Plausibility Checking of Formal Business Process Specifications in Linear Temporal Logic

Christoph Czepa[1], Huy Tran[1], Uwe Zdun[1],
Thanh Tran Thi Kim[2], Erhard Weiss[2], and Christoph Ruhsam[2]

[1] University of Vienna, Faculty of Computer Science, Software Architecture Research Group,
Währingerstraße 29, 1090 Vienna, Austria
{christoph.czepa,huy.tran,uwe.zdun}@univie.ac.at
[2] Isis Papyrus Europe AG, Alter Wienerweg 12, 2344 Maria Enzersdorf, Austria
{thanh.tran,erhard.weiss,christoph.ruhsam}@isis-papyrus.com

Abstract. Many approaches for keeping business processes in line with constraints stemming from various sources (such as laws and regulations) are based on Linear Temporal Logic (LTL). Creating LTL specifications is an error-prone task which entails the risk that the formula does not match the intention of its creator. This paper proposes a *plausibility checking* approach for LTL-based specifications. The proposed approach can provide confidence in an LTL formula if plausibility checking is passed. If the formula does not pass the plausibility checks, a counterexample trace and the truth values of both the LTL formula and the plausibility specification are generated and can be used as a starting point for correction.

1 Introduction

Keeping business processes in line with requirements stemming from various sources (e.g., laws, regulations, standards, internal policies, best practices) has become an important research field due to increasing flexibility demands in business process management, especially in knowledge-intensive environments. In recent years, both academia and industry are working towards solutions for enabling the flexible handling of business processes while providing support to meet necessary requirements. Two closely related categories of such supporting approaches have been extensively investigated. Firstly, there are compliance enabling approaches developed for checking semantic constraints (also called compliance rules) at runtime (e.g., [13]) and design time (e.g., [3]) of business processes. Secondly, constraint-based business processes models (also called declarative workflows) are defined in which a set of constraints is used to describe the business process and these constraints become the basis for the enactment of the process. A prominent example for the declarative workflow approach is Declare [16] which provides a graphical front end with mappings to Linear Temporal Logic (LTL) [17] as underlying formalism[3]. Temporal logics, such as LTL and CTL (Computation Tree

[3] The graphical notations of Declare can also be translated to other formalisms, such as Event Calculus [15].

In: S. España, M. Ivanović, M. Savić (eds.): Proceedings of the CAiSE'16 Forum at the 28th International Conference on Advanced Information Systems Engineering, Ljubljana, Slovenia, 13-17.6.2016, published at http://ceur-ws.org

Logic), are established ways to describe desired system properties for verification. Especially LTL has become a *de facto* standard for defining system specifications due to its extensive use in model checking (e.g., [18]) and the possibility to automatically translate LTL formulas to nondeterministic finite automata (NFA) for runtime verification on finite traces [6].

The creation of LTL formulas is a challenging and error-prone task that requires considerable knowledge of and experience with LTL. It is hardly surprising that higher levels of abstraction, such as the property specification patterns proposed by Dwyer et al. [7], are often preferred over authoring new LTL formulas. There are two major issues when solely trying to rely on a pattern-based approach. Firstly, formal patterns that precisely match the intention of the user might not be available. Hence, manually defining the constraint by modifying or combining existing patterns or by creating a new specialized LTL formula might be required. Secondly, if an existing candidate pattern has been identified, how can the user be sure that her or his intention is really met by the pattern? Either the meaning of the pattern could be misinterpreted or the LTL formula might contain errors. Such problems in the specifications could result in severe consequences (e.g., legal issues due to the violation of compliance requirements). Thus, it is highly important to provide better support for creating correct specifications.

Plausibility checking aims at supporting technical users during the creation process of LTL formulas and at increasing the confidence in LTL formulas by ensuring that an LTL specification matches the user's intention. Whenever an LTL formula is created or modified, the user also creates a plausibility specification which is used to check whether an LTL formula is contradictory to this plausibility specification. The approach performs reasoning on finite traces by Complex Event Processing (CEP)—with plausibility specifications encoded as Temporal Queries (TQs) on top of Event Processing Language (EPL)—and Nondeterministic Finite Automata (NFA)—representing LTL formulas. We discuss the practical use of our approach by the following scenario: The implementation of a new constraint pattern stemming from an EPA (Environmental Protection Agency) compliance document [9].

