Electronics and Information Technologies, 2025, 32, 151–170
http://publications.lnu.edu.ua/collections/index.php/electronics/index

UDC: 004.4

DEFINITION AND FORMALIZATION OF THE SOFTWARE FUNCTIONAL STATE CONCEPT THROUGHOUT THE DEVELOPMENT LIFE CYCLE

Mariia Lyashkevych* , ***Vasyl Lyashkevych*** , ***Roman Shuvar***
Department of System Design
Faculty of Electronics and Computer Technologies
Ivan Franko National University of Lviv,
50 Drahomanova Str., 79005 Lviv, Ukraine

Lyashkevych, M.Y., Lyashkevych, V.Y., & Shuvar, R.Y. (2025). Definition and Formalization of the Software Functional State Concept Throughout the Development Life Cycle. *Electronics and Information Technologies*, 32, 151–170. https://doi.org/10.30970/eli.32.11

ABSTRACT

Background. Today, software is a critically important component of any information system. Its development requires significant resources and complex technical solutions, and the development of technologies is so rapid that not all concepts and definitions in the field of software are clearly formalized. This is especially true for the software functional state (SFS) throughout the software development life cycle (SDLC), as predicting all possible states is virtually impossible due to the dynamic nature of environments, changing requirements, component interactions, and the behavior of project participants. This creates a challenge for formalizing, analyzing, forecasting, monitoring, and managing these states.

Materials and Methods. The definition and formalization of SFSs encompass concepts from state theory in computer science, as well as quality models from international standards ISO/IEC 25010:2011 and the State Standard of Ukraine ISO/IEC 9126-1:2005. The defined concepts of SFS and SFS during SDLC are formalized mathematically, which allows building dynamic models of state evolution during SDLC based on the stochastic transition function. To build models, attributes such as functional compliance, reliability, vulnerability, testability, and others have been developed in combination with event-driven, finite-state machine, and state-driven models. Also presented are different types of SFS and their relationship with SDLC.

Results and Discussion. The research results include the formalization of SFS, the development of evaluation metrics, and practical recommendations for SFS analytics at all stages of SDLC, which enable proactive control of the quality, reliability, security, and compliance of software systems.

Conclusion. The formalization of the concept of SFSs, including their types, properties, and parameters, allowed for a reasonable connection to the SDLC phases. The proposed metrics and recommendations contribute to the development of SFS analytics, ensuring both the theoretical integrity of the approach and its practical applicability in the tasks of monitoring, analysis and predicting SFS. This methodology creates a new foundation for self-learning SDLC-oriented ecosystems in which SFSs are predicted, assessed and managed automatically in real-time.

Keywords: software development life cycle, software functional state, functional suitability, software state prediction, software functional state analytics, software state characteristics

INTRODUCTION

The SDLC is a system model that describes the sequence of processes for creating, implementing, and maintaining a software product [1]. Each phase [2] has its own goals, artefacts, and typical risks. The structure of SDLC phases may vary depending on the software development methodology and may include some stages [2]. An advanced comparison of the software development methodologies represented in [3].

However, modern practice proves that these phases are not isolated: they are interconnected by data and knowledge flows. That is why analysis, forecasting, or continuous monitoring of SFS becomes a key condition for ensuring quality and security throughout the SDLC. The methodology determines not only the sequence of phases but also the transition mechanisms between them. It defines the methods of SFS monitoring, types of possible defects, and risk criteria. As a result, monitoring, analysis, and prediction of SFS become key mechanisms for ensuring quality, reliability, security, and providing context throughout the entire SDLC.

Due to the demand for extension to the formal SFS definition, it is difficult to establish a universal model for assessment, diagnostics, prediction, quality management, and other analytics. In most standards [4-5], especially ISO/IEC 25010:2011 [6], quality is considered as a set of characteristics such as reliability, safety, functional suitability, efficiency and others, but no mechanism for transitioning between system states is defined, which leads to a gap between the process level and the functional level and its context of the SDLC model.

Commonly, the SFS is understood as a set of parameters that describe the current behavior, performance and quality of the system relative to its objectives, resources, and environment at a given point in time [1]. This concept is key in building intelligent process monitoring systems, multi-agent SDLC control systems, and LLM-oriented systems, and its importance suggests that modern methodology must shift from a static understanding of processes to dynamic modelling of software states. However, modern quality standards, such as ISO/IEC 25010:2011, focus primarily on quality characteristics such as functional suitability, effectiveness, compatibility, reliability, security, availability, maintainability, and portability [6]. These models lack a formal definition of the system state and therefore cannot establish a general mechanism for assessing the current SFS or predicting its changes.

The architecture of modern systems is becoming increasingly complex, ranging from microservices and multi-agent systems to distributed solutions running in the cloud or hybrid environments. Such systems typically involve continuous component changes, version updates, scaling, and load balancing, thus creating a dynamic state space [7]. Also, in modern development methodologies, especially DataOps or DevSecOps, continuous monitoring and security are based on automated metrics and predictive models [8]. Without a clear state formalization, it is difficult to build systems for degradation detection, fault prediction, or adaptive recovery.

In the context of artificial intelligence and LLMs, a new type of software is emerging where a system's behavior is determined by dynamic knowledge, context, and learning outcomes. This requires describing the state not only at the code or process level, but also at the interaction level between knowledge, model, and context [9].

In recent years, scientific publications have been actively researching methods for modelling the behavior of software systems through SFS [10]. Model-based multi-objective optimization helps address complex trade-offs in software architecture quality attributes, guiding refactoring decisions and highlighting key research challenges and future directions [11].

Existing quality models, stochastic SDLC frameworks [12], and performance monitoring methods are mainly designed for manually developed software systems, or at most, for development processes supported by partial code generation tools such as code

completion assistants or systems like Copilot. In such an environment, the behavior, architecture, and quality attributes of the software or other uncertainties [13] are largely determined by human-installed solutions, while automation tools play a supporting role. Therefore, the concept of software state in these models is often simplified to a set of observable technical characteristics or performance metrics, without explicitly including a generation environment for creating software artifacts.

Modern development paradigms increasingly rely on partially or fully automated code generation, including LLMs, agent-based development pipelines, and adaptive DevSecOps workflows. In this environment, the behavior and quality of a software system are not merely the result of static design artifacts or performance metrics, but rather the result of dynamic interactions between generating models, clues, learned knowledge, configuration strategies, and the ever-changing SDLC environment. Existing stochastic SDLC models [12] and degradation or risk prediction methods do not explicitly model this context dependency and therefore lack the ability to represent software state as a function of the technical execution and generation context.

