Ontological Control

Marcus Bjäreland*
Dept of Comp and Info Science
Linköping University
S-581 83 Linköping, Sweden
marbj@ida.liu.se

George Fodor
ABB Industrial Systems
Department ISY/AMC
S-721 67 Västeras, Sweden
george.fodor@seisy.mail.abb.com

Abstract

Execution monitoring has been defined as an agent's process of observing the world for discrepancies between the actual world and its internal representation, and recovering from such discrepancies. Although there exists a body of research in this area, few researchers have taken interest in *classifications* of the discrepancies, especially the distinction between discrepancies due to unobservable uncontrollable actions, and discrepancies due to the fact that the representation does not reflect the world in an adequate way. In *Ontological Control* we are interested in object-level execution systems, i.e. control systems that are in immediate contact with the world via sensors and actuators.

In sufficiently complex control situations there are always a number of unmodelled assumptions, which necessarily must hold for the control system to be valid. In this paper we show how violations of such *ontological assumptions* can be correctly classified in a PLC setting, and we define a recovery procedure that can guide the controller to a control goal whenever this is possible.

Introduction

Execution monitoring has been defined as an agent's process of observing the world for discrepancies between the actual world and its internal representation, and recovering from such discrepancies (De Giacomo, Reiter, & Soutchanski 1998). Although there exists a body of research in this area, e.g. (Ambrose-Ingerson & Steel 1988; Fikes, Hart, & Nilsson 1972; Haigh & Veloso 1996; Williams & Nayak 1996), few researchers have taken interest in *classifications* of the discrepancies, especially the distinction between discrepancies due to unobservable uncontrollable actions, and discrepancies due to the fact that the representation does not reflect the world in an adequate way. In *Ontological Control*, introduced in (Driankov & Fodor 1993; Fodor 1998), we are interested in object-level execution systems, i.e. control systems that are in immediate contact with the world via sensors and actuators.

The knowledge a control system designer has about a particular system has to be an abstraction of the reality. Independently of the formal precision of such an abstraction, there are *always* certain constraints which specify under what conditions the application of this abstract knowledge to a particular system is valid. Ontological knowledge is the abstracted knowledge underlying the choice of controller program. The contraints for the validity of the program *assumed* and not *modelled* will be called *ontological assumptions.* In this paper we show how violations of ontological assumptions (VOA:s) can be correctly classified in a PLC (Programmable Logic Controller) setting, and we define a recovery procedure that can guide the controller to a control goal whenever this is possible.

Overview

In Figure 1 we can see a generic view of a controlled dynamic system, where a plant may be influenced either by observable and controllable actions invoked by a controller or by observable but uncontrollable actions, for example invoked by an operator or another controller. The output from the sensors in the plant is processed by a database that detects catastrophic faults that occur in the plant, e.g. by detecting if sensor values get out of pre-specified boundaries. The database also filters the sensor output, and sends it to the controller as plant output. The controller also recieves output from the actuators, and decides from the plant output and the actuator output what to do next. This paper covers the functionality of the *Ontological Controller* (OC), which is a device that classifies any deviation, and suggests a new control action to the controller. In Ontological Control we assume that the controller is programmed with a set of *optimal goal paths*

*This research has been supported by ABB ISY/AMC, the Wallenberg Laboratory for Information Technology and Autonomous Systems (WITAS), and the Swedish Research Council for Engineering Sciences (TFR).

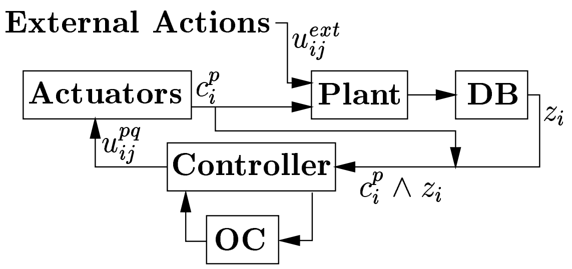

Figure 1: The control system

(plans) each with an initial state, a goal state, and a sequence of actions and states between them. By *state* we mean a set of sensor-value vectors described by a formula in propositional logic. The optimal goal paths are sequential control algorithms designed by an engineer or a planner, and are optimal for some reason, e.g. for low use of some resource, or simply that it appears to do the job better than other available alternatives. From the set of optimal goal paths we can construct the *control graph* by merging the sets of vertices and the sets of edges of the optimal goal paths. In this way we can view the execution of the controller as graph traversal.

