $\psi \Rightarrow (\psi_1 \vee ... \vee \psi_r)$ to the state and add ψ_j to Set
- If $\psi = F(\psi')$ then add $\psi \Rightarrow \psi' \vee X(\psi)$ to the state and add ψ' to Set
- If $\psi = G(\psi')$ then add $\psi \Rightarrow \psi'$ and $\psi \Rightarrow X(\psi)$ to the state and add ψ' to Set
- If $\psi = (\psi')U\psi''$ then add $\psi \Rightarrow (\psi'' \vee (\psi' \wedge X(\psi)))$ to state and ψ'' and $\psi' \wedge X(\psi)$ to Set
- If $\psi = (\psi')W\psi''$ then add $\psi \Rightarrow (\psi'' \vee (\psi' \wedge X(\psi)))$ to state and ψ'' and $\psi' \wedge X(\psi)$ to Set

Figure 5: Unwinding into disjunctions

Intuitively, among the resulting formulas of the unwound state, the unit rule propagation consists in taking a non disjunctive formula f and trying to find its negation in one of the operand of a disjunctive formula. If it is found, this negation is then unsatisfiable, and temporarily 'erased'. For instance in figure 8 the occurrence of $G(\neg i)$ 'erase' the operand $\neg G(\neg i)$ in the unwound formula $G(\neg i) \Rightarrow \neg i$. While it remains only one operand in the disjunction, this 'unit' formula can be propagated and so on. Then the necessary occurrence of $XF(i)$ is entailed by the occurrence of both 'unit' formulas $F(i)$ and $\neg i$ applied to the disjunction $F(i) \Rightarrow (i \vee XF(i))$. Watcher technique uses a lazy access to formulas to boost the unit rule propagation. The principle is just to watch only two operands per disjunction.

B. Classical conflict handling

It may also happen that a path ends in a state with a formula and its negation as for instance an occurrence of an order o and the absence of it $\neg o$. In this case, the algorithm analyses the cause of the conflict as in boolean SAT-solver by backtracking along the recorded propagation of occurrences. We refer to [45] for more details about backtracking in SAT-solver. We now present the main method.

C. Basic Solver

Figure 6 shows the main method of the algorithm called Solver. We encode the nature of the state (nstate) of the automata (unwound state ($= 0$), prestate ($= 1$), prestate to unwind ($= 2$)). The solver will be, in the case ($= 2$), populated by new formulas by unwinding. In the other cases, a unit rule and classical conflict detection is launched. A classical backtrack is triggered in case of a conflict, otherwise if it is possible, a choice of formula following a heuristic is done. Once all the choices have been made (fullstate) then a SCC-search function is called. Otherwise the Solver is recursively called. The SCC-search function is similar to the computation of strongly connected components and uses deep first search numbers (see, for more details, [44]). If a SCC is found to not fulfill one promise then the function triggers an analysis of temporal conflict.

D. Temporal conflict handling

In order to avoid to revisiting the same conflict in other states the so called conflict learning [45] method is drastically used in boolean SAT-solvers. However, such analysis does not hold in SCC, since the flavor or the conflict is temporal. We propose to add a temporal conflict detection and learning (see Figure 8). While infinite unsatisfiable path is found reaching a SCC -in the figure 4 it corresponds to the both below states- the path necessarily does not fullfil a promise. Then an

```
Solver ::
if nstate=2 then
  unwind
else
  unit-rule and classical-conflict-detection
  if conflict then
    backtrack
  end if
  if  nstate = 0 then
    if no conflict then
      make a choice of literal
    end if
    if fullstate then
      SCC-search
    else
      Solver
    end if
  else
    SCC-search
  end if
end if
```

Figure 6: LTL-solver

analysis of the cause of the non fullfilment of the promise is done into the SCC. Figure 7 shows the detection algorithm of the temporal conflict and figure 8 shows the SCC[2] with the cause (depicted in color blue) of the non fullfillement of the promise of the insurance subscription i. In the SCC, the algorithm computes a backward fixpoint from the negation of the promise $\neg i$ for any states along the recorded propagations. After reaching the fixpoint containing relevant formulae (in figure 8 the multiset of $\{G(\neg i), \neg i, XG(\neg i), G(\neg i), \neg i, XG(\neg i)\}$) we can find $G(\neg i)$ as the 'cause' of the non fullfilment at the root state of the SCC. Then, the method erases the states of this SCC. Trigger of a classical backtracking at precedent state (the second state Figure 4) starting from the conflicting formulae $F(i)$ and $G(\neg i)$. We also learn the formula $\neg F(i) \vee \neg G(\neg i)$.

E. Heuristics

Our algorithm is compatible with many heuristic of SAT-solver [45]. The early choice of formula affecting many disjunctive ones is of course possible.

