A Verified Approach to Checking Real-Time Patterns on Fiacre Programs

Nouha Abid, Silvano Dal Zilio

▶ **To cite this version:**

Nouha Abid, Silvano Dal Zilio. A Verified Approach to Checking Real-Time Patterns on Fiacre Programs. Doctoral Symposium of FM 2012, Aug 2012, Paris, France. hal-01790120

HAL Id: hal-01790120
https://laas.hal.science/hal-01790120

Submitted on 11 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Verified Approach to Checking Real-Time Patterns on Fiacre Programs

Nouha Abid[1,2] and Silvano Dal Zilio[1,2]

[1] CNRS ; LAAS ; 7 avenue colonel Roche, F-31400 Toulouse, France
[2] Université de Toulouse; Toulouse, France

Abstract. An issue limiting the adoption of model-checking technologies by the industry is the ability, for non-experts, to express their requirements using the property languages supported by verification tools. This has motivated the definition of dedicated assertion languages for expressing temporal properties at a higher level. However, only a limited number of these formalisms support the definition of timing constraints and even fewer of them take into account the complexity of checking properties on a system. In this paper, we propose a complete framework that includes: the definition of timed patterns; a tool chain for checking timed properties; and methods to prove the correctness of our verification approach.

1 Introduction

An issue limiting the adoption of model-checking technologies by the industry is the difficulty, for non-experts, to express their requirements using the specification languages supported by the verification tools. Indeed, there is often a significant gap between the boilerplates used in requirements statements and the low-level formalisms used by model-checking tools; the latter usually relying on temporal logic. This limitation has motivated the definition of dedicated assertion languages for expressing properties at a higher level. However, only a limited number of assertion languages support the definition of timing constraints and even fewer are associated to an automatic verification tool, such as a model-checker. Moreover, few works consider the problem of checking the correctness of the verification approach. Who check the model-checkers!

In this paper, we propose a complete framework that includes: the definition of timed patterns; an approach for checking timed properties; and methods for proving the correctness of our verification approach. We have integrated our framework in a toolchain for Fiacre [5], a formal modelling language for real-time, reactive systems. Our first contribution is the definition of a set of patterns that extends the specification language of Dwyer [8] with hard, real-time constraints. For example, we define a pattern "**Present** A **after** B **within** $[0, 4]$" to express that event A must occur within 4 units of time (u.t.) of the first occurrence of B, if any. Our main objective is to propose an alternative to timed extensions of temporal logic during model-checking. Our patterns are designed to express general timing constraints commonly found in the analysis of real-time systems (such as compliance to deadlines; event duration; bounds on the worst-case traversal time; etc.). They are also designed to be simple in terms of both

clarity and computational complexity. In particular, each pattern should correspond to a decidable model-checking problem.

Our second contribution is the definition of a verification approach based on model-checking. Our approach is based on the use of observers in order to transform the verification of timed patterns into the verification of simpler LTL formulas. We define a set of observers for each pattern and compare them in order to choose the best one in practice. Our goal is to provide the most efficient observer in terms of verification time and system size growth. While the use of observers for model-checking is quite common, our contribution is original in several ways. The most novel aspect is that we formally check that our observers are *correct* (they provide an answer that is sound with respect to the semantics of patterns) and *non-intrusive*, meaning that observers cannot interfere with the observed system.

Our final contribution is to provide an integrated model-checking tool chain that can be used to check timed requirements. In our tool chain, Fiacre is used to express the model of the system while verification activities ultimately relies on Tina [4], the TIme Petri Net Analyzer. (Fiacre models have basically the same expressiveness than an extension of Time Petri Nets with data variables and priorities.)

2 Technical Backgrounds

Fiacre: we consider systems modeled using Fiacre, a formal specification language designed to represent both the behavioral and timing aspects of real-time systems. We give some examples of Fiacre programs in Sect. 4. Fiacre supports two of the most common communication paradigms: communication through shared variable and synchronization through (synchronous) communication ports. In the latter case, it is possible to associate time and priority constraints to communication over ports. Fiacre programs are stratified in two main notions: processes, which models structured activities, and components, which describes a system as a composition of processes, possibly in a hierarchical manner. A process is defined by a set of control states. Each state is associated with a set of transitions (introduced by the keyword **from**), that declares how variables are updated and which transitions may fire. The language includes deterministic constructs available in classical programming languages (assignments, conditionals, sequential composition, ...); non-deterministic constructs (choice and non-deterministic assignments); communication events on ports; and jump to next state.

