Continuous Governance – a novel dialect for IT automation

Raphael Hans
Stuttgart, Germany
Raphael.Hans@continuous-governance.org

***Abstract*—Continuous governance (CG) is proposed as an automation dialect for the operation of IT services. An automation is written as a finite state machine**

$$(Q, \Sigma, \delta, q_0). \quad (1)$$

Observability, controllability, and a policy-as-code trace are properties of this design. Moreover, such automaton can be executed in parallel. Its multi-threading capabilities makes it applicable for any enterprise size. The design is demonstrated on a real-world inspired automation use-case. In summary, the presented work gives an outlook for a novel IT service operation.

Keywords—automation, state machine, governance, IT, AI

I. Introduction

Software development is a craft which evolves over time [1]. In recent decades, centralized versioning concepts such as Subversion are substituted by distributed ones such as *Git.* Organizational *models* are changing from *Waterfall* to role-based *Agile* [2]. Many of these software-development-driven patterns have transformed into domain-agnostic enterprise and organizational methods [3].

A continuation of this thinking is continuous deployment, as represented by tools such as Argo CD [4]. Here, modern infrastructure-as-code methods are used to adapt software-development patterns to service operations. This is a complete shift in a modern DevOps engineer's deployment routine and replaces legacy deployment pipelines based on classical CI tools. This is surprising for the following reason:

While Ops in DevOps has evolved over time into its own automation dialect, Dev has no automation dialect of its own.

This whitepaper proposes a dedicated automation dialect for DevOps engineers in IT environments, called "continuous governance". It focuses on an end-to-end practice, starting at the DevOps engineer's policies to be implemented and continuing through the calculated events of the compiled automation system during its execution. This is a governance- and audit-oriented approach that aims to match daily DevOps practice and responsibility more closely. In fig. 1, its conceptual difference from existing automation approaches is indicated.

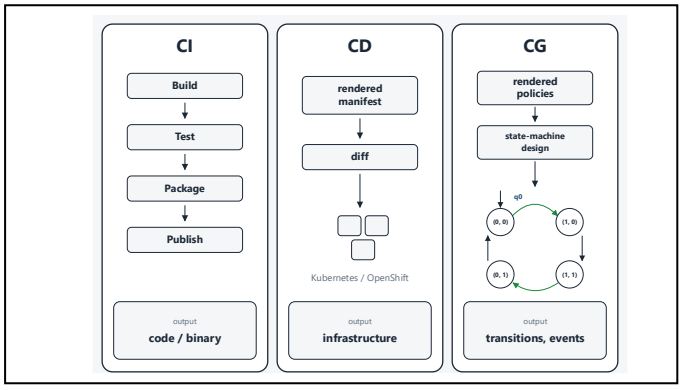


Fig. 1. CI pipeline, CD manifest sync, and CG state-machine design with policy sync

II. State Machine

The automation is written as a finite state machine of the form

$$(Q, \Sigma, \delta, q_0), \quad (2)$$

where

- Q is a finite set of states
- Σ is a finite set of events
- $\delta: Q \times \Sigma \rightarrow Q$ maps a state and an event to the successor state

which is its typical representation [5]. Any automation problem is expressed by a desired state *r* and a current state *y*. The automaton state is that pair

$$q = (r, y), \quad (3)$$

While *r* is derived from the IT environment, *y* is measured from the target system, the *Service-under-Control* (*SuC*). *Q* is the finite set of pairs the problem distinguishes. The pair at the start reads

$$q_0 = (r_0, y_0). \quad (4)$$

Drift is the difference between the desired state and the current state

$$e = \Delta(r, y). \quad (5)$$

Where *r* and *y* are predicates rather than numbers, *e* is the set of predicates on which they disagree. The event $\sigma \in \Sigma$ is selected from *e*. The transition is the map already given

$$\delta(q, \sigma) = q' \quad (6.1)$$
$$q' = (r', y'). \quad (6.2)$$

r′ is the *desired state* at the successor, derived from the IT environment. *y′* is the *current state* measured from the *SuC* after *σ*. The state concept above does not hard-code one workflow. It evaluates a declared machine. Each automation problem is one declaration of the states according to (3) and of the transitions according to (6). In the following, the core properties of such a system are investigated.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

III. Observability & Controllability

Any IT automation task that can be expressed as $q = (r, y)$ with a corresponding set of transitions $\delta(q, \sigma) = q'$ can be represented by the presented state machine design. To such state machines, the same fundamental control theoretical principles apply. For a DevOps engineer, this means the following for an automation task:

- observability: the current and desired state of the *SuC* can be observed and/or calculated.
- controllability: events $\delta(q, \sigma) = q'$ can be implemented to set the *SuC* to the desired state

Both properties correspond to the control-engineering definitions of observability and controllability of a control system [6]. On that basis, it can be assumed that any engineering automation task in practice fulfills the criteria above, and that the state machine design can be applied.

