

Single State Trackability of Discrete Event Systems

Valentin Bouziat, Xavier Pucel, Stéphanie Roussel, Louise Travé-Massuyès

▶ **To cite this version:**

Valentin Bouziat, Xavier Pucel, Stéphanie Roussel, Louise Travé-Massuyès. Single State Trackability of Discrete Event Systems. DX'19 – 30th International Workshop on Principles of Diagnosis, Nov 2019, Klagenfurt, Austria. hal-02383500

HAL Id: hal-02383500
https://laas.hal.science/hal-02383500

Submitted on 27 Nov 2019

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.

Single State Trackability of Discrete Event Systems

Valentin Bouziat[1] and **Xavier Pucel**[1] and **Stéphanie Roussel**[1] and **Louise Travé-Massuyès**[2]
[1] ONERA / DTIS, Université de Toulouse, F-31055 Toulouse – France
firstname.lastname@onera.fr
[2] LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
louise@laas.fr

Abstract

Specific requirements must guide the design of autonomous systems as they are increasingly present in our everyday environment. Their properties must be carefully defined and checked to guarantee safety, security and dependability. In this paper, we adopt a discrete event system modelling framework and focus on properties that are related to diagnosis. A new property called *single state trackability* is introduced. While available observations may lead to an ambiguous estimate, i.e. several admissible state candidates, this property assesses the possibility of reducing the estimate to a single state without this leading to a dead-end in the continuation of the execution. A single state estimate advantageously facilitates decision making and allows the use of a deterministic planner in the autonomous architecture. We provide a necessary and sufficient condition for single state trackability of a discrete event system and we propose a recursive algorithm to check this property. The algorithm is validated with a set of benchmarks.

1 Introduction

Self-awareness is one of the essential properties of autonomous systems : it heavily impacts decision making and can be critical for the survival of the system. The ability of a system to react to unexpected events depends on its capacity to evaluate its current state. We focus on systems whose dynamics can be represented with a discrete event modelling formalism.

In this context, state tracking and diagnosis have been the target of many works [1; 2; 3; 4]. In general, observations are not enough to guaranty state observability [5; 6], which means that state estimation is ambiguous and returns several estimates at each time step. Consequently, the number of possible state candidates grows exponentially as time goes by. Most works tackle this problem by selecting a limited number of best candidates according to some preference criterion, for example probabilities like in [7; 8]. Yet, when the real state is not among the selected subset, this may lead to a dead-end in the continuation of the tracking. The solution proposed by [8] to this problem is to backtrack and recover the state trajectory that allows estimation to resume.

In this paper, we reduce to an extreme the number of estimates and we propose to keep only one, as in [9]. We call such an estimator a *single state estimator*. The reasons for this are several. First, autonomous architectures have a very limited amount of memory and it is not desirable to store the complete history of the system execution as it grows over time. Second, estimation must be incremental, only based on the previous estimate and the current observation at each time step. Last but not least, a single-state estimate advantageously facilitates decision making and allows the use of deterministic planners, task allocators, etc. in the autonomous architecture. This last point is also a step towards explainability of autonomous systems as autonomous decisions are based on only one estimate.

In [9], we analyse whether a single state estimator specified by a set of preferences is subject to dead-ends. When this is the case, modifying the preferences, i.e. the estimation strategy, may solve the dead-end issue [10]. However, there exists systems for which every possible single-state estimator are subject to dead-ends. This means that there is no way to estimate the state of such systems in real time without potentially facing a dead-end in the continuation of the execution. When backtracking is not an option for the considered system (for instance because of real time constraints), the occurrence of a dead-end might cause the loss of the system. In this paper, we aim at characterizing such systems and we define a new property called *single state trackability* that assesses the existence of a single-state estimator that does not lead to dead-end. Our main contribution is to provide a necessary and sufficient condition for single state trackability of a discrete event system and we propose a recursive algorithm to check this property at design time.

This paper is structured as follows. Related work is covered in Section 2. Section 3 introduces our modeling formalism for discrete event systems and incremental estimators. Then, the property of *single state trackability* is introduced in Section 4. Investigations are conducted in order to check this property in Section 5. An algorithm for checking this property is proposed in Section 6 and validated on experimental systems in Section 7.

2 Related work

While the problem of single state trackability is new, the related notions of DES observability and diagnosability have been the subject of several papers. [5] addresses observability under partial state and total event observation. The notions of (weak) indistinguishable states account for all future observations, and (weak) observability is achieved when all

pairs of states are (weakly) indistinguishable. The notion of coobservability is defined similarly with past observations. Strong coobservability implies that the current state can be uniquely determined from the observation history, which is much stronger than single state trackability.