Thus, there is a methodological gap between traditional state-based quality or risk models and the requirements of generative and agent-oriented software development. Existing methods cannot provide a single state representation that can simultaneously capture execution attributes, quality and risk characteristics, contextual dependencies, and semantics of SDLC stages, while supporting probabilistic predictive modeling, interstage thinking, and state transitions.

Bridging this gap requires extending the concept of software state from static functions to context-dependent, multidimensional functional representations. Without such state representations, some important practical problems cannot be systematically solved. Especially in the early stages of the SDLC (such as requirements analysis or architecture design), when software artifacts are still partially or fully generated, it becomes impossible to reliably predict the transition from an unstable state to a failure state. Similarly, it remains inappropriate to compare different SDLC strategies or generative development processes based on expected state trajectories rather than isolated metrics. Ultimately, because recovery, regeneration, and mitigation measures depend not only on technical metrics but also on the underlying build and execution environment, it is impossible to effectively reconcile DevSecOps with stateful or intelligent development processes.

To address these issues, this paper proposes a unified abstraction of software functional states that covers the software development lifecycle phases and runtime execution, while explicitly considering context and generation dimensions. Furthermore, a probabilistic model of SFS evolution is proposed, which allows for prediction and inter-stage reasoning, as well as a state classification framework validated through Monte Carlo simulations, demonstrating the separability and robustness of SFS under uncertainty.

Therefore, the extended formalization of the SFS concept is a prerequisite for constructing the theoretical and applied foundation of software intelligent analysis and monitoring. It integrates methods such as systems analysis, artificial intelligence, ontology modelling, knowledge engineering, and other analytics. This concept lays the foundation for developing new SDLC models that ensure quality and security not only at the process level but also at the operational status level of software products with its context.

MATERIALS AND METHODS

It is important to emphasize that the proposed formal approach is not intended to replace the traditional SDLC or quality models for manually developing software systems. Instead, it extends them to modern development environments, where software artifacts are generated, in whole or in part, by automated agents, large language models (LLMs),

or hybrid human-machine workflows. In this environment, SFS cannot be fully characterized by artifact integrity or execution time metrics alone, because the fundamental determinants of system behavior are embedded in the knowledge that generates prompts and searches, configuration strategies, and the ever-changing SDLC context.

The problem of determining the SFSs is one of the key ones in theoretical computer science and software systems engineering. It is related to the fact that modern software systems are highly complex and are being characterised by high dynamics, distribution, adaptability, and context dependence. Unlike hardware systems, where the state of additional physical parameters is determined, in software, it is formed through a set of variables that describe behavior, internal resources, calculation logic, and interaction with the environment. Also, there is a lot of contextual information throughout the entire SDLC.

Existing stochastic SDLC models and degradation or risk prediction systems can successfully explain uncertainties in process execution or operational behavior. They typically assume that software artifacts are fixed or implicitly controlled by humans. Therefore, uncertainty is mainly modeled at the level of defect frequency, failure rate, or process delay. Conversely, in generative development, uncertainty permeates multiple levels, including the variability of generated code, semantic drift in requirement interpretation, and context-aware decision-making by autonomous agents. These factors exceed the representativeness of classic stochastic models, thus necessitating the introduction of context-dependent state function abstractions.

Historical approaches to “state” description

The initial conceptions of the SFS emerged within the context of automata theory, where a system was viewed as a deterministic or non-deterministic finite automaton. This model defines a system as a finite set of states: $S = \{s_1, s_2, \ldots, s_n\}$, between which discrete transitions occur under the influence of input signals or events from the input alphabet S. [14-15] Classic models, such as the Mealy machine and the Moore machine, became the fundamental mathematical basis for formally describing the behaviour of software systems as objects that react to external events and internal state changes. Harel (1987) extended this approach by proposing Statecharts, a hierarchical graphical notation for complex systems with parallelism and nested states [16]. This approach later became the basis for UML State Machine Diagrams, which are used today to model the behavior of software components [17].

In the 1990s, the problem of determining the state became practical in the V&V (verification and validation) model, where each design phase corresponds to a specific state of the system: from requirements to testing [18]. Spiral, represented in [19], viewed the state as the result of risk iteration with the advent of agile methods. The concept of state began to be seen as a dynamic context of tests, tasks, and requirements at a certain point in time [20].

The concept of “state” in computer science

In the classic definition of computer science, state refers to the collection of all stored data and context relating to the current behavior of a system at a given point in time. Officially, in [21] stated: “In computer science, the state of a program or computational system is a complete description of its current condition, including all stored information that can affect future behavior”. Therefore, a state is a snapshot of all variables, structures, and contexts that determine how the system will respond to incoming events.

For software, the “state” definition applies not only to data in memory or files, but also to the internal logical state of modules, the execution state of processes, active services, configuration settings, and component states. Therefore, SFS reflects the current state of an application, including its internal data, behavior, and readiness to perform functions.

"State" as an object of control and diagnostics

In the theory of system reliability, the concept of SFS is interpreted as the result of the interaction between the system and the environment, which determines the system's performance [22].

In modern software engineering, state is defined through a set of variables that describe the following characteristics:

$$S = \{ (q_i, r_i, c_i, e_i, t_i) \mid i = 1 \ldots n \}, \quad (1)$$

where q_i – logical state of the process, r_i – resource load, c_i – configuration parameters, e_i – external influences, t_i – time indicators.

This representation enables monitoring and prediction using state estimation, time series forecasting, and anomaly detection methods [23]. In DevOps architecture, this is achieved through an Application Performance Monitoring (APM) system that collects metrics on CPU, memory, latency, errors, and creates an instantaneous state model [24].

The relationship between "state" and the qualitative software characteristics

The standard [6] defines a set of software quality characteristics, describing the state of software in terms of functionality, efficiency, reliability, security, compatibility, availability, maintainability, and portability. A key characteristic previously associated with the concept of functional status is "Functional Suitability". This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following components [6]:

- Functional completeness – the degree to which the set of functions covers all the specified tasks and intended users' objectives.
- Functional correctness – the degree to which a product or system provides accurate results when used by intended users.
- Functional appropriateness – the degree to which the functions facilitate the accomplishment of specified tasks and objectives.

