# Input-Complete Finite State Machines with Explicit Failure States: A Verification-Oriented Approach for ROPS Control Systems*

Del Prete Ernesto, Gattamelata Davide, Puri Daniele, Vita Leonardo, Failla Francesco, Pera Fabio

*Abstract*— **Finite State Machines (FSMs) are widely used to develop control logic for safety-critical and embedded systems. One FSM property that should be verified is whether the implementation is partial, meaning that in some states not every possible input combination is handled explicitly. If these cases are missed, the system may reach an undefined state or may behave unsafely during run time. For agricultural tractors fitted with foldable, assisted and automatic Roll-Over Protective Structures (ROPS), such gaps can expose operators to significant risk. Unexpected combinations of operator inputs or sensor failures can cause safety-related functions to act unpredictably or fail to switch to a safe state. This paper describes a JSON-based format for specifying finite-state machines, along with a matching method for generating and analyzing code. The approach makes the transition function total by adding an explicit failure state that handles any input combination that is not otherwise defined. When designers must list all states, inputs, outputs, and expected failure behavior, FSM modelling becomes a repeatable method for writing safety-related control software. This supports worker protection and suits compliance-focused engineering under the current European regulatory framework for machinery software safety.**

## I. Introduction

Finite State Machines remain one of the most common abstractions for designing and developing control logic in embedded systems, industrial controllers, and safety-critical applications. While the mathematical model of a deterministic finite automaton assumes a total transition function, real-world implementations often behave as partial FSMs, implicitly leaving some combinations of state and inputs unhandled in real-world it is possible that, especially in complex FSM with many inputs, often they are defined only partially implicitly leaving some combinations of state and inputs unhandled. In practical code, this is frequently encoded as incomplete chains of conditional statements (e.g., if/else if), where the else branch either does nothing, or produces behavior that is not explicitly analyzed or documented.

For safety-critical systems, such partiality is problematic: unhandled input combinations may lead to undefined behavior, latent faults, or violation of safety requirements [1]. In addition, classical verification techniques and coverage metrics may fail to expose these gaps, especially when they focus on state, transition, or path coverage derived from expected use scenarios rather than from the full input space.

At the same time, the European Union is revising its legal framework for machinery safety with Regulation (EU) 2023/1230, which replaces Directive 2006/42/EC and explicitly extends safety requirements to software and digital "safety components", including stand-alone safety-related control software [2]. The new regulation introduces obligations on software integrity, cybersecurity, and the handling of substantial modifications, aligning with horizontal instruments such as the Cyber Resilience Act [3].

This paper addresses both the technical and regulatory motivations by proposing:

1. A JSON-based specification format for FSMs (Mealy machines [4]) that explicitly enumerates inputs, outputs, transitions, and a distinguished failure state.
2. A code generation scheme that transforms a pattern-specified partial FSM into a total FSM, where every undefined combination of state and inputs transitions to the failure state.
3. A lightweight analysis that checks input-completeness and flags unintended reachability of the failure state from supposedly "safe" states.
4. A discussion of how this approach supports compliance-oriented engineering under the new EU machinery and software safety framework, particularly for safety-related digital components [2][3].

The approach is demonstrated on an actual controller deployed on a tractor, where it has been used to generate the FSM that manages the behavior of a rollover protective structure (ROPS) with limit switches and unlock sensors (see Figure 1).

This real-world application shows that the method is directly relevant to machinery safety and, in particular, to protecting workers in the event of rollover or incorrect operation of the ROPS. In this sense, the proposed FSM-based approach can be regarded as a general methodology for the development of safety-related control software for machinery, such as

*Research supported by INAIL.

E. Del Prete is with the National Institute for Insurance against Accidents at Work, Rome, Italy (phone: +393355878514; e-mail: e.delprete@inail.it).

D. Gattamelata is with the National Institute for Insurance against Accidents at Work, Rome, Italy (e-mail: d.gattamelata@inail.it).

D. Puri is with the National Institute for Insurance against Accidents at Work, Rome, Italy (e-mail: d.puri@inail.it).

L. Vita is with the National Institute for Insurance against Accidents at Work, Rome, Italy (e-mail: l.vita@inail.it).

F. Failla is with the National Institute for Insurance against Accidents at Work, Rome, Italy (e-mail: f.failla@inail.it).

