Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020)
Main Track

Strategy Synthesis for Data-Aware Dynamic Systems with Multiple Actors

Massimiliano de Leoni[1] , **Paolo Felli**[2] , **Marco Montali**[2]
[1]University of Padua - Padua, Italy
[2]Free University of Bozen-Bolzano - Bolzano, Italy
deleoni@math.unipd.it, {pfelli,montali}@unibz.it

Abstract

The integrated modeling and analysis of dynamic systems and the data they manipulate has been long advocated, on the one hand, to understand how data and corresponding decisions affect the system execution, and on the other hand to capture how actions occurring in the systems operate over data. KR techniques proved successful in handling a variety of tasks over such integrated models, ranging from verification to online monitoring. In this paper, we consider a simple, yet relevant model for data-aware dynamic systems (DDSs), consisting of a finite-state control structure defining the executability of actions that manipulate a finite set of variables with an infinite domain. On top of this model, we consider a data-aware version of reactive synthesis, where execution strategies are built by guaranteeing the satisfaction of a desired linear temporal property that simultaneously accounts for the system dynamics and data evolution.

1 Introduction

Many complex systems can be described in terms of their constituent activities, and how the resulting execution is affected by the data that these activities manipulate. This is the case of integrated models for business processes and data, extensively studied in the last two decades within business process management and database theory (Reichert 2012; Calvanese, De Giacomo, and Montali 2013). AI techniques have been extremely successful to handle a variety of reasoning tasks over such integrated models, ranging from static analysis (Bagheri Hariri et al. 2013; De Masellis et al. 2017; Abdulla et al. 2019) to online monitoring (De Masellis, Maggi, and Montali 2014; Maggi et al. 2011).

In parallel with this line of research, a number of weaker data-aware models were developed, with a twofold intention: on the one hand, to capture and formally analyze decision-driven processes operating over non-persistent data (Batoulis, Haarmann, and Weske 2017); on the other hand, to identify models that can be learned from event data using process mining techniques (van der Aalst 2016). This led to a balanced model for data-aware dynamic systems expressive enough to capture decision-driven processes (de Leoni, Felli, and Montali 2018), and simple enough to be learned from event data by combining control-flow discovery and decision tree mining techniques (Mannhardt et al. 2016). In this model, a finite-state control structure defines the behavior of the system in terms of the actions that are allowed in each control state. These actions operate over a finite set of variables with infinite domain and comparison predicates, and their executability is also guarded by the current values of these variables. The evolution of the system is then nondeterministic when it comes to resolve decision points with multiple enabled conditions, or to choose which value to pick when updating the content of a variable. While in reality these different sources of nondeterminism are typically controlled by multiple, autonomous actors, verification of these models has so far always adopted the simplifying assumption that these actors cooperate.

In this work, we relax this assumption, and show how to combine well-established strategy synthesis approaches for temporal specifications (Vardi 1996; Pnueli and Rosner 1989) with faithful data abstraction methods, towards a *constructive, readily implementable technique for strategy synthesis in data-aware dynamic systems* (DDSs).

Technically, we look at the usual reactive approach (Pnueli and Rosner 1989) captured as a two-player adversarial setting, which gives us the ability to consider an actor, operating within the decision-driven processes, which can only control a subset of the actions and, similarly, is responsible for updating a subset of the variables. The objective is to compute a refinement of the DDS which meets the desired specification on every trace. Such objective is here captured as the synthesis of strategies that induce these refinements. Computing refinements is of particular importance in a data-aware setting, as the interplay of actions and data may lead to unforeseen results that are not evident on the model itself. Execution strategies are built by guaranteeing the satisfaction of a specification expressed as a linear temporal property that simultaneously accounts for the system dynamics and data evolution. Differently from the usual case of simple propositional labels, we need to consider here an alphabet that is composed not only of action symbols but is also enriched with data conditions including variable-to-constant and variable-to-variable comparisons. For instance, we may want to check the formula $\Diamond(x > 0)$ which, intuitively, is satisfied by runs of a DDS in which the variable x eventually takes a positive value. In line with the intuition that "each system execution must eventually terminate", we adopt a finite-trace semantics and consider a temporal language that is based on LTL$_f$ (De Giacomo and Vardi 2013).

The paper is organised as follows. In Section 2 we discuss the relevance of our approach and related work. In Section 3 we formalise DDSs and their executions. In Section 4 we present our specification language and its semantics, defining when a DDS satisfies a formula. In Section 5 we introduce our adversarial setting and strategy synthesis task and in Section 6 we define a finite abstraction of the runs of a DDS. Finally, we show how to adapt an automata-based technique to the synthesis of strategies for a given formula.

2 Related Work

The approach studied here has a general applicability, and the formal verification of temporal specifications in a data-aware setting is beneficial for and explored in several application domains, including Business Process Management (de Leoni and Mannhardt 2019; van der Aalst 2016; Reichert 2012), Model-driven Engineering (see e.g. (Tikhonova 2017)), or when there is the need to verify the composition of different machines, e.g., to produce chip wafers (van Gool et al. 2006). In particular, the verification of data-aware business processes is gaining momentum, triggered by the latest standards for modelling data and decisions, such as the DMN standard (Decision Model and Notation) by OMG (the Object Management Group, an international not-for-profit technology standards consortium). Works such as (Yousfi, Batoulis, and Weske 2019; Maggi et al. 2011; Calvanese, De Giacomo, and Montali 2013; De Masellis et al. 2017; de Leoni, Felli, and Montali 2018) are example of this large repertoire. In spite of the fact that business processes typically involve multiple, possibly non-cooperating parties, very little research in this context was dedicated to date towards strategic reasoning. Our model is analogous to existing models in the literature (Mannhardt et al. 2017), and is hence directly applicable.

From the logical point of view, our work has connections with known verification settings such as that of constraint LTL (CLTL) (Demri and D'Souza 2007) and of gap-order constraints (e.g., (Bozzelli and Pinchinat 2014)), although these adopt an infinite-runs semantics.

Transitional gap order constraints systems (Bozzelli and Pinchinat 2014) are abstract model of counter machines with infinite branching and constraints of the form $x - y < k$, where x, y are integers variables or constants and k is a natural number. In this setting, decidability results exist for the satisfiability and verification of fragments of CTL*, which are PSPACE-complete. Notwithstanding the distinct infinite/finite semantics, there are certainly relationships between our setting and theirs, although the nature of our synthesis task with partial controllability on variables is not addressed, to the best of our knowledge, by this literature.

The CLTL framework is closer to the model we study here, hence we provide a more detailed comparison. CLTL allows one to specify linear temporal properties with constraints on variables, such as $\Diamond(x < OOx)$, which intuitively expresses that, eventually, the value of a variable x is smaller than its value in the second-next state. Also for the case of real-valued variables, CLTL is more expressive than the logic we consider here, where constraints do not contain arbitrary LTL temporal operators and can only compare the current and next values of variables. Notwithstanding the difference in terms of finite vs infinite semantics, these logics are similar in spirit. Concrete models of CLTL formulae are sequences of variable valuations that are abstracted by sequences of frames (i.e., maximally consistent sets of constraints), which is a similar idea to the abstraction technique used in this paper to tame the data dimension. However, the task considered in (Demri and D'Souza 2007) and following work is satisfiability. In order to exploit these results, one could follow a procedure similar to the one for standard LTL/LTL$_f$. First, one would need to represent a DDS as a finite structure whose transitions are labelled by CLTL formulae restricted to depth at most two (such as $x = y$ or $Ox > x$), capturing data tests and updates needed for our purposes, as well as laws of inertia used to preserve the content of variables that are not updated. Second, one would need to translate the DDS itself, conceived as an automaton, into a CLTL formula to be conjoined with the property to verify (Demri and D'Souza 2007, Sec. 8). However, this approach would only tackle verification and it would not be sufficient for operatively computing strategies even in a single-actor setting. Indeed, the existence of a sequence of frames (abstracting valuation sequences) does not allow one to extract a strategy that selects only those valuation sequences that are models of the formula. In fact, at each step one cannot incrementally compute the valuation sequence (towards building a CLTL model, and in turn a strategy) by simply selecting a valuation that is a solution of the variable conditions in the next frame. Instead, one needs to process the entire frame sequence so as to foresee at once all the possible future executions.

Moreover, there are no actions in CLTL. Encoding actions through either variables or constants, and then labelling transitions only by constraints, is not a viable approach. Indeed, this would introduce unwanted nondeterminism, beyond that already present in the model (which is due to non-controllable actions and variable updates). This would both fail to correctly model the process at hand and also prevent us from defining controllable executions. Instead of tailoring a CLTL-based approach to account for these issues, we study a faithful abstraction that is natively defined over the concrete runs of DDSs, so that we can directly solve the more general problem of strategy synthesis, obtaining results on verification as a by-product. Local consistency, a fundamental property assumed for CLTL frame sequences, is here enforced during the computation of the abstraction. By adopting this approach, we are able to describe the exact required computations (which are directly implementable), following a well-understood technique, in a modelling framework that is of direct applicability for real application domains based on data/decisions and actions.

3 Data-Aware Dynamic Systems

Data-aware dynamic systems (DDSs) consist of a finite-state structure with state-transitions that are labelled not only by action symbols, as in regular state-transition systems, but also by conditions expressed on a finite set of variables.

This is a representation equivalent to that introduced in previous work (de Leoni, Felli, and Montali 2018; Felli, de Leoni, and Montali 2019), where bounded *Petri nets with* *data* are used. It is well known that, when a Petri net is bounded (i.e., when there is a bound on the number of tokens in the places of each reachable marking), by interpreting concurrency as interleaving we can represent the execution semantics of the net, namely its control-flow, as a finite-state transition system (Reisig 2013). These Petri nets with data are known to be as expressive as BPMN (the Business Process Model and Notation) with case data and without events and messages. BPMN is the de-facto modelling standard for the BPM domain for which such nets were devised (Kalenkova et al. 2019).