Therefore, the SFS within the ISO/IEC 25010 standard is a dynamic representation of these three components at a given point in time. A product is a good product if it can perform all the necessary functions correctly and efficiently. If some functions work partially or incorrectly, or if some functions are missing, then the product is abnormal or defective. Obviously, the quality standards assume that software quality assessment systems are static, and they capture properties at a certain point in time, but do not allow modelling the evolution of states.

Definitions in international and national standards

Functional suitability is defined as the main characteristics that describe the functional efficiency of a system under given conditions [4]. This approach permits assessment of the functional implementation degree, accuracy, and relevance, namely the parameters which make up the SFS.

In the previous version of the quality model, functional suitability was interpreted as: "The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions". In [5], the emphasis is on "capability," that is, the potential SFS in relation to its purpose.

This standard [25] introduces metrics for evaluating the characteristics of functionality, reliability, and maintainability. It effectively transforms the concept of SFS into a quantitative model through indicators, such as: "percentage of functions that perform correctly", "number of failed operations per function", "execution completeness ratio". That is, SFS can be measured numerically.

Regarding the SFSs or states when the software performs or does not perform its functions, the latest national standard [26] explains the functional suitability, which refers to the extent to which the software provides functions that meet stated and implied needs under given conditions of use.

"State" in the context of adaptive and intelligent systems

The development of adaptive systems and autonomous agents has given rise to the concept of runtime models, which describe the current state of a system and can be observed and modified during execution [27].

Kephart and Chess, in their paper [28], defined the state in the concept of autonomous computation as a set of controllable properties of a system that can be measured, compared to a reference value, and adjusted without human intervention.

In adaptive and multi-agent systems, SFSs are viewed as information projections of an agent's behavior in space, such as environment, objectives, actions, and resources [29]. This allows states to be formalized through ontology, logical predicates, or vector representations.

In the context of LLMs and generative AI systems, SFSs include not only technical parameters but also cognitive parameters such as current context, query history, word segmentation parameters, weights of inner layers, state cache, etc. [30] Bubeck et al. (2023) pointed out that the behavior of LLM is a stochastic function of current knowledge and context state, and therefore can be modeled as a stateful system [9].

In retrieval augmented generation systems and multi-agent LLM environments, "states" determine action readiness, response reliability, and contextual consistency. This allows us to view not only SFSs technically but also semantic states, thereby determining the level of cognitive consistency among agents [31].

Ukrainian scientific approaches

In the Ukrainian scientific space, the concept of SFS transcends classical automata theory and has been applied to practical problems such as risk assessment, energy modeling, ontology-based diagnostics, and context-aware monitoring in decision support systems. Furthermore, the authors of reference [12] demonstrate that the probabilistic GERT-based integration model can integrate artificial intelligence intervention into the hybrid SDLC, transforming continuous integration/continuous delivery (CI/CD) telemetry data into interpretable schedule risk predictions, thereby reducing rework cycles, shortening delivery time distribution, and supporting scenario-driven optimization under cognitive uncertainty.

In article [32], a fuzzy model of risk assessment as a function of software product state change has been formed. In that model, the concept of software state is used in its fuzzy risk assessment model. SFS is treated as a variable, and the risk level of the software product depends on this variable as a function of state changes.

The analysis of scientific publications shows that existing methods fail in providing a complete formal description of the SFS, whether process-based, behavioral, standard or cognitive ones. In most models, software state is treated as a set of internal variables or performance levels, but the factors below are not considered:

- External context of the runtime environment as infrastructure, users and security events.
- Ontological relationships between states, such as hierarchical relationships, compatibility, and transitivity.
- Evolutionary dynamics of states, such as development, degradation, and adaptation.
- Intelligent decision-making processes that influence states.

Therefore, a new formal model is required to describe the SFS as a multidimensional entity in a space of technical, behavioral, cognitive, and semantic features. Such a model

will provide a unified state representation for monitoring tasks, quality assessment, risk management, and the real-time adaptation of software systems. Thus, the SFS will represent a “snapshot” of all characteristics that determine how the system reacts to inputs, performs its functions, and maintains quality in accordance with the ISO/IEC 25010 quality characteristics such as functional suitability, reliability, efficiency, security, and maintainability.

RESULTS AND DISCUSSION

Existing stochastic software development lifecycle models mainly adopt a process-centric perspective, modeling the transitions between lifecycle stages or activities, but do not explicitly represent the internal functional states of the software system. Quality models (such as those conforming to ISO/IEC 25010) provide a static snapshot of software characteristics but do not model state evolution or contextual dependencies. Runtime monitoring and performance degradation prediction methods focus on the runtime phase and treat the Software Development Lifecycle (SDLC) environment as exogenous or fixed. In contrast, the SFS model proposed in this paper adopts a state-centric perspective, covering the SDLC phase and runtime execution, integrating quality, risk, and contextual information, and explicitly supporting prediction and probabilistic reasoning. This positioning makes the model particularly suitable for environments involving some or all of the code generation, where software behavior stems from both execution dynamics and the generation context.

Formalisation of definitions

The SFS is not only a technical execution property but a dynamic multidimensional construction combining:

- System properties such as performance, stability, reliability, and others.
- Requirement compliance, such as functional and non-functional requirements.
- Current suitability, quality, risk, and security profile.
- Operational and contextual conditions such as environment, workload, configuration, knowledge context, etc.

Software functional state is a set of parameters that reflect the current quality, operability, security, integrity, and contextual compliance of a software system at a given moment in time, relative to its functional requirements, architectural constraints, resources, and operating environment.

The SFS of the system at time *t* is defined as:

$$S(t) = \langle \phi(t), \mu(t), \sigma(t), \rho(t), \kappa(t), \tau(t) \rangle, \quad (2)$$

where: $\phi(t)$ – level of functional adequacy or requirement fulfillment; $\mu(t)$ – performance indicators such as response time, resource utilization; $\sigma(t)$ – stability indicators such as failure rate, MTBF; $\rho(t)$ – risk and security indicators such as vulnerabilities, threat probability; $\kappa(t)$ – contextual and cognitive coherence for intelligent or LLM-based systems; $\tau(t)$ – SDLC phase context such as project initiation and planning, requirement analysis, architectural design, development, testing, deployment, maintenance and operations, team and management.

