

DOI 10.70286/EOSS-19.10.2026.003.124-131

DWELL-TIME GUARD FOR FUNCTIONALLY STABLE STRATEGY SWITCHING IN SOFTWARE CONTROL SYSTEMS

Moroz Mykola

PhD student

Bychkov Oleksii

Doctor of Technical Sciences, Professor

Department of Software Systems and Technologies

Faculty of Information Technology

Taras Shevchenko National University of Kyiv, Ukraine

Abstract. Event-driven control software adapts by switching strategies when monitors report changes. Even if every strategy is stable on its own, switching too often can drive the controlled variables away from the goal and violate functional stability. We embed the classical dwell-time condition into a strategy-selection design pattern as a guard. Each strategy declares its guaranteed decay rate in a contract, an impact matrix bounds how much one switch can worsen the error measure, and the guard forbids switches closer together than the number of steps computed from these two quantities. On a two-strategy benchmark of individually stable but non-normal dynamics, 129 of 150 unguarded runs diverged, whereas every run converged for dwell times of 6 steps or more; the computed bound (10 steps) exceeded the worst-case threshold (7 steps) by a factor of 1.4. A debounce filter calmed random detector flicker but failed under periodic requests, which the guard handled; the two are complementary.

Keywords: dwell time, functional stability, software control systems, strategy switching, design pattern.

Introduction. Software that controls physical objects such as drones, robots or industrial plants increasingly adapts at run time: when a monitor reports a lost target, a degraded sensor or a failed actuator, the system activates another control strategy [1]. Event-driven architectures make this adaptation modular, because modules subscribe to events and choose a strategy in response [2]. Our earlier design pattern follows this scheme and aims at functional stability, i.e. the ability of a system to maintain its designated functions within specified quality bounds under disturbances. Formally, a system is functionally stable if, for each of its designated functions f , the quality measure $Q_f(t)$ stays above a minimum threshold $Q_f^{\min}$ at all times (the strong form) or returns to that level within a finite time (the weak form) [2].

The base pattern, however, does not limit how often the selector may change the strategy, and switching between individually stable strategies can itself destabilize a system [3]. A small example shows how. Let the two-component vector x_k be the tracking error at step k (for instance, the horizontal and vertical offsets of the target

from its desired position), and let strategy q transform it from one step to the next by a matrix F_q : $x_{k+1} = F_q x_k$. Such a linear model describes the closed-loop error dynamics: the diagonal entries show how fast each component decays on its own, the off-diagonal entries how strongly the components drive each other. Take two strategies with

$$F_1 = \begin{pmatrix} 0.7 & 1 \\ 0 & 0.7 \end{pmatrix}, \quad F_2 = \begin{pmatrix} 0.7 & 0 \\ 1 & 0.7 \end{pmatrix} \quad (1)$$

Each matrix is triangular, so its eigenvalues are the diagonal entries, 0.7 and 0.7. The spectral radius, i.e. the largest modulus of an eigenvalue, is the asymptotic factor by which repeated multiplication by a matrix changes the error. Here it is 0.7, which is below one, so each strategy alone drives the error to zero. If the strategies alternate at every step, the error is multiplied every two steps by $F_2 F_1$, whose eigenvalues are approximately 1.85 and 0.13. Its spectral radius 1.85 exceeds one, so the error grows by about 1.85 times every two steps. In software terms, a detector that flickers between "target found" and "target lost" makes the controller chatter between its nominal and recovery strategies, and the error grows although both strategies are correct.

Practitioners counter chattering with debouncing, hysteresis or rate limiting, which are tuned empirically and carry no guarantee for the dynamics of the controlled object. Control theory offers quantitative conditions instead: multiple Lyapunov functions [4], a minimum dwell time between switches [5], average dwell time [6], its discrete-time form [7] and a systematic treatment [8]; control-theoretic software engineering advocates such guarantees for adaptive software [9]. The condition itself is thus classical. What software practice usually lacks is a way to obtain its parameters from software artifacts and to enforce it architecturally. This paper addresses that gap.