Before defining the DDSs themselves, we need to define the shape of data conditions. We restrict to the domain of reals $\langle \mathbb{R}, P\rangle$, where $P = \{<, >, =, \neq, \leq, \geq\}$ is a set of binary comparison predicates on $\mathbb{R}$, although our technique would seamlessly work with additional finite domains such as $\langle\{\mathit{true}, \mathit{false}\}, \{=, \neq\}\rangle$ or further domains that are dense, e.g. $\langle \mathbb{Q}, P\rangle$. This restriction is required to guarantee that, for each operator $\odot \in P$ that induces a total ordering on the domain values, if $k \odot k'$ then there exists another value k'' such that $k \odot k'' \odot k'$. Intuitively, this property guarantees that there is an infinite supply of fresh domain values in each closed interval (k, k'), a property we require for our interval-based abstraction when the process we are modelling is cyclic. Analogous conditions are required for related approaches in the literature, such as the completion property in (Demri and D'Souza 2007). For arbitrary additional domains we also require the predicates to be effectively computable and closed under negation (as we often need to negate data conditions).

Given a set V of variables and $v \in V$, we write v^r or v^w to denote that v is read (resp., written) by an action, hence we consider two variable sets V^r and V^w defined as $V^r = \{v^r \mid v \in V\}$ and $V^w = \{v^w \mid v \in V\}$. This provides the basic building block for defining data conditions constraining the evolutions of the DDS, as well as the information on the current values of such variables. We call the former *guards* (data conditions for defining the executability of actions) and the latter *constraints* ("static" data conditions).

Definition 1. *Given a finite set V, the set Guards_V of possible* guards *is the set containing:*
- $v \odot k$ *iff* $v \in (V^r \cup V^w)$*,* $k \in \mathbb{R}$ *and* $\odot \in P$*;*
- $v_1 \odot v_2$ *iff* $v_1 \in (V^r \cup V^w)$*,* $v_2 \in V^r$ *and* $\odot \in P$*;*

A guard of the form $(v^r \odot k)$, with $k \in \mathbb{R}$, captures a condition which will be associated to some action (see Definition 3), requiring that the current value of the variable v is compared to k through $\odot$. For instance, $(\mathtt{a}^r \geq \mathtt{0})$ expresses that the current value of $\mathtt{a}$ is greater or equal to $\mathtt{0}$. Similarly, $(v^w \odot k)$ denotes the condition imposing a restriction on the new value of variable v, that is being written. For instance, $(\mathtt{a}^w > \mathtt{0})$ specifies that the new value of $\mathtt{a}$ will be positive. Guards of the form $(v_1^r \odot v_2^r)$ and $(v_1^w \odot v_2^r)$ are analogous, but compare v_1 to the current value of variable v_2. In this paper, k is only used to denote constants.

Variables in V that are read and written by a guard g are respectively denoted by $\mathit{read}(g)$ and $\mathit{write}(g)$. For example, $\mathit{read}((\mathtt{a}^r \odot \mathtt{b}^r)) = \{\mathtt{a}, \mathtt{b}\}$, $\mathit{write}((\mathtt{a}^w \odot \mathtt{b}^r)) = \{\mathtt{a}\}$, $\mathit{read}((\mathtt{a}^w \odot \mathtt{b}^r)) = \{\mathtt{b}\}$, $\mathit{write}((\mathtt{a}^r \odot \mathtt{b}^r)) = \emptyset$.

Definition 2. *Given a finite set V, the set $\mathcal{C}_V$ of possible* constraints *is the set containing:*
- $v \odot k$ *iff* $v \in V$*,* $k \in \mathbb{R}$ *and* $\odot \in P$*;*
- $v_1 \odot v_2$ *iff* $v_1, v_2 \in V$ *and* $\odot \in P$*.*

A constraint allows to relate a variable with a constant or with another variable. The distinction between guards and constraints is only needed to facilitate the technical content and ease the notation and terminology. Given either a guard or a constraint c, we denote by $\neg c$ the guard or constraint in which the predicate in c is replaced by its negation.

A *guard variable assignment* is a function $\beta : (V^r \cup V^w) \mapsto \mathbb{R}$, which assigns a value to read and written variables. As the name suggests, these assignments are used to specify the values of variables for evaluating the guards associated to actions, as we intuitively described above. In general, this requires to compare previous and current values. Given a guard variable assignment β and a guard g of the form $(v^r \odot k)$, we say that g is satisfied when variables are substituted as per β, i.e., $k' = \beta(v^r)$ and $k' \odot k$. If the guard is of the form $(v_1^r \odot v_2^r)$, this requires that $k_1 \odot k_2$ with $k_1 = \beta(v_1^r)$, $k_2 = \beta(v_2^r)$. The case for $(v_1^w \odot v_2^r)$ is analogous. This is denoted by writing $g_{[\beta]} = \mathtt{true}$.

For instance, a guard $(v^w > v^r)$ imposes that v is updated with a value greater than its current value. Given a guard variable assignment β with $\beta(v^w) = 3$ and $\beta(v^r) = 2$, we have that $(v^w > v^r)_{[\beta]} = \mathtt{true}$.

A *constraint variable assignment* is instead a function $\alpha : V \mapsto \mathbb{R}$, which assigns a value to each variable $v \in V$. The difference with a guard variable assignment β is that while β is used for evaluating action guards, a constraint variable assignment holds the current value of each variable in V.

With these notions at hand, we now formally define DDSs as labelled transition systems equipped with a finite set of variables and with guards defining data-aware preconditions and effects attached to the actions.

Definition 3. *A DDS $\mathcal{B} = \langle B, b_0, \mathcal{A}, T, F, V, \alpha_0, \mathit{guard}\rangle$ is a labelled transition system where:*
- *B is a finite set of system states, with b_0 the initial one;*
- *$\mathcal{A}$ is a finite set of actions;*
- *$T : B \times \mathcal{A} \mapsto B$ is a transition function, and we denote that the DDS reaches a new system state b' through the execution of a from a system state b by $b \xrightarrow{a} b'$;*
- *$F \subseteq B$ is the set of final states, for which we require that no outgoing transition exists (but not all states without outgoing transitions must be in F);*
- *V is a finite set of variables;*
- *α_0 is the initial constraint variable assignment;*
- *$\mathit{guard} : \mathcal{A} \mapsto \mathit{Guards}_V$ specifies a guard for each action.*

It follows that each action a in a DDS is always assigned the same guard $\mathit{guard}(a)$. If needed, however, it is always possible to introduce a fresh copy a' of a whenever we need to model distinct guards from distinct system states of the DDS, i.e., so that $\mathit{guard}(a) \neq \mathit{guard}(a')$.

Given $a \in \mathcal{A}$, as a shorthand we write $\mathit{read}(a) \doteq \{v \in V \mid v \in \mathit{read}(\mathit{guard}(a))\}$, and analogously $\mathit{write}(a) \doteq \{v \in V \mid v \in \mathit{write}(\mathit{guard}(a))\}$.

A system state in F, when reached, indicates the conclusion of the execution. This allows us to model the condi-

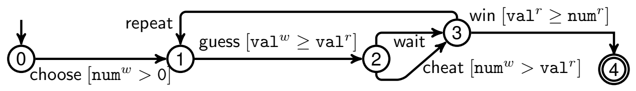
repeat
win $[\mathtt{val}^r \geq \mathtt{num}^r]$
guess $[\mathtt{val}^w \geq \mathtt{val}^r]$
wait
0
1
2
3
4
choose $[\mathtt{num}^w > 0]$
cheat $[\mathtt{num}^w > \mathtt{val}^r]$

Figure 1: A simple DDS $\mathcal{B}$ with $F = \{4\}$, $\alpha_0(\mathtt{num}) = \alpha_0(\mathtt{val}) = \mathtt{0}$. Guards are between brackets (assuming a tautological guard when none is specified).

tions, on the control-flow and data variables, under which the execution may terminate. This is done for simplicity: arbitrary final states as in regular transition systems are possible, as it is sufficient to introduce additional transitions from states that are intended to be final to sink final states.

Let us consider as an example the simple DDS in Figure 1. A number is arbitrarily chosen and assigned to variable num by executing an action choose, then a second number is guessed and assigned to variable val via an action guess, with the condition that the chosen value for val is greater or equal to the current value of this variable (i.e., greater or equal to the value chosen in the last iteration, if this exists, or 0 if no previous iteration exists). If the guessed number is indeed at least as large as the chosen number, then some prize is won by executing an action win. Before this test, however, an action cheat can be executed to make the guess always wrong. An action wait can be executed instead of cheating, with no effect on variables.

Next, we define the runs of a DDS. The set of possible *configurations* of $\mathcal{B}$ is the set of all pairs (b, α) where b is a system state of $\mathcal{B}$ and α is the (current) constraint variable assignment. Intuitively, from a configuration (b, α), an action a can be executed reaching the new configuration (b', α') if: *(i)* the guard associated to a is satisfied, *(ii)* $b \xrightarrow{a} b'$, and *(iii)* the new values of variables specified by α' result from updating the variables according to the guard of a. A pair (a, β) where $a \in \mathcal{A}$ and β is a guard variable assignment is called *action firing*.

Definition 4. *A DDS $\mathcal{B} = \langle B, b_0, \mathcal{A}, T, F, V, \alpha_0, guard\rangle$ evolves from configuration (b, α) to configuration (b', α') by action firing (a, β) iff $b \xrightarrow{a} b'$ and:*

- *$\beta(v^r) = \alpha(v)$ for every variable $v \in read(a)$ that is read, i.e., the value assigned by the guard variable assignment β must coincide with that in (b, α);*
- *the new constraint variable assignment α' is as α but updated as per β for the variables that are written. Namely, for all $v \in V$, we have $\alpha'(v) = \alpha(v)$ if $v \notin write(a)$, otherwise $\alpha'(v) = \beta(v^w)$;*
- *$guard(a)_{[\beta]} = \mathtt{true}$: the guard of a is satisfied by β.*

Essentially, an action firing fully specifies an action execution: it specifies the action label and all the variable values *before and after* the action is executed. For instance, referring again to Figure 1, from the initial configuration $(\mathtt{0}, \{\alpha(\mathtt{num}) = \mathtt{0}, \alpha(\mathtt{val}) = \mathtt{0}\})$, the action firing (choose, β) so that $\beta(\mathtt{num}^w) = \mathtt{7}$ results in the new configuration $(\mathtt{1}, \{\alpha'(\mathtt{num}) = \mathtt{7}, \alpha'(\mathtt{val}) = \mathtt{0}\})$.