Although the components of the SFS vector may appear heterogeneous—combining performance metrics, quality metrics, risk metrics, contextual consistency, and SDLC stage information – this heterogeneity reflects the fundamental nature of software states in generative and adaptive systems. From the point of view of systems theory and state assessment, the state of a system is determined not by the homogeneity of its parameters, but by the completeness of the information necessary to predict future behavior. In generative software systems, this predictive completeness cannot be achieved without

explicitly modeling the context and lifecycle dimensions. Therefore, the SFS vector is constructed as a multi-layered representation of states, including core execution states, derived quality and risk states, and contextual SDLC embeddings, which together define the evolution of the system.

The integral quality of the state is defined as:

$$Q(S) = w_\phi \phi + w_\mu \mu + w_\sigma \sigma + w_\kappa \kappa - w_\rho \rho, \quad (3)$$

where w_i are the weighting coefficients correspond to a specific system class.

A software functional state during the software development life cycle is the predicted or actual configuration of suitability, quality, risk, security, and operability parameters of a software system formed at each SDLC phase under the influence of development methodology and artefacts, team decisions, and environmental factors.

This state has a dual nature:

- Predictive (forecastive) – derived from software requirements and architectural analysis with prior distribution $\pi_0(S)$.
- Empirical (consequential) – refined by real observations during development, testing, and other stages with posterior distribution $\pi_k(S)$.

It is not enough to simply know the types of SFS (**Table 1**) during SDLC and their relationships (**Table 2**).

Table 1. **Types of SFS**

SFS name	Description	Typical indicators	Formal conditions
Operational (Normal), S_N	System functionality meets requirements	SLA met, no critical bugs, vulnerabilities	$Q(S) \geq 0.9,\ \rho < 0.1$
Degraded, S_D	Partial loss of performance or efficiency	Higher latency, resource saturation, non-critical delays	$0.7 \leq Q(S) < 0.9$
Vulnerable, S_V	Functionality with security weaknesses	CVEs, authentication flaws, data exposure	$0.6 \leq Q(S)$ $Q(S) < 0.8, \rho > 0.3$
Anomalous, S_A	System behavior deviates from expected norms	Unusual requests, unstable metrics	$metric\ deviation > 3\sigma$
Defective (Buggy), S_B	Logical or functional faults without a crash	Incorrect output, UI errors	$test\ failure,$ $but\ system\ operable$
Critical (Failure), S_F	System crash or total loss of functionality	Downtime, data loss, fatal exceptions	$Q(S) < 0.4, \sigma < 0.5$
Recovering, S_R	Recovery after failure or degradation	Rollback, restart, autoscaling	$S_F \rightarrow^{recover} S_R$
Transitional (Testing), S_T	Temporary unstable state during update or deployment	Active deployment or migration	$during\ CI/CD\ or$ $configuration\ change$
Uncertain, S_U	A scope doesn't fit expectations	Timelines mismatch, employee attrition	$Sched.perf.ind. < 1:$ $behind\ schedule$

Table 2. **Relation between SDLC phases and SFS**

SDLC phase	Possible states	Typical transition causes
Project initiation and planning	S_U, S_A, S_N	Scope definition, agreement with stakeholders
Requirement analysis	S_N, S_B, S_V	Ambiguity, inconsistency, and missing requirements
Architectural design	S_N, S_D, S_V	Architectural anti-patterns, design errors
Development	S_N, S_B, S_D	Coding errors, dependency issues
Testing	S_B, S_F, S_R	Test instability, coverage gaps
Deployment	S_T, S_R, S_N	CI/CD misconfiguration, misallocation
Maintenance and operations	S_N, S_D, S_A, S_V, S_F	Load spikes, attacks, degradation, failure
Team, management	S_U, S_N	Team composition with required skills

The model must take into account the fact that with each development phase and its corresponding stage, we are getting closer to expectations. That is, at each step of the requirements analysis, architectural design, planning, or testing, we are increasingly excluding undesirable states. The proposed SFS abstraction can be understood as a unified layer between classical quality assessment, risk modeling, and generative software engineering. In the traditional model, these aspects are analyzed independently: quality models assess consistency, risk models estimate failure probabilities, and generative mechanisms are treated as implementation tools. In contrast, SFS views quality degradation, risk escalation, and inconsistency as different manifestations of the same underlying state evolution process. This perspective enables us to reason about software behavior at various stages of the SDLC, including stages where executable artifacts are incomplete or continuously regenerated.

Dynamic state evolution models are important for modelling, analyzing, and predicting the behavior and evolution of complex systems over time. These models capture the interactions between components and the influence of external factors, thus supporting informed decision-making and effective management. Simple models do not perform both prediction and simulation, allowing researchers to predict system behavior under different conditions. By revealing internal mechanisms and feedback loops, dynamic models can enhance our understanding of complex adaptive systems.

In engineering and applied sciences, dynamic models are crucial for system design and control, ensuring system efficiency and safety. They also facilitate hypothesis testing, providing a structured approach to evaluating system responses. In fields such as economics, biology, and urban planning, dynamic models are being used as decision support tools, guiding policy or strategy development.

State transitions, in the dynamic state evolution model, during the SDLC are determined by a stochastic process:

$$S_{t+1} = \delta(S_t, A_t, E_t), \tag{4}$$

where: A_t – set of actions within SDLC phases and their stages; E_t – external factors such as requirement changes, load variations, or environment.

Transition probability is being calculated:

$$P(S_{t+1} = S_j \mid S_t = S_i, \quad A_t = a) = p_{ij}(a), \tag{5}$$

Hence, the SDLC can be represented as a graph of state transitions, forming an appropriate digital model of the development process.

As discussed above, the SFS is a complex function which has some characteristics. The target set of characteristics depends on the specific software, chosen methodology, technology stack, etc. The core characteristics of SFS with typical metrics for estimation are shown in **Table 3**.

Table 3. **Core characteristics of SFS**

Characteristic	Definition	Typical Metric
Availability	The time system remains operational	$MTBF\ /\ (MTBF\ +\ MTTR)$
Stability	Sensitivity to changes and faults	$Performance\ variance\ \sigma$
Reliability	Probability of fault-free execution	$R(t) = e^{-\lambda t}$
Security	Probability of absence of exploitable vulnerabilities	$1 - CVSS_{norm}$
Code Quality	Structural and logical correctness	$Cyclomatic\ Complexity, Coverage$
Contextual Coherence	Consistency with the environment and the knowledge base	$Semantic\ Consistency\ Index$
Risk	Probability–impact product	$R_i = P_i \cdot I_i$
Viability	Time share in healthy states	$L(S) = (T_{S_N} + T_{S_D})\ /\ T_{total}$

The proposed set of indicators may be refined in the future depending on the objectives of the SFS study.