Aim and objectives. The aim is to make strategy switching in software control systems compatible with functional stability by bounding the switching rate with a guard whose parameter follows from declared properties of the strategies. The objectives are to map the quantities of the dwell-time condition onto the pattern, to design the guard and the modified selector, to evaluate the guard and the conservatism of its bound, and to compare it with a debounce filter.

Results and discussion. We explain the theory (Section 1), the modified selector (Section 2), the experiments (Section 3) and the limitations (Section 4) in turn.

1. From strategies to the dwell-time condition

We use the target-tracking scenario of [2] as the running example. Its strategies are Nominal (track the target) and Recover (search for it after a loss). The state x is the tracking error and the goal is $x = 0$. Its size is measured by the Euclidean norm $\|x\|$, the length of the error vector, which the software computes from the measured error components. We let the quality of the tracking function decrease with $\|x\|$, so that the threshold $Q^{\min}$ corresponds to a tolerance radius ε_F and the quality bound holds when $\|x\| \leq \varepsilon_F$. Convergence of x to zero, which the guard secures, therefore gives the weak form of functional stability for every $\varepsilon_F > 0$ in the disturbance-free case; the strong form would also require a bound on the transient and is not claimed here.

While strategy q is active, the error evolves as $x_{k+1} = F_q x_k$ at discrete steps k , and the selector chooses the active strategy in response to events such as "target

moved" or "target lost". Every strategy is good on its own, so the question is what switching between them costs and how software can bound that cost. Table 1 lists the quantities introduced below together with the pattern elements that carry them.

Table 1 – From theory to the pattern: what each quantity means and where its value comes from

Quantity	Meaning in a software control system	Pattern element	How the value is obtained
Certificate $V_q(x)$	Non-negative measure of the distance from the goal from the viewpoint of strategy q ; zero only at $x = 0$	certificate(state)	Chosen by the designer, e.g. $x^T P_q x$
Decay rate a_q (contract)	Factor by which the certificate value shrinks per step while q is active; the smaller, the faster	StrategyContract, rejected if $a_q \geq 1$	Derived analytically or estimated from logs
Impact coefficient μ_{pq} ; impact matrix; $\mu_{\max}$	Largest factor by which the certificate value can increase at a switch from p to q ; the matrix holds all ordered pairs, $\mu_{\max}$ is its maximum	ImpactMatrix, reports uncovered pairs	Derived analytically or calibrated: maximal logged ratio V_q/V_p times a safety margin
Dwell time N_D	Minimum number of steps between two switches	DwellTimeGuard	Computed from $a_{\max}$ and $\mu_{\max}$ by (5), rounded up strictly

To reason about convergence to $x = 0$ with one number instead of a vector, each strategy q carries a certificate $V_q(x)$: a non-negative function of the error that measures, from the viewpoint of q , how far the system is from the goal (in control theory, a Lyapunov function). It must bound the error from below, $c_1 \|x\|^2 \leq V_q(x)$ with $c_1 > 0$, so that $V_q \rightarrow 0$ implies $x \rightarrow 0$. A typical choice is the quadratic form $V_q(x) = x^T P_q x$ with a symmetric positive definite matrix P_q , a weighted sum of the squared error components (for $P_q = I$ it equals $\|x\|^2$). A contract is the promise a strategy makes about its certificate: while q is active, the certificate value shrinks at least by the factor a_q at every step,

$$V_q(F_q x) \leq a_q V_q(x), \quad 0 < a_q < 1 \quad (2)$$

The decay rate a_q is smaller for faster strategies. For quadratic certificates the promise can be checked by calculation: if P_q solves $F_q^T P_q F_q - a P_q = -I$, then $V_q(F_q x) = a V_q(x) - \|x\|^2 \leq a V_q(x)$. The contract lets the pattern reason about a strategy without reading its code, and a strategy whose declared rate is not below one is rejected at registration.