[2]For convenience the left-right neighbouring is now switched to a above-below one

```
Cause ::
INI: Vector= ¬Promise
while ∃e ∈ scc ∧ e not marked do
  mark e ; v = e.parents ∩ scc
  while l ∈ v ∧ l not marked  do
    Vector.push(l)
  end while
end while
learn( Vector, promise)
erase SCC
Backtrack
```

Figure 7: Analysis of the temporal conflict

Classical disjunctive subsumption is possible and purging of learned disjunction is also allowed while space problem happens. To quickly find a temporal conflict (eg. on $F(i)$), we first choose the promise i rather than postponing it to a strict future $XF(i)$.

F. Correctness and completeness of the algorithm

Theorem 2: The algorithm terminates, it is correct and complete.

sketch of the proof:

Our algorithm is similar to a deep first search of satisfiable SCC in a LTL tableau as in [19]. The main difference is that we prune the state space by learning boolean and temporal conflict. Another important issue is the use of globally learned disjunction (holding at any state of the tableau or any time). On the contrary to boolean learned formula [3] *which holds only in the presence of original clauses, the learned formula of the algorithm records all the necessarily formulas to hold globally.*

IV. Extraction of unsatisfiable core

In order to compute unsatisfiable core from the algorithm we reuse the idea of [46]. The method is based on the following notifications: (1) the existence of a learned formula which is learned from boolean conflict is the set of formulas involved while backtracking and learning, (2) the existence of learned formula which is learned from temporal conflict is, for each state of the corresponding SCC, the set of formulas in the state computed while

[3]These formulas are disjunction of literals and called clauses

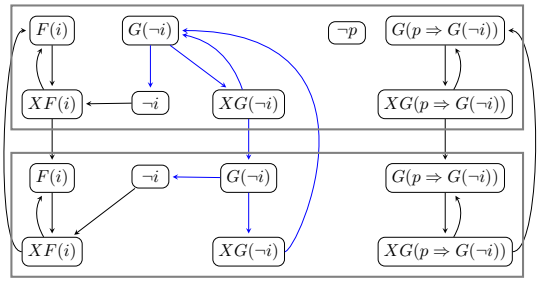

Figure 8: Detection of conflicting subformulae

the backward fixpoint computation. Recording the history of all these implications as in [46], we can later efficiently extract, by backwarding from the terminal conflict[4], original formulas that are involved in the unsatisfiability (see Figure 9). If ϕ

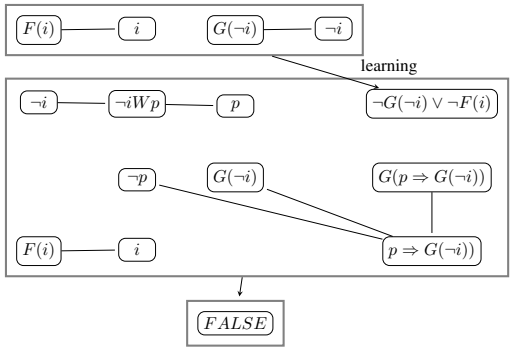

Figure 9: Conflicting subformulae

is an unsatisifiable formula, S the set of involved formulae[5], the subformulae f of ϕ occuring in S or being an operand of a disjunction of S are exactly those subformulae involved in the conflict. We return the corresponding unsatisfiable core formula.

$$\phi' = \phi(f \leftarrow TRUE, f \notin S \cup op_{\vee}(S))$$

In Figure 9, a rectangle represents a conflict and contains the involved subformulae and learned formulae. The arrow shows learning link between conflict and formulae. Each line between subformulae of ϕ shows a relation of subformulae. i

[4]the algorithm terminates on a conflict iff the formula is unsatisfiable

[5]S may contain unwound formulae

Conflicting Rules	LTL core
	T
	T
r3.a	$F(i)$
r3.b	$(\neg i)Wp$
r3.c	$G(p \Rightarrow G(\neg i))$

Figure 10: Conflicting rules and LTL core

occurs because it is a subformula of $F(i)$ and because it is an operand of the unwound formula $F(i) \Rightarrow i \vee XF(i)$ involved in the temporal conflict. The above rectangle represents the involved subformulae in the first temporal conflict of our running example figure 8. The second rectangle shows a second temporal conflict (not shown in our running example). This last conflict uses the learned formula $\neg G(\neg i) \vee \neg F(i)$ required by the first conflict. The second conflict leads finally to a contradiction. By backwarding from FALSE, we can extract all the subformulae involved in the conflict of the specification. Our method provides an unsatifiable core of the running example figure 10. The algorithm can also deal with deadline and provide fine unsatisfiable core as shown in the following section.

V. Complex example

We now present an example of debugging a complex contract with deadline. The contract Figure 11 specifies a composite business process. Some underpining temporal patterns are classical as reaction or deadline.