Timed Traces: patterns are used to express timing and behavioral constraints on the execution of a system. We base the semantics of patterns on the notion of *timed traces*, which are sequences mixing events and time delays. In our context, the observable events are fired transitions, the state of processes and the value of variables. We use a dense time model, meaning that we consider rational time delays and work both with strict and non-strict time bounds. In this setting, a timed trace σ is a possibly infinite sequence of events $A, B, \ldots$ and duration $d(\delta)$ with $\delta \in \mathbb{Q}^+$. Given a finite trace σ and a—possibly infinite—trace σ', we denote $\sigma\sigma'$ the *concatenation* of σ and σ'. The semantics of Fiacre can also be defined as a set of timed traces. Given a real-time pattern P, we say that a system T satisfies the requirement P if all the traces of T hold for P.

MTL and FOTT: the semantics of a pattern will be expressed as the set of all timed traces where the pattern holds. A first approach to define set of traces is to use timed extensions of Linear Temporal Logic, for instance MTL [13]. Nonetheless, this partially defeats one of the original goal of patterns, that is to circumvent the use of temporal logic in the first place. For this reason, we propose an alternative way for defining the semantics of patterns that relies on first-order formulas over traces. For instance, we can define the "scope" σ **after** A–that defines the part of a trace σ located after the first occurrence of A–as the trace σ_2 such that $\exists \sigma_1 . (\sigma = \sigma_1 A \sigma_2) \wedge (A \notin \sigma_1)$. We believe that this second approach may ease the work of engineers that are not trained with formal verification techniques.

3 A Catalog of Real-Time Patterns

We have defined a catalog of patterns, using a hierarchical classification borrowed from Dwyer [9]. Patterns are built from five categories (existence, absence, universality, response and precedence) or can be composed using logical connectives. In each category, generic patterns may be specialized using *scope modifiers*—such as before, after, between—that limit the range of the execution trace over which the pattern must hold. Finally, timed patterns are obtained using one of two possible kinds of *timing modifiers* that limit the possible dates of events referred in the pattern: **Within** I which is used to constrain the delay between two given events to be in the time interval I and **For duration** D which is used to constrain the length of time during which a given condition holds (without interruption) to be greater than D.

Due to the somewhat large number of possible alternatives, we restrict this catalog to some examples of presence, absence and response patterns. Patterns that are not described here can be found in a long version of this paper [1]. For each pattern, we give its denotational interpretation based on First-Order formulas over Timed Traces (denoted FOTT in the following) and a logical definition based on MTL.

Present A **after** B **within** I

This is an example of existence pattern*, that is patterns used to express that, in every trace of the system, some events must occur. The pattern holds for traces such that* A *occurs* t_0 *units of time after the first occurrence of* B*, with* $t_0 \in I$*. The pattern is also satisfied if* B *never holds.*

MTL def.: $(\neg B)\ \mathbf{W}\ (B \wedge \mathit{True}\ \mathbf{U}_I\ A)$

FOTT def.: $\forall \sigma_1, \sigma_2 \,.\, (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4 \,.\, \sigma_2 = \sigma_3 A \sigma_4 \wedge \Delta(\sigma_3) \in I$

Present first A **before** B **within** I

(This is an example of the ***before*** *scope modifier.) If* B *occurs then the first occurrence of* A *holds* t_0 *u.t. before the first occurrence of* B *with* $t_0 \in I$*. (The difference with* ***Present*** B ***after*** A ***within*** I *is that* B *should not occur before the first* A*.)*

MTL def.: $(\Diamond B) \Rightarrow ((\neg A \wedge \neg B) \mathbf{U} (A \wedge \neg B \wedge (\neg B \mathbf{U}_I B)))$
FOTT def.: $\forall \sigma_1, \sigma_2 \,.\, (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \Rightarrow \exists \sigma_3, \sigma_4 \,.\, \sigma_1 = \sigma_3 A \sigma_4 \wedge A \notin \sigma_3 \wedge \Delta(\sigma_4) \in I$

Absent A **before** B **for duration** D

This is an example of absence pattern*, that is patterns used to express that some condition should never occur. The pattern holds if no A occurs less than D u.t. before the first occurrence of B or if B does not occur.*