Switching from strategy p to strategy q leaves the error unchanged but replaces the measure: the same error may have a larger certificate value under q than under p . The impact coefficient $\mu_{pq} \geq 1$ bounds this increase,

$$V_q(x) \leq \mu_{pq} V_p(x) \text{ for all } x \quad (3)$$

and the impact matrix is the table of these coefficients for all ordered pairs of strategies. Its largest entry $\mu_{\max}$ is the worst-case increase caused by one switch. For quadratic certificates the tightest coefficient is the largest eigenvalue of $P_p^{-1}P_q$, i.e. the largest ratio $V_q(x)/V_p(x)$ over all error directions. Otherwise the coefficients are calibrated: the software logs the certificate values of the old and the new strategy around each switch and keeps the largest ratio V_q/V_p multiplied by a safety margin, because a sample maximum underestimates the true worst case. Strategy pairs with too few observed switches are reported as uncovered.

The two effects can now be balanced. If a strategy is kept for at least N_D steps (the dwell time), its certificate value falls at least by the factor $a_{\max}^{N_D}$, where $a_{\max}$ is the largest decay rate among the strategies, and the next switch raises it at most by the factor $\mu_{\max}$. Over one dwell-and-switch cycle the certificate value is therefore multiplied at most by

$$\rho = \mu_{\max} a_{\max}^{N_D} \quad (4)$$

If $\rho < 1$, the decay outweighs the increase at switches: the certificate value at the start of each dwell interval is at most ρ times its value at the start of the previous one, so it tends to zero geometrically, and by $c_1 \|x\|^2 \leq V_q(x)$ the error x tends to zero for every sequence of switches that respects the dwell time. Taking logarithms, $\rho < 1$ means $N_D \ln a_{\max} < -\ln \mu_{\max}$; dividing by $\ln a_{\max} < 0$ reverses the inequality and gives the dwell-time condition

$$N_D > \ln \mu_{\max} / \ln(1/a_{\max}) \quad (5)$$

in words: the strategy must stay long enough for its decay to compensate the increase caused by the switch. The guard of Section 2 enforces this lower bound on the time between switches.

Three remarks follow. First, (5) is strict because $\rho = 1$ gives no decrease, so the guard uses the smallest integer above the bound. Second, (5) is sufficient but not necessary, hence conservative; Section 3 measures by how much. Third, $\mu_{\max}$ enters logarithmically: each doubling of it adds only $\ln 2 / \ln(1/a_{\max})$ steps (1.4 here), so coarse calibration of the impact matrix suffices, whereas $a_{\max}$ close to one inflates N_D sharply. Persistent disturbances would leave the error in a neighbourhood of zero (the ultimate bound) instead of driving it to zero. We study the disturbance-free case, which isolates the effect of switching. Result (5) is the discrete-time form of classical dwell-time results [5, 6, 7]. Our contribution is its embedding into the pattern.

2. Modification of the pattern: the dwell-time guard

In the base pattern [2] an event manager delivers events by publish-subscribe. A module that receives an event asks its strategy selector for a strategy (the Strategy pattern [1]) and activates it, and the selector may change the strategy at every event. The modification makes the quantities of Section 1 explicit and enforceable: every strategy declares a contract and a certificate, an impact matrix holds μ_{pq} , and a dwell-time guard in the selector enforces N_D (Table 1). A pre-deployment audit computes $a_{\max}$, $\mu_{\max}$, N_D and ρ from the declared values and flags the configuration as unfit for

deployment if $\rho \geq 1$, if some pair of strategies is uncovered or, when disturbances are declared, if the ultimate bound exceeds ε_F .