General formalization

SFS creates a hierarchy of system viability:

$$S_N > S_D > S_V > S_A > S_B > S_F, \tag{6}$$

where “>” indicates higher operational integrity.

Valid transitions form a stochastic graph of SDLC:

$$S_N \to S_D \to S_V \to S_F \to S_R \to S_T \to S_N, \tag{7}$$

Uncertain, anomalous, and defective states (S_U ,S_A ,S_B) may emerge at any SDLC phase as early indicators of potential failures.

Prior distributions of SFSs are various at different SDLC phases and their stages but can be estimated. For example, at the “Requirement analysis” phase, a prior distribution of SFS is estimated as:

$$\pi_0(S) = P(S \mid X^{(req)}), \tag{8}$$

where $X^{(req)}$ is a vector of requirement features such as completeness, security, consistency, ambiguity, etc.

As the SDLC progresses, the posterior estimate is updated using new observations O_k:

$$\pi_k(S) \propto P(O_k \mid S) \sum_{S'} P(S')\pi_{k-1}(S'). \tag{9}$$

Hence, the SFS in SDLC evolves from a probabilistic forecast to an empirically measurable condition. State Function in SDLC:

$$S_{t+1} = f\left(S_t, X^{(req)}, O_{1:t}, A_t, E_t\right), \tag{10}$$

where f describes the predictive-reactive evolution of the SFS.

Optimization criterion:

$$max_\pi E\left\{\sum_t \gamma^t [Q(S_t) - \rho(S_t) - C(A_t)]\right\}, \tag{11}$$

where policy π determines actions to maintain viable states.

SFS is a dynamic, multidimensional characteristic that describes the aggregate of quality, operability, stability, security, and contextual coherence parameters of a software system at a specific moment, formalized as a vector model S_t that evolves under the influence of internal processes and external environmental factors.

SFS in the SDLC is a stochastic function describing the sequence of transitions between SFS types:

$$\Omega = (S_N, S_D, S_V, S_A, S_B, S_F, S_R, S_T), \tag{12}$$

The statement (11) is driven by SDLC activities and artefacts – from forecastive prediction at the requirement phase to empirically validated runtime observations during operations.

Estimation of the proposed definitions

The estimation metrics of the proposed SFS and SFS during SDLC definitions should be aligned with the formal model (3) and their stochastic evolution through SDLC phases. The estimation metrics aim to:

- Quantify the current or predicted SFS.
- Detect deviations, degradation, or transitions between states.
- Support automated decision-making, diagnostics, and optimization within SDLC.
- Correlate process-level artefacts, such as requirements, commits, tests, and metrics with state-level variables.
- Detect component-level and state-level dependencies across all phases of the SDLC.
- Other software and development methodologies specifics.

The estimation dimensions represent the key measurable components of a software's functional state. They capture functional adequacy $\phi(t)$ through requirement compliance, performance $\mu(t)$ via runtime efficiency, and stability $\sigma(t)$ through reliability under perturbations. Risk and security $\rho(t)$ quantify exposure to faults or threats, while contextual coherence $\kappa(t)$ measures semantic and environmental consistency.

Finally, the lifecycle phase $\tau(t)$ situates all parameters within the SDLC process context, ensuring temporal and methodological traceability. Each SFS parameter has its own measurable indicators and a normalised score in a range of $[0, 1]$ (**Table 4**).

Composite indices for SFS estimation integrate multiple quality, performance, risk, and stability metrics into a unified score. They provide a holistic view of software health, simplify complex multidimensional assessments, enable trend detection, support automated decision-making, and allow early prediction of degradation across the SDLC.

SFS quality index $FSQI$:

$$FSQI(t) = w_\phi RC \cdot TC + w_\mu RE + w_\sigma \frac{MTBF}{MTBF_{max}} + w_\kappa SCI - w_\rho REX, \tag{13}$$

where range is $[0, 1]$.

The interpretation is:

- $FSQI \geq 0.9 \rightarrow Operational(Normal)$.
- $0.7 - 0.9 \rightarrow Degraded$.
- $< 0.7 \rightarrow Vulnerable$.

Dynamic risk index DRI:

$$DRI(t) = \sum_{i=1}^{k} P_i \cdot I_i \cdot w_i, \tag{14}$$

where P_i – probability of event i, I_i – impact magnitude, w_i – weight.

The interpretation is:

- $DRI > 0.4 \rightarrow risk\ accumulation\ trend, requiring\ mitigation\ actions$.

SFS stability index $FSSI$:

$$FSSI(t) = 1 - \frac{\sigma_{perf}(t) + \sigma_{error}(t)}{\sigma_{max}}. \tag{15}$$

The interpretation is:

- $Higher\ values = better\ stability$.
- $Values < 0.5\ indicate\ transition\ toward\ S_D\ or\ S_F$.

SDLC functional integrity LFI, which measures how consistently the system remains in acceptable states across SDLC phases:

$$LFI(t) = \frac{1}{T_{total}} \sum_{phases} Q(S_{phase}) \cdot \Delta t_{phases}. \tag{16}$$

Composite SDLC state metric $CLSM$ allows for end-to-end lifecycle monitoring:

Table 4. **Quantitative Metrics Calculations**

Dimension	Metric	Calculation	Range	Interpretation
Functional adequacy, $\phi(t)$	Requirement Coverage, RC	$RC = \frac{N_{implemented}}{N_{specified}}$	$[0,1]$	Degree of implemented requirements
	Requirement Consistency, $RCon$	$RCon$ = NLP-based coherence score between requirements	$[0,1]$	Higher = fewer conflicts
	Test Pass Ratio, TPR	$TPR = \frac{N_{passed}}{N_{total}}$	$[0,1]$	Functional correctness
	Technical Coherence, TC	$TC = 1 / \frac{N_{adapted}}{N_{expected}}$	$[0,1]$	The right tech stack, the higher = the fewer mismatches
Perform., $\mu(t)$	Response Efficiency, RE	$RE = 1 - \frac{t_{resp}}{t_{SLA}}$	$[0,1]$	Lower latency → higher RE
	Resource Utilization Efficiency, RUE	$RE = 1 - \frac{CPU + MEM}{100}$	$[0,1]$	Optimal performance balance
Stability, $\sigma(t)$	Mean Time Between Failures, $MTBF$	$MTBF$ = Empirical	$[0,\infty)$	Higher = more stable
	Variance of Key Metrics, $\sigma_{per\ f}$	$\sigma_{per\ f}$ = standard deviation of latency/load	norm. $[0,1]$	Lower = more stable
Security/ risk, $\rho(t)$	Vulnerability Index, VI	$VI = 1 - \frac{\Sigma(CVSS_i \cdot w_i)}{\Sigma w_i \cdot 10}$	$[0,1]$	1 = no vulnerabilities
	Risk Expo-sure, REX	$REX = P(Event) \times Impact$	$[0,1]$	Expected severity of failure
	Attack Surface Ratio, ASR	$ASR = \frac{N_{exposed\ interfaces}}{N_{total\ interfaces}}$	$[0,1]$	Lower = more secure
Contextual coherence, $\kappa(t)$	Semantic Consistency Index, SCI	$SCI = sim(v1, v2)$	$[0,1]$	High value = coherent with context
	Configuration Drift, CD	$CD = 1 - sim(conf_{runtime}, conf_{baseline})$	$[0,1]$	0 = perfect match
Lifecycle robustness, $\tau(t)$	Phase Consistency, PC	Entropy of transitions $PC = -\Sigma\ p_i\ log\ p_i$	$[0, logN]$	Lower = more deterministic SDLC
	Temporal Integrity, TI	TI = delay between planned and actual phase completion	norm. $[0,1]$	Schedule stability

$$CLSM(t) = \frac{1}{N_{phases}} \sum_{k=1}^{N_{phases}} (FSQI_k - DRI_k), \quad (17)$$

where range is $[0, 1]$.

The interpretation is:

- $CLSM > 0.7 \rightarrow SDLC\ is\ healthy\ (high\ functional\ integrity)$.
- $CLSM\ between\ 0.4 - 0.7 \rightarrow partial\ degradation$.
- $CLSM < 0.4 \rightarrow instability\ or\ risk\ accumulation$.

At the moment, we are setting the thresholds for SFS classification that are defined by analyzing normalized quality, risk, and stability metrics such as $FSQI$, DRI and $FSSI$across multiple SDLC phases and mapping their statistical distributions to empirically observed states. The entire table with the thresholds is shown in **Table 5**.

Table 5. **Thresholds for SFS classification**

SFS	$FSQI$ range	DRI range	$FSSI$ range	Qualitative interpretation
Normal, S_N	0.90 – 1.00	< 0.10	> 0.85	Fully stable, performant, secure; system meets or exceeds all requirements.
Degraded, S_D	0.78 – 0.90	0.10 – 0.22	0.70 – 0.85	Minor performance decline or partial overload without failure; functionality intact.
Vulnerable, S_V	0.66 – 0.78	0.22 – 0.38	0.60 – 0.78	The system is functional, but operates under elevated security or reliability risk.
Anomalous, S_A	variable (≈ 0.55–0.85)	spikes > 0.45 (> 3σ events)	0.45 – 0.70 (fluctuating)	Behavioral deviations from expected norms; potential precursor to defect or attack.
Defective, S_B	0.60 – 0.75 (local)	< 0.25	0.55 – 0.75	Logical or functional errors appear without a full crash, incorrect outputs.
Failure, S_F	< 0.55	> 0.45	< 0.45	Critical system disruption, downtime, or data loss requires immediate recovery.
Recovering, S_R	0.58 – 0.72 (transient ↑)	decreasing 0.30 → 0.20	increasing 0.55 → 0.75	System restoring from failure toward normal; self-healing or restart in progress.
Transitional, S_T	0.60 – 0.80 (transient)	0.20 – 0.35	0.55 – 0.70	Short-term unstable configuration during deployment, migration, or CI/CD.
Uncertain, S_U	0.68 – 0.76	0.25 – 0.35	0.60 – 0.70	State of scope misalignment, ambiguous requirements, or resource/timeline mismatch.

To prove the formalization, we currently validate thresholds via dynamic modelling and Monte Carlo simulations of SDLC (**Fig. 1**), comparing predicted and actual state transitions, ensuring consistent separability and stability of states across iterations and empirical test datasets.