2 Related Work

To the best of our knowledge, only very few studies exist on keeping LTL formulas in line with the users' understanding of the formula. Salamah et al. [19] propose to use a set of manually created test cases to check the plausibility of pattern-generated LTL formulas. However, this involves the user in the generation process of all the sample traces and the expected truth values at the end of this traces. As a result, the count of test cases remains marginal because the manual specification of test cases is time-consuming. Yan et al. [23] claim to keep natural language requirements and their corresponding formulas consistent by translating specifications in structured English grammar to LTL, mainly by mapping English words to LTL operators. While the approach provides relief for specifying discrete time properties, the direct mapping of LTL operators to words does not really simplify the creation process of LTL specifications. Thus, there is still the risk to create formulas that contradict the actual intention of the creator.

Other plausibility checking approaches do not focus on the consistency between the users' intention and its actual formal representation in LTL, but check the internal consistency of LTL formulas (e.g., Barnat et al. [5]). Vacuity detection is concerned with avoiding tautologies and subformulas that are not relevant for the satisfaction of the formula (e.g., Simmonds et al. [20]). Consistency checking of LTL formulas means finding contradicting parts of a formula or contradictions in sets of formulas (e.g., compliance rule collections) that are generally unsatisfiable (e.g., Awad et al. [4]). While these plausibility checking approaches are also very important, their focus is entirely different from the approach presented in this paper.

3 Temporal Queries (TQs)

We devise Temporal Queries (TQs) as an abstraction layer on top of EPL (Event Processing Language) [10]—an event query language— for supporting plausibility checking. A temporal query is of the form $e \Rrightarrow r$ where e is an temporal expression and r is a truth value. The expression formed by the operator $\Rrightarrow$ implies that there is a change of the truth value caused by the temporal expression e and the resulting truth value is r which has one of the following states:

- $\bot$ stands for *temporarily violated*,
- $\top$ stands for *temporarily satisfied*,
- $\bot_{\mathcal{P}}$ stands for *permanently violated*,
- $\top_{\mathcal{P}}$ stands for *permanently satisfied*.

The aforementioned set of states is used in Declare [14] and other automata-based approaches [6] and also proposed for runtime verification of LTL formulas on finite traces in general [12]. Thus, we adopt these states as they are sufficient for our plausibility checking approach. Once a permanent state is reached for a constraint instance, it cannot be left anymore.

A temporal expression e can contain events and the following expressions:

- $e_1 \rightsquigarrow e_2$, where both e_1 and e_2 contain at least one non-negated event, means e_1 *eventually leads to* e_2 ('$\rightsquigarrow$' is equivalent to the '->' operator in EPL),
- $\neg e_1 \; \mathcal{U} \; e_2$, where both e_1 and e_2 contain at least one non-negated event, means e_1 *does not occur until* e_2 *eventually occurs* ('$\mathcal{U}$' is equivalent to the '$until$' operator in EPL).

Furthermore, a temporal expression e can contain the following boolean operators and quantifiers:

- $\neg e$ is a negation (*not e*), initialized as true and changing to false once e occurs,
- $e_1 \wedge e_2$ is a logical conjugation (e_1 *and* e_2),
- $e_1 \vee e_2$ is a logical disjunction (e_1 *or* e_2),
- $\forall e$ means *for every e* ('$\forall$' is equivalent to the '$every$' operator in EPL).

For example, the query $\forall(a \rightsquigarrow b) \Rrightarrow \top$ implies a truth value change to *temporarily satisfied* for every time event a is followed by event b. Without the $\forall$ quantifier, the query

$(a \rightsquigarrow b) \Rrightarrow \top$ observes just the first occurrence where *a* leads to *b* and stops thereafter. The following query $a \rightsquigarrow \forall(b \rightsquigarrow c) \Rrightarrow \top$ will denote a change to *temporarily satisfied* for every occurrence of b leads to c after the first occurrence of a. Another query $\forall(a \rightsquigarrow \neg d \wedge \neg b \,\mathcal{U}\, c \rightsquigarrow d) \Rrightarrow \bot_P$ indicates a change to *permanently violated* when c is not preceded by b in between every occurrence of a and d.

4 Plausibility Checking Approach

Authoring LTL formulas usually starts with thinking about some kind of constraint or requirement in natural language. In this case, the user intends to create an LTL formula that matches the description in natural language. Alternatively, a natural language description and the corresponding LTL formula could already be existent and the business user is interested in finding out whether the LTL formula is a plausible representation of the natural language description. For the creation of plausibility specifications, we propose to use Temporal Queries (TQs) which are a way for specifying truth value changes while observing finite traces, like the execution trace of a business process instance. Additionally to the creation of the TQs, it is necessary to define the initial truth value of the specification (i.e., temporarily satisfied or temporarily violated) otherwise the truth value of the plausibility specification would be undefined until a TQ causes a truth value change. Up to four TQs, namely one for each possible truth value state ($\bot$, $\top$, $\bot_{\mathcal{P}}$, and $\top_{\mathcal{P}}$), and an initial truth value (either $\top$ or $\bot$) are defined to represent a *plausibility specification*.