In [6] only some events are observed, and a system is said to be observable when an observer that tracks all the possible states regularly visits states with only one candidate, i.e. with a bounded period. The notion of delayed observability is similar, but allows for the uniquely known state to be in the past. The definition of resilient observers captures the principle of robustness to perturbated observations. Observability requires the knowledge of the exact system state, which is not necessary for single state trackability. Conversely, single state trackability requires that among two undistinguishable states, one explains all the observations produced by the other, which is not necessary in observability definitions.

Diagnosability, as defined in [11], differs from our approach by several aspects. First, it targets permanent faults, while our approach can be applied to estimate the presence of intermittent faults as well. Second, it requires the complete construction of a diagnoser that poses a scalability problem, that is mitigated in [12]. Finally, it accounts for a bounded delay between the occurrence of a fault and its diagnosis. While this idea is interesting and makes for a realistic requirement, there is no universal way to extend it to intermittent phenomena. Examples include [13] and [14], which we found hardly relevant from our point of view about autonomous systems.

3 State estimation of Discrete Event Systems

In this paper we adopt a modeling approach similar to model-checking, where states are defined as assignments on a set of Boolean variables $\mathtt{V_S}$. Some variables are observed and form a set of Boolean variables $\mathtt{V_0}$ such that $\mathtt{V_0} \subseteq \mathtt{V_S}$. Variables that are not observed are estimated. We assume that the system follows discrete dynamics where each time step lasts the same duration. The set of possible system states is noted S, and each state $s \in S$ is represented by a Boolean assignment to all variables in $\mathtt{V_S}$. Consequently, $S \subseteq 2^{\mathtt{V_S}}$.

Definition 1 (Discrete event system)**.** *A discrete event system (DES) is represented by a tuple (S, Δ, s_0) where S is set of system states, $\Delta \subseteq S \times S$ is the transition relation and $s_0 \in S$ is the initial state.*

Definition 2 (Execution language)**.** *For a discrete event system (S, Δ, s_0), the* execution language *$\mathcal{L}(\Delta) \subseteq S^*$ is the set of state sequences starting with s_0 that satisfy Δ:*

$$\mathcal{L}(\Delta) = \{(s_0, s_1, \cdots, s_n)| \\ n \in \mathbb{N}^+, i \in [0, n-1], (s_i, s_{i+1}) \in \Delta\}$$

Notations : Let $seq \in \mathcal{L}(\Delta)$ be a state sequence. $seq[i]$ refers to the i^{th} state of the sequence beginning at initial state $seq[0] = s_0$. The sequence's size is denoted $|seq|$.

An assignment on $\mathtt{V_0}$ is called an *observation* and O denotes the set of all possible observations. We denote obs the function from S to O that returns the observation associated to a state and we naturally extend it from S^* to O^* as follows: $|obs(seq)| = |seq|$ and $obs(seq)[i] = obs(seq[i])$ for $i \in [0, |seq|[$.

At each time step the system sends an observation to the estimator. Formally, the estimator receives as input a sequence of observations, and produces a sequence of estimations. At each time step, the estimator selects a unique estimated state among the set of estimation candidates.

Definition 3 (Observation language)**.** *The observation language $\mathcal{L}_{obs}(\Delta) \subseteq O^*$ of a DES is the set of all consistent observation sequences.*

$$\mathcal{L}_{obs}(\Delta) = \{obs(seq)|seq \in \mathcal{L}(\Delta)\}$$

In our approach, an estimator only takes into account the previous estimated state and the current observation.

Definition 4 (Set of estimation candidates)**.** *Given a DES (S, Δ, s_0), a state $s \in S$ and an observation $o \in O$, we define the set of estimation candidates $cands(s, o)$ as the set of states in which the system could be, assuming it was in state s at the previous time step, and produces observation o at the current time step. Formally:*

$$cands(s, o) = \{\hat{s} \in S|(s, \hat{s}) \in \Delta \text{ and } obs(\hat{s}) = o\}$$

Definition 5 (Estimation function)**.** *Let (S, Δ, s_0) be a DES. An* estimation function *is a function $estim : (S \times O) \to S$ that selects a unique estimation candidate, i.e. for every $s \in S$, and every $o \in O$, if $cands(s, o) \neq \emptyset$ then $estim(s, o) \in cands(s, o)$.*

An estimator is completely defined by its estimation function. It receives observations as inputs, and the estimated state is reused at the next time step to compute the next estimation. We assume that the initial system state is known to the estimator.