F. Pera is with the National Institute for Insurance against Accidents at Work, Rome, Italy (e-mail: f.pera@inail.it).

agricultural tractors, providing the software counterpart of safe machine design practices aimed at protecting workers [5][6][7][8]. The approach is general enough to be applied to a broader class of embedded controllers.

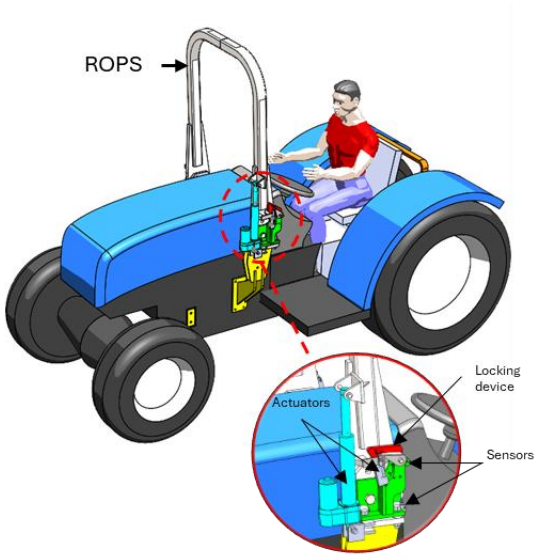


*Figure 1. ROPS structure on a tractor with actuators, sensors and locking devices.*

## II. JSON-BASED FSM SPECIFICATION

### A. Model Structure

We adopt a JSON-based format that serves both as an implementation-neutral specification and as an input to the code generator and analysis tool. Conceptually, the FSM model we adopted consists of:

- a finite set of states $S$;
- a distinguished failure state $s_{\text{fail}} \in S$;
- input variables $x_1, \dots, x_n$ with finite domains $V_1, \dots, V_n$, forming the input space $I = V_1 \times \cdots \times V_n$;
- output variables $y_1, \dots, y_m$ with finite domains $O_1, \dots, O_m$;
- an initial state $s_0 \in S$;
- for each state $s$, an ordered list of transition rules (patterns).

In JSON, each input is declared with a name and a set of values (e.g., "values": ["0","1"]), and each output is declared with a name and its enumerated values (e.g., "MOTORE_SU", "MOTORE_GIU", "OFF"). The failure_state field names the distinguished error state, while transitions groups rules per source state.

Each transition rule for a state $s$ includes:

- an input list of length $n$, where each entry is either a concrete value in $V_i$ or null to denote a "don't care";
- a target state;
- an output list of length $m$, where each entry is either a concrete output value or null (to denote "no change").

This specification gives a short, clear description of a finite state machine and supports defining inputs through patterns. This approach makes the states and transitions clear and easy to track. It also creates a tangible deliverable that can be saved in a configuration management system and included with the technical documentation, helping meet the traceability requirements in Regulation (EU) 2023/1230 [2].

### B. Safety-Related Example

As an example (see Figure 2), we consider a controller for a rollover protective structure (ROPS) with:

- binary inputs for upward/downward commands, limit switches, unlock sensors and timeout;
- symbolic outputs for motor control, block/unblock actuators, and error codes;
- states for start-up, moving up/down, fault detection, lock/unlock sequences, and error conditions.

The JSON specification uses patterns with wildcards (or *don't cares*) to describe when the controller should:

- move the motor up or down;
- transition into intermediate fault-detection states;
- signal specific error codes (e.g., timeouts or inconsistent sensor combinations);
- enter a generic error state when an unexpected input combination occurs.

Recent studies on agricultural machinery examine how ROPS design and tractor layout can improve operator safety and comfort, with an emphasis on lowering rollover risk and reducing incorrect use of foldable or low-profile ROPS frames [5]. Here, the FSM-based development flow plays a similar role on the software side as the mechanical and structural safety upgrades: earlier studies [6][7] focus on design changes that make tractors and ROPS frames safer by design, while this method focuses on checking that the control software for automatic ROPS actuation and related safety functions is correct and behaves in a fail-safe way.