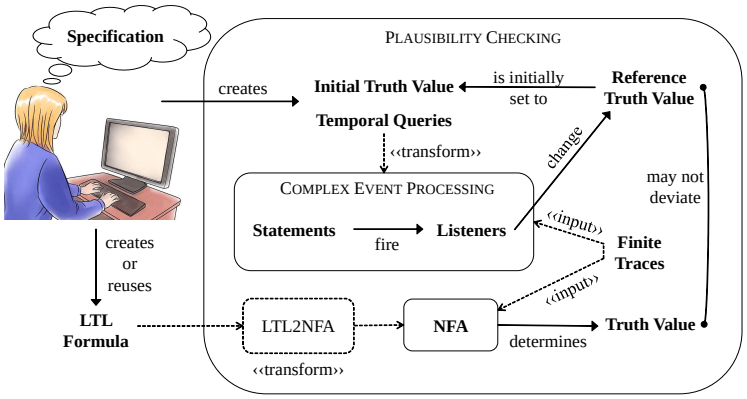

Fig. 1. Approach Overview

An overview of *Plausibility Checking* is shown in Figure 1. Plausibility checking requires two inputs, namely an LTL formula and its corresponding plausibility specifications. The LTL formula is then transformed into a nondeterministic finite automaton (NFA). The plausibility specifications consist of an initial truth value and TQs. The TQs are transformed to *event query statements* and *listeners* that can be used by a Complex Event Processing (CEP) engine.

In particular, there can be up to four listeners, namely, one for each runtime verification state. Once a *permanent* state is reached, the truth value becomes immutable. Both the NFA and CEP receive inputs which are the elements of finite traces. These inputs will lead to changes of both the *Truth Value* and the *Reference Truth Value*. A change of the *Reference Truth Value* of the plausibility specification occurs once a listener is triggered because the temporal query matches the current trace. The *Truth Value* reflects the current acceptance state of the automaton.

In order to achieve a positive plausibility checking result, there must not be any deviation between the *Truth Value* and the *Reference Truth Value* for all inputs. A large set of test cases can be created and checked automatically. There are two options: *Option 1:* All words over the alphabet of the NFA having sufficient length can be used as inputs. A moderate maximum trace length between 7 and 10 is in most cases we encountered so far already sufficient. *Option 2:* Only a subset of traces with greater maximum size is created randomly and checked automatically as well. The alphabet of the automaton always consists of the variables of the LTL formula and a single additional variable that functions as a surrogate for all other variables that are not part of the LTL formula. If the formula does not meet the plausibility specifications, a counterexample trace and the truth values of both the LTL formula and the plausibility specification are being made available as a starting point for the correction of the LTL formula. The approach has been fully implemented in a prototype which makes use of the open source CEP engine *Esper* [11] and the `LTL2NFA` algorithm [6].

In this way, the plausibility checking can be leveraged for aiding users during the creation of a new constraint patterns as well as for analyzing existing patterns to gain confidence in the proposed LTL representation of the pattern.

5 Running Example

Pattern collections, most notably the Property Specification Patterns by Dwyer et al. [7], contain a large number of patterns that fit in many cases. However, such collections are far from being complete. If no suitable pattern is available to realize a certain constraint, a new formula must be created. We are now going to illustrate this scenario by a practical example extracted from our prior research work related to capturing and formalizing real-world compliance requirements [21].

Let us consider a compliance guidance document [9] published by the United States Environmental Protection Agency (EPA) regarding buildings built before or after 1978—since then lead-based paints are prohibited residential and public buildings in USA—that states "*In housing built before 1978, you must: Distribute EPA's lead pamphlet [...] to the owner and occupants before renovation starts.*". This rule involves three tasks, namely, checking whether the house is built before 1978, distributing EPA's lead pamphlets, and starting the renovation process. It makes sense to distribute the lead pamphlet in case the house was built before 1978. Hence, a confirmation that the housing was built before 1978 must *coexist* with the distribution of lead pamphlets in the same process instance *before* the renovation is started. Now we are facing the problem that the pattern catalog by Dwyer et al. [7] does not introduce *Coexistence* patterns. The declarative workflow approach Declare [16], as another important source of patterns,

does offer a Coexistence pattern but only in the *global-* instead of the needed *before-*scoped variant. To the best of our knowledge, the *Coexistence Before* pattern does not exist, thus it must be newly created. During this process, we will leverage the proposed plausibility checking approach.