However the rule (c9) is not usually found in current property pattern. This a so called weak fairness. Intuitively, It means that until an event, it is possible to trigger infinitely event. Such pattern can be found in [3]. We obtain the translation in LTL figure 12.

Running our solver on this example allows to extract the following unsatisfiable core figure 13. Even if additional clauses are added, the result remains the same.

The unsatisfiable core explains that inconsistency is involved by rule (c1), (c2), (c4), (c5), (c7). Further, only assuming the occurrence of a payment without information about the non-repudiation is

- (c0) An order must occur
- (c1) A payment with non-repudiation must occcur
- (c2) An insurance submitting must occur
- (c3) A good delivery must occur.
- (c4) Insurance before payment is forbidden.
- (c5) If a payment occurs, it must occur only after the third day from the order.
- (c6) Delivery before Payment is forbidden .
- (c7) After two days from the order the Insurance subscription is forbidden
- (c8) A Golden customer must receive items before three days from the time the payment is accomplished.
- (c9) From the order, customer can change multiple times its order until good delivery

Figure 11: Complex contract

Rule	LTL translation
c0	$F(o)$
c1	$F(p \wedge nr)$
c2	$F(i)$
c3	$F(g)$
c4	$(\neg i)W(p)$
c5	$(\neg p)W(o \wedge \neg p \wedge X(\neg p) \wedge XX(\neg p))$
c6	$(\neg g)W(p)$
c7	$G(o \Rightarrow XX(G(\neg i))$
c8	$gold \Rightarrow (G(p \Rightarrow (Xg \vee XX(g)))$
c9	$G(o \Rightarrow ((ch? \Rightarrow F(o))Ug))$

Figure 12: Complex LTL rules

sufficient to imply a conflict. Thus, the algorithm enables to extract precise subconstraint into a rule.

VI. Conclusion

In order to detect which compliance rules are conflicting, we have provided a temporal conflict driven solver for LTL. We have shown how to extract unsatisfiable core from it. We apply our method to debug a DecSerFlow example and also to debug a contract with deadline. Detecting conflicts in rules is critical for human interactive contract management systems. Moreover, our method pinpoints temporal issues in any automatic tool which is sensitive to the consistency of many

C-Rule	LTL core
	T
c1	$F(p \wedge T)$
c2	$F(i)$
	T
c4	$(\neg i)W(p)$
c5	$(\neg p)W(o \wedge \neg p \wedge X(\neg p) \wedge XX(\neg p))$
	T
c7	$G(o \Rightarrow XX(G(\neg i))$
	T
	T

Figure 13: Conflicting rules

evolving heterogeneous policies such as regulatory laws, internal business rules, security or privacy. Part of our future work is to use the method in order to highlight which part (eg. task, logical constraints...) of an ongoing business process (given in language such as BPEL or BPMN) is conflicting with a given rule. This is actually not provided by getting a counter example, as it is done in current compliance checking.

Temporal logics may be mixed with deontic logics for contract formalization [47], [48], [8]. The extension of our method to deontic modality used in contracts appears straightforward, and we are now focusing on this issue. Although LTL express deadline properties, it lacks succinteness. On the contrary real time temporal logic use region abstraction to get efficient result. Future work will tackle the problem of unsatisfiable core for real time logic. Another interesting issue is to enhance the performance by using parallel processing.

References

[1] R. G. Ross, "Expressing business rules," in *SIGMOD Conference*, 2000, pp. 515–516.

[2] Z. Milosevic, S. W. Sadiq, and M. E. Orlowska, "Towards a methodology for deriving contract-compliant business processes," in *Business Process Management*, 2006, pp. 395–400.

[3] United States Code, "Sarbanes-oxley act of 2002, pl 107-204, 116 stat 745," Codified in Sections 11, 15, 18, 28, and 29 USC, July 2002.

[4] A. Fuxman, J. Mylopoulos, M. Pistore, and P. Traverso, "Model checking early requirements specifications in tropos," in *RE*, 2001, pp. 174–181.

[5] M. P. Papazoglou and B. Kratz, "A business-aware web services transaction model," in *ICSOC*, 2006, pp. 352–364.

[6]

[7] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari, "Declarative specification and verification of service choreographiess," *TWEB*, vol. 4, no. 1, 2010.

[8] S. Fenech, G. J. Pace, and G. Schneider, "Automatic conflict detection on contracts," in *ICTAC*, 2009, pp. 200–214.

[9] A. Marconi and M. Pistore, "Synthesis and composition of web services," in *SFM*, 2009, pp. 89–157.

[10] E. A. Emerson., "Temporal and modal logic," *Handbook of theoretical computer science*, vol. B : formal models and semantics, 1990.

[11] R. Alur and T. A. Henzinger, "Logics and models of real time: A survey," in *REX Workshop*, 1991, pp. 74–106.