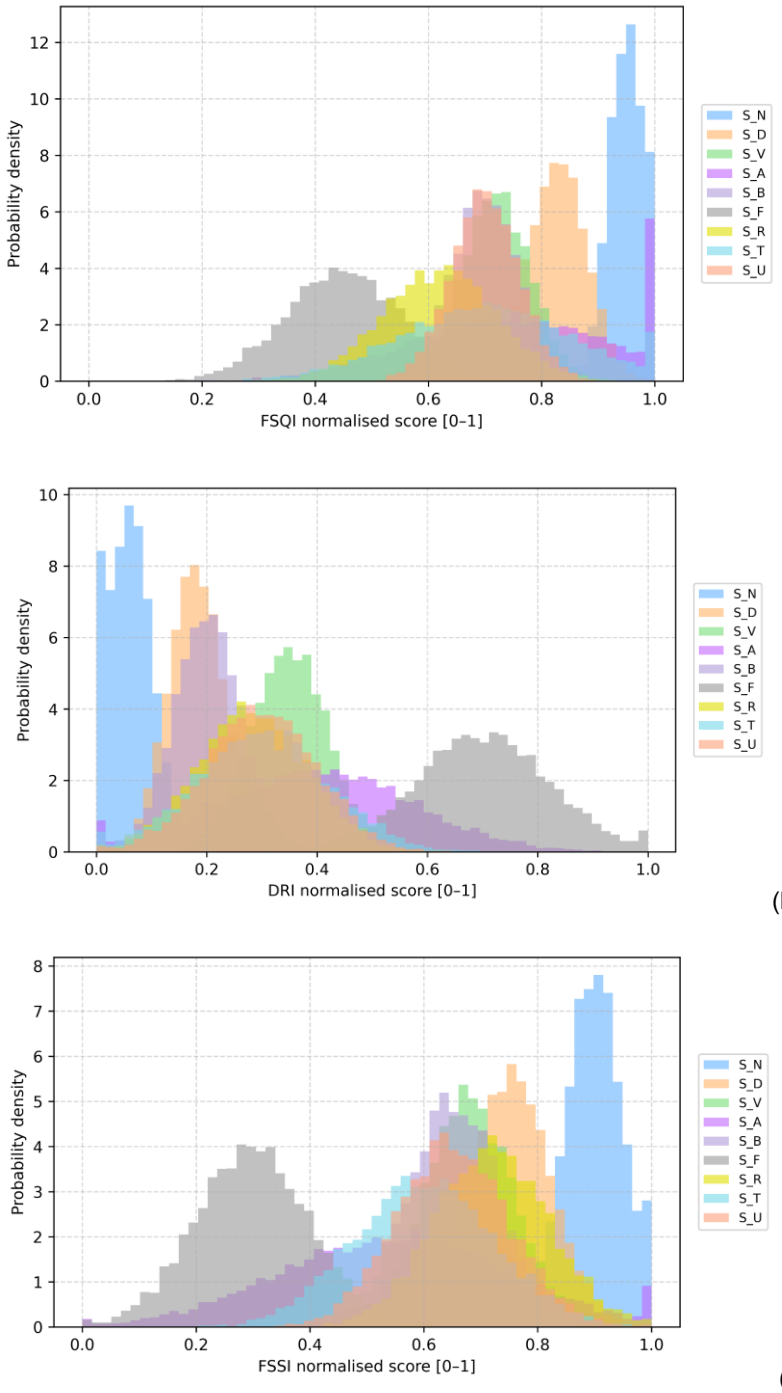

Fig. 1. Results of Monte Carlo simulation for $FSQI$ (a); DRI (b) and $FSSI$ (c).

The generated distributions of $FSQI$, DRI and $FSSI$ across the nine functional software states: S_N (marked as 'S_N' in **Fig. 1**), $S_D, S_V, S_A, S_B, S_F, S_R, S_T, S_U$) demonstrate that the proposed mathematical formalization reliably produces distinct, separable, and meaningful clusters that reflect the expected behavior of software under varying levels of quality, risk, and stability. Each distribution captures the theoretical assumptions of the model while revealing realistic overlaps in transitional or ambiguous operational conditions.

The simulations correctly reproduce the expected real-world fuzziness of these states. It has shown that no single metric is sufficient, but the vectors together produce reliable decision boundaries.

The results achieved provide an effective and applicable baseline for application in SFS analytics, real-time SDLC monitoring, and decision support. Monte Carlo modelling and updated classification thresholds can reliably distinguish SFS through measurable indicators $FSQI$, DRI and $FSSI$ thereby enabling continuous assessment of quality, risk, and stability. These models can be integrated into CI/CD pipelines, DevSecOps control panels, and as digital twin models of SDLC to predict state transitions, detect early performance degradation, and automatically execute recovery strategies.

From the perspective of LLM-based and agentic SDLCs, the concept of SFS takes on additional meaning. In these systems, software behavior is influenced by nondeterministic generative processes, evolving internal representations, and external knowledge sources. Therefore, SFS should not only include execution-level attributes, but also semantic consistency, contextual validity, and consistency between the produced artifacts and the software development lifecycle goals. The proposed SFS model provides a formal basis for representing these dimensions without binding the method to a specific generative technique.

CONCLUSION

This paper proposes a formal and computational framework for modeling the SFS throughout the SDLC, extending classic state-based and quality-oriented approaches to context-aware and probabilistic representations.

The results show that traditional models are insufficient to cope with new development paradigms that involve some or all of the code generation, in which software behavior depends on execution technology and the production environment.

The proposed SFS abstraction enables predictive analysis, interphase thinking, and state-aware orchestration of DevSecOps and agent-based development pipelines, providing a foundation for intelligent SDLC monitoring and decision support.

The next step is to transform the theoretical model into a working platform for the implementation of the proposed model in real-world SDLCs, integrating it with generative processes, and evaluating its effectiveness in large-scale industrial environments.

ACKNOWLEDGMENTS AND FUNDING SOURCES

The authors received no financial support for the research, writing, and publication of this article.

COMPLIANCE WITH ETHICAL STANDARDS

The authors declare that the research was conducted in the absence of any conflict of interest.

AUTHOR CONTRIBUTIONS

Conceptualization, [M.L.]; methodology, [M.L.]; validation, [R.S., V.L.]; formal analysis, [M.L.]; investigation, [M.L.]; resources, [M.L.]; data curation, [M.L.]; writing – original draft preparation, [M.L.]; writing – review and editing, [V.L.]; visualization, [M.L.]; supervision, [R.S., V.L.].

All authors have read and agreed to the published version of the manuscript.

REFERENCES

[1] Lyashkevych, M. Y., Lyashkevych, V. Y., & Shuvar, R. Y. (2025). Security and other risks related to LLM-based software development. Ukrainian Journal of Information Technology, 7(1), 86–96. https://doi.org/10.23939/ujit2025.01.086.

[2] Lyashkevych, M. Y., Rohatskiy, I. Y., Lyashkevych, V. Y., & Shuvar, R. Y. (2024). Software risk taxonomy creation based on the comprehensive development process. Science and Technology: New Horizons of Development 209 – Electronics and Information Technologies, 1(27), 59–71. https://doi.org/10.30970/eli.27.5.

[3] Hossain, Mohammad. (2023). Software Development Life Cycle (SDLC) Methodologies for Information Systems Project Management. International Journal For Multidisciplinary Research. https://doi.org/10.36948/ijfmr.2023.v05i05.6223.

[4] State Enterprise “UkrNDNC.” (2016). DSTU ISO/IEC 25010:2016 – Systems and software engineering – Systems and software quality requirements and evaluation – System and software quality model. Kyiv, Ukraine: SE “UkrNDNC.” (In Ukrainian; translated title.)

[5] State Committee of Ukraine for Technical Regulation and Consumer Policy. (2005). DSTU ISO/IEC 9126-1:2005 – Information technology – Software product quality – Part 1: Quality model. Kyiv, Ukraine: Derzhspozhyvstandart Ukrainy. (In Ukrainian; translated title.)

[6] International Organization for Standardization. (2011). ISO/IEC 25010:2011 – Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Geneva, Switzerland: ISO. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[7] Jamshidi, P., Pahl, C., Lewis, J., & Tilkov, S. (2020). Microservices: The journey so far and challenges ahead. IEEE Software, 38(1), 24–31. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8354433.

[8] R. Yedida and T. Menzies, "How to Improve Deep Learning for Software Analytics (a case study with code smell detection)," 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA, 2022, pp. 156-166, doi: https://doi.org/10.1145/3524842.3528458.

[9] Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Li, Y., Lundberg, S., Nori, H., & others. (2023). Sparks of artificial general intelligence: Early experiments with GPT-4. arXiv preprint arXiv:2303.12712. https://doi.org/10.48550/arXiv.2303.12712.