Definition 6 (Estimation sequence)**.** *Let (S, Δ, s_0) be a DES, $estim : (S \times O) \to S$ an* estimation function*, and $sobs \in \mathcal{L}_{obs}(\Delta)$ an observation sequence. The* estimation sequence *for $sobs$ is the unique state sequence $sest$ such that:*

- *$sest[0] = s_0$ and*
- *for $i \in [1, |sobs|[$, if $cands(sest[i-1], sobs[i]) \neq \emptyset$ then $sest[i] = estim(sest[i-1], sobs[i])$ else $sest$ is undefined.*

4 Single state trackability

The simple fact that an estimator selects a single state estimate creates scenarios where the estimate can differ from the real system state, and later the system produces an observation that is inconsistent with the previously estimated state. In such scenarios, the set of estimation candidates is empty, the estimation function is then undefined and the estimator is unable to produce an estimation[1]. Formally, if we note s the system state and $\hat{s}$ the state estimate, if $s \neq \hat{s}$, the system may evolve in a state s' and produce an observation $obs(s') = o$ such that $cands(\hat{s}, o) = \emptyset$. We call such a situation a *dead-end*, and the observable path is called a *dead-end path*.

Definition 7 (Dead-end path)**.** *Let (S, Δ, s_0) be a DES, $estim : (S \times O) \to S$ an estimation function, $sobs \in \mathcal{L}_{obs}(\Delta)$ an observable sequence of length k, and $o \in O$ a continuation of $sobs$, i.e. $sobs.o \in \mathcal{L}_{obs}(\Delta)$; $sobs.o$ is a* dead-end path *if and only if:*

[1]Note that such scenarios happen not only in our single state approach, but for any approach that does not keep all the estimation candidates in memory.

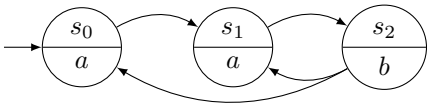


Figure 1: A non SST DES with states $S = \{s_0, s_1, s_2\}$, and observations $O = \{a, b\}$. Δ and obs are shown as in a Moore machine.

- *the estimation sequence for $sobs$ is defined: $sest = (s_0, \hat{s_1}, \cdots, \hat{s_k})$;*
- *there exists no estimation candidate after observation o, i.e $cands(\hat{s_k}, o) = \emptyset$.*

Dead-end paths illustrate the situation where the estimator assumes something about the system's real state, and discovers later that this assumption was false. This can be a problem if some important decision was made as a consequence of this assumption. Most of the time, it is impossible (and not necessary) to know the full system state to operate it. This is why we introduce the concept of *single state trackability*, i.e. the ability to estimate the system state and never encounter a dead-end path.

Definition 8 (Single state trackability)**.** *A DES is* Single state trackable[2] *(SST) if and only if there exists an estimation function $estim : (S \times O) \to S$ such that no observable sequence $sobs \in \mathcal{L}_{obs}(\Delta)$ is a dead-end path.*

Example 1. *Let us consider the DES represented in Figure 1, described as a Moore automaton [15] with no input alphabet.*

Let the system produce the observation sequence (a, a, b). For the first 3 steps, estimation is trivial as the system can only go through the state sequence (s_0, s_1, s_2). But for the observation sequence (a, a, b, a), the set of candidates is then $cands(s_2, a) = \{s_0, s_1\}$, and a choice needs to be made. Let us consider the two possible estimation functions $estim_0(s_2, a) = s_0$ and $estim_1(s_2, a) = s_1$, and the two observation sequences (a, a, b, a, a) and (a, a, b, a, b) respectively produced by state sequences $(s_0, s_1, s_2, s_0, s_1)$ and $(s_0, s_1, s_2, s_1, s_2)$.

With $estim_0$, the observation sequence (a, a, b, a) is estimated as (s_0, s_1, s_2, s_0), however since $cands(s_0, b) = \emptyset$, the observation sequence (a, a, b, a, b) is a dead-end.

With $estim_1$, the observation sequence (a, a, b, a) is estimated as (s_0, s_1, s_2, s_1), and since $cands(s_1, a) = \emptyset$, the observation sequence (a, a, b, a, a) is a dead-end.

Since all the possible estimation functions encounter dead-ends, the system is not SST.

5 Checking single state trackability

In this section we introduce some necessary conditions and one equivalent condition for checking single state trackability.

Definition 9 (Reachable states)**.** *A state $\hat{s}$ is reachable via the observation sequence $sobs \in \mathcal{L}_{obs}(\Delta)$ if and only if there exists a state sequence $seq \in \mathcal{L}(\Delta)$ such that $obs(seq) = sobs$, and seq ends with $\hat{s}$. The set of states reachable via $sobs$ is noted $reach(sobs)$.*

Note that while the set of observation sequences is infinite, the set of all possible $reach(sobs)$, $sobs \in \mathcal{L}_{obs}(\Delta)$ is finite and a subset of 2^S. This set can be enumerated by constructing the so-called powerset automaton with respect to obs. This property is used in Algorithm 2.

[2]In this paper, "trackable" and "trackability" always refer to single state trackable and single state trackability respectively.