An action firing (a, β) as in Definition 4 is termed *legal* in (b, α) because, intutively, β "agrees with" α. We denote a legal action firing (a, β) from a configuration (b, α) to a configuration (b', α') by writing $(b, \alpha) \xrightarrow{a,\beta} (b', \alpha')$. We also extend this definition to *runs* of the form $\rho = (b_0, \alpha_0) \xrightarrow{a_1,\beta_1} \ldots \xrightarrow{a_n,\beta_n} (b_n, \alpha_n)$. A run of $\mathcal{B}$ is a run as above starting from (b_0, α_0). As we consider runs of unbounded length but finite, we denote the length of a run ρ by $length(\rho)$. We also denote the positions of a run ρ, i.e., the configurations traversed by ρ, as $\rho[i]$, provided that $i \in [0, length(\rho){-}1]$. We use the abbreviation $last(\rho) \doteq length(\rho){-}1$.

Finally, we characterize all possible evolutions of a given DDS by considering all its possible runs.

Definition 5. *Given a DDS $\mathcal{B} = \langle B, b_0, \mathcal{A}, T, F, V, \alpha_0, guard\rangle$, the reachability graph of $\mathcal{B}$ is the graph $RG_{\mathcal{B}} = \langle W, w_0, E\rangle$ where:*

- *W is the (possibly infinite) set of configurations reachable by legal runs from $w_0 = (b_0, \alpha_0)$;*
- *E is a transition function so that $(b, \alpha) \xrightarrow{a,\beta} (b', \alpha')$ is an edge iff (a, β) is a legal action firing from (b, α) in $\mathcal{B}$.*

From now on, we use sets of constraints as in Definition 2 (henceforth called *constraint sets*), to encode conditions on the variables of the DDS, at each step. A *solution of a constraint set* C is a constraint variable assignment α such that for each $(v \odot k) \in C$ we have $\alpha(v) \odot k$ and, for each $(v_1 \odot v_2) \in C$, we have $\alpha(v_1) \odot \alpha(v_2)$.

4 Specification Language

Given a DDS $\mathcal{B}$, let $\mathcal{L}_{\mathcal{B}}$ be the language with grammar:

$$\psi = \mathtt{true} \mid C \mid b \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \langle a\rangle\psi \mid \Diamond\psi \mid \Box\psi$$

where $a \in \mathcal{A}$, C is a constraint set over the variables in $\mathcal{B}$ and $b \in B$ is a system state of $\mathcal{B}$. We now give the semantics on finite runs on $RG_{\mathcal{B}}$, for expressing properties on these runs. For brevity, in what follows it is often convenient to represent a constraint variable assignment α as a constraint set. Hence we define $C_\alpha \doteq \bigcup_{v \in V}\{(v = \alpha(v))\}$.

Intuitively, a formula $\psi = C$ is true when C is satisfiable together with the current constraint variable assignment α in the run of $RG_{\mathcal{B}}$, i.e., constraint variable assignment is a solution of C ($C \cup C_\alpha$ is satisfiable). Similarly, an atomic formula b requires the current system state to be b. $\langle a\rangle\psi$ requires that ψ is true in the run after executing action a (in the next configuration, which must exist). $\Box$ and $\Diamond$ are read as 'for each step in the run' and 'eventually in the run'.

Definition 6. *Given a run ρ in $RG_{\mathcal{B}}$ and a formula ψ in $\mathcal{L}_{\mathcal{B}}$, we say that ρ satisfies ψ, written $\rho \models \psi$, iff $\rho, 0 \models \psi$ according to the following semantics, where $i \in [0, last(\rho)]$:*

$\rho, i \models \mathtt{true}$
$\rho, i \models C$ *iff* α is a solution of C, with $\rho[i] = (b, \alpha)$
$\rho, i \models b$ *iff* $\rho[i] = (b, \alpha)$ for some α
$\rho, i \models \neg\psi$ *iff* $\rho, i \not\models \psi$
$\rho, i \models \psi_1 \wedge \psi_2$ *iff* $\rho, i \models \psi_1$ and $\rho, i \models \psi_2$
$\rho, i \models \psi_1 \vee \psi_2$ *iff* $\rho, i \models \psi_1$ or $\rho, i \models \psi_2$
$\rho, i \models \langle a\rangle\psi$ *iff* $\rho[i] \xrightarrow{a,\beta} \rho[i{+}1]$ and $\rho, i{+}1 \models \psi$
$\rho, i \models \Diamond\psi$ *iff* $\exists j$ s.t. $i \leq j \leq last(\rho)$ and $\rho, j \models \psi$
$\rho, i \models \Box\psi$ *iff* $\forall j$ s.t. $i \leq j \leq last(\rho)$ we have $\rho, j \models \psi$

A run ρ of $RG_{\mathcal{B}}$ is *terminal* iff it ends in a configuration (b, α) so that $b \in F$. As final system states in $\mathcal{B}$ have no outgoing edges, terminal runs cannot be extended further.

A <u>*witness*</u> for a formula ψ in the reachability graph $RG_\mathcal{B}$ is a terminal run of $RG_\mathcal{B}$ satisfying ψ.

A <u>*verification problem*</u> is a pair $\langle \mathcal{B}, \psi \rangle$ where $\mathcal{B}$ is a DDS and ψ is the formula in $\mathcal{L}_\mathcal{B}$ which we want to verify. Hence our first reasoning task, denoted as *T1*, is the following:

Definition 7. *Given a verification problem $\langle \mathcal{B}, \psi \rangle$, (T1:) check whether there exists a witness for ψ in $RG_\mathcal{B}$.*

Further, we define when a formula is true in $RG_\mathcal{B}$. This brings together the usual notion of satisfaction of linear temporal formulae with a correctness requirement for DDSs inspired to that of "possibility of termination" for data-aware processes (Felli, de Leoni, and Montali 2019), i.e., that a final system state can always be reached.

Definition 8. *Given $RG_\mathcal{B}$, we say that $\psi \in \mathcal{L}_\mathcal{B}$ is true in $RG_\mathcal{B}$, written $RG_\mathcal{B} \models \psi$, iff every run in $RG_\mathcal{B}$ is a prefix of a terminal run, and $\rho \models \psi$ for every terminal run ρ in $RG_\mathcal{B}$.*

5 Adversarial Setting

Before detailing our technique, we introduce a further reasoning task (*T2*), then show that the very same approach works for both *T1* and *T2*. We first give an intuition. Assume that the system is in fact a multi-agent system with a single-agent perspective, i.e., consider an agent (actor) ag that has some control on the execution of the DDS, so that all other aspects that cannot be controlled by ag are considered as controlled by the *environment* (an abstract antagonist). Our objective is to: *(i)* characterise which are the fragments (or refinements) of $RG_\mathcal{B}$ that ag can enforce; *(ii)* represent such fragments as *execution strategies* for ag; *(iii)* define that ag can force a formula to be true in $RG_\mathcal{B}$ when such formula is true in a possible fragment that ag can enforce.

Formally, consider a subset $A_{ag} \subseteq \mathcal{A}$ of actions and a subset $V_{ag} \subseteq V$ of variables that are *under the control of* ag. Intuitively, only actor ag can decide to execute actions in $\mathcal{A}_{ag}$ and similarly it is the responsibility of actor ag to decide the new value of variables in V_{ag} whenever these are written. To avoid conflicts, we impose one restriction on DDSs: all actions available from any system state $b \in B$ (i.e. the set $\{a \mid b \xrightarrow{a} b' \in T\}$) are controlled either by the actor or by the environment. We refer to the former case by saying that ag controls b, and similarly we say that ag controls (b, α) in $RG_\mathcal{B}$ iff ag controls b.

Definition 9. *Given $\mathcal{B}$, the actor ag can <u>enforce</u> a set of action firings X from a configuration (b, α) of $RG_\mathcal{B}$ iff:*
- *if ag controls (b, α) then for all $(a, \beta) \in X$ either:*
  - *$write(a) = \emptyset$ or $write(a) \subseteq V_{ag}$ or*
  - *for every β', $(b, \alpha) \xrightarrow{a,\beta'} w$ implies $(a, \beta') \in X$;*
- *if ag does not control (b, α) then for all $(a, \beta) \in X$ either:*
  - *$write(a) \subseteq V_{ag}$, or*
  - *for every β', $(b, \alpha) \xrightarrow{a,\beta'} w$ implies $(a, \beta') \in X$;*

  and there is no action a' so that $(b, \alpha) \xrightarrow{a',\cdot} w$ for which there is no β' so that $(a', \beta') \in X$.

Intuitively, the actor can always act so that, no matter how the environment evolves, the selected action firing will be in X. Since action firings are deterministic, this implies that the actor can force the next configuration to be in a certain subset. We denote the set of all sets of action firings that can be enforced by ag from a configuration w (of the form (b, α)) as $Ctrl^w_{ag}$. Note that it is often possible that the set of action firings that can be enforced by ag from a configuration w is not unique (i.e., $|Ctrl^w_{ag}| > 1$) but it is closed under union. Finally, $Ctrl^w_{ag}$ does not depend on the sequence of configurations (called history) that led to w.

Then, we turn to define the executions that an actor can enforce. We can model this as a <u>*system strategy*</u> for ag: it is a partial function $strat$ which, given the history $w_0, \ldots, w_n$ of configurations of $RG_\mathcal{B}$, either returns a set of legal action firings that can be enforced by ag or it is undefined (if w_n is terminal). If we define $Ctrl^W_a \doteq \{Ctrl^w_{ag} \mid w \in W\}$ then $strat : W^* \mapsto Ctrl^W_a$ so that $strat(w_0, \ldots, w_n) \in Ctrl^{w_n}_{ag}$.

We denote by $RG^{strat}_\mathcal{B}$ the (refined) reachability graph obtained by executing $strat$ on $RG_\mathcal{B}$, that is, so that a run $w_0 \xrightarrow{a_1,\beta_1} \cdots \xrightarrow{a_n,\beta_n} w_n$ of $RG_\mathcal{B}$ is in $RG^{strat}_\mathcal{B}$ iff $(a_{i+1}, \beta_{i+1}) \in strat(w_0, \ldots, w_i)$ for $i \in [0, n{-}1]$. Since multiple functions $strat$ may exist then multiple refinements are possible, but we do not need to actually compute them. They are needed to define task *T2*: an <u>*adversarial problem*</u> is a tuple $\langle \mathcal{B}, \psi, V_{ag}, A_{ag} \rangle$ where $\langle \mathcal{B}, \psi \rangle$ is a verification problem, $V_{ag} \subseteq V$ and $A_{ag} \subseteq A$.