A brief description of the pattern in natural language, such as *before c: a coexists with b*, outlines what we aim for. At first, we create appropriate *Temporal Queries (TQs)* for the pattern. On the one hand, we need a TQ $(a \rightsquigarrow b \rightsquigarrow c) \vee (b \rightsquigarrow a \rightsquigarrow c) \Rrightarrow \top_{\mathcal{P}}$ that turns the truth value to *permanently satisfied* and another TQ $(\neg c \wedge \neg b\, \mathcal{U}\ a \rightsquigarrow \neg b\, \mathcal{U}\ c) \vee (\neg c \wedge \neg a\, \mathcal{U}\ b \rightsquigarrow \neg a\, \mathcal{U}\ c) \Rrightarrow \bot_{\mathcal{P}}$ that turns the truth value to *permanently violated*. Creating these formulas is relatively straightforward. The pattern becomes permanently satisfied when a and b, in any order, occur before c. A permanent violation of the pattern occurs if one of the following conditions is satisfied.

- there is no c and no b until a occurs and thereafter is again no b until c occurs.
- there is no c and no a until b occurs and thereafter is again no a until c occurs.

On the other hand, we need to create the LTL formula. As soon as we believe that the LTL formula corresponds to the meaning of the pattern, we can run the plausibility check. For example, when we perform plausibility checking on $\mathcal{F}\ c \rightarrow ((\neg c\, \mathcal{U}\ a) \wedge (\neg c\, \mathcal{U}\ b))$, our approach reports a plausibility issue in relation with the trace $[c]$. The plausibility specification is still in its initial state, namely *temporarily satisfied* while the LTL formula indicates already a violation. We intended that the formula becomes only violated if either *a* exists before *c* but *b* does not, or *b* exists before *c* but *a* does not. Since this is not the case for the trace $[c]$, the plausibility checking approach correctly identifies an issue. Hence, we must revise the LTL formula. Eventually, we come up with the formula $(\mathcal{F}\ c \rightarrow ((\neg c\, \mathcal{U}\ a) \rightarrow (\neg c\, \mathcal{U}\ b))) \wedge (\mathcal{F}\ c \rightarrow ((\neg c\, \mathcal{U}\ b) \rightarrow (\neg c\, \mathcal{U}\ a)))$ that passes the plausibility checks. Therefore, we have found a plausible representation of the pattern in LTL. Now, we can encode the compliance requirement in LTL as

$$(\mathcal{F}\ \text{“Renovation started”} \rightarrow ((\neg\text{“Renovation started”}\ \mathcal{U}\ \text{“Housing build before 1978 confirmed”}) \rightarrow (\neg\text{“Renovation started”}\ \mathcal{U}\ \text{“Distribute pamphlet finished”})))$$
$$\wedge\ (\mathcal{F}\ \text{“Renovation started”} \rightarrow ((\neg\text{“Renovation started”}\ \mathcal{U}\ \text{“Distribute pamphlet finished”}) \rightarrow (\neg\text{“Renovation started”}\ \mathcal{U}\ \text{“Housing build before 1978 confirmed”}))).$$

To complete the formalization of the compliance rule, it is additionally required that an execution of the *Housing build before 1978* task happens before the start of the renovation process. This can be realized by using the already existing *Precedence Global* pattern [1].

6 Discussion

Although the motivation and reason for proposing plausibility checking is related to business process management, the approach is applicable to other domains, such as the verification of software in general. The current plausibility checking approach assumes a single state per instant of time which has been sufficient for plausibility checking

problems that we encountered by now. Multiple states per instant are currently not considered due to the resulting exponential blowup of traces. Even if working with a single state per instant, the main drawback of the approach is the exponential growth of the number of traces. It is, however, in the scenarios we encountered so far, sufficient to work with moderate maximum trace lengths because the count of propositional variables that are present in most semantic constraints is usually low (cf. [8]) and most issues are already discoverable in short traces (cf. Section 6). Alternatively, if LTL formulas involve a high count of propositional variables, the proposed approach still can perform a large quantity of plausibility checks with randomly generated longer traces, which is an improvement over generating only a few test cases manually.