[12] A. Awad, G. Decker, and M. Weske, "Efficient compliance checking using bpmn-q and temporal logic," in *BPM*, 2008, pp. 326–341.

[13] A. Ghose and G. Koliadis, "Auditing business process compliance," in *ICSOC*, 2007, pp. 169–180.

[14] K. Xu, Y. Liu, and C. Wu, "Bpsl modeler - visual notation language for intuitive business property reasoning," *Electr. Notes Theor. Comput. Sci.*, vol. 211, pp. 211–220, 2008.

[15] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*, 1999, pp. 411–420.

[16] C. Giblin, A. Y. Liu, S. Müller, B. Pfitzmann, and X. Zhou, "Regulations expressed as logical models (realm)," in *JURIX*, 2005, pp. 37–48.

[17] V. Schuppan, "Towards a notion of unsatisfiable cores for ltl," in *FSEN*, 2009, pp. 129–145.

[18] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *PSTV*, 1995, pp. 3–18.

[19] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli, "A decision algorithm for full propositional temporal logic," in *CAV*, 1993, pp. 97–109.

[20] O. G. Edmunds Clarke and D. A. Peled, "Model checking," *MIT Press*, 1999.

[21] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking:10^20 states and beyond," *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992.

[22] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using sat procedures instead of bdds," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, ser. DAC '99. New York, NY, USA: ACM, 1999, pp. 317–320. [Online]. Available: http://doi.acm.org/10.1145/309847.309942

[23] J. Barnat, L. Brim, and P. Rockai, "Scalable multi-core ltl model-checking," in *SPIN*, 2007, pp. 187–203.

[24] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, ser. GLSVLSI '08. New York, NY, USA: ACM, 2008, pp. 77–82. [Online]. Available: http://doi.acm.org/10.1145/1366110.1366131

[25] E. Torlak, F. S.-H. Chang, and D. Jackson, "Finding minimal unsatisfiable cores of declarative specifications," in *FM*, 2008, pp. 326–341.

[26] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable boolean formula." in *In Prelim. Proc. Sixth Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.

[27] I. Lynce and J. P. M. Silva, "On computing minimum unsatisfiable cores," in *SAT*, 2004.

[28] É. Grégoire, B. Mazure, and C. Piette, "On finding minimally unsatisfiable cores of csps," *International Journal on Artificial Intelligence Tools*, vol. 17, no. 4, pp. 745–763, 2008.

[29] A. Cimatti, A. Griggio, and R. Sebastiani, "A simple and flexible way of computing small unsatisfiable cores in sat modulo theories," in *SAT*, 2007, pp. 334–339.

[30] S. Schlobach and R. Cornet, "Non-standard reasoning services for the debugging of description logic terminologies," in *IJCAI*, 2003, pp. 355–362.

[31] K. Y. Rozier and M. Y. Vardi, "Ltl satisfiability checking," *STTT*, vol. 12, no. 2, pp. 123–137, 2010.

[32] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *FMCAD*, 2000, pp. 108–125.

[33] K. L. McMillan, "Interpolation and sat-based model checking," in *CAV*, 2003, pp. 1–13.

[34] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman, "Computational challenges in bounded model checking," *STTT*, vol. 7, no. 2, pp. 174–183, 2005.

[35] R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M. Y. Vardi, "Efficient ltl compilation for sat-based model checking," in *ICCAD*, 2005, pp. 877–884.

[36] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded ltl model checking," *CoRR*, vol. abs/cs/0611029, 2006.

[37] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962. [Online]. Available: http://doi.acm.org/10.1145/368273.368557

[38] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–215, July 1960. [Online]. Available: http://doi.acm.org/10.1145/321033.321034

[39] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *STTT*, vol. 4, no. 2, pp. 224–233, 2003.

[40] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik, "Exploiting resolution proofs to speed up ltl vacuity detection for bmc," in *FMCAD*, 2007, pp. 3–12.

[41] K. S. Namjoshi, "An efficiently checkable, proof-based formulation of vacuity in model checking," in *CAV*, 2004, pp. 57–69.

[42] M. Fisher, "A resolution method for temporal logic," in *IJCAI*, 1991, pp. 99–104.

[43] C. Dixon, M. Fisher, and H. Barringer, "A graph-based approach to resolution in temporal logic," in *ICTL*, 1994, pp. 415–429.

[44] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[45] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *DAC*, 2001, pp. 530–535.

[46] L. Zhang and S. Malik, "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications," in *DATE*, 2003, pp. 10 880–10 885.

[47] J. Broersen, F. Dignum, V. Dignum, and J.-J. C. Meyer, "Designing a deontic logic of deadlines," in *DEON*, 2004, pp. 43–56.
[48] F. Dignum and R. Kuiper, "Combining dynamic deontic logic and temporal logic for the specification of deadlines," in *HICSS (5)*, 1997, pp. 336–346.