Definition 10. *Given $\langle \mathcal{B}, V_{ag}, A_{ag}, \psi \rangle$ (T2:) compute a system strategy strat for ag so that $RG^{strat}_\mathcal{B} \models \psi$ (see Def. 8).*

For instance, in Fig. 1 we might be interested in checking whether the actor ag can win with (i) a guess smaller than 3 and (ii) with `num` $=$ `val`. Clearly, if ag controls actions `wait` and `cheat`, and variable `val`, then (ii) is possible. For (i), the actor also needs to control at least `num`, so that it can choose its value to be smaller than 3. However, we cannot compute strategies on $RG_\mathcal{B}$ because this structure is infinite-state in general (its runs are infinite in number and in length).

6 Constraint Graph

In this section we provide a *finite, faithful representation* of all the possible runs in $RG_\mathcal{B}$, adopting a form of interval abstraction for the domain of variables, so that we can solve our tasks on such abstraction rather than on $RG_\mathcal{B}$.

First, in order to model the effects of action firings on the current values of variables, we need a procedure to update constraint sets according to the guard $g = guard(a)$ of the selected action a. We denote such operation as $C' = C \oplus g$ and we give a simple pseudocode in Algorithm 1.

In the algorithm we write `saturate`(C) to denote a *saturation* procedure that returns another constraint set C' containing all the constraints logically implied by C (using only variables and constants in C). Note that $C' = C \oplus g$ is a constraint set, as it does not contain guards. As the constraint language is based on comparisons, it follows that finitely many constraints can be added. To guarantee uniqueness, we assume a canonical ordering of variables and constraints.

Algorithm 1 is as follows: if the guard does not involve written variables, the constraint corresponding to g is simply added. If it involves a written variable (line 6) then we temporarily add to C a new constraint in which we use a

Algorithm 1: Procedure for computing $C' \doteq C \oplus g$

```
1  input: constraint set $C$ and guard $g$
2  if $g = (v^r \odot k)$ then
3      $C' \leftarrow C \cup \{(v \odot k)\}$
4  else if $g = (v_1^r \odot v_2^r)$ then
5      $C' \leftarrow C \cup \{(v_1 \odot v_2)\}$
6  else if $g = (v^w \odot k)$ or $g = (v^w \odot v_2^r)$ then
7      if $g = (v^w \odot k)$ then
8          $C' \leftarrow C \cup \{(\bar{v} \odot k)\}$
9      else if $g = (v^w \odot v_2^r)$ then
10         $C' \leftarrow C \cup \{(\bar{v} \odot v_2)\}$
11     $C' \leftarrow$ saturate($C'$)
12     foreach $c = (v \odot x)$ or $c = (x \odot v)$ in $C'$ do
13         $C' \leftarrow C' \setminus \{c\}$
14     foreach $(\bar{v} \odot x)$ in $C'$ do
15         $C' \leftarrow C' \setminus \{(\bar{v} \odot x)\}$
16         if $x \neq v$ then $C' \leftarrow C' \cup \{(v \odot x)\}$
17     return $C'$
18 return saturate($C'$)
```

distinguishable, "primed" copy $\bar{v}$ of variable v and then remove from C all constraints that mention v (here x is used to denote either a variable or a constant). This is needed to properly reason on guards of the form $(v^w \odot v^r)$. After saturating, we remove all the previous constraints on v and replace all occurrences of $\bar{v}$ with v, unless x is v (as this would generate a constraint $(v \odot v)$ – see line 16).

Let us denote by $Const_{\mathcal{B}}$ the set of constants appearing in the guard of at least one action in $\mathcal{B}$: $Const_{\mathcal{B}} = \{k \in \mathbb{R} \mid \exists a \in \mathcal{A}, v \in V.\ guard(a) \text{ is either } (v^w \odot k) \text{ or } (v^r \odot k)\}$. Also, let $Const^{+}_{\mathcal{B}} \doteq Const_{\mathcal{B}} \cup \{+\infty, -\infty\}$, which we assume ordered. For $\mathcal{B}$ as in Figure 1, $Const^{+}_{\mathcal{B}} = \{-\infty, \mathtt{0}, +\infty\}$.

We now define the set of intervals partitioning the domain of variables, which we will use for our interval abstraction.

Definition 11. *The* representative intervals *for a DDS $\mathcal{B}$ and a variable v is the set of all constraint sets that capture a partitioning of all possible values of v. It is denoted by $Intervals^v_{\mathcal{B}}$, defined as the union of $\{(v = k)\}$ for each $k \in Const_{\mathcal{B}}$ and $\{(v > k_1), (v < k_2)\}$ for any two successive $k_1, k_2 \in Const^{+}_{\mathcal{B}}$.*

For the simple DDS in Figure 1, the set $Intervals^v_{\mathcal{B}}$, for each v, is $\{\{(v < \mathtt{0})\}, \{(v = \mathtt{0})\}, \{(v > \mathtt{0})\}\}$. Note that this set is the same for each $v \in V$, but one could optimise by analysing the structure of $\mathcal{B}$ (see end of section).

Definition 12. *The set of* two-variable constraint sets *of a DDS $\mathcal{B}$ is the set $C^2_{\mathcal{B}}$ of all satisfiable constraint sets C in which, for each action a so that $guard(a)$ is of the form $(v_1^r \odot v_2^r)$, either $(v_1 \odot v_2)$ or $\neg(v_1 \odot v_2)$ is in C.*

Intuitively, this set of constraint sets simply represents all possible (consistent) ways in which all guards of the form $(v_1^r \odot v_2^r)$, associated to actions in $\mathcal{B}$, can be guessed to be either true of false. For the simple DDS in Figure 1, $C^2_{\mathcal{B}} = \{\{(\mathtt{val} \geq \mathtt{num})\}, \{(\mathtt{val} < \mathtt{num})\}\}$. We will use these constraint sets to explicitly reason by cases, by guessing all possible combinations of values of these guards.

Finally, we finitely represent the reachability graph $RG_{\mathcal{B}}$ of DDS $\mathcal{B}$ as a so-called *constraint graph* where constraints are used in place of constraint variable assignments.

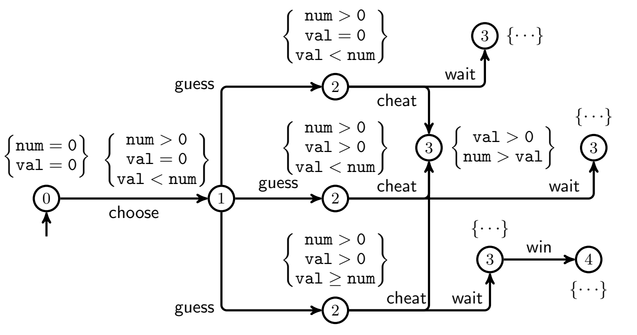

Figure 2: $CG_{\mathcal{B}}$ for $\mathcal{B}$ as in Figure 1 (constraint sets not minimised). Dots denote unchanged sets. The action repeat is omitted.

Definition 13. *Given $\mathcal{B} = \langle B, b_0, \mathcal{A}, T, F, V, \alpha_0, guard \rangle$, the* constraint graph *$CG_{\mathcal{B}}$ of $\mathcal{B}$ is a tuple $\langle S, s_0, \gamma_{\mathcal{B}} \rangle$ where:*

- *$S \subseteq B \times 2^{\mathcal{C}_V}$ is a set of states, called* nodes *to disambiguate the terminology w.r.t configurations in $RG_{\mathcal{B}}$. Each (b, C) holds the current system state b and a constraint set C capturing the current conditions on data;*
- *$s_0 = (b_0, C_0) \in S$ is the initial node, with $C_0 = C_{\alpha_0}$;*
- *$\gamma_{\mathcal{B}} \subseteq S \times \mathcal{A} \times S$ is the set of arcs, defined with S by mutual induction as follows. A transition $((b, C), a, (b', C'))$ is in $\gamma_{\mathcal{B}}$ iff $b \xrightarrow{a} b'$ and one of the following cases applies:*
  - *(i) $write(a) = \{v\}$;*
    (ii) there exists $C'' \in Intervals^v_{\mathcal{B}}$ and $C''' \in C^2_{\mathcal{B}}$ s.t. $C' = (C \oplus guard(a)) \cup C'' \cup C'''$ is satisfiable.
  - *(i) $write(a) = \emptyset$;*
    (ii) $C' = C \oplus guard(a)$ is satisfiable.

Intuitively, in the first case an edge exists in $CG_{\mathcal{B}}$ for each possible way in which a representative interval and a two-variable constraint set can be selected (guessed), so that the new set is satisfiable. Analogously to $RG_{\mathcal{B}}$, a run of $CG_{\mathcal{B}}$ is a sequence $\rho = (s_0, C_0) \xrightarrow{a_1} \cdots \xrightarrow{a_n} (s_n, C_n)$. The graph above is finite-state: the second component of the state space is always bounded by construction, as a DDS has finitely many guards, inducing finitely many constraints.

An example is shown in Figure 2. At each step, it considers explicitly all possible intervals for the values of written variables, as well as the truth value of two-variable constraints. The latter are guessed when variables are written (e.g., three possible outcomes exist for action guess), although one could easily optimise the computation to avoid considering guesses on variables that cannot affect the execution. This would require defining a dependency graph between variables, and we regard it as an optimisation.

6.1 Verification of Constraint Graphs

We now define a new semantics for the specification language $\mathcal{L}_{\mathcal{B}}$ in Section 4, using $CG_{\mathcal{B}}$ as interpretation structure. Differently from the previous case, a run here traverses nodes, each of the form (b, C). Accordingly, the semantics of a formula differs because each such constraint set C accounts for a set of possible solutions (i.e., a set of possible

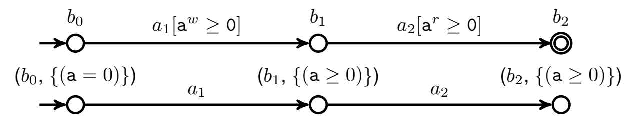
b_0
$a_1[\mathtt{a}^w \geq 0]$
b_1
$a_2[\mathtt{a}^r \geq 0]$
b_2
$(b_0, \{(\mathtt{a} = 0)\})$
a_1
$(b_1, \{(\mathtt{a} \geq 0)\})$
a_2
$(b_2, \{(\mathtt{a} \geq 0)\})$

Figure 3: A simple DDS $\mathcal{B}$ (above) and its $CG_{\mathcal{B}}$ (below).

constraint variable assignments α). As we will show, there is a relationship between the formulae that $RG_{\mathcal{B}}$ and $CG_{\mathcal{B}}$ satisfy, giving us a practical way of verifying formulae on $RG_{\mathcal{B}}$ by means of $CG_{\mathcal{B}}$, which is finite-state.