MTL def.: $\Diamond B \Rightarrow (A \Rightarrow (\Box_{[0,D]} \neg B)) \mathbf{U} B$
FOTT def.: $\forall \sigma_1, \sigma_2, \sigma_3, \omega \,.\, (\sigma = \sigma_1 \omega \sigma_2 B \sigma_3 \wedge B \notin \sigma_1 \omega \sigma_2 \wedge \Delta(\sigma_2) \leqslant D) \Rightarrow \neg A(\omega)$

A **leadsto first** B **within** I

This is an example of response patterns*, that is patterns used to express "cause–effect" relationship. The pattern holds if every occurrence of A is followed by a B within time interval I (considering only the first occurrence of B after A).*

MTL def.: $\Box(A \Rightarrow (\neg B) \mathbf{U}_I B)$
FOTT def.: $\forall \sigma_1, \sigma_2 \,.\, (\sigma = \sigma_1 A \sigma_2) \Rightarrow \exists \sigma_3, \sigma_4 \,.\, \sigma_2 = \sigma_3 B \sigma_4 \wedge \Delta(\sigma_3) \in I \wedge B \notin \sigma_3$

4 Checking the Correctness of Patterns Implementation

We propose to use *observers* for checking real-time patterns. In our tool chain, an observer is a Fiacre (sub-)program that monitors the behavior of the system and that can signal whether a requirement holds or not. For example the observer can enter in a special "error state". To check a timed pattern P, on a system, T, we graft the observer for P to the Fiacre program—we denote the resulting program ($T \parallel P$)—and test the reachability of error states using an LTL model-checker. Therefore, observers are used to transform the verification of timed patterns on T into the verification of simpler reachability properties on $T \parallel P$.

More details about observers are given in [2], where we have defined several observers for each pattern. The idea is not to provide a generic way of generating the observers from a formal definition of the pattern. Rather, we seek, for each pattern, to come up with the best possible candidate in practice.

In this paper, we give an example of observer for the pattern "**Present** A **after** B **within** $[d1, d2[$" (see Listing 1.1). The process *observer* monitors a system through two special ports, A and B, of type sync. The process is initially in state *idle* and moves to state *start* when event B occurs. When in state *start* for more than d_1 u.t., the observer moves to state *watch* (this is the meaning of the **wait** operator). The **select** operator is a non-deterministic choice, with **unless** coding priorities. Hence, in state *watch*, the observer moves to *stop* if an A occurs, unless $d_2 - d_1$ u.t. elapses, in which case it moves

to *error*. As a consequence, the pattern is false whenever process *Present* can reach state *error*. This is equivalent to checking the LTL formula : $\Box\neg$(Present/state error).

```
process Present [A, B : sync] is
   states idle, start, watch, error, stop
   init to idle
   from idle   B ;  to start
   from start  wait [d1,...[ ; to watch
   from watch  select
                  A ; to stop
               unless
                  wait [d2–d1, ...[ ; to error
               end
```

Listing 1.1. Observer for **Present** A **after** B **within** $[d1, d2]$

To prove that the observer is correct, we need to prove that, for every system T, the program ($T \parallel$ Present) can reach *error* if and only if there is a trace σ in T where the pattern does not hold. In [2], we define a theoretical framework to prove exactly this kind of properties. Efforts are also under way to mechanize these proofs using the Coq proof assistant. Nonetheless, formal proofs of correctness can be quite tedious. Therefore, to detect possible problems with an observer early on (that is, before spending a lot of efforts doing a formal proof of correctness) we also propose a "graphical verification method". This is akin to debugging our observers.

Universal Program. In the remainder of this section, we take the example of the observer *Present*. The idea is to check the observer on a particular Fiacre program, called the universal program, that can generate all possible combinations of delays and events A and B. (This is much simpler than testing the observer with all possible programs.) We give an example of universal program in Listing 1.2. The process *Universal* has only one state and three possible transitions. Each transition changes the value of a shared integer variable, x. The first and second transitions of *Universal* can be fired without time constraints. In our context, A will be linked to the event "setting x to 1" and B to "setting x to 2". The third transition reset the value of x to 0 immediately.