As noted in (Pell *et al.* 1996) it is seldom the case in practice that execution begins in an initial state and goes on from there, until a goal state is reached. Typically, it is unknown before the automated execution is started where in the control graph the state of the plant can be found. In this paper we define an operation, the *goal seeking operation* (GSO), which does not only pin point the current state (which is called the *synchronization* operation), but also guides the execution to the optimal goal path leading to the best possible goal state.

If a deviation (a *de-synchronization*) from the optimal goal paths is detected (by noticing that the actual plant state does not correspond to the expected post-condition of the latest action), there are two cases we are interested in distinguishing between. First, when the cause of the fault is *external*; that something unexpected happened but that the control graph still is corroborated. We will call this cause *Unexpected External Action* (UEA). It is possible to view theses de-synchronizations as effects of unobservable and uncontrollable actions. Second, when the fault is due to *Violation of Ontological Assumptions* (VOA), that is, when some unmodelled aspect of the system influence the control of the system.

Preliminaries

A *plant* is a quadruple $P = \langle X \cup X', D, U, U^{ext} \rangle$, where $X \cup X' = \{x_1, \ldots, x_n\}$ is a set of *plant signal variables* (plant signals), where $X \cap X' = \emptyset$, $D = \{D_1, \ldots, D_n\}$ is a set of plant signal *domains* each being a set of real values (finite or infinite, countable or uncountable) such that the domain of x_i is D_i, U is a set of *control action symbols* (control actions), and U^{ext} is a set of *external action symbols* (external actions).

The plant signals denote sensor output, and, thus, the domains are determined by the accuracy and range of the sensors. The members of X' will be called *actuator signals*, and denote signals from the actuators of the plant. The members of U denote actions that can be invoked by a controller, while the members in U^{ext} denote actions that only can be invoked externally, e.g. by an operator. Below, we will assume a fixed plant P.

A *plant signal constraint* (plant constraint) is a relation between a variable, $x_i \in X$, and an element, $v \in D_i$. We will only be interested in the relations $<$, $\leq$, $=$, $\geq$, and $>$. Constraints for $x_i \in X'$ will be called *actuator constraints.*

A *syntactic plant state* (state) is a conjunction $y = c \wedge z$, where c is a conjunction of actuator constraints (*control configuration formula*), and z is a boolean combinations of plant constraints (*proper plant formula*). The set of all states for a plant P will be denoted Y_P. We define two functions, $conf$ and *plant* on Y_P, such that for $y = c \wedge z \in Y_P$, $conf(y) = c$ and $plant(y) = z$.

The distinction between c and z can be viewed as a causality, i.e. in the same sense that the actuators has effects in the physical plant, the actuator output can be seen as having effects on the plant output. We are always interested in controlling the plant output via the control configurations.

We use sub- and superscripts on control actions to mark that they are connecting certain states. For example, by u_{ij}^{pq} we denote the control action from $y_i = c_i^p \wedge z_i$ to $y_j = c_j^q \wedge z_j$. Similarily, for external actions u_{ik}^{ext} for the external action from $y_i = c_i^p \wedge z_i$ to $y_k = c_k^p \wedge z_k$. The sub- and superscripted actions may be syntactically distinct, but may still denote the same physical actions (see the running example below). We remove indices whenever ambiguity can be avoided.

A *goal path* is a non-empty graph $O_i = \langle Y_i, U_i \rangle$, where $Y_i = \{y_{i_1}, \ldots, y_{g_i}\} \subseteq Y$ and $U_i = \{u_{i_1 i_2}, u_{i_2 i_3}, \ldots, u_{i_{m-1} g_i}\} \subseteq U$, where all y:s are distinct. Thus, we use the indices to place particular control actions in the goal paths. The first state in a goal path is called the *initial state* and the last the *goal state.* An *external action transition graph*, is a directed graph, $\langle Y', U' \rangle$, where $Y' \subseteq Y$ and $U' \subseteq U^{ext}$. States connected by a control action will be called *consecutive*.

A *controller* for P is a tuple $C = \langle G, T^{ext} \rangle$, where $G = \{O_1, \ldots, O_m\}$ is a set of goal paths with distinct goal states, and T^{ext} is an external action transition graph. Below we will assume a fixed controller C.