In the semantics below we also resort to a constraint set (denoted by A) to hold all the "assumptions", initially empty, that are "collected" along a run ρ. We provide an explanation after the definition. Finally, we use $\langle\text{-}\rangle\psi$ as a shorthand for $\bigvee_{a \in \mathcal{A}} \langle a \rangle \psi$.

Definition 14. *Given a run ρ in $CG_{\mathcal{B}}$ and a formula $\psi \in \mathcal{L}_{\mathcal{B}}$, we say that ρ satisfies ψ, written $\rho \models \psi$, iff $\rho, 0 \models_{\emptyset} \psi$, where $i \in [0, last(\rho)]$ and A is a constraint set, with:*
$\rho, i \models_A \mathtt{true}$ *iff A is satisfiable*
$\rho, i \models_A C$ *iff $\rho[i] = (b, C')$ and $C \cup A \cup C'$ is satisfiable*
$\rho, i \models_A b$ *iff $\rho[i] = (b, C)$ and $C \cup A$ is satisfiable*
$\rho, i \models_A \neg\psi$ *iff $\rho, i \not\models_A \psi$*
$\rho, i \models_A C \wedge \psi$ *iff $\rho, i \models_A C$ and $\rho, i \models_{A \cup C} \psi$*
$\rho, i \models_A \psi_1 \wedge \psi_2$ *iff $\rho, i \models_A \psi_1$ and $\rho, i \models_A \psi_2$*
$\rho, i \models_A \psi_1 \vee \psi_2$ *iff $\rho, i \models_A \psi_1$ or $\rho, i \models_A \psi_2$*
$\rho, i \models_A \langle a \rangle \psi$ *iff $\rho[i] \xrightarrow{a} \rho[i{+}1]$ and $\rho, i{+}1 \models_{A \oplus guard(a)} \psi$*
$\rho, i \models_A \Diamond\psi$ *iff $\rho, i \models_A \psi$ or ($i{<}last(\rho)$ and $\rho, i \models_A \langle\text{-}\rangle\Diamond\psi$)*
$\rho, i \models_A \Box\psi$ *iff $\rho, i \models_A \psi$ and ($i{=}last(\rho)$ or $\rho, i \models_A \langle\text{-}\rangle\Box\psi$)*

As explained before the definition, the set A is the constraint set encoding the "assumptions", initially empty, that are "collected" along the run ρ, as formalised above. Intuitively, whenever a constraint set C is in conjunction with a subformula (cf. the case for $\rho, i \models_A C \wedge \psi$), the constraints are first checked to be consistent with the current constraint set C' in the node of $CG_{\mathcal{B}}$ (together with the assumptions A collected so far); then ψ is evaluated by adding C to A.

For an example clarifying this point, consider the very simple DDS $\mathcal{B}$ as in Figure 3, shown with its corresponding constraint graph $CG_{\mathcal{B}}$. Assume $V = \{\mathtt{a}\}$ and $\alpha_0(\mathtt{a}) = 0$. Consider now a formula $\psi = \langle a_1 \rangle((\mathtt{a} = 2) \wedge \langle a_2 \rangle(\mathtt{a} = 3))$. It is intuitively evident that this formula must be false: action a_2 does not update the value of $\mathtt{a}$ after this is written by a_1, so $\mathtt{a}$ cannot be equal to 2 and then, next, be equal to 3. A naive approach, which would simply check the two constraints $(\mathtt{a} = 2)$ and $(\mathtt{a} = 3)$, independently from each other, against the constraint sets in the second and third node of $CG_{\mathcal{B}}$ would erroneously deduce that the formula is true. Indeed $\{(\mathtt{a} = 2), (\mathtt{a} \geq 0)\}$ and $\{(\mathtt{a} = 3), (\mathtt{a} \geq 0)\}$ are both satisfiable. Instead, by applying the semantics, in order to check that $\rho, 0 \models_{\emptyset} \psi$ we first add $guard(a_1)$ to the (empty) set of assumptions, then verify that $\rho, 1 \models_{\{(\mathtt{a} \geq 0)\}} \{(\mathtt{a} = 2)\}$, which is true. Then, we add $(\mathtt{a} = 2)$ to the assumptions, i.e. compute $A' = \{(\mathtt{a} \geq 0)\} \cup \{(\mathtt{a} = 2)\}$, and check whether $\rho, 2 \models_{A'} \{(\mathtt{a} = 3)\}$, which is false.

The justification for the case $\rho, i \models_A \langle a \rangle \psi$ is analogous: the set A cannot just be incrementally updated, because assumptions must be dropped when they become *obsolete* after a variable update. We enforce this by adding (via $\oplus$) the guard of the actions that are executed at each step along the run. As one can verify on Algorithm 1, this removes any current information about the variables that are updated. This is also the reason why the semantics of the operators $\Diamond$ and $\Box$ is given using this inductive, next-step case.

If $\rho \models \psi$ in $CG_{\mathcal{B}}$ then there exists at least one run of $RG_{\mathcal{B}}$ that satisfies ψ and which is, intuitively, *abstracted* by ρ (informally, it has the same action symbols and traverses configurations with a constraint variable assignment that is, step by step, a solution of the constraint set in the nodes of $CG_{\mathcal{B}}$ – see the proof of Thm. 1).

Analogously to $RG_{\mathcal{B}}$, a run ρ of $CG_{\mathcal{B}}$ is *terminal* iff it ends in a node (b, C) so that $b \in F$. As final system states in $\mathcal{B}$ have no outgoing edges, terminal runs cannot be extended. We redefine accordingly the notion of *witness* for a formula ψ on $CG_{\mathcal{B}}$: it is a terminal run of $CG_{\mathcal{B}}$ satisfying ψ.

Theorem 1. *There exists a witness for ψ in $RG_{\mathcal{B}}$ iff there exists a witness for ψ in $CG_{\mathcal{B}}$.*

Proof. (sketch) We can show that, when only considering the actions available at each step, $RG_{\mathcal{B}}$ and $CG_{\mathcal{B}}$ are bisimilar. Moreover, an ad-hoc variant of bisimulation exists so that (b, α) and (b', C) are bisimilar (being action-deterministic, this implies $b = b'$) iff, in addition, α is a solution of C. Hence each node of $CG_{\mathcal{B}}$ corresponds to a set of configurations in $RG_{\mathcal{B}}$ and, similarly, a run in $CG_{\mathcal{B}}$ corresponds to a set of runs in $RG_{\mathcal{B}}$.

The theorem then is a direct consequence of such result. The complete proof is omitted here due to lack of space, but we include the definition of the variant of bisimulation. This requires $R \subseteq S \times W$ to be such that: $\langle (b, C), (b', \alpha) \rangle \in R$ implies that (1) $b = b'$ and α is a solution of C; (2) for any $(b', \alpha) \xrightarrow{a, \beta} (b'', \alpha')$ there exists in $CG_{\mathcal{B}}$ an edge $(b, C) \xrightarrow{a} (b'', C')$ and $\langle (b'', C'), (b'', \alpha') \rangle \in R$; (3) for any edge $(b, C) \xrightarrow{a} (b', C')$ in $CG_{\mathcal{B}}$ there exists $(b', \alpha) \xrightarrow{a, \beta} (b'', \alpha')$ in $RG_{\mathcal{B}}$ and $((b', C'), (b'', \alpha')) \in R$. One can easily show by induction from the initial configuration and node that one such R always exists, and thus that $RG_{\mathcal{B}}$ and $CG_{\mathcal{B}}$ are bisimilar according to this definition. As a consequence, (i) these structures have the same branching structure when one only considers action symbols; and (ii) it is always possible to find a witness $\rho' = (s_0, C_0) \xrightarrow{a'_1} \cdots \xrightarrow{a'_n} (s_n, C_n)$ of ψ for $CG_{\mathcal{B}}$ which is so that one such relation R holds step-by-step w.r.t. the corresponding witnesses $\rho = (b_0, \alpha_0) \xrightarrow{a_1} \cdots \xrightarrow{a_n} (b_n, \alpha_n)$ on $RG_{\mathcal{B}}$. Formally, R as above exists s.t. $\langle \rho[i], \rho'[i] \rangle \in R$ and $a_i = a'_i$ for each $0 \leq i \leq |\rho|$. For each witness on $CG_{\mathcal{B}}$ multiple witnesses may exist in $RG_{\mathcal{B}}$. $\Box$

7 A Technique for Computing Strategies

We first describe all the required steps, then we detail and exemplify each of them. The procedure is based on a classical synthesis approach using deterministic and nondeterministic finite-state automata (DFA and NFA), which in our case have a more complex, "data-aware" alphabet:

1. Transform the formula ψ into an equivalent DFA $\mathcal{D}_{\psi}$;
2. Transform $CG_{\mathcal{B}}$ into a DFA $\mathcal{D}_{\mathcal{B}}$;
3. Compute a (special) product, denoted $\mathcal{D}_{\mathcal{B}} \bowtie \mathcal{D}_{\psi}$;

4. Define two *DFA games*, introduced later, on $\mathcal{D}_{\mathcal{B}} \bowtie \mathcal{D}_{\psi}$, so that there exists a witness for ψ ($T1$) iff the first game can be won and there exists a strategy $strat$ for ag with $RG_{\mathcal{B}}^{strat} \models \psi$ iff the second game can be won ($T2$).

1. DFA for a formula. Any formula $\psi \in \mathcal{L}_{\mathcal{B}}$ can be transformed into an equivalent LTL$_f$ (De Giacomo and Vardi 2013) formula over the alphabet Σ, defined as the union of the system state symbols B that appear in ψ (and their negation), of the action symbols in $\mathcal{A}$ that appear in ψ (and their negation) and the set of constraints that encode intervals for each variable of the DDS, considering the constants of ψ, i.e. the set: $\bigcup_{v \in V} Intervals_{\psi}^{v}$, where $Intervals_{\psi}^{v}$ is computed similarly to Def. 11 but with respect to the set of constants $\{k \mid k$ appears in a constraint of $\psi\}$. Σ is clearly finite.