IV. Scalability

Any IT automation task that can be expressed as $q = (r, y)$ with the transitions $\delta(q, \sigma) = q'$ separates into two atomic classes of calculations. The state and the event are calculated separately from each other. This is a split of the state machine into the data processing paths it requires and the transitions it enforces.

A. *Populator - data import and processing*

A populator is a general data importer and processor, and one required data processing path. It can read from any remote system, or process locally cached data. It calculates r and y in generic form. That result is the state as given in eq. 3

$$q = (r, y).$$

B. *Generator - drift calculation and event generation*

A generator is a general drift calculator for event emission, and one enforced transition. It calculates an individual drift. It reads the state q already produced, applies one event σ selected from e, and confirms its action. In other words, it emits

$$\delta(q, \sigma) = q',$$

and confirms

$$q' = (r', y'),$$

for the state machine. One generator applies one event.

C. *Assembly - Shared memory*

A populator and a generator are atomic: each calculates either the state or one event. Assembled, they are the finite state machine, with states $q = (r, y)$ and transitions $\delta(q, \sigma) = q'$.

All populators and generators are coupled by a shared memory, typically in the form of a database. The tables read and written by the populators and generators are the state of the machine. A further data processing path is a further populator. A further enforced transition is a further generator.

D. *Summary*

With that concept, multi-threading is built in since each state machine transition and event generation can be calculated in an isolated thread.

Note, the state machine is evaluated on tables. Their processing time is $O(n_m)$ with the table size n_m. For each member in that table, transitions and events are expected to be calculated within µs, as shown by a prototypical measurement. This is an expected performance compared to other domains, e.g. numerical simulation [7]. This enables state machine reaction times within seconds for enterprise sizes of up to 500,000 associates.

V. Policy-as-Code

Any IT automation task that can be expressed as $q = (r, y)$ with the set of transitions $\delta(q, \sigma) = q'$ is machine-readable and can be tracked by a common revision-management system. It thereby fulfills policy-as-code practice [8]. In more detail, states and transitions represent the formally abstracted policies applied to the *SuC*. Their verbal description, formal abstraction, and implementation belong to a single implementation trace, which must be actively governed in practice.

VI. Scenario: access management to a resource

A. *The Scenario*

In the following, the state machine design is applied to an IT-related use case: automated access management.

In this scenario, access for an employee to a resource is granted on a *SuC* through an Active Directory group membership. This is representative of granting employees permissions to a document, a binary, source code, or other material served by the *SuC*. The input is a reference Active Directory group. Its member set is r, derived from the enterprise Identity Provider (*IdP*). The member set currently on the resource is y, fetched from the platform, the *SuC*. The output is the events that add or remove a member on that resource.

The term e is the drift of the two sets. Members in r but not in y are to be added. Members in y but not in r are to be removed. Σ contains *add(m)*, *remove(m)*, and *none*, where m is one member drawn from the drift e. It follows

$$\delta((r, y), add(m)) = (r, y \cup \{m\}), \quad (7.1)$$
$$\delta((r, y), remove(m)) = (r, y \cup \setminus \{m\}), \quad (7.2)$$
$$\delta((r, y), none) = (r, y), \quad (7.3)$$

The definitions above are based on sets, but can be rewritten as atomic member predicates

$$y \in \{0, 1\}, \quad (8.1)$$
$$\rho \in \{0, 1\}, \quad (8.2)$$

y represents the membership state on the *SuC*. r represents the membership *state* in the reference Active Directory group. The *state* is $q = (r, y)$. The four *states* are *(0, 0)*, *(1, 0)*, *(0, 1)*, and *(1, 1)*. The initial *state* is

$$q_0 = (0, 0). \quad (9)$$

The drift *e* is empty at *(0, 0)* and at *(1, 1)*, and there σ = *none*. At *(1, 0)* the predicate is *add*. At *(0, 1)* the predicate is *remove*. Each member *m* is one copy of this machine

From q_0, an employee added to the AD group on the *IdP* reaches *(1, 0)*. δ applies *add* and reaches *(1, 1)*. σ = *none* holds *(1, 1)*. An employee removed from the AD group on the *IdP* reaches *(0, 1)*. δ applies *remove* and returns to q_0. A manual change on the resource is an error: *(1, 1)* goes to *(1, 0)*, and *(0, 0)* goes to *(0, 1)*.