For C we construct the *control graph*, $\langle S, V \rangle$, which is the component-wize union of the goal paths in G. If we include the external actions in a control graph, we call this an *extended control graph*. Thus, the dynamics of the control system can be described as graph traversal. Since the control graph can be divided into paths different from those in the G component, we call the goal paths in G *optimal goal paths* and all other paths *non-optimal goal paths*. We define the *reachability predicate*, R, taking two states as arguments, where $R(y_i, y_j)$ holds iff there is a (optimal or non-optimal) path from y_i to y_j in the control graph. The function $d(y_i, y_j)$ denotes the graph theoretic distance between y_i and y_j, if $R(y_i, y_j)$ holds, and ∞ otherwise.

Finally, we define a *control session* as a triple $\langle C, y, < \rangle$, where C is a controller, y is a goal state in some goal path of C (the *current goal state*), and $<$ is a total order on the goal states in C. This order is called a *priority order on goal states.*

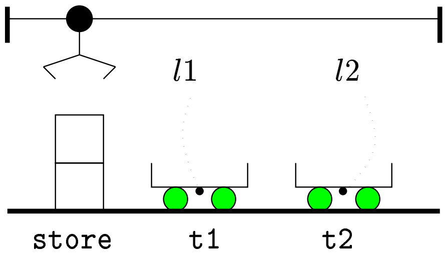

Figure 2: The plant.

Running Example

We will now look at an example of a plant depicted in Figure 2. We control a robot arm (our actuator) that is supposed to move boxes from `store` to the trolleys `t1` and `t2`. The actuator output is the value of a variable *pos* (for the position of the robot arm) with domain $D_{pos} = \{\mathtt{store}, \mathtt{t1}, \mathtt{t2}\}$, and the other plant signals give values to two variables, $l1$ and $l2$, with domains $D_{l1} = D_{l2} = \{0, 1, 2\}$, denoting the number of boxes on respective trolley. We have three control actions, MS for the action of moving the arm to the store, $M1$ for the action of gripping a block and moving it to position `t1`, and $M2$ for moving a block to position `t2`. It would be possible to include external actions such as actions that empty a trolley, but for readability we neglect such actions. If it is a physical fact that no more than two blocks can occur in the system simultaneously, the relevant states can be found in Table 1. For readability, the indices of the states denote the

State	*pos* =	$l1$ =	$l2$ =
y^s_{00}	`store`	0	0
y^s_{10}	`store`	1	0
y^s_{01}	`store`	0	1
y^1_{10}	`t1`	1	0
y^1_{20}	`t1`	2	0
y^2_{01}	`t2`	0	1
y^2_{02}	`t2`	0	2
y^2_{11}	`t2`	1	1
$y_{g_{20}}$	`store`	2	0
$y_{g_{11}}$	`store`	1	1
$y_{g_{02}}$	`store`	0	2

Table 1: The relevant states.

values of the variables, e.g.

$$y^2_{11} = \overbrace{pos = \mathtt{t2}}^{c^2} \wedge \overbrace{l1 = 1 \wedge l2 = 1}^{z_{11}}$$

The first state is the initial state of the system, and three last states are the goal states. The remaining states (for example y^2_{20}) represent "non-relevant" states. We have the following three optimal goal paths:

$$y^s_{00} \overset{M1}{\Longrightarrow} y^1_{10} \overset{MS}{\Longrightarrow} y^s_{10} \overset{M1}{\Longrightarrow} y^1_{20} \overset{MS}{\Longrightarrow} y_{g_{20}}$$
$$y^s_{00} \overset{M1}{\Longrightarrow} y^1_{10} \overset{MS}{\Longrightarrow} y^s_{10} \overset{M2}{\Longrightarrow} y^2_{11} \overset{MS}{\Longrightarrow} y_{g_{11}}$$
$$y^s_{00} \overset{M2}{\Longrightarrow} y^2_{01} \overset{MS}{\Longrightarrow} y^s_{01} \overset{M2}{\Longrightarrow} y^2_{02} \overset{MS}{\Longrightarrow} y_{g_{02}}$$

In Figure 3 we can see the resulting control graph (without the control actions).