To obtain the automaton, we use the following algorithm. First, the formula is taken as a formula in LTL$_f$ where all symbols in Σ are treated as propositional symbols. To handle our next operator, we rewrite any subformula of the form $\langle a\rangle\psi'$ as $\bigcirc(a \wedge \psi')$, where $\bigcirc$ is the (strong) next operator. Similarly, $\langle\text{-}\rangle\psi'$ becomes $\bigcirc\psi'$. Second, we compute a NFA as for LTL$_f$, with size at most exponential in the size of the formula (De Giacomo and Vardi 2013). Third, the NFA is determinized, which is again an exponential step (Rabin and Scott 1959). Importantly, we require this automaton to be *complete*: from each configuration, a transition is defined for every possible propositional interpretation of the alphabet, i.e., every symbol appears in its positive or negated form (note that negated constraints are still constraints).

Finally, we operate two transformations: we remove spurious edges labelled with more than one non-negated action or system state symbol to reduce the size. Then, we replace each edge in the DFA that is labelled with constraints by a set of edges as follows. These edges are obtained by substituting to the set of constraints C' of the form $(v \odot k)$, for the same v (where k is a constant and thus is used to compute $Intervals_{\psi}^{v}$), a set $C \in Intervals_{\psi}^{v}$ so that $C \cup C'$ is satisfiable, in any possible way. We can optimise again by removing edges labelled with unsatisfiable sets.

2. DFA for the constraint graph. Similarly, we transform $CG_{\mathcal{B}} = \langle S, s_0, \gamma_{\mathcal{B}}\rangle$, with $s_0 = (s_0, C_0)$, into a DFA $\mathcal{D}_{\mathcal{B}} = \langle 2^{\Sigma}, S_{\mathcal{B}}, s_{\mathcal{B}}^0, \delta_{\mathcal{B}}, F_{\mathcal{B}}\rangle$. The objective of this construction is for this DFA to have the same alphabet of $\mathcal{D}_{\psi}$, so that a special, "data-preserving" product can be computed. To achieve this, we take each node (b, C) of $CG_{\mathcal{B}}$ as the label of each edge in the DFA that is reaching that node (we add an initial dummy state from which an edge towards s_0 exists). $\mathcal{D}_{\mathcal{B}}$ is computed as follows: Σ is as above, with the addition of a special symbol $init$ to $\mathcal{A}$ (assuming $write(init) = \emptyset$); the set of states $S_{\mathcal{B}}$ of the DFA is the set $S \cup \{s_{\mathcal{B}}^0\}$ of nodes of the constraint graph plus the dummy initial state $s_{\mathcal{B}}^0$; the transition function is $\delta_{\mathcal{B}} : S_{\mathcal{B}} \times 2^{\Sigma} \mapsto S_{\mathcal{B}}$, so that:

- $(s_0, C_0) = \delta(s_{\mathcal{B}}^0, \{init, b_0\} \cup C_0)$;
- for any other state: $(b', C') = \delta((b, C), \varsigma_{\mathcal{B}})$ iff:
  - $(b, C) \xrightarrow{a} (b', C')$ is in $CG_{\mathcal{B}}$;
  - $\varsigma_{\mathcal{B}} = \{a, b'\} \cup C'$
- final states $F_{\mathcal{B}}$ are the nodes (b, C) with $b \in F$ in $\mathcal{B}$.

This is a DFA, since a successor node is encoded in full in the transition label $\varsigma_{\mathcal{B}}$: $\mathcal{D}_{\mathcal{B}}$ reads the action and the resulting system state and constraint set.

3. Product. Finally, we compute a special product $\mathcal{D}_{\mathcal{B}} \bowtie \mathcal{D}_{\psi}$, where each state encodes the state of both automata, with the addition of a third component that holds a set of assumptions (about constraint variable assignments). The intuition is that we need to take as assumptions the constraints labelling edges of $\mathcal{D}_{\psi}$, which must be assumed to hold in following steps (see the semantics in Def. 14, which is also based on assumptions). Technically, the product $\mathcal{D}_{\mathcal{B}} \bowtie \mathcal{D}_{\psi}$ is a DFA $\langle 2^{\mathbf{\Sigma}}, \mathbf{S}, \mathbf{s}^0, \boldsymbol{\delta}, \mathbf{F}\rangle$ obtained as follows:

- $\mathbf{\Sigma}$ is the alphabet of symbols, constructively defined below. It includes the set $\mathcal{A}$ of DDS actions and a finite set of symbols of the form $pick(\cdot)$, defined by mutual induction with the transition function $\boldsymbol{\delta}$;
- $\mathbf{S} = S_{\psi} \times S_{\mathcal{B}} \times \mathcal{C}_V$, i.e. a state $\langle s_{\psi}, s_{\mathcal{B}}, A\rangle$ encodes the current states of $\mathcal{D}_{\psi}$ and $\mathcal{D}_{\mathcal{B}}$, and assumptions A – recall that $\mathcal{C}_V$ is the set of possible constraint sets;
- $\mathbf{s}^0 = \langle s_{\psi}^0, s_{\mathcal{B}}^0, \emptyset\rangle$ is the initial state;
- $\boldsymbol{\delta} : \mathbf{S} \times 2^{\mathbf{\Sigma}} \mapsto \mathbf{S}$ is defined such that an arc $\langle s'_{\psi}, s'_{\mathcal{B}}, A'\rangle = \boldsymbol{\delta}(\langle s_{\psi}, s_{\mathcal{B}}, A\rangle, \varsigma)$ exists iff the following holds:
  - ς_{ψ} and $\varsigma_{\mathcal{B}}$ exist s.t. $s'_{\psi} = \delta_{\psi}(s_{\psi}, \varsigma_{\psi})$, $s'_{\mathcal{B}} = \delta_{\mathcal{B}}(s_{\mathcal{B}}, \varsigma_{\mathcal{B}})$;
  - $constr(\varsigma_{\psi} \cup \varsigma_{\mathcal{B}})$, denoting the set of constraints only (excluding other symbols in $\mathbf{\Sigma}$), is satisfiable;
  - if $(\varsigma_{\psi} \cap \mathcal{A}) \neq \emptyset$ then $(\varsigma_{\psi} \cap \mathcal{A}) = (\varsigma_{\mathcal{B}} \cap \mathcal{A}) = \{a\}$, i.e., a is the only action symbol labelling the edges of $\mathcal{D}_{\psi}$ and of $\mathcal{D}_{\mathcal{B}}$. Moreover, if $a \in \varsigma_{\mathcal{B}}$ then $\neg a \notin \varsigma_{\psi}$;
  - if $(\varsigma_{\mathcal{B}} \cap \mathcal{A}) = \{a\}$ then $A \cup C \cup \{guard(a)\}$ is satisfiable, with $(b, C) = s_{\mathcal{B}}$. This requires that the assumptions A do not contradict $guard(a)$, since $guard(a) \cup C$ is satisfiable by construction and $A \cup C$ is satisfiable (this is true for $\mathbf{s}^0$ and for any reachable state: see next);
  - the new set of assumptions is:
  $$A' = \begin{cases} (A \oplus guard(a)) \cup constr(\varsigma_{\psi}) & \text{if } write(a) \neq \emptyset \\ A \cup constr(\varsigma_{\psi}) & \text{otherwise} \end{cases}$$
  with $\{a\} = \varsigma \cap \mathcal{A}$. The set A is first updated to remove assumptions that are made obsolete by the action, then updated with constraints labeling the edge of $\mathcal{D}_{\psi}$;
  - $A' \cup C'$ is satisfiable, for $(b', C') = s'_{\mathcal{B}}$, i.e., the new set of assumptions A' is compatible with the new constraint set in $\mathcal{D}_{\mathcal{B}}$;
  - Finally, the set of symbols ς labelling the transition of the product is as follows: $\varsigma = \{a\}$ if $write(a) = \emptyset$, otherwise $\varsigma = \{a, pick(v, C' \cup A')\}$ with $\{v\} = write(a)$. In other words, whenever a variable v is written by a, we label the transition not only by a but also by a symbol $pick(v, C' \cup A')$ which makes explicit the new set of constraints that are in the next state of this product.
- $\mathbf{F} = \mathbf{S} \cap (F_{\psi} \times F_{\mathcal{B}} \times \mathcal{C}_V)$: if the components corresponding to states of $\mathcal{D}_{\psi}$ and $\mathcal{D}_{\mathcal{B}}$ are both final then the state is final.

4. DFA game. Next, we adapt the general form of synthesis to our setting, resorting to a DFA game between two players, called the environment and the controller, where the latter represents the actor ag. In this kind of games, we assume a set $\mathcal{X}$ of uncontrollable propositions (i.e., which are under the control of the environment player), and a set $\mathcal{Y}$ of controllable propositions that are under the control of the

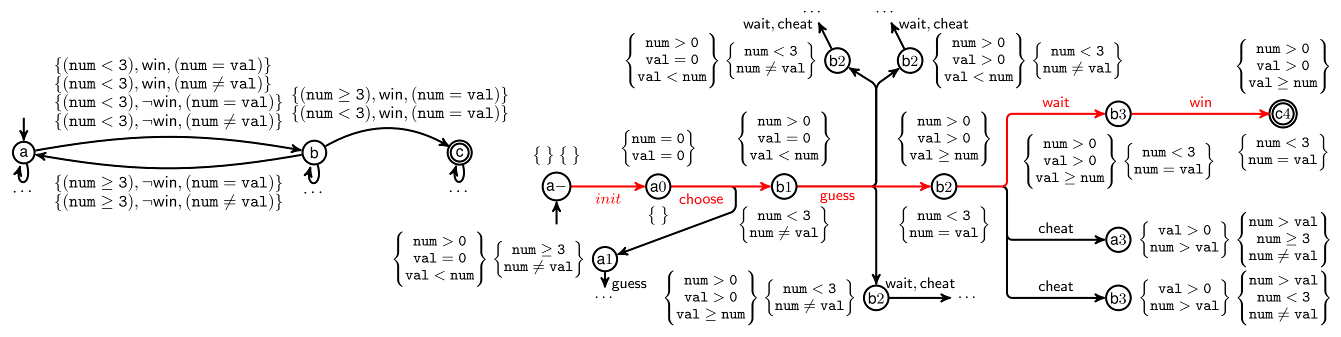
{(num < 3), win, (num = val)}
{(num < 3), win, (num ≠ val)}
{(num < 3), ¬win, (num = val)}
{(num < 3), ¬win, (num ≠ val)}
{(num ≥ 3), win, (num = val)}
{(num < 3), win, (num = val)}
{(num ≥ 3), ¬win, (num = val)}
{(num ≥ 3), ¬win, (num ≠ val)}
a
b
c
init
choose
guess
wait
win
cheat
wait, cheat
a0
a1
b1
b2
b3
c4