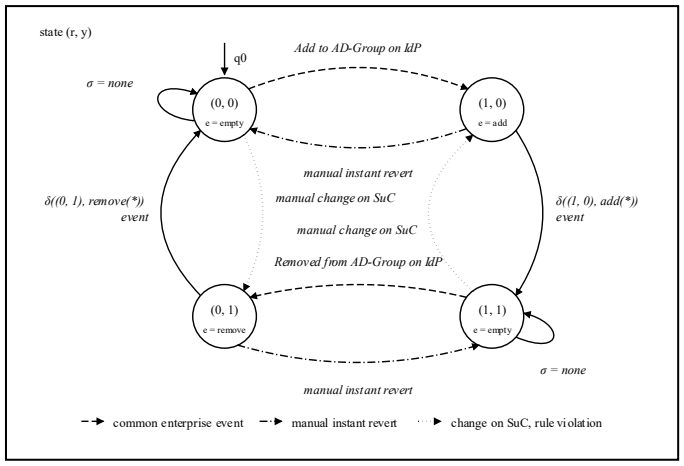


Fig. 2. The membership machine for one member. The state is (r, y). q_0 = (0, 0).

B. Summary

Access management is a simple, straightforward example of how typical automation and development tasks can be expressed as a state machine. This applies to any task that can be executed in finite steps, where the overall setup is observable and controllable. In other words, no limitation can be assumed in IT practice.

VII. Conclusion

Section 2 builds the formal framework for the state machine. Here, the automation task is formulated as a controller-oriented tracking problem and requires only basic control engineering properties to be applicable in a real-world IT scenario. Moreover, it is shown, that such a set of equations can be calculated in individual threads. This is of relevance to ensure acceptable processing times for large enterprises. Moreover, the formalism of such state machine can be interpreted as Policy-as-Code since the set of equations is machine-readable and represent the enterprise policies one-to-one. Finally, the dialect is applied to a simple member synchronization task, which is a classical service automation task in IT.

The presented work shows that state machine design is not limited to embedded devices or real-time applications. A state machine can operate network applications. That is beneficial since the operational policies and the compiled state-machine match in practice and can be governed easily.

The introduced automation dialect is highly abstract and fits the latest trends in agentic coding. More precisely, one goal of the formalism is to be the mutual language between human and AI agents. The AI agent's context is human curated knowledge, e.g. this paper. Based on recent progress, it can be expected that AI assistance is applicable end-to-end from modelling the state machine, to auditing the overall operation – while governance is remains with a human.

The presented automation dialect is the basis of an engine prototype for network applications. This prototype already operates an internal service platform at the Robert Bosch GmbH and combines unusual aspects in service operation. The compiled state machine works as

- Automation backend
- Audit and governance surface
- Cyber-Security incident handler

More use-cases are expected to be explored in future.

VIII. Future Work

While controllers find applications in engineering, biology, or chemistry [9], the application in IT is not yet known to the best of the author's knowledge. Upcoming work will focus on

- Concept hardening of "Continuous Governance".
- Release of prototypical engine, incl. real-world application examples
- Further investigations on assisted modelling and coding according to ISO 42001 in the context of end-to-end IT operations

References

[1] M. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017

[2] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[3] K. Dikert, M. Paasivaara, and C. Lassenius, "Challenges and success factors for large-scale agile transformations: A systematic literature review," *Journal of Systems and Software*, vol. 119, pp. 87–108, 2016.

[4] Argo CD Project, "Core concepts," *Argo CD Documentation*. [Online]. Available: https://argo-cd.readthedocs.io/en/stable/core_concepts/. [Accessed: Oct. 24, 2026].

[5] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston, MA, USA: Pearson, 2007.

[6] R. E. Kalman, "Mathematical Description of Linear Dynamical Systems," *Journal of the Society for Industrial and Applied Mathematics, Series A: Control*, vol. 1, no. 2, pp. 152–192, 1963

[7] R. Hans, F. Panik, and H.-C. Reuss, "1-D+1-D PEM fuel cell stack model for advanced hardware-in-the-loop applications," *SAE Int. J. Alternative Powertrains*, vol. 5, no. 1, pp. 1–12, 2015, Art. no. 2015-01-1779

[8] R. Chandramouli, Implementation of DevSecOps for a Microservices-Based Application With Service Mesh, NIST Special Publication 800-204C. Gaithersburg, MD, USA: National Institute of Standards and Technology, Mar. 2022,

[9] D. Del Vecchio, A. J. Dy, and Y. Qian, "Control theory meets synthetic biology," *J. R. Soc. Interface*, vol. 13, no. 120, Art. no. 20160380, Jul. 2016, doi: 10.1098/rsif.2016.0380