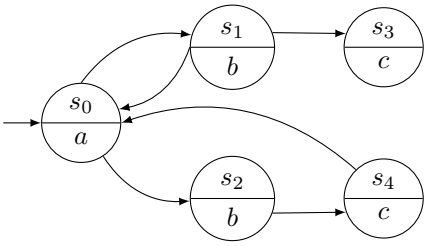


Figure 2: A DES in which all transitions are non-blocking, but that is still not SST.

Definition 10 (Non-blocking states (NBS))**.** *Let (S, Δ, s_0) be a DES, and $sobs \in \mathcal{L}_{obs}(\Delta)$ an observation sequence. A state $\hat{s}$ is* non-blocking *for $sobs$ if and only if it is reachable via $sobs$ and for every subsequent observation o, there is an estimation candidate from $\hat{s}$. Formally, $\hat{s} \in reach(sobs)$ is* non-blocking *if and only if:*

$$\forall o \in O,\ \textit{if } sobs.o \in \mathcal{L}_{obs}(\Delta) \textit{ then } cands(\hat{s}, o) \neq \emptyset$$

A state reachable via $sobs$ but which is not non-blocking for $sobs$ is called a *blocking state*. Note that a state $\hat{s}$ may be non-blocking for some observation sequence $sobs_1$ and blocking for another sequence $sobs_2$. Non-blocking states are important because estimators must always select them, or they will encounter a dead-end path, as stated in the following propositions.

Proposition 1 (Non blocking state condition)**.** *If there exists an observation sequence $sobs \in \mathcal{L}_{obs}(\Delta)$ such that $reach(sobs)$ contains only blocking states, then the system is not trackable.*

Intuitively, Proposition 1 means that for any estimation function, the estimated sequence for $sobs$ ends with a state in $reach(sobs)$. If it contains only blocking states, then whatever state $\hat{s} \in reach(sobs)$ is chosen, there exists a continuation $sobs.o \in \mathcal{L}_{obs}(\Delta)$ that is a dead-end path.

Proposition 2 (Non blocking transition condition)**.** *Let $sobs \in \mathcal{L}_{obs}(\Delta)$ be an observation sequence and $o \in O$ an observation such that $sobs.o \in \mathcal{L}_{obs}(\Delta)$. If the system is single state trackable, then there exists a pair of states $(s_1, s_2) \in \Delta$ such that s_1 is non-blocking for $sobs$ and s_2 is non-blocking for $sobs.o$.*

Proposition 2 extends Proposition 1 to transitions, and could be extended further to paths of arbitrary length. However, when constructing the set of all $reach(sobs)$ to check Proposition 1, and particularly when constructing a transition between $reach(sobs)$ and $reach(sobs.o)$, it is straightforward to verify Proposition 2 on-the-fly. Checking it for longer paths can be done efficiently in a dynamic programming style algorithm.

Extending the NBS property to paths of any length does not provide a sufficient condition for trackability. In the DES presented in Figure 2, it is always possible for a given observation sequence to associate a state sequence respecting the condition of proposition 2. If we take the observation sequence (a, b, a, b, c) produced by the non-blocking state sequence $(s_0, s_1, s_0, s_2, s_4)$, however, it is impossible to construct a function $estim : (S \times O) \to S$ allowing

to borrow this sequence since starting from s_0 and receiving observation b, we would have to choose sometimes s_1, sometimes s_2. To provide a necessary and sufficient condition for trackability, we check that not only observable sequences, but the full observable language is supported by some estimator.

Definition 11 (Estimator accepted language)**.** *Let (S, Δ, s_0) be a DES, $sobs \in \mathcal{L}_{obs}(\Delta)$ an observation sequence and $estim : (S \times O) \to S$ an estimation function. The language accepted by this estimation function, denoted $\mathcal{L}_{obs}(estim)$, is the set of all possible observation sequences for which there exists an estimation sequence (i.e. that are not dead-ends). Formally :*

$$\begin{aligned}\mathcal{L}_{obs}(estim) = \{sobs \in \mathcal{L}_{obs}(\Delta) \mid (|sobs| > 1) \\ \text{and } \exists sest \in \mathcal{L}(\Delta) \\ \text{s.t. for } i \in [1, |sobs|], obs(sest[i]) = sobs[i] \\ \text{and } estim(sest[i-1], sobs[i]) = sest[i]\}\end{aligned}$$

It now becomes apparent that finding dead-ends for a given estimator can be done with an algorithm similar to the algorithm for checking equality of regular languages.

Proposition 3 (Trackability condition)**.** *A DES (S, Δ, s_0) is* single state trackable *if and only if there exists an estimation function whose accepted language equals the observation language of the system:*

$$\exists\, estim : (S \times O) \to S, \mathcal{L}_{obs}(estim) = \mathcal{L}_{obs}(\Delta)$$

The proof is straightforward: if proposition 3 is satisfied, then the estimation function provides an estimation sequence for all elements of $\mathcal{L}_{obs}(\Delta)$. Thus there are no dead-ends. The main difficulty lies in finding such an estimation function, or proving that it does not exist.