Figure 4: Left: $\mathcal{D}_\psi$ for $\psi = \Diamond((\texttt{num} < 3) \land \langle \mathsf{win} \rangle (\texttt{val} = \texttt{num}))$, requiring the chosen real to be smaller than 3 and next (by executing win) the guess to be exact. Dots are used for labels not already labelling other outgoing edges. Right: a fragment of the game (with only action symbols labelling arcs) showing a winning run. States are associated to two constraint sets, not minimised for clarity, i.e., the constraint set C as in $\mathcal{D}_\mathcal{B}$ and the constraint set A. State labels refer to the states of $\mathcal{D}_\psi$ and $\mathcal{D}_\mathcal{B}$. Note how the action guess generates four possible outcomes, although only three are in $CG_\mathcal{B}$ (see Figure 2): two outcomes disambiguate between the cases in which either $\texttt{num} = \texttt{val}$ or $\texttt{num} \neq \texttt{val}$ is added to the set of assumptions A. A winning strategy exists if at least $\texttt{num}, \texttt{val} \in V_{ag}$ and wait, cheat $\in \mathcal{A}_{ag}$. Trivially, a sequence of controller game moves guaranteeing to satisfy ψ is $\{\}$, $\{\mathsf{pick}(\texttt{num}, \{\texttt{num} > 0, \texttt{num} < 3, \ldots\})\}$, $\{\mathsf{pick}(\texttt{val}, \{\texttt{num} = \texttt{val}, \ldots\})\}$, $\{\mathsf{wait}\}$, $\{\}$.

controller (with $\mathcal{X} \cap \mathcal{Y} = \emptyset$). The objective is to control, at each step, the values of variables in $\mathcal{Y}$ in such a way that for all possible values of those in $\mathcal{X}$ a certain formula is true.

Formally, the specification of the game is given by a DFA G of the form $G = (2^{\mathcal{X}\cup\mathcal{Y}}, Q, q_0, \gamma, Q_F)$, where: *(i)* Q are the states of the game and q_0 is the initial state; *(ii)* $\gamma : Q \times 2^{\mathcal{X}\cup\mathcal{Y}} \mapsto Q$ is the (partial) transition function so that, given the current state q and a choice X and Y of propositions in $\mathcal{X}$ and $\mathcal{Y}$, respectively, for the environment and the controller, then $\gamma(q, X \cup Y)$ is the resulting game state; *(iii)* Q_F are the final states, which are winning for the controller.

The game evolves as follows: from a current state q (initially, q_0) the controller chooses values Y for which $\gamma(q, X \cup Y)$ is defined for at least one propositional interpretation of variables in $\mathcal{X}$, then the environment choses X among these. A play is a word in $(2^{\mathcal{X}\cup\mathcal{Y}})^*$. Plays of the game are of the form $\pi = (X_1, Y_1)(X_2, Y_2) \cdots (X_n, Y_n)$, where (X_i, Y_i) stand for the propositional interpretation at the i-th position in π. Given π, the corresponding game run ρ_π is the sequence of game states $q_0 \xrightarrow{X_1 \cup Y_1} \cdots \xrightarrow{X_n \cup Y_n} q_n$ so that $\gamma(q_i, X_i \cup Y_i) = q_{i+1}$, for $i \in [0, n-1]$.

Definition 15. *A* strategy *for the controller is a function $f : (2^{\mathcal{X}})^* \mapsto 2^{\mathcal{Y}}$ that, given $X_1, \ldots, X_n$ with $X_i \subseteq \mathcal{X}$ for $i \in [1, n]$, decides which propositions in $\mathcal{Y}$ to set to true next.*

A play π as above is induced by a strategy f iff $Y_1 = f(\bar{\epsilon})$ and $Y_i = f(X_1, \ldots, X_{i-1})$ for $i \in [1, n]$, where $\bar{\epsilon}$ denotes the empty sequence. A play π is *winning* if ρ_π ends in a state in Q_F. A *winning strategy* is a strategy f inducing only plays that are winning.

We now give a sound and complete technique to solve realizability for these games. The *controllable preimage* $Pre^c(Q')$ of a set Q' of states $Q' \subseteq Q$ is the set of states such that there exists a choice of values for symbols in $\mathcal{Y}$ such that for all choices of values for those in $\mathcal{X}$ (for which a successor exists), the game progresses to states in Q'. This notion can be seen as the counterpart of the notion of 'enforcing' in Def. 9. Formally, $Pre^c(Y, Q') = \{q \in Q \mid \forall X \in 2^{\mathcal{X}}.\ \text{if}\ q' = \gamma(q, X \cup Y)\ \text{then}\ q' \in Q'\}$ is the set of game states from where, if the controller chooses Y, then no matter how the environment chooses X the resulting game state will be in Q'. Then, let $Pre^c(Q')$ be the set $\{q \in Q \mid \exists Y, X.\ \gamma(q, X \cup Y) \land q \in Pre^c(Y, Q')\}$, i.e., the set of states from which the controller can force the game to reach a state in Q' (by choosing some Y).

We define the set $Win(G)$ of winning states of a DFA game G as the least fixpoint $Win(G) \doteq \mu Z.(Q_F \cup Pre^c(Z))$, which captures the reachability of final states in Q_F. This gives rise to the computation: $Win_0 = Q_F$, $Win_{i+1} = Win_i \cup Pre^c(Win_i)$, $Win(G) = \bigcup_{i \le |Q|} Win_i$. Intuitively, from the states in $Win(G)$ the controller player can force the game to eventually reach a state in Q_F, i.e., a terminal state (final for both $\mathcal{D}_\psi$ and $\mathcal{D}_\mathcal{B}$).

Theorem 2. *G admits a winning strategy iff $q_0 \in Win(G)$.*

We now define DFA games for capturing *T1*-*T2*. In fact, we can elegantly capture a variety of scenarios, assigning symbols (for actions and variables) to either player.

Given $\mathcal{D}_\mathcal{B} \bowtie \mathcal{D}_\psi$ represented as $\langle 2^{\mathbf{\Sigma}}, \mathbf{S}, \mathbf{s}^0, \boldsymbol{\delta}, \mathbf{F} \rangle$, we define the DFA game $G_\psi = (2^{\mathcal{X}\cup\mathcal{Y}}, \mathbf{S}, \mathbf{s}_0, \boldsymbol{\delta}, \mathbf{F})$, with $\mathcal{Y} = \mathbf{\Sigma}$ and $\mathcal{X} = \emptyset$. Intuitively, the controller has full control on the execution of $\mathcal{B}$, selecting both the actions and the conditions on the written variables.

Theorem 3. *Given $\langle \mathcal{B}, \psi \rangle$, there exists a witness for ψ in $RG_\mathcal{B}$ iff there exists a winning strategy for game G_ψ.*

Proof. (sketch) If there is no winning strategy then ag cannot enforce winning paths in the game (as they are maximal: final states in $\mathcal{D}_\mathcal{B}$ have no outgoing edges). By construction of the product automaton, game paths correspond to runs in $CG_\mathcal{B}$ and in turn these abstract runs in $RG_\mathcal{B}$ (see the proof sketch of Thm. 1). Then no witness for ψ can exist. In fact, it can be shown by induction on the structure of the formula ψ that there exists a run ρ of $CG_\mathcal{B}$ that corresponds to a game run of G_ψ ending in an accepting state iff $\rho \models \psi$. This is obvious if one considers that the game is obtained as the product of the two automata for $CG_\mathcal{B}$ and ϕ. Here, 'corresponds' means that the game run is simply projected on the constraint graph. The other direction follows by the fact that runs of $CG_{\mathcal{B}}$ abstract runs of $RG_{\mathcal{B}}$ (again, by Thm 1). $\square$

Finally, given an adversarial problem $\langle \mathcal{B}, \psi, V_{ag}, A_{ag} \rangle$ and $\mathcal{D}_{\mathcal{B}} \bowtie \mathcal{D}_{\psi}$ as before, let G^{a}_{ψ} be the game equal to G_{ψ}, but with $\mathcal{Y} = A_{ag} \cup \{pick(v, \cdot) \mid v \in V_{ag}\}$ and $\mathcal{X} = \mathbf{\Sigma} \setminus \mathcal{Y}$. Namely, the controller is given control of the actions in A_{ag} and variables in V_{ag}, while the environment controls the rest.

Theorem 4. *Given $\langle \mathcal{B}, \psi, V_{ag}, A_{ag} \rangle$ as before, there exists $strat$ s.t. $RG^{strat}_{\mathcal{B}} \models \psi$ iff a winning strategy for G^{ag}_{ψ} exists.*

Proof. (sketch) This result follows from the fact that the controller player only controls symbols in $\mathbf{\Sigma}$ for actions in $\mathcal{A}_{ag}$ and symbols $pick(v, \cdot)$ for each variable $v \in V_{ag}$. Intuitively, if ψ mentions a two-variable constraint then this will appear "as is" in $\mathcal{D}_{\psi}$, whereas other constraints will induce a discretisation of the intervals of values. Then, we can show (i) that a winning strategy f for G^{ag}_{ψ} can be readily transformed into a system strategy $strat$; then (ii) that this $strat$ induces a set of runs of $RG_{\mathcal{B}}$, corresponding to the game paths selected by f, which satisfy the following: at each step (with history ending in a configuration w of $RG_{\mathcal{B}}$), the set of action firings returned by $strat$ is a set in $Ctrl^{w}_{ag}$ as defined in Def. 9. These steps are commented below. $\square$

How to compute a winning strategy when $q_0 \in Win(G)$, where G is either G_{ψ} or G^{ag}_{ψ}, and how to relate it to system strategies $strat$? The first step is to extract f from $Win(G)$, assuming $q_0 \in Win(G)$. Since $Win(G)$ simply captures a winning region characterised by a reachability property, we are guaranteed that, if $q_0 \in Win(G)$, then there exists a way for the actor ag (the controller player) to reach a state in Q_F irrespective of the choices made by the environment (since Pre^c is used to define $Win(G)$). In fact, $Win(G)$ encodes any possible way to do so. By applying f we need to progress by stratification, i.e., from a member of $Win_{i+1} \setminus Win_i$, for some $i \geq 0$, to a member of Win_i. Thus, we can chose the strategy f in any possible way such that, for a game play $\pi = (X_1, Y_1) \cdots (X_{n-1}, Y_{n-1})$, then $f(X_1, \ldots, X_{n-1}) = Y_n$ implies that for any X_n for which $\pi' = \pi \cdot (X_n, Y_n)$ is a game play, then if $last(\rho_{\pi}) \in (Win_{i+1} \setminus Win_i)$ then $last(\rho_{\pi'}) \in Win_i$. This is done by annotating states with the index j by which they are first included in a set Win_i, for $j = i$.