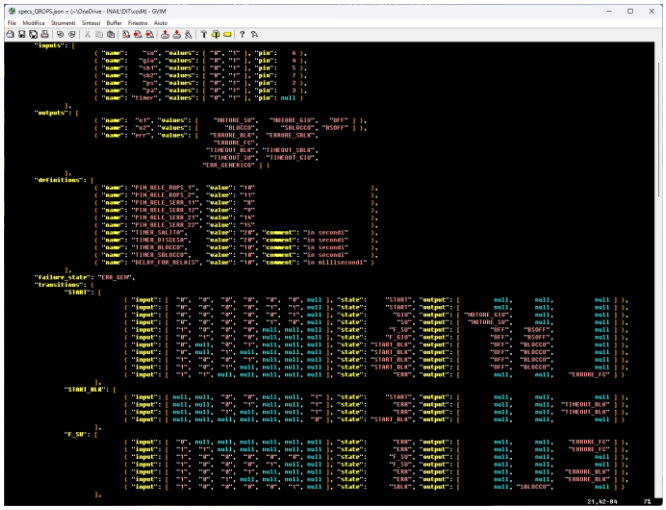

*Figure 2. JSON File for ROPS specification*

## III. Formal Semantics

### A. Pattern-Specified Partial FSM

Formally, let $S$ be the state set, $S_f \subseteq S$ the set of failure states, and the set of input vectors as defined above. For each $s \in S$, the JSON specification gives a finite ordered list of patterns:

$$T(s) = [t_1, \dots, t_k].$$

Each pattern $t_j$ is a triple $(p_j, s'_j, o_j)$, where:

- $p_j = (p_j^1, \dots, p_j^n)$, with $p_j^i \in V_i \cup \{\text{null}\}$;
- $s'_j \in S$ is the target state;
- $o_j \in O \cup \{\text{null}\}^m$ is the output vector.

For an input vector $x = (x^1, \dots, x^n) \in I$, we say that $x$ matches pattern $p_j$ if:

$$\forall i \in \{1, \dots, n\}: p_j^i = \text{null or } p_j^i = x^i.$$

This defines a partial transition function $\delta_p: S \times I \rightharpoonup S$:

- if there exists at least one pattern $t_j \in T(s)$ such that $x$ matches $p_j$, then $\delta_p(s, x)$ is the target state $s'_j$ of the first such pattern in list order;
- otherwise, $\delta_p(s, x)$ is undefined.

The corresponding partial output function $\lambda_p: S \times I \rightharpoonup O$ is defined analogously from the output vectors. This matches the view of partial FSMs used in conformance testing and test-suite completeness analysis [9][10].

### B. Total FSM with Explicit Failure State

In order to get an FSM that behaves deterministically for all state and input combinations, we compile this partial model into a total FSM by introducing an explicit failure state $s_{\text{fail}}$ and a default transition:

$$\delta(s, x) = \begin{cases} \delta_p(s, x) & \text{if } \delta_p(s, x) \text{ is defined,} \\ s_{\text{fail}} & \text{otherwise,} \end{cases}$$

with a similarly defined total output function $\lambda$ that provides a default failure output when the machine enters $s_{\text{fail}}$state. The result is a deterministic FSM with a total transition function $\delta: S \times I \to S$, in which all unspecified behaviors are mapped to a known failure configuration [9][11].

This compilation is conceptually close to hardware design practices in which synthesis tools provide "safe state" options that force the machine into a known state when an illegal state or transition is reached.

## IV. Implementation and Code Generation

The following code realizes the semantics above. For each state, the tool generates an *if/else if* chain that checks the patterns in order and a final *else* branch behaving as the default transition to the failure state. In C-like pseudocode:

```
switch (state) {
   case S:
   if (matches_pattern_1(inputs)) {
   state = TARGET_1;
   outputs = OUTPUT_1;
} else if (matches_pattern_2(inputs)) {
   state = TARGET_2;
   outputs = OUTPUT_2;
}
  /* ... additional else-if clauses ... */
} else {
   state = FAILURE_STATE;
   outputs = FAILURE_OUTPUT;
} break;
  /* other states ... */
}
```

Here, *matches_pattern_j* tests the input vector against pattern $p_j$, ignoring positions marked as wildcards. This structure is simple enough for low-resource embedded targets, and the default *else* branch directly corresponds to the mapping of undefined inputs to $s_{\text{fail}}$.

## V. Verification Oriented Analysis

Given the JSON specification, the tool checks input-completeness by enumerating all combinations and verifying that any combination is managed. In addition, the system verifies that the generated code is input complete providing all the input combinations and checking if the FSM

enters the failure state for all generated combinations, like discussed in [12]. This strategy avoids bugs in code generation.