To efficiently check trackability, our approach consists in evaluating partially defined estimation functions. We define dead-ends for such partial estimation functions, and introduce a proposition that will be used in our algorithm.

Definition 12 (Extension of partial estimation functions)**.** *Let $estim : (S \times O) \to S$ be an estimation function, and let $pestim$ be a partial function from $(S \times O)$ to S. $estim$* extends *$pestim$ if and only if for every couple (s, o) from $S \times O$ such that $pestim(s, o)$ is defined, then $pestim(s, o)$ = $estim(s, o)$.*

A partial function that can be extended in an estimation function is called a partial estimation function*.*

Definition 13 (Dead-end for partial functions)**.** *Let $pestim$ be a partial estimation function from $(S \times O)$ to S, $sobs$ be an observation sequence and $sobs.o \in \mathcal{L}_{obs}(\Delta)$ a continuation. $sobs.o$ is a dead-end path for $pestim$ if and only if there exists a state sequence $sest = (s_0, \ldots, \hat{s_k})$ such that $pestim(sest[i-1], sobs[i])$ is defined and equals $sest[i]$ for $i \in [1, |sobs|]$, and $cands(\hat{s_k}, o) = \emptyset$.*

Proposition 4. *If $sobs \in \mathcal{L}_{obs}(\Delta)$ is a dead-end for a partial estimation function $pestim$, then $sobs$ is a dead-end for every estimation function $estim : (S \times O) \to S$ that extends $pestim$.*

6 Algorithm

We describe an algorithm for checking the single state trackability of a system, based on a search for dead-ends for partially defined estimation functions. Our algorithm is organized in two components: the first one produces partially defined estimation functions and the second one checks if a given partial estimation function satisfies Proposition 3. The algorithm is illustrated in Figure 3.

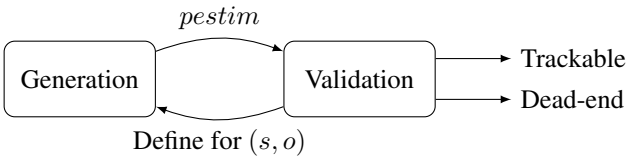


Figure 3: Algorithm structure.

The generation component recursively produces partial estimation functions $pestim$ and sends them for validation. The validation component has 3 outcomes: "Trackable" means the system is trackable with the current $pestim$; "Dead-end" means the current $pestim$ has a dead-end; "Define for (s, o)" means $pestim$ for the input pair (s, o). According to the validation result, the generation component may return or recursively produce other partial estimation functions.

6.1 Estimation function generation

The generation component provides the *GenEstim* function described in Algorithm 1 that starts the algorithm and calls the validation function. At the first iteration, the generation component produces the empty partial estimation function (i.e. defined on $\emptyset$), and sends it to the validation component.

If the validation component returns "Trackable", it means that the current partial estimation function $pestim$ defines an estimator that accepts $\mathcal{L}_{obs}(\Delta)$, and that the pairs for which $pestim$ is undefined are not used for estimation. Thus, every extension of $pestim$ satisfies Proposition 3, and the system is single state trackable. We can stop the algorithm and return true.

If the validation component returns "Dead-end", it means that there exists a dead-end path for $pestim$. By Proposition 4, no extension of this partial function can satisfy Proposition 3. We just stop the recursion.

If the validation component returns "Define for (s, o)", it means that there exists a pair (s, o) such that s is reachable, that $cands(s, o)$ contains several candidates, and that $pestim$ is undefined for (s, o). In this case, we need to check if there exists an estimation option for (s, o) that satisfies Proposition 3, so we recursively generate them.

Since the algorithm recursively explores all the estimation options, if at the end it has found no partial estimation function $pestim$ that satisfies Proposition 3, then we are certain that the system is not single state trackable.

6.2 Estimation function validation

The validation component contains a function CheckEstim that checks whether the language accepted by a given partial estimation function is equal to the observation language of the system. The algorithm is based on a slight modification of the classical algorithm for testing regular language equality [16] in order to account for cases where there are several estimation candidates, and the partial estimation function is undefined.