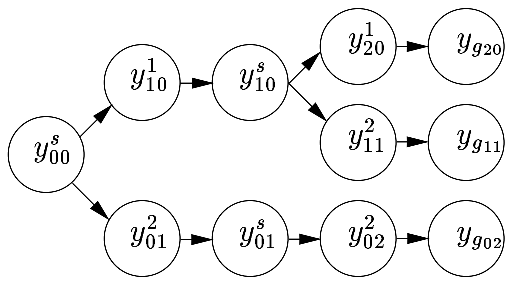

Figure 3: The control graph.

Interpretations

We assume that the physical plant signals are read at particular *sampling instances*, $t_k = k \cdot h$, where k is an integer and h is a certain sampling period. For the purpose of our presentation we will denote the sequence of sampling instances, $\langle t_k, t_{k+1}, \ldots, t_{k+m}\rangle$ by $\langle t, t+1, \ldots, t+m\rangle$.

A *sample*, σ_t, for a plant P is a function from plant signals to the corresponding domains. The sample is constructed from the vector of signals read from the physical plant for the variables at time point t. The set of all possible samples for a plant P is denoted Σ_P. An *interpretation*, ζ, is a function from samples and formulae to truth values, and it is defined as follows: For $x_i \in X$, $\odot \in \{<, \leq, =, \geq, >\}$, $v_i \in D_i$, and $y_k, y_l \in Y$, we have

$$\begin{aligned}
\zeta(\sigma_t, x_i \odot v_i) = 1 &\quad \text{iff} \quad \sigma_t(x_i) \odot v_i \text{ holds} \\
\zeta(\sigma_t, y_k \wedge y_l) = 1 &\quad \text{iff} \quad \zeta(\sigma_t, y_k) = 1 \text{ and } \zeta(\sigma_t, y_l) = 1 \\
\zeta(\sigma_t, y_k \vee y_l) = 1 &\quad \text{iff} \quad \zeta(\sigma_t, y_k) = 1 \text{ or } \zeta(\sigma_t, y_l) = 1 \\
\zeta(\sigma_t, \neg y_k) = 1 &\quad \text{iff} \quad \zeta(\sigma_t, y_k) = 0
\end{aligned}$$

With slight abuse of notation we let $\zeta(\sigma_t, a) = 1$, for a control action a, iff the controller invoked a at time point t. We say that a formula, y, is *consistent* iff there exists a $\sigma \in \Sigma$ such that $\zeta(\sigma, y) = 1$. If we , at time point t, recieve a sample σ_t from the plant such that for a state y $\zeta(\sigma_t, y) = 1$, we say that y *materializes* at t.

Definition (Well-Designed Controller) Given a plant P and a controller C for P, C is *well-designed* if the following conditions holds:

1. every state $y \in S$ is consistent, and
2. for every sample $\sigma \in \Sigma_P$ there exists exactly one $c \in \{conf(y) \mid y \in S\}$ such that $\zeta(\sigma, c) = 1$.

$\square$

As seen in the running example there may be more than than one state with the same configuration.

The last constraint above implies that if two states have different configurations, their respective configurations can never be satisfied by the same sample. It is easy to verify that the controller in the running example is well-designed.

De-synchronization

When a controller recieves a sample from the plant and actuators, and there is a possibility that more than one state may be satisfied by the sample, and that the sample does not fulfill our expectations, we say that the controller is *de-synchronized*. Typically, this occurs when the automation initially is turned on, or when external actions influence the plant. In the first case, we cannot trust the actuator output, since changes in the physical plant may have been done manually. For the second case, it is vital to distinguish between the case where the de-synchronization is caused by an external action, or by a discrepancy between how the plant actually reacts to actuator influences, and how the reaction is modelled in the controller. Thus, we have four causes of de-synchronization that we need to distinguish between:

- **Expected External Action** (EEA) – de-synchronization due to an action in U^{ext}.
- **Unexpected External Action** (UEA) – de-synchronization due to unmodelled changes in the physical plant.
- **Timing Errors** (TE) – de-synchronization due to too long sampling periods, that is sampling periods that allow for (expected or unexpected) external actions to be invoked in a new state before the new state has materialized in the controller.
- **Violation of Ontological Assumption** (VOA) – de-synchronization due discrepancies between the plant behaviour and the modelled behaviour.

The distinction between the four causes cannot be made without a specification of the dynamics of the execution system in use. By dynamics we mean the precise circumstances under which state transitions occur. For example, in certain execution systems (e.g. in PLC systems) we can safely assume that the sampling period can be set to be sufficiently short, which would enable us to neglect the possibility of TE.