## VI. APPLICATION TO A TRACTOR ROPS CONTROL SYSTEM

This approach was used to design the control logic for an automatic roll-over protective structure (ROPS) installed on an agricultural tractor. In this project, we used the JSON specification and the related tool to build the FSM that controls ROPS lowering, raising, locking, and unlocking (see Figure 3). The FSM was derived from the available sensor inputs, such as limit switches and unlock sensors, and from the system's power conditions. The FSM runs on the tractor's control unit (an Arduino UNO) and makes sure the mechanism moves only when the input conditions are safe. It checks for sensor readings that do not match each other and switches to clear error states when it detects faults or when a timeout occurs.

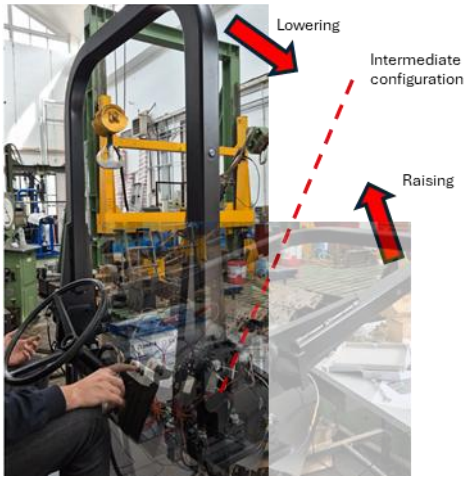


*Figure 3. Foldable ROPS main functions.*

From an occupational safety view, this case shows that the method works and can be applied to protect workers. By listing every state and input pair and sending unexpected conditions to planned failure states, the ROPS controller lowers the chance of unsafe motion or undefined behavior that could put the operator at risk during operations or roll-over events. Therefore, this FSM design, implementation, and verification flow fits with recent hardware and structural approaches aimed at making agricultural tractors safer, because it offers a clear and testable method for building the safety-related software that controls ROPS operation and helps reduce injuries from rollovers or improper use of the protection system [7][8].

## VII. ALIGNMENT WITH EU MACHINERY SOFTWARE SAFETY

Under Regulation (EU) 2023/1230, some software counts as a safety component if it carries out a safety function, whether it does so on its own or as part of machinery. For this type of software, manufacturers are required to conduct a risk assessment that considers how the software behaves under normal and abnormal conditions, ensure safeguards are in place to protect against misuse that can reasonably be anticipated, including intentional malicious interference, record the control system and its safety functions in the technical file, with enough detail to support review and compliance.

This JSON FSM model, together with the analyses described here, directly supports these tasks. The model lays out the system's states, transitions, and failure states in a form that can be included in the technical documentation. The input-completeness and failure-state checks provide objective support that every input combination is either handled directly or routed to a safe failure path [2][10]. Recent commentaries and guidance on the new Machinery Regulation, along with the Cyber Resilience Act and related initiatives, stress that safety-related software needs protection from data corruption and cyberattacks that could change inputs or internal system states. An FSM that sends any out-of-spec or corrupted input pattern to a defined failure state, instead of ignoring it or reacting in an inconsistent way, makes the system more predictable and reliable. It lowers the risk of security issues that can come from undefined behavior by limiting the number of places where such problems can occur, allows unusual conditions to be identified by reporting clear error messages and system states and supports monitoring and incident reporting because state changes can be recorded and then compared with diagnostic data [2][3]. Standards like IEEE 1012 emphasize that safety-related software should go through planned, systematic verification and validation to reduce risk and support dependable behavior [10]. In the EU machinery framework, a similar idea shows up in the requirement to show that control systems meet the EHSR throughout their entire life cycle [2]. This approach provides a formal model that combines a finite-state machine with its semantics, which can be reviewed as part of the risk assessment, automatically generated reports can be added to the technical file to document that all required inputs were provided and that any failures occurred in a controlled way, support for change impact analysis: when the JSON specification is updated, the analysis can be run again to identify newly unsafe inputs or new paths that could lead to failure states [2][3].

## VIII. CONCLUSION

This paper presents a practical approach to the design, development and verification of finite state machines for machinery control software. By specifying FSMs in JSON format with pattern-based transitions and a designated failure state, and by compiling them into total FSMs with explicit failure behavior, we enable:

- systematic checking of input-completeness and failure-state safety;
- simple, reliable code generation for embedded targets;
- creation of formal artefacts and analysis reports that support compliance with Regulation (EU) 2023/1230 and related EU instruments on software safety and cybersecurity.