The approach is to simulate the execution of the estimator and the system, while ensuring that they are synchronized on the same observation sequences. Since for every observation sequence $sobs$, there exists (at most) one unique estimation sequence, we only need to keep track of a single

Algorithm 1 Generation component: GENESTIM function

```
1: Input
2:    Σ = (S, Δ, s_0): a DES
3: Output
4:    boolean: Σ is single state trackable
5: function GENESTIM(Σ, pestim)
6:    switch CHECKESTIM(Σ, pestim, s_0, {s_0}) :
7:       case Trackable : return true
8:       case Dead-end : return false
9:       case Define for (s,o) :
10:         for c in cands(s, o) do
11:            ext ← pestim ∪ ((s, o), c)
12:            if GENESTIM(Σ, ext) then return true
13:         end for
14:         return false
```

Algorithm 2 Validation component: CHECKESTIM function

```
1: Input
2:    Σ = (S, Δ, s_0) : a DES
3:    estim: a partial estimation function
4:    estSt ∈ S : the estimator state
5:    sysSts ⊆ S : the possible system states
6: Output
7:    "Trackable", "Dead-End" or "Define for (s, o)"
8: Global
9:    visited ∈ S × 2^S ← ∅
10: function CHECKESTIM(Σ, estim, estSt, sysSts)
11:    if (estSt, sysSts) ∈ visited then
12:       return Trackable
13:    visited ← visited ∪ (estSt, sysSts)
14:    nextObs ← {obs(s')|∃s ∈ sysSts, (s, s') ∈ Δ}
15:    for o in nextObs do
16:       estNxts ← NEXTESTSTATES(estSt, o)
17:       sysNxts ← {s' | obs(s') = o ∧
18:                                  ∃s ∈ sysSts, (s, s') ∈ Δ}
19:       if estNxts = ∅ then
20:          return Dead-end
21:       else if estNxts = {estNxt} then
22:          rec ←
23:                CHECKESTIM(Σ, estim, estNxt, sysNxts)
24:          switch rec :
25:             case Dead-End : return Dead-End
26:             case Define for (s, o) : return Define for
    (s, o)
27:             case Trackable : continue
28:       else
29:          return Define for (estSt, o)
30:    end for
31:    return Trackable
```

estimator state. Thus the estimator state reached via an observation sequence $sobs$ is associated with the set of system states $reach(sobs)$ (see Definition 9).

The algorithm recursively explores pairs $(estSt, sysSts)$ where $sysSts = reach(sobs)$ for some $sobs \in \mathcal{L}_{obs}(\Delta)$, and where $estSt \in sysSts$. To ensure termination, a global variable "visited" stores the pairs that have already been explored. The algorithm looks for sequences $sobs$ for which the estimator is not defined, so the termination condition is met only if no such sequence exists. In this case, the languages are equal, and we return "Trackable" (lines 11 to 13).

At each iteration, CheckEstim calculates the observations that can be produced by the system (line 14). Then for every such observation, it calls the `NextEstStates`$(estSt, o)$ (line 16) function defined as follows. If $pestim$ is defined for $(estSt, o)$, it returns $\{pestim(estSt, o)\}$ else it returns $cands(estSt, o)$. The algorithm then tests the number of possible estimator states: If $estNxts = \emptyset$ (line 19): this means that, if $sobs$ is the observation sequence that led to this recursive call, $sobs.o$ is a dead-end path (see Definition 7), we (recursively) return "Dead-end".

If $estNxts = \{estNxt\}$ (line 21): this means that for the current pair $(estSt, o)$, either there is a unique successor or that $pestim$ is defined. In this case we continue the search with a recursive call.

If $|estNxts| > 1<$ (line 28): there are several estimation candidates, and $pestim$ is undefined for $(estSt, o)$. We (recursively) return "Define for $(estSt, o)$" so that the generation component will generate estimation functions defined for this pair.

6.3 Performance

The performance of the algorithm described above can be significantly enhanced with a few mechanisms. A preliminary check is added to verify propositions 1 and 2 for every set of states that can be reached via some observable sequence.

In our experiments, the main source of complexity is the number of partial estimation functions to be tested. The only mechanism we have to prune partial estimation functions is to find dead-end paths as early as possible.

First, in Algorithm 1, upon detection of a Dead-End, one can try to remove from $pestim$ estimation triplets that are not used in the dead-end path, and memorize this trimmed partial estimation function. This way, by Proposition 4, we can test at line 10 whether some extensions are future functions that extend the one we just memorized, and spare some calls to Algorithm 2.

Second, there exist partial estimation functions for which there are both dead-end paths and undefined pairs. In these cases, in Algorithm 2, we may return either "Dead-end" or "Define for (s, o)" according to the order of traversal at line 15. Instead of immediately returning "Define for (s, o)", we store it in memory, and pursue the search for a dead-end. When we encounter a dead-end we immediately return it. When the recursive search finishes without finding any dead-end, if we have encountered an undefined pair (s, o) we return "Define for (s, o)" otherwise we return "Trackable". This favors early dead-end detection.

The properties of blocking states of Propositions 1 and 2 can also be used to reduce the search space: in Algorithm 1 at line 10, one can skip blocking states as they will definitely lead to a dead-end.