We should note that in the absence of TE, VOA:s are the potentially dangerous cause of de-synchronization, since the system is probable to encounter the same VOA every time the optimal goal path, where it occurred a first time, is executed. A recovery procedure, such as the GSO defined below, can only remedy the fault behaviour temporarily; when a VOA occurs, a warning system should alert an operator.

The OC will have a state-transition graph representation of the system and it will recieve states from the controller *when state changes has occurred*, not else. Based on that information the OC will classify causes of de-synchronizations, if such occur.

Dynamics

There are a number of general properties of dynamics which we assume are shared by every execution system. These properties are basically codifying the system setup depicted in Figure 1.

Property D1 If a de-synchronization is detected and it can be explained by an external action, the external action is assumed to be the culprit. For example, let $\zeta(\sigma_t, y_i^p) = 1$ and $\zeta(\sigma_t, u_{ij}^{pq}) = 1$, then we would expect $\zeta(\sigma_{t+n}, y_j^q) = 1$ for some n, but we note that $\zeta(\sigma_{t+n}, y_j^q) = 0$ and $\zeta(\sigma_{t+n}, y_k^p) = 1$, for $k \neq j$. Now, if there exists an external action u_{ik}, we set $\zeta(\sigma_{t+n}, u_{ik}^{ext}) = 1$.□

Property D2 External actions do not change the actuator output. For an external action $u^{ext} \in U^{ext}$ this means that if u^{ext} was executed in a state y_i the resulting state, say y_k will satisfy $conf(y_i) \equiv conf(y_k)$ □

Property D3 Unless an external action is invoked, an action invokation always succeeds such that the actuators never fail to produce the expected output. This means that, unless an external action is invoked at t, if $\zeta(\sigma_t, u^{pq}) = 1$ then $\zeta(\sigma_{t+n}, c^q) = 1$. □

We will now define the dynamics of the specific system we are studying, and put more specific contraints on the control graphs. In this paper we will particularily consider *PLC dynamics*.

PLC Dynamics

In the PLC case we can (fairly) safely assume that the sampling period is sufficiently short. Thus, we do not have to consider TE. The dynamics of a PLC can be described as follows:

Definition (PLC Dynamics)
An execution system is said to have *PLC Dynamics* if it exibits the following behaviour:
Assume that the state y_i has materialized, and that there is a control action u_{ij} on the optimal goal path to the current goal. Then the action u_{ij} is executed exactly until a sample recieved from the plant and actuators does not satisfy y_i. □

In the terminology of (Nilsson 1994) we, thus, assume all actions to be *energized* rather than *ballistic*.

Now, to able to distinguish between the causes of de-synchronization we will need the following property of the control graph:

Property PLC4 Two consecutive states in any optimal goal path never have the same control configuration. □

Property PLC4 implies that we do not allow *pure logging*, that is, that every transition has to be due to actual change in the actuator output.

Finding the Causes of De-Synchronization

We assume that we have an execution system with PLC dynamics, and that the system has a well-designed controller with properties D1-D3, and PLC4. For this system it is possible to formally define how the three interesting causes of de-synchronization can be detected. To be able to use properties D1-D3, we will need to start by checking for EEA, then check for UEA, and finally check for VOA. The checking can be done locally, i.e. we only need to consider two states: y_i which we assume materialized at time point t, y_j which was expected to materialize at $t + n$ due to some control action u_{ij}. If a de-synchronization has occurred we know that we have recieved a sample σ_{t+n} such that $\zeta(\sigma_{t+n}, y_j) = 0$.

1. If there exists an external action u_{ik}^{ext} and $\zeta(\sigma_{t+n}, y_k) = 1$, we say that the de-synchronization is caused by EEA.
   Motivation: This follows immediately from property D1.

2. If $\zeta(\sigma_{t+n}, conf(y_i)) = 1$ we say that the de-synchronization is caused by UEA.
   Motivation: By the second constraint for well-designed controllers we know that two state with different control configurations can never be satisfied by the same sample. By property PLC4 we know that y_i and y_j do not have the same control configuration, since they are consecutive. Since $\zeta(\sigma_{t+n}, conf(y_i)) = 1$ the control configuration has not changed which only occurs at state transitions due to external actions (by D2). Since the de-synchronization is not caused by a EEA, we say that it was caused by UEA.