The second step is, given a winning strategy f for G, to compute the corresponding system strategy $strat$ for $RG_{\mathcal{B}}$. One way is to relate f to the runs of $RG_{\mathcal{B}}$ that correspond to runs in G induced by f. This however is impractical and requires lengthy definitions. More operatively, we directly apply f to the current game run ρ_{π} and compute $f(X_1, \ldots, X_{n-1}) = Y$, so that the game play $\pi \cdot (X, Y)$ results from the selection of X by the environment. For both tasks (*T1*-*T2*), depending on the membership in $\mathcal{A}_{ag}$ and V_{ag}, either the actor or the environment chooses the action a which is the only action symbol in $Y \cup X$ (this is unique by the construction of the product). Similarly, if $write(a) \neq \emptyset$ then either of them chooses a guard variable assignment β so that the resulting constraint variable assignment (as in Def. 4) is a solution of $C \cup A$, with $pick(write(a), C \cup A)$ in $Y \cup X$. If needed, one can then obtain the system strategy $strat$ by returning at each step all possible couples (a, β) that can be obtained in this way, although this is not necessary for the execution of f.

The procedure is correct, and it shows how to apply the same synthesis technique to *T1*-*T2*, as $\langle \mathcal{B}, \psi \rangle$ is equivalent to the adversarial problem $\langle \mathcal{B}, \psi, V, \mathcal{A} \rangle$. The computed strategy always induces terminal runs of $RG_{\mathcal{B}}$ that satisfy ψ.

Theorem 5. *Given a winning strategy f for G_{ψ} or G^{ag}_{ψ}, the strategy $strat$ computed as above is s.t. $RG^{strat}_{\mathcal{B}} \models \psi$.*

This follows from the fact that, being f based on Pre^c, no subset of game edges can be enforced by f unless such edges correspond, in $RG_{\mathcal{B}}$, to a set of action firings that can be enforced by ag. Indeed, in the definition of $\mathcal{D}_{\mathcal{B}} \bowtie \mathcal{D}_{\psi}$ each edge is either labelled by an action a, when no variable is written, or it is labelled by $\{a, pick(write(a), C \cup A)\}$ for some a, A, C. Four cases then exist for each such action a, depending on whether ag controls a and $write(a)$. These are the same cases as in Definition 9. Clearly, a single winning strategy f computed as above is so that, when executed on $RG_{\mathcal{B}}$, a single action $a \in \mathcal{A}_{ag}$ at the time will be selected from states controlled by ag.

Complexity. Regarding complexity, the hardness of the problem is given by the synthesis problem for LTL$_f$ on nondeterministic domains, which corresponds to the special case in which ψ has no variables, $V \neq \emptyset$ and $V_{ag} = \emptyset$. This is doubly exponential in the formula – see, e.g., (De Giacomo and Rubin 2018). Likewise, computing $\mathcal{D}_{\psi}$ in our approach has cost doubly exponential in the size of ψ and $\mathcal{D}_{\mathcal{B}}$ has size linear in $\mathcal{B}$ although at most exponential in $|V|$ (the number of possible constraint sets). $\mathcal{D}_{\mathcal{B}} \bowtie \mathcal{D}_{\psi}$ is obtained by a polynomial cross-product of these structures, although we require a number of calls to a constraint solver for determining whether a transition exists in the product, according to the definition in Section 7.

8 Conclusions

We have illustrated an automata-based technique for computing winning strategies for data-aware dynamic systems, for temporal specifications that also capture constraints on the data that these systems manipulate. We achieved this by combining interval-based data abstraction techniques with standard automata-based constructions for verification. The novelty is not related to the use of automata-based techniques for two-player adversarial games, but in the enrichment of these techniques with a 'data-aware' feature. This setting is relevant for several application domains, such as Business Process Management, that do not allow the data aspects to be simply abstracted away upfront.

The construction used in this paper *can be directly implemented.* In future work we will implement the approach, study possible optimisations, and evaluate it in practical application domains. As our model of the system representation is analogous to data-aware process models in the literature (cf., e.g., (de Leoni and Mannhardt 2019)), natural candidates are data-aware models for business processes, for which state-of-the-art "forward" execution and analysis tools exist, but not verification nor strategy synthesis tools.

Acknowledgments

This work has been partially supported by the unibz CRC project DACoMan and the unibz project VERBA.

References

Abdulla, P. A.; Aiswarya, C.; Atig, M. F.; and Montali, M. 2019. Reachability in database-driven systems with numerical attributes under recency bounding. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '19, 335–352. ACM.

Bagheri Hariri, B.; Calvanese, D.; De Giacomo, G.; Deutsch, A.; and Montali, M. 2013. Verification of relational data-centric dynamic systems with external services. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, 163–174. ACM.

Batoulis, K.; Haarmann, S.; and Weske, M. 2017. Various notions of soundness for decision-aware business processes. In *ER*, volume 10650 of *Lecture Notes in Computer Science*, 403–418. Springer.

Bozzelli, L., and Pinchinat, S. 2014. Verification of gap-order constraint abstractions of counter systems. *Theoretical Computer Science* 523:1–36.

Calvanese, D.; De Giacomo, G.; and Montali, M. 2013. Foundations of data-aware process analysis: A database theory perspective. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, 1–12. ACM.

De Giacomo, G., and Rubin, S. 2018. Automata-theoretic foundations of FOND planning for LTLf and LDLf goals. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI '18, 4729–4735. AAAI Press.

De Giacomo, G., and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, 854–860. AAAI Press.

de Leoni, M., and Mannhardt, F. 2019. Decision discovery in business processes. In Sakr, S., and Zomaya, A. Y., eds., *Encyclopedia of Big Data Technologies*. Springer.

de Leoni, M.; Felli, P.; and Montali, M. 2018. A holistic approach for soundness verification of decision-aware process models. In *ER*, volume 11157 of *Lecture Notes in Computer Science*, 219–235. Springer.

De Masellis, R.; Di Francescomarino, C.; Ghidini, C.; Montali, M.; and Tessaris, S. 2017. Add data into business process verification: Bridging the gap between theory and practice. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI '17, 1091–1099. AAAI Press.

De Masellis, R.; Maggi, F. M.; and Montali, M. 2014. Monitoring data-aware business constraints with finite state automata. In *Proceedings of the 2014 International Conference on Software and System Process*, ICSSP '14, 134–143. Association for Computing Machinery.

Demri, S., and D'Souza, D. 2007. An automata-theoretic approach to constraint LTL. *Information and Computation* 205(3):380–415.

Felli, P.; de Leoni, M.; and Montali, M. 2019. Soundness verification of decision-aware process models with variable-to-variable conditions. In *19th International Conference on Application of Concurrency to System Design, ACSD 2019, Aachen, Germany, June 23-28, 2019*, 82–91. IEEE.

Kalenkova, A.; Burattin, A.; de Leoni, M.; van der Aalst, W. M. P.; and Sperduti, A. 2019. Discovering high-level BPMN process models from event data. *Business Process Management Journal* 25(5):995–1019.

Maggi, F. M.; Montali, M.; Westergaard, M.; and van der Aalst, W. M. P. 2011. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *BPM*, volume 6896 of *Lecture Notes in Computer Science*, 132–147. Springer.

Mannhardt, F.; de Leoni, M.; Reijers, H. A.; and van der Aalst, W. M. P. 2016. Decision mining revisited - discovering overlapping rules. In *CAiSE*, volume 9694 of *Lecture Notes in Computer Science*, 377–392. Springer.

Mannhardt, F.; de Leoni, M.; Reijers, H. A.; and van der Aalst, W. M. P. 2017. Data-driven process discovery - revealing conditional infrequent behavior from event logs. In *CAiSE*, volume 10253 of *Lecture Notes in Computer Science*, 545–560. Springer.

Pnueli, A., and Rosner, R. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, 179–190. Association for Computing Machinery.

Rabin, M. O., and Scott, D. S. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development* 3(2):114–125.

Reichert, M. 2012. Process and data: Two sides of the same coin? In *On the Move to Meaningful Internet Systems: OTM 2012*, 2–19. Berlin, Heidelberg: Springer Berlin Heidelberg.

Reisig, W. 2013. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer.

Tikhonova, U. 2017. *Engineering the dynamic semantics of domain specific languages*. Ph.D. Dissertation, Department of Mathematics and Computer Science. Proefschrift.

van der Aalst, W. M. P. 2016. *Process Mining - Data Science in Action, Second Edition*. Springer.

van Gool, L.; Punter, T.; Hamilton, M.; and van Engelen, R. 2006. Compositional mda. In Nierstrasz, O.; Whittle, J.; Harel, D.; and Reggio, G., eds., *Model Driven Engineering Languages and Systems*, 126–139. Berlin, Heidelberg: Springer Berlin Heidelberg.

Vardi, M. Y. 1996. *An automata-theoretic approach to linear temporal logic*. Berlin, Heidelberg: Springer Berlin Heidelberg. 238–266.

Yousfi, A.; Batoulis, K.; and Weske, M. 2019. Achieving business process improvement via ubiquitous decision-aware business processes. *ACM Transactions on Internet Technology* 19(1):14:1–14:19.