7 Experiments

We tested our approach on an example inspired from the autonomous robotics framework PLEXIL [17]. This framework is organised around the concept of actions, that have a complex hierarchical workflow. We use a simplified sequential action workflow described in Example 2.

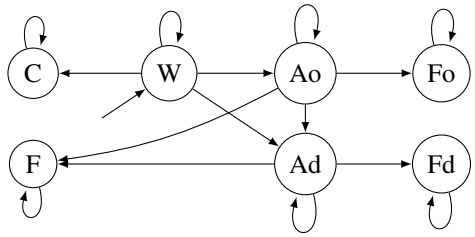


Figure 4: The workflow of an action, that can be (W)aiting, (C)ancelled, (A)ctive (o)k, (F)inished (o)k, (F)ailed, (A)ctive (d)elayed, or (F)inished (d)elayed.

Example 2. *We consider a robotics framework where robot plans consist in a sequence of actions. The workflow of an action is illustrated in Figure 4. We consider a robot with a plan composed of two sequential actions: move and inspect, whose states are represented by variables mv and ins. The robot's health status is described by three Boolean variables $hnav$, $hsens$ and $hpow$ representing respectively whether the navigation, sensor and power supply functions perform normally at each time step. Another Boolean variable $pert$ indicates whether the robot is subject to perturbations (slippery terrain, obstacle, wind) at each time step. We note $move = W$ to express that the move action is in state W, and denote $Y(v)$ the value at the previous time step for variable $v \in \{mv, ins, hnav, hsens, hpow, pert\}$. We use the function $start(v) = K$ that means $Y(v) \neq K \wedge v = K$. The system behaviour is described by the automata for each action, plus the following constraints:*

$$mv \in \{W, Ao, Ad\} \rightarrow ins = W \quad (1)$$
$$mv \in \{C, F\} \rightarrow ins = C \quad (2)$$
$$start(mv = Fo) \rightarrow ins = Ao \quad (3)$$
$$start(mv = Fd) \rightarrow ins = Ad \quad (4)$$
$$start(mv = Ad) \rightarrow (\neg hnav \vee \neg hpow \vee pert) \quad (5)$$
$$start(mv = F) \rightarrow \neg hpow \quad (6)$$
$$start(ins = Ad) \rightarrow (\neg hsens \vee \neg hpow \vee pert) \quad (7)$$
$$start(ins = F) \rightarrow \neg hpow \quad (8)$$
$$hnav \vee hsens \rightarrow hpow \quad (9)$$

Action ins must remain in W while action mv executes (1)*. If mv fails or is cancelled, ins is cancelled* (2)*. ins starts at the moment when mv finishes* (3), (4)*. A delay in mv (resp. ins) can be explained by a navigation (resp. sensor) or power supply problem, or a perturbation* (5) *(resp.* (7)*). A failure in mv or ins can only be explained by a problem in the power supply* (6)*,* (8)*. A problem in the power supply propagates to the sensor and navigation* (9)*.*

Variables mv and ins are observable, i.e. the state of each action is known. Variables $hnav$, $hsens$, $hpow$ and $pert$ are estimated. The set of states S is the set of valuations for all variables, the obs function restricts a valuation to variables mv and ins. For example, the initial state $(mv = W, ins = W, hnav, hsens, hpow, \overline{pert})$ yields the observation $(mv = W, ins = W)$.

Our algorithm is implemented in Scala, and executed on an Intel® Xeon(R) W-2123, 3.60GHz 8 core processor, with memory limited to 4 gigabytes. The system as described in Example 2 is trackable, so we introduced some modifications to make it non-trackable. We made actions show the same observation in their "Ao" and "Ad" states.

Model	States	Succ.	Result	Time (s)
Example 2	112	13.7	yes	66
Example 2-modified	112	13.7	no	0.4
Valve controller	209	15.5	no	0.4
Valve driver	51	22.3	yes	56

Table 1: Computation times for checking trackability. Column "States" indicates the number of states in the system, "Succ." the average number of outgoing transitions for each state, "Result" whether the system is trackable, and "Time" the computation time in seconds.

We also modelled the example systems from [11] (Valve controller) and [7] (Valve driver).

Results are presented in Table 1 and show that non-trackable systems are detected very quickly. This is due to our preliminary check described in section 6.3 that catches it early. While propositions 1 and 2 are not sufficient to ensure trackability, they catch many cases. For trackable systems, the computation time is related to the number of partial estimation functions tested. Note that it can be made faster by taking into account operational requirements to limit the space of estimation functions to explore.