3. If $\zeta(\sigma_{t+n}, conf(y_j)) = 1$ we say that the de-synchronization was caused by VOA.
   Motivation: Obviously neither EEA or UEA has occurred since the control configuration $conf(y_i)$ is not satisfied. Instead, the control action u_{ij} has changed the control configuration appropriately, by property D3, i.e. we have detected the correct changes in the actuators but these are not reflected by the effects of the actuators on the plant.

Theorem 1 For an execution system with PLC dynamics, with a well-designed controller with properties D1-D3, and PLC4, we can classify any de-synchronization as EEA, UEA, or VOA.
Proof: Assume that y_i materialized at time point t, y_j which was expected to materialize at $t + n$ due to

some control action u_{ij} which was executed in y_i at t. If a de-synchronization has occurred we know that we have recieved a sample σ_{t+n} such that $\zeta(\sigma_{t+n}, y_j) = 0$.

It is clear that if either of $\zeta(\sigma_{t+n}, conf(y_i)) = 1$ or $\zeta(\sigma_{t+n}, conf(y_j)) = 1$ holds the classification can be done as described above. Assume instead that neither $conf(y_i)$ nor $conf(y_j)$ are satisfied by σ_{t+n}. This clearly violates property D3, since either u succeeds to change the control configuration or it does not. It it does σ_{t+n} will satisfy $conf(y_j)$, and if it does not σ_{t+n} will satisfy $conf(y_i)$. □

We will examplify the detection of UEA and VOA.

Running Example Cont'd

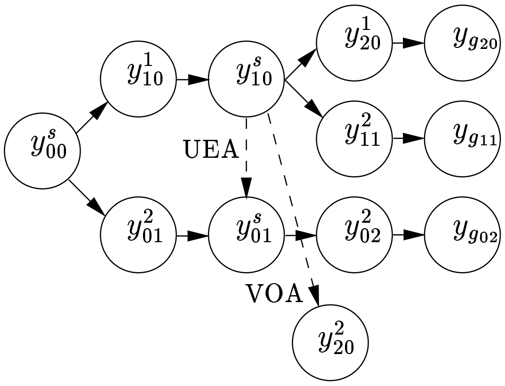

Figure 4: VOA and UEA.

Assume that the current goal is $y_{g_{11}}$ and that the plant is in state y^s_{10}. According to the second optimal goal path the action $M2$ should be invoked and the expected next state is y^2_{11}. The de-synchronizations are depicted in Figure 4.

UEA: If an operator would move the box on the first trolley to the second the state y^s_{01} would materialize. Since there is no expected external action connecting y^s_{10} and y^s_{01} and the states have the same control configuration ($pos = s$) the conclusion is that the de-synchronization is due to a UEA.

VOA: One of the ontological assumptions in our examples is that the sensors $l1$ and $l2$ on the trolleys correspond to the positions `t1` and `t2`, respectively. If the trolleys would change places while executing $M2$, the resulting state would be y^2_{20} (since the arm would put a block on position `t2`, where, now, the sensor $l1$ is situated), which is one of the erroneous states. But, since the expected control configuration $pos = 2$ materialized, but the expected proper plant formula did not, the conclusion is that a VOA has occurred.

Synchronization with the Goal Seeking Operation (GSO)

When de-synchronizations occur, and are detected and classified by the OC, the OC invokes the GSO and overrides the controller by sending action invokation signals to the actuators. As mentioned above, when the controller is turned on, the control configuration output is obsolete and the controller must rely solely on the proper plant part of the sample. Thus, it is unclear what state the plant is in relative to the controller. The task of choosing the "best" state that is satisfied by the sample is called *synchronization*. The best state is a state from where the goal state can be reach so that as much of the optimal goal path to the goal state as possible is traversed. This notion of "best" is motivated by the intuition that every step in an optimal goal paths is necessary for the controller to achieve the goal.

As we will see below, there are other situations where we also need to synchronize the plant.

In the *Goal Seeking Operation* (GSO) we implement the synchronization and best state notions as follows:

Algorithm (GSO)
Input is a control session, $\langle C, y_{g_i}, < \rangle$, and a sample, σ. Output is **nop**, **fail**, or a control action u.