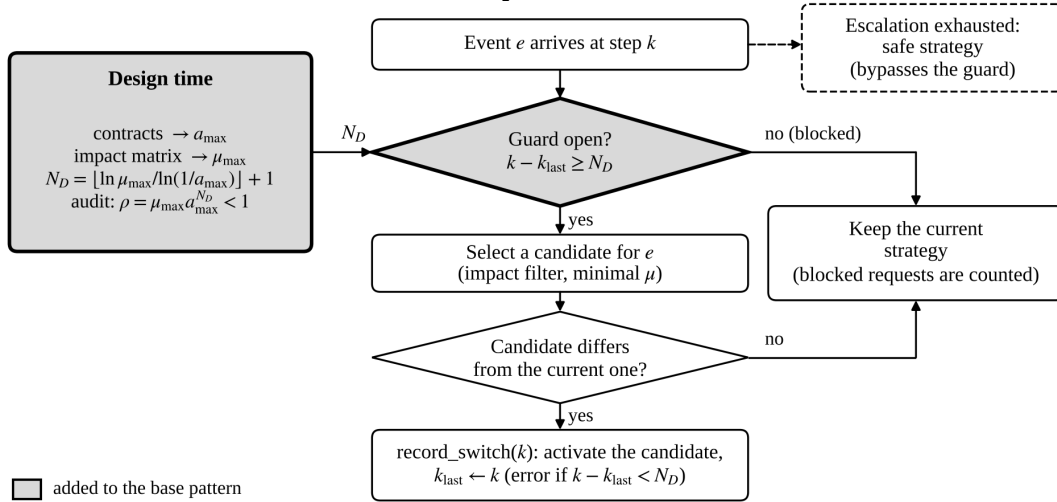


Figure 1. Decision flow of the strategy selector with the dwell-time guard (shaded elements are added to the base pattern; k_{last} is the step of the previous switch)

Figure 1 shows how the modified selector treats one event. The guard is checked first, so no other criterion can override it: if fewer than N_D steps have passed since the last switch, the current strategy is kept and the request is counted as blocked. Otherwise the base-pattern steps follow (candidate lookup, impact filter, choice), and a changed strategy is recorded as a switch. Recording is the only way to change the strategy, and it raises an error when attempted earlier than N_D steps after the previous switch, so the condition holds by construction rather than by testing. The only exemption is a transition to the safe strategy once the bounded escalation of unhandled events to higher levels is exhausted, which reacts to a failure instead of adapting (Section 4). Premature requests are discarded, not queued.

3. Experimental evaluation

Benchmark. The strategies of (1) serve as Nominal (F_1) and Recover (F_2). Their certificates are $V_q(x) = x^T P_q x$, where P_q solves $F_q^T P_q F_q - 0.6 P_q = -I$. By Section 1 this guarantees $V_q(F_q x) \leq 0.6 V_q(x)$. The tightest decay rate is $a_q = \lambda_{\max}(P_q^{-1} F_q^T P_q F_q) = 0.599$ and the tightest impact coefficient is $\mu_{pq} = \lambda_{\max}(P_p^{-1} P_q) = 162.4$ in both directions, where $\lambda_{\max}$ is the largest eigenvalue. The large value reflects the strongly non-normal dynamics (the error can grow temporarily before it decays): an error that is small for one strategy can be large for the other. A switch can thus raise the certificate value up to 162-fold, while one step lowers it by only 40 %. Condition (5) turns this balance into $N_D > 9.93$, hence $N_D = 10$ and $\rho = 0.96$, which the audit confirms. Contracts, impact matrix, guard and selector belong to the pattern's implementation, through which the experiments run.

Events and metrics. The true regime of the detector flips with probability 1/50 per step and each report is wrong with probability 0.2 (flicker). A report “target visible” maps to Nominal and “target lost” to Recover. The selector follows the reports, with hysteresis and impact filter switched off to isolate the guard. The initial error is a

random unit vector, a run lasts 600 steps, and each configuration is tested on 150 runs with independent seeds. A run converges if $\|x\| \leq 10^{-3}$ at its end (the weak form of functional stability with tolerance radius $\varepsilon_F = 10^{-3}$) and diverges if $\|x\|$ exceeds 10^6 at any step. Since $\|x_0\| = 1$, these thresholds mean a thousandfold reduction and a millionfold growth of the error length. Mismatch, the fraction of steps in which the active strategy differs from the true regime, measures the quality of adaptation.