Graphical Verification. Using our tool chain, we can generate the (finite) state graph for the program (*Universal* $\parallel$ *Present*). We display the resulting graph in Fig. 1 for the special case where $d_1 = 4$ and $d_2 = 5$. The state graph is generated with a "discrete time" abstraction, where special transitions (labeled with i) are used to model the flow of time: a transition i between two states denotes that 1 u.t. has passed. This is obtained using the Tina tool [4] with the flag `-F 1`, which is used to generate the state graph including dense time models.

To debug the pattern "**Present** A **after** B **within** $[4, 5[$" we can simply check (visually) that after the first occurrence of a B (in state 2 of Fig. 1), and after 4 units of time (after 4 transitions labeled i), the system reachs a state (numbered 13 in the state graph) such that: (1) if we do not see an A before 1 u.t. pasess then we have an error; and (2) if we see an A then we will never see an error.

```
process Universal(&x : nat) is
   states s0
   from s0
      select
         x := 1; to s0
      [] x := 2; to s0
      unless
         on (x <> 0) ; wait [0,0]; x := 0 ; to s0
      end
component Main is
   var x : nat := 0
   port A : sync is value (x == 1),
        B : sync is value (x == 2)
   par Universal(&x) || Present[A, B] end
```

Listing 1.2. Universal program in Fiacre

This graphical verification method has some drawbacks. It relies on a discrete time model and it only works for fixed time intervals (we have to fix the value of d_1 and d_2). Nonetheless, it is usually enough to catch some errors in the observer before we try to prove the observer correct more formally.

In practice, we do not simply rely on a visual inspection of the state graph. We can use properties expressed in the μ-calculus to check the state graph (the Tina toolset includes a model-checker for the μ-calculus called muse). Informally, we can define the set of traces where *Present* holds using the union of two regular languages: first the traces where B never occurs (the expression $(\neg B)^*$); then the traces where there is an A 4 u.t. after the first B (the expression $(\neg B)^* \cdot B \cdot ((\neg i)^* \cdot i)^4 \cdot A \cdot \top^*$). Each expression can be interpreted as a μ-calculus formula. For instance, the absence of B corresponds to the formula $\mu\, X.(\langle \neg B \rangle \vee X)$, which is true for the states 0 and 1 in the state graph. Likewise, using a maximal fix-point formula, we can define the "errored states", that is the states where an error has been signaled or is inevitable. Finally, it is enough to check that all valid traces cannot lead to an errored state and that all invalid traces (the complement language) inevitably lead to an errored state.

5 Related Work and Contributions

We base our approach on the original catalog of specification patterns defined by Dwyer [8]. Their patterns language is still supported, with several tools, an online repository of examples [9] and the definition of the Bandera Specification Language [7] that provides a structured-English language front-end. A recent study by Bianculli et al. [6] show the relevance of this pattern-based approach in an industrial context.

Some works consider the extension of patterns with hard real-time constraints. Konrad et al. [12] extend the patterns language with time constraints and give a mapping from timed pattern to TCTL and MTL. Nonetheless, they do not consider the complexity of the verification problem (they do not study the implementability of their approach). Another related work is [11], where the authors define observers based on Timed Automata for each pattern. However, they consider a less expressive set of patterns (without the "for duration" modifier) and they have not integrated their language

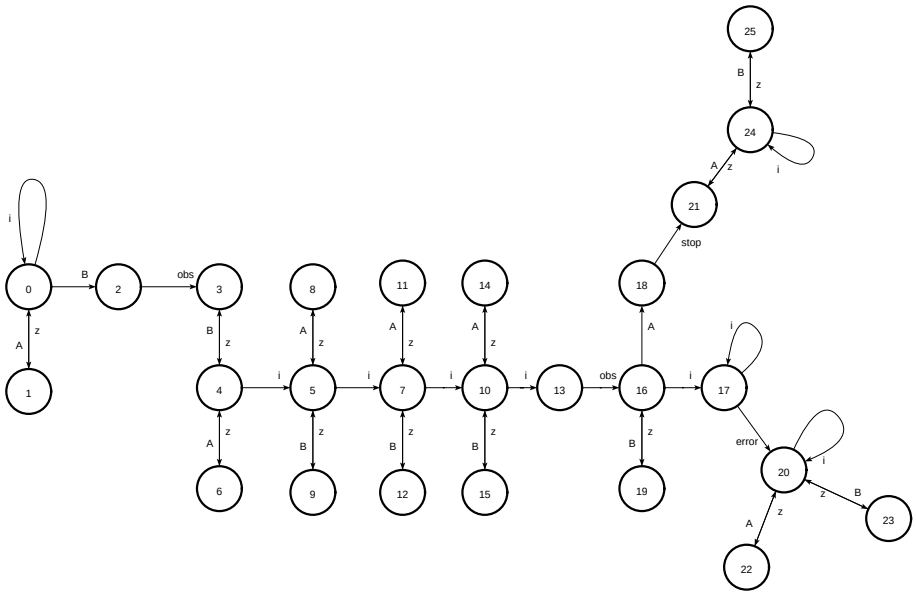

Fig. 1. State graph for the (*Universal* || *Present*)