1. Generate the set of states

$$S' = \{y \in S \mid \zeta(\sigma, plant(y)) = 1 \text{ and } R(y, y_{g_i}) \text{ holds.}\}$$

2. If $y_{g_i} \in S'$, then RETURN **nop**.

3. If $S' = \emptyset$ then, if y_{g_i} is the least element according to $<$ then RETURN **fail** and STOP, else let $y_{g_{i-1}}$ be the new goal, where $y_{g_{i-1}}$ is the next goal with lower priority than y_{g_i} according to $<$, and GOTO 1.

4. Let $O_i = \langle Y_i, U_i \rangle$ be the optimal goal path with goal state y_{g_i}. Let $K = S' \cap Y_i$.

5. If $K \neq \emptyset$, pick the state $y_i \in K$ with highest value on $d(y_i, y_{g_i})$, RETURN the control action u_{ij}, and STOP.

6. Let y_{i_1} denote the initial state in O_i, and connect a number k_y with each $y \in S'$ as follows:

$$k_y := \min\{d(y_{i_1}, y') \mid R(y, y') \text{ holds, and } y' \text{ is on } O_i\}.$$

7. Let

$$S_{min} = \{y \in S' \mid \text{For all } y' \in S', k_y \leq k_{y'}\}.$$

A unique state on the optimal goal path to y_{g_i} is reachable from every member of S_{min}. Let this unique state be y_o.

8. Let

$$S'_{min} = \{y \in S_{min} \mid \text{For all } y' \in S_{min}, d(y, y_o) \leq d(y', y_o)\}.$$

9. Pick any member $y_i \in S'_{min}$ and RETURN an action u_{ij}, such that $d(y_j, y_o) = d(y_i, y_o) - 1$.

□

In step 1 we remove all candidate states from which the current goal state is not reachable. Then if the goal state is a candidate, we ssume that the goal is reached. If no candidates remain (step 3) we have to change the goal state, and we pick the next state with lower priority. If the current state had the lowest priority, the controller cannot decide what action signal to send. In step 4 we construct the set of all candidates that belong to the optimal goal path leading to the current goal, and in step 5 we pick the state as close to the initial state in that optimal goal path as possible, and return the action leading from it. In step 6 we know that there are no optimal goal paths leading to the goal state, so we start to concentrate on non-optimal goal paths by (in step 7) looking at states from which minimal *entries*, i.e. states as close to the initial state of the optimal goal path as possible, is reachable. From those states we can maximize the number of traversals in the optimal goal path. Since the optimal goal paths are totally ordered we will get a unique entry. Next, in step 8, we generate the set of states from where the distance to the entry is minimal. Now, the remaining states have equal (minimal) distance to the entry. It is possible to constrain the choice of state and action further, but in this paper we will not, so the algorithm arbitrarily chooses on state in the remaining set and returns an action that is expected to lead one step closer to the entry.

If we would get the output $l1 = 0 \wedge l2 = 1$ in our example, we can see that there are two candidates for optimal goal paths. If our current goal is $y_{g_{02}}$, the GSO will be able to guide the execution to this goal, but if the goal is $y_{g_{20}}$ it will fail.

Properties of the GSO

The matching performed in step 1 can be done in $\mathcal{O}(n)$ time for every formula, where $n = |X|$. Thus, if $m = |S|$, step 1 can be done in $\mathcal{O}(nm)$ time. The reachability computation in steps 1 and 7 can be done in $\mathcal{O}(m)$, and in the worst case we might have to perform it for every state, which yields $\mathcal{O}(m^2)$ time. By using Dijkstra's algorithm (which runs in $\mathcal{O}(m^2)$) the distance function can be computed in $\mathcal{O}(m^3)$ (since we have fixed y_o). However, the worst case appears when there is a path to a unique state on the optimal goal path from every other state, which obviously cannot be the case.

Application

The ideas presented here will be implemented for the ABB ISY steel roll flatness control system STRESSOMETER$^{\text{TM}}$, which is equipped with 64 magnetico-elastic sensors on one roller, and a number of hydraulic tilting devices and cooling systems as possible actuators. This application is today controlled by a standard industrial PLC system (ABB MasterPiece 200/1) with 400 input variables, and programmed with around 1 million lines of code in FBD (Function Block Diagram) like language. The system is divided into a large number of blocks with $10 - 20$ inputs (from the sensors) and $10 - 20$ outputs (to the actuators), which run concurrently. One OC will be connected to each of these blocks which will yield an average state set of 60 states (most states in the state space defined by the sensors are error states).