8 Conclusion

This paper motivates and defines single state trackability for partially observed discrete event systems. This property states that it is possible to track the execution of a system by keeping a unique state in memory, without ever losing consistency with its dynamics. Some related conditions are provided along with an algorithm for checking this property. Experimental results exemplify this algorithm applied to autonomous robots. The algorithm is a proof of concept of the approach but could be improved in many ways, for instance by defining specific heuristics to make it more efficient. Larger benchmarks should also be tested.

Through this paper, single state trackability is achieved by finding one estimation function whose observation language matches that of the system. In general, as there might be many such functions, we could look for the best estimation function in regard to other properties, for example estimation correctness for some variables or at some instants.

Work about observability and diagnosability often account for a possible bounded delay between events and their observation or diagnosis. If we could account for delay in the estimation process, we could address a more general single state trackability definition. The theoretical study of the complexity of the single state trackability existence is also part of our prospects.

In addition, we are convinced that our work is strongly linked with controller synthesis [18], in particular when framed in the game theory framework [19; 20]. These links suggest future work for precise comparison and possible enhancement with ideas from these different areas.

Finally, automatic or semi-automatic synthesis of estimation functions is a direct application of this work, for example by representing estimation function with compact languages such as in [9].

References

[1] WM Wonham, Kai Cai, and Karen Rudie. Supervisory control of discrete-event systems: A brief history. *Annual Reviews in Control*, 2018.

[2] Janan Zaytoon and Stéphane Lafortune. Overview of fault diagnosis methods for discrete event systems. *Annual Reviews in Control*, 37(2):308–320, 2013.

[3] Alban Grastien, Marie-Odile Cordier, and Christine Largouët. Incremental diagnosis of discrete-event systems. In *Proceedings of the 29th International Workshop on Principles of Diagnosis DX'05*, Pacific Grove, CA, USA, 2005.

[4] Alban Grastien, J Rintanen Anbulagan, Jussi Rintanen, Elena Kelareva, et al. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, volume 22, page 305, Vancouver, British Columbia, 2007.

[5] Peter J Ramadge. Observability of discrete event systems. In *Proceedings of the 25th IEEE Conference on Decision and Control CDC'86*, pages 1108–1112, Athens, Greece, 1986. IEEE.

[6] C. M. Ozveren and A. S. Willsky. Observability of discrete event dynamic systems. *IEEE Transactions on Automatic Control*, 35(7):797–806, July 1990.

[7] C. Brian Williams and P Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, Portland, Oregon, 02 1996.

[8] James Kurien and P Pandurang Nayak. Back to the future for consistency-based trajectory tracking. In *Proceedings of the 17th AAAI Conference on Artificial Intelligence*, pages 370–377, Austin, Texas, USA, 2000.

[9] Valentin Bouziat, Xavier Pucel, Stéphanie Roussel, and Louise Travé-Massuyès. Preferential discrete model-based diagnosis for intermittent and permanent faults. In *Proceedings of the 29th International Workshop on Principles of Diagnosis DX'18*, Warsow, Poland, August 2018. CEUR Workshops Proceedings.

[10] Valentin Bouziat, Xavier Pucel, Stéphanie Roussel, and Louise Travé-Massuyès. Preference-based fault estimation in autonomous robots: Incompleteness and meta-diagnosis - extended abstract. In *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019)*, pages 1841–1843, Montreal, Canada, May 2019.

[11] Meera Sampath, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on automatic control*, 40(9):1555–1575, 1995.

[12] Anika Schumann, Yannick Pencolé, et al. Scalable diagnosability checking of event-driven systems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence IJCAI'00*, volume 7, pages 575–580, Hyderabad, India, 2007.

[13] Olivier Contant, Stéphane Lafortune, and Demosthenis Teneketzis. Diagnosis of intermittent faults. *Discrete Event Dynamic Systems*, 14(2):171–202, 2004.

[14] Johan De Kleer. Diagnosing multiple persistent and intermittent faults. In *Proceedings of the 21th International Joint Conference on Artificial Intelligence IJCAI'19*, Pasadena, CA, USA, June 2009.

[15] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[16] Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

[17] Vandi Verma, Tara Estlin, Ari Jónsson, Corina Pasareanu, Reid Simmons, and Kam Tso. Plan execution interchange language (plexil) for executable plans and command sequences. In *Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space iSAIRAS'05*, Munich, Germany, 2005.

[18] Cédric Pralet, Gérard Verfaillie, Michel Lemaître, and Guillaume Infantes. Constraint-based controller synthesis in non-deterministic and partially observable domains. In *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 681–686, Amsterdam, The Netherlands, 2010.

[19] Doyen Laurent and Jean-François Raskin. Games with imperfect information: Theory and algorithms. In *Lectures in Game Theory for Computer Scientists*, page 185–212, Cambridge, 2011.

[20] Dietmar Berwanger and Anup Basil Mathew. Infinite games with finite knowledge gaps. *Inf. Comput.*, 254(P2):217–237, June 2017.