The real-world deployment on a tractor automatic ROPS control system further demonstrates that the approach can be

effectively used as part of the safety concept for machinery, contributing directly to worker protection by enforcing predictable, fail-safe behavior of safety-related control functions. From a broader perspective, the approach can be regarded as a general methodology for developing safety-related control software for machinery, and in particular for agricultural machines, acting as the software counterpart of the safe machine designs proposed in recent works on tractor and ROPS safety by Gattamelata, Puri, Vita and co-authors [5][6][7][8].

The results are applicable to a broad range of foldable ROPS protective structures featuring assisted actuation and locking systems, independently of their kinematics, structural configuration, or specific design characteristics.

Future work includes extending the approach to hierarchical or concurrent FSMs, introducing symbolic methods for large input spaces, integrating with model-based testing frameworks, and mapping analysis results more directly onto the documentation structures and risk-assessment workflows expected by EU machinery and cybersecurity regulations [2][3][10][13].

## REFERENCES


[1] G. J. Myers, C. Sandler, and T. Badgett, The Art of Software Testing, 3rd ed. Hoboken, NJ, USA: Wiley, 2012.

[2] Regulation (EU) 2023/1230 of the European Parliament and of the Council of 14 June 2023 on machinery, and repealing Directive 2006/42/EC. Official Journal of the European Union, 2023, https://eur-lex.europa.eu/eli/reg/2023/1230/oj/eng

[3] European Commission. "Cyber Resilience Act." Proposal COM(2022) 454 final, 2022, https://data.consilium.europa.eu/doc/document/ST-12429-2022-INIT/en/pdf

[4] Mealy, G. H. (1955). “A Method for Synthesizing Sequential Circuits.”, Bell System Technical Journal, 34(5), 1045–1079.

[5] Gattamelata D, Vita L, Puri D, Pessina D, Galli L, An assisted folding rear rollbar for agricultural tractors: the QROPS. Lecture notes in Civil Engineering - Safety, Health and Welfare in Agriculture and Agro-Food Systems. Springer Nature Switzerland; 2024. Vol. 521 pp 195–204, Springer ISBN 978-3-031-63503-8, DOI: https://doi.org/10.1007/978-3-031-63504-5

[6] Gattamelata, D.; Puri, D.; Vita, L.; Fargnoli, M. A Full Assistance System (FAS) for the Safe Use of the Tractor’s Foldable Rollover Protective Structure (FROPS). AgriEngineering 2023, 5,218-235. https://doi.org/10.3390/agriengineering5010015

[7] Gattamelata, D., Laurendi, V., Vita, L., Galli, L.E., Pessina, D. (2023). Design, Setup and Virtual Test of a Three-Section Foldable Rear Rollbar for a Low-Power Tractor. In: Ferro, V., Giordano, G., Orlando, S., Vallone, M., Cascone, G., Porto, S.M.C. (eds) AIIA 2022: Biosystems Engineering Towards the Green Deal. AIIA 2022. Lecture Notes in Civil Engineering, vol 337. Springer, Cham. https://doi.org/10.1007/978-3-031-30329-6_59

[8] Gattamelata, D., Puri, D., Vita, L., and Fargnoli, M., 2022, July. Safety and ergonomics enhancement of the foldable rollover protective structures. In 5th European International Conference on Industrial Engineering and Operations Management, https://doi.org/10.46254/EU05.20220484

[9] A. Petrenko and N. Yevtushenko, "Conformance tests as checking experiments for partial nondeterministic FSM," in Formal Approaches to Software Testing, Berlin, Germany: Springer, 2005, pp. 118–133.

[10] IEEE Computer Society, IEEE Std 1012–2016: Standard for System, Software, and Hardware Verification and Validation. Piscataway, NJ, USA: IEEE, 2016.

[11] A. Petrenko, "Checking experiments with protocol machines," in Proc. IFIP Int. Workshop Protocol Test Syst., 1991, pp. 83–94.

[12] J. Tretmans, "Test generation with inputs, outputs and quiescence", Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems: Springer-Verlag, pp. 127-146.

[13] M. Utting and B. Legeard, “Practical Model-Based Testing: A Tools Approach. Burlington,” MA, USA: Morgan Kaufmann, 2007. ISBN: 978-0-12-372501-1