7 Conclusion and Future Work

This paper proposes an approach for plausibility checking of LTL specifications based on Nondeterministic Finite Automata (NFA) and Complex Event Processing (CEP). The approach has been discussed in the context of a practical scenario, namely the creation of a new constraint pattern stemming from a compliance document. Existing pattern-based approaches, such as Declare, can benefit from our approach. Whenever it becomes necessary to extend the set of supported constraints, our approach can enable support. Not only runtime verification or declarative workflow techniques can benefit but also design time approaches since specifications for model checking are often encoded in LTL. In our future work, we plan to further evaluate the approach through performance, scalability and user experiments and to apply our approach to other formalisms that have, for example, a notion of quantitative time [2].

Acknowledgement. The research leading to these results has received funding from the FFG project CACAO, no. 843461 and the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001. This paper contains an image licensed under CC [22].

References

[1] Property Pattern Mappings for LTL. `http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`, last accessed: May 18, 2016

[2] Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. Software Engineering 41(7), 620–638 (July 2015)

[3] Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In: 6th International Conference on Business Process Management (BPM). pp. 326–341. Springer, Berlin, Heidelberg (2008)

[4] Awad, A., Weidlich, M., Weske, M.: Consistency checking of compliance rules. In: Business Information Systems, vol. 47, pp. 106–118. Springer (2010)

[5] Barnat, J., Bauch, P., Brim, L.: Checking sanity of software requirements. In: Software Engineering and Formal Methods, LNCS, vol. 7504, pp. 48–62. Springer (2012)

[6] De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on ltl on finite traces: Insensitivity to infiniteness. In: AAAI. pp. 1027–1033. AAAI Press (2014)
[7] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: 21st International Conference on Software Engineering (ICSE). pp. 411–420. ACM (1999)
[8] Elgammal, A., Turetken, O., van den Heuvel, W.J., Papazoglou, M.: Formalizing and applying compliance patterns for business process compliance. Software & Systems Modeling pp. 1–28 (2014)
[9] EPA: Small Entity Compliance Guide to Renovate Right. http://www2.epa.gov/sites/production/files/documents/sbcomplianceguide.pdf, last accessed: May 18, 2016
[10] EsperTech Inc.: EPL Reference. http://www.espertech.com/esper/release-5.1.0/esper-reference/html/event_patterns.html, last accessed: May 18, 2016
[11] EsperTech Inc.: Esper. http://www.espertech.com/esper/, last accessed: May 18, 2016
[12] Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Logics of Programs. pp. 196–218. Springer (1985)
[13] Ly, L.T., Maggi, F.M., Montali, M., Rinderle-Ma, S., van der Aalst, W.M.: Compliance monitoring in business processes: Functionalities, application, and tool-support. Information Systems 54, 209 – 234 (2015)
[14] Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.P.: Runtime verification of ltl-based declarative process models. In: Second International Conference on Runtime Verification (RV). pp. 131–146. Springer (2012)
[15] Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring business constraints with the event calculus. ACM Trans. Intell. Syst. Technol. 5(1), 17:1–17:30 (Jan 2014)
[16] Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: BPM Workshops. pp. 169–180. Springer (2006)
[17] Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on. pp. 46–57 (Oct 1977)
[18] Rozier, K.Y.: Survey: Linear temporal logic symbolic model checking. Comput. Sci. Rev. 5(2), 163–203 (May 2011)
[19] Salamah, S., Gates, A., Roach, S., Mondragon, O.: Verifying pattern-generated ltl formulas: A case study. In: Model Checking Software, LNCS, vol. 3639, pp. 200–220. Springer (2005)
[20] Simmonds, J., Davies, J., Gurfinkel, A., Chechik, M.: Exploiting resolution proofs to speed up ltl vacuity detection for bmc. International Journal on Software Tools for Technology Transfer 12(5), 319–335 (2010)
[21] Tran, T., Weiss, E., Ruhsam, C., Czepa, C., Tran, H., Zdun, U.: Embracing process compliance and flexibility through behavioral consistency checking in acm: A repair service management case. In: AdaptiveCM 15 (August 2015)
[22] Wikiphoto: http://www.wikihow.com/Become-a-Software-Engineer, licensed under: http://creativecommons.org/licenses/by-nc-sa/3.0/ Last accessed: May 18, 2016
[23] Yan, R., Cheng, C.H., Chai, Y.: Formal consistency checking over specifications in natural languages. In: DATE '15. pp. 1677–1682 (2015)