Discussion

As mentioned in the Introduction, this paper builds on (Driankov & Fodor 1993; Fodor 1998). That work is, however, specifically focused on PLC-systems, which means that a number of assumptions are implicit. In this paper we have generalized and formalized those ideas by formally stating the necessary assumptions, and using PLC dynamics only as an example. Moreover, constraints on the state set in the original papers has been considerably weakened.

In industry, new technology often gets rejected due to high introduction costs; It is not considered economical to spend a lot of time reimplementing or reengineering systems that already are proven to work, unless the benefits are very high. This important issue is considered in our work. The OC:s are supposed to be connected to any existing controller exhibiting the properties described above. PLC programs in state transition-based languages of SFC (Sequential Function Chart) type (e.g. Grafcet) can quite easily be translated into the graphs used in this paper. For RLL (Relay Ladder Logic) programs there exists a number of papers (e.g. (Falcione & Krogh 1993)) describing how the programs can be translated into SFC. We conjecture that such translations from FBD can be automated as well.

Related Work

In (De Giacomo, Reiter, & Soutchanski 1998) the high-level robot programming language GOLOG is extended with execution monitoring. However, it is explicitly assumed that discrepancies between sensor information and the internal representation always is due to UEA (exogenous actions). This work is one of the few examples in the literature where explicit assumptions about the causes of de-synchronizations are made.

Livingstone (Williams & Nayak 1996) is an execution monitoring system (or, configuration management system) with two distinct stages: Mode Identification (MI) and Mode Reconfiguration (MR). Livingstone uses a model which is represented by a modal propositional logic that has transitions system semantics (Manna & Pnueli 1992). MI identifies the current state (configuration) of the system, and MR generates a control action. Basically, this is exactly what the GSO does. MI does not classify the cause of discrepancies, it generates the best candidate (in a probabilistic sense), or none if an unrecoverable error has occurred.

Another interesting approach from the Qualitative Reasoning community is MIMIC (e.g. (Dvorak & Kuipers 1989)), where a qualitative model of the plant is used for simulation, and discrepancies between simulation results and actual plant output is analyzed in correlation to a fault model.

Conclusion

In this paper we have presented Ontological Control which is a control "paradigm" for execution systems. The basic problem is to handle de-synchronizations, i.e. to find the best candidate of the states that is satisfied by the actual plant state. When the controller can follow the pre-specified optimal goal paths, no action is taken by the ontological controller. It is when a de-synchronization occurs that it becomes active. It's first task is to classify the cause of the de-synchronization, and if it is a VOA, to alert an operator. The second is to guide the system towards the best possible goal.

The first task can be achieved if the controller never outputs the same signal for two consecutive state-transitions. The second task is performed by the GSO, which is a recovery procedure defined in this paper. These two task are to be implemented in an Ontological Controller.

References

Ambrose-Ingerson, J., and Steel, S. 1988. Integrating planning, execution and monitoring. In *AAAI'88*.

De Giacomo, G.; Reiter, R.; and Soutchanski, M. 1998. Execution monitoring of high-level robot programs. In *Working Notes of the 4th Symposium on Logical Formalizations of Commonsense Reasoning*.

Driankov, D., and Fodor, G. 1993. Ontological real-time control. In *Proceedings of the First European Congress on Fuzzy and Intelligent Technologies*.

Dvorak, D., and Kuipers, B. 1989. Model-based monitoring of dynamic systems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. Los Altos, CA, USA: Morgan Kaufmann.

Falcione, A., and Krogh, B. 1993. Design recovery for relay ladder logic. *IEEE Control Systems* 13(2):90–98.

Fikes, R.; Hart, P.; and Nilsson, N. 1972. Learning and executing generalized robot plans. *Artficial Intelligence* 4(3):251–288.

Fodor, G. 1998. *Ontologically Controlled Autonomous Systems: Principles, Operations and Architecture*. Kluwer Academic.

Haigh, K., and Veloso, M. 1996. Interleaving planning and robot execution for asynchronous user requests. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*.

Manna, Z., and Pnueli, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag.

Nilsson, N. 1994. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research* (1):139–158.

Pell, B.; Gat, E.; Keesing, R.; Muscettola, N.; and Smith, B. 1996. Plan execution for autonomous spacecraft. In *Working Notes of the AAAI Fall Symposium on Plan Execution*.

Williams, B., and Nayak, P. 1996. A model-based approach to reactive self-configuring systems. In *AAAI'96*.