Effect of the guard. Without the guard ($N_D = 0$), 129 of 150 runs diverged (after a median of 333 steps) and 1 converged: the controller switched 33 times per 100 steps and the error grew although both strategies are stable (Fig. 2a). From $N_D = 6$ on, every run converged (95 % upper bound on the failure rate: 2.0 %; Table 2, Fig. 2b). Random flicker is, however, not the worst case. For this pair of strategies the worst case is known exactly: a dwell block F_q^n contracts the Euclidean norm precisely for $n \geq 7$, so every admissible switching sequence converges when $N_D \geq 7$, whereas strictly alternating switching with a smaller N_D does not, as the spectral radius ρ_{per} in Table 2 shows. A periodic request stream with half-period N_D realizes this case. In simulation it diverged for $N_D \leq 5$, grew slowly for $N_D = 6$ and converged for N_D from 7 to 10. The worst-case threshold is therefore $N^* = 7$: the computed bound exceeds it by a factor of 1.4 and the random-flicker threshold by a factor of 1.7. The price of the guard is adaptation quality: switches fall from 33 to 8 per 100 steps while mismatch rises from 0.20 to 0.41.

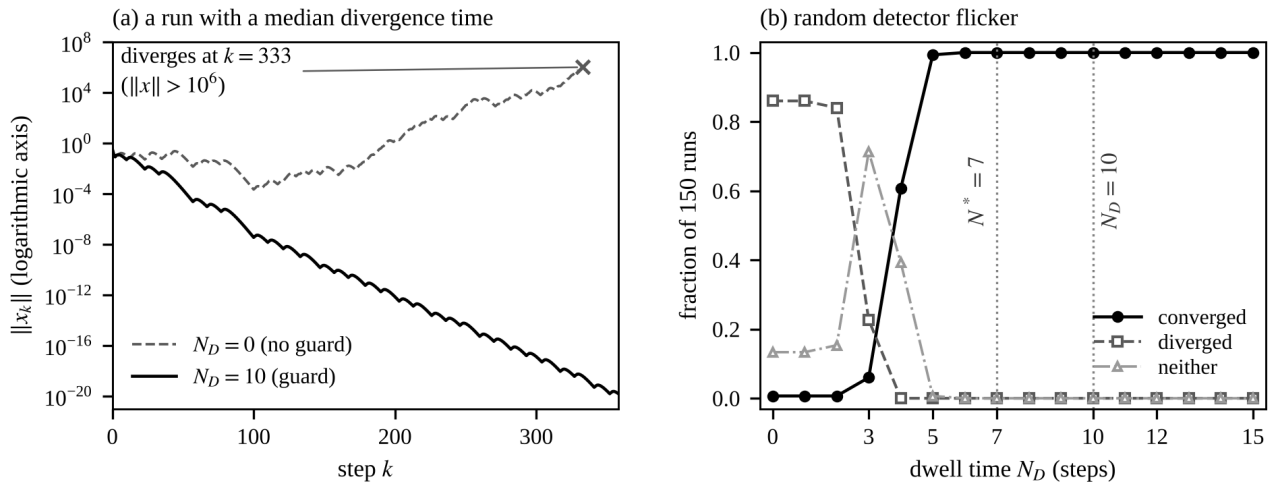


Figure 2. Effect of the guard. (a) One run with the median divergence time of the unguarded runs and identical detector reports in both cases; $\|x_k\|$ is plotted on a logarithmic axis, so a straight line means geometric growth or decay. (b) Fractions of the 150 runs per N_D that converged, diverged or did neither; dotted lines: worst-case threshold N^* and computed bound N_D

Table 2 – Sweep over the dwell time under random detector flicker

N_D	ρ_{per}	Converged	Diverged	Neither	Switches per 100 steps	Mismatch
0	–	1	129	20	32.9	0.20
3	2.39	9	34	107	19.8	0.31
4	2.00	91	0	59	16.5	0.34
5	1.50	149	0	1	14.2	0.36
6	1.04	150	0	0	12.5	0.38
7	0.69	150	0	0	11.2	0.39
10	0.16	150	0	0	8.3	0.41
15	0.01	150	0	0	5.8	0.43

*Data generated by the author: 150 runs per row. "Neither": not converged and not diverged by step 600. ρ_{per} : spectral radius of strictly alternating switching with dwell N_D .