inside a tool chain or proved the correctness of their observers. By contrast, we have defined a framework that has been used to prove the correctness of some of our observers [2].

We can also compare our approach with works concerned with observer-based techniques for the verification of real-time systems. Consider for example the work of Aceto et al. [3] based on the use of test automata to check properties on timed automata. In this framework, verification is limited to safety and bounded liveness properties since the authors focus on properties that can be reduced to reachability checking. In the context of Time Petri Net, Toussaint et al. [14] propose a verification technique similar to ours, but only consider four specific kinds of time constraints.

Compared to these related works, we make several contributions. We extend the specification patterns language of Dwyer et al. with two modifiers for real-time constraints. For each pattern, we give a precise definition based on different formalisms. We also address the problem of checking the validity of a pattern on a real-time system using model-based techniques: our verification approach is based on a set of observers. Using this approach, we reduce the problem of checking real-time properties to the problem of checking simpler LTL properties on the composition of the system with an observer. We have defined a set of observers and compare them in order to obtain the most efficient one in practice. Another contribution is the definition of a framework to prove that observers are correct and non-intrusive, meaning that they do not affect the system under observation. Finally, concerning tooling, our work is integrated in a complete verification tool chain for the Fiacre modelling language and can therefore be used in conjunction with Topcased [10], an Eclipse based toolkit for system engineering.

6 Conclusion and Perspectives

We define a complete framework including the definition of a set of high-level specification patterns for expressing requirements on systems with hard real-time constraints, a verification method and a framework to verify the correctness of the verification method. Our approach eliminates the need to rely on model-checking algorithms for timed extensions of temporal logics that—when decidable—are very complex and time-consuming. In the future, we plan to define a compositional patterns inspired by the "denotational interpretation" used in the definition of patterns. The idea is to define a lower-level pattern language, with more composition operators, that is amenable to an automatic translation into observers (and therefore can dispose with the need to manually prove the correctness of our interpretation).

References

1. N. Abid, S. Dal Zilio, and D. Le Botlan. Real-Time Specification Patterns and tools. In *Proc. of FMICS–17th Int. Workshop on Formal Methods for Industrial Critical Systems*, 2012.
2. N. Abid, S. Dal Zilio, and D. Le Botlan. Verification of Real-Time Specification Patterns on Time Transitions Systems. In *Proc of VECoS–6th Int. Workshop on Verification and Evaluation of Computer and Communication Systems*, 2012.
3. L. Aceto, A. Burgueño, and K. G. Larsen. Model checking via reachability testing for timed automata. In *Proc. of TACAS*, vol. 1384 of *LNCS*. Springer, 1998.
4. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool Tina – construction of abstract state spaces for Petri nets and time Petri nets. *Int. J. of Production Research*, 42:14, 2004.
5. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *Proc. of ERTS*, 2008.
6. D. Bianculli and C. Ghezzi and C. Pautasso and P. Senti. Specification Patterns from Research to Industry: a Case Study in Service-based Applications. In *the 34th International Conference on Software Engineering*. IEEE, 2012.
7. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN Software Model Checking Workshop*, vol. 1885 of *LNCS*. Springer, 2000.
8. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of ICSE*, 1999.
9. M. B. Dwyer, L. Dillon. Online Repository of Specification Patterns. At `http://patterns.projects.cis.ksu.edu/`
10. P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. The TOPCASED project: a Toolkit in Open source for Critical Aeronautic SystEms Design. In *Proc. of ERTS*, 2006.
11. V. Gruhn and R. Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.
12. S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proc. of ICSE*, ACM, 2005.
13. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2:255–299, 1990.
14. J. Toussaint, F. Simonot-Lion, and J.-P. Thomesse. Time constraints verification methods based on time Petri nets. In *Proc. of FTDCS*. IEEE, 1997.