Guard versus debounce. A common alternative accepts a request only after it has persisted for m consecutive steps. Table 3 compares it ($m = 3$) with the guard ($N_D = 10$) under the flickering detector and under a periodic stream with half-period 4. On flicker the filter alone removes the chattering and keeps the mismatch low (0.08), whereas the guard alone also converges but adapts more slowly (0.41). The periodic stream reverses the picture: it passes the filter, which fails for every half-period from 3 to 6, and only the guard guarantees convergence. Combined, they converge in all cases with mismatch 0.11.

Table 3 – Guard versus debounce filter

Configuration	Flicker: converged	Flicker: mismatch	Periodic requests (half- period 4): converged
None ($N_D = 0$, $m = 1$)	1/150	0.20	0/50
Debounce only ($m = 3$)	150/150	0.08	0/50
Guard only ($N_D = 10$)	150/150	0.41	50/50
Debounce and guard	150/150	0.11	50/50

*Data generated by the author: 150 flicker runs and 50 periodic runs per row.

4. Discussion

The computed bound is conservative but of the right order: it must hold for every switching sequence and exceeds the worst-case threshold by a factor of 1.4. The guard is a rate limiter, not a filter: it bounds the switching rate whatever the event stream does, but it reacts to the first request after it opens, which may be noise (Table 3). A debounce filter has the opposite properties, so the two are complementary. Limitations: the benchmark is synthetic, linear and disturbance-free; forced transitions to the safe strategy bypass the guard, so a complete design must prevent them from repeating (for instance by holding the safe mode until a reset); and the guard secures only one condition of functional stability, not detection latency or recovery time.

Conclusions. We embedded the classical dwell-time condition into a strategy-selection pattern as a guard whose parameter follows from two declared quantities: the decay rate in each strategy's contract and the switching impact in the impact matrix. On the benchmark the guard removed the divergence caused by fast switching. The computed dwell time was within a factor of 1.4 of the worst-case threshold; and, unlike a debounce filter, the guard held under periodic request streams, at the cost of slower adaptation. Future work: guards based on average dwell time, disturbances with ultimate bounds, run-time calibration of the impact matrix and a physical plant.

References

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley.
2. Bychkov, O. S., & Moroz, M. V. (2026). A design pattern for enabling functional stability in software systems. *Radio Electronics, Computer Science, Control*, (1), 90–102. <https://doi.org/10.15588/1607-3274-2026-1-9>
3. Liberzon, D., & Morse, A. S. (1999). Basic problems in stability and design of switched systems. *IEEE Control Systems Magazine*, 19(5), 59–70. <https://doi.org/10.1109/37.793443>
4. Branicky, M. S. (1998). Multiple Lyapunov functions and other analysis tools for switched and hybrid systems. *IEEE Transactions on Automatic Control*, 43(4), 475–482. <https://doi.org/10.1109/9.664150>
5. Morse, A. S. (1996). Supervisory control of families of linear set-point controllers, Part 1: Exact matching. *IEEE Transactions on Automatic Control*, 41(10), 1413–1431. <https://doi.org/10.1109/9.539424>
6. Hespanha, J. P., & Morse, A. S. (1999). Stability of switched systems with average dwell-time. *Proceedings of the 38th IEEE Conference on Decision and Control*, 3, 2655–2660. <https://doi.org/10.1109/CDC.1999.831330>
7. Geromel, J. C., & Colaneri, P. (2006). Stability and stabilization of discrete time switched systems. *International Journal of Control*, 79(7), 719–728. <https://doi.org/10.1080/00207170600645974>
8. Liberzon, D. (2003). Switching in systems and control. Birkhäuser. <https://doi.org/10.1007/978-1-4612-0017-8>
9. Filieri, A., Maggio, M., Angelopoulos, K., D'Ippolito, N., Gerostathopoulos, I., Hempel, A. B., et al. (2015). Software engineering meets control theory. In 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (pp. 71–82). IEEE. <https://doi.org/10.1109/SEAMS.2015.12>