[10] S. Silva, A. Tuyishime, T. Santilli, P. Pelliccione and L. Iovino, "Quality Metrics in Software Architecture," 2023 IEEE 20th International Conference on Software Architecture (ICSA), L'Aquila, Italy, 2023, pp. 58-69, doi: https://doi.org/10.1109/ICSA56044.2023.00014.

[11] D. Di Pompeo and M. Tucci, "Quality Attributes Optimization of Software Architecture: Research Challenges and Directions," 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), L'Aquila, Italy, 2023, pp. 252-255, doi: https://doi.org/10.1109/ICSA-C57050.2023.00061.

[12] Semenov, S., Tsukur, V., Molokanova, V., Muchacki, M., Litawa, G., Mozhaiev, M., & Petrovska, I. (2025). Mathematical Model of the Software Development Process with Hybrid Management Elements. Applied Sciences, 15(21), 11667. https://doi.org/10.3390/app152111667.
[13] Li, Can & Grossmann, Ignacio. (2021). A Review of Stochastic Programming Methods for Optimization of Process Systems Under Uncertainty. Frontiers in Chemical Engineering. 2. 622241. https://doi.org/10.3389/fceng.2020.622241.
[14] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). Introduction to automata theory, languages, and computation (3rd ed.). Pearson/Addison Wesley.
[15] Lewis, H. R., & Papadimitriou, C. H. (1998). Elements of the theory of computation (2nd ed.). Prentice Hall.
[16] Harel, D. (1987). Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9.
[17] OMG (Object Management Group). (2017). Unified modeling language (UML) specification (Version 2.5.1, OMG Formal Document No. 17-12-01). https://www.omg.org/spec/UML/2.5.1/.
[18] Forsberg, K., Mooz, H., & Cotterman, H. (2005). Visualizing project management: Models and frameworks for mastering complex systems (3rd ed.). Wiley. https://www.wiley.com/en-us/Visualizing+Project+Management%3A+Models+and+Frameworks+for+Mastering+Complex+Systems%2C+3rd+Edition-p-x000260487.
[19] Boehm, B. (1988). A spiral model of software development and enhancement. Computer, 21(5), 61–72. https://doi.org/10.1109/2.59.
[20] Beck, K. (2005). Extreme programming explained: Embrace change (2nd ed.). Addison-Wesley.
[21] State (computer science). (2024). Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/State_(computer_science).
[22] Musa, J. D. (1998). Software reliability engineering: More reliable software, faster and cheaper. McGraw-Hill.
[23] Menzies, T., & Zimmermann, T. (2023). Software analytics in DevOps. IEEE Transactions on Software Engineering, 49(3), 512–530. https://doi.org/10.1109/TSE.2022.3175113.
[24] Kim, G., Debois, P., Willis, J., & Humble, J. (2021). The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations (2nd ed.). IT Revolution Press.
[25] International Organization for Standardization. (2011). ISO/IEC 25010:2011 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. ISO. https://www.iso.org/standard/35746.html.
[26] DSTU ISO/IEC 25010:2025. (2025). Systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — System and software quality model (ISO/IEC 25010:2023, IDT). Kyiv: SE “UkrNDNC”. https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=116491.
[27] Cheng, B. H. C., de Lemos, R., Giese, H., Müller, H., Shaw, M., & Uchitel, S. (Eds.). (2009). Software engineering for self-adaptive systems (Vol. 5525). Springer. https://doi.org/10.1007/978-3-642-02171-8.
[28] Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. Computer, 36(1), 41–50. https://doi.org/10.1109/mc.2003.1160055.
[29] Wooldridge, M. (2002). An introduction to multiagent systems. John Wiley & Sons. https://uranos.ch/research/references/Wooldridge_2001/TLTK.pdf.

[30] Bommasani, R., Hudson, D. A., Adeli, E., Agrawal, P., Ahuja, S., Argyriou, A., ... Liang, P. (2022). On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258. http://arxiv.org/abs/2108.07258.
[31] Tran, Khanh-Tung & Dao, Dung & Nguyen, Minh-Duong & Pham, Viet & O'Sullivan, Barry & Nguyen, Hoang. (2025). Multi-Agent Collaboration Mechanisms: A Survey of LLMs. 10.48550/arXiv.2501.06322. https://doi.org/10.48550/arXiv.2501.06322.
[32] Pomorova, O. Fuzzy system of the evaluation and prediction of overall risks in software development [Text] / O. Pomorova, M. Lyashkevych //Proceedings of the 6th International Conference ACSN-2013. – Lviv: Ukraine Technology, 2013. – Pp.126-129.

ВИЗНАЧЕННЯ ТА ФОРМАЛІЗАЦІЯ КОНЦЕПЦІЇ ФУНКЦІОНАЛЬНОГО СТАНУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПРОТЯГОМ УСЬОГО ЖИТТЄВОГО ЦИКЛУ РОЗРОБКИ

Марія Ляшкевич**, *Василь Ляшкевич*, *Шувар Роман
Кафедра системного проектування
Факультет електроніки та комп'ютерних технологій
Львівський національний університет імені Івана Франка,
вул. Драгоманова 50, 79005, Львів, Україна

АНОТАЦІЯ

Вступ. У сучасному світі програмне забезпечення (ПЗ) є критично важливим компонентом будь-якої інформаційної системи. Його розробка вимагає значних ресурсів і складних технічних рішень, а розвиток технологій настільки швидкий, що не всі поняття та визначення в галузі ПЗ є чітко формалізованими. Це особливо стосується функціонального стану ПЗ (ФСПЗ) упродовж життєвого циклу розробки (SDLC), адже передбачити всі можливі ФСПЗ практично неможливо через динамічність середовищ, зміни вимог, взаємодію компонентів і поведінку учасників проєкту. Це створює виклик для формалізації, аналізу, прогнозу та моніторингу та управління цими станами.

Матеріали та методи. Визначення та формалізації ФСПЗ охоплюють концепції із теорії станів у комп'ютерних науках, моделі якості з міжнародних стандартів ISO/IEC 25010:2011 та ДСТУ ISO/IEC 9126-1:2005. На основі цих джерел реалізуються формалізовані підходи до опису та відстеження змін у ФСПЗ протягом SDLC. Визначені поняття ФСПЗ та ФСПЗ протягом SDLC формалізовані математично, що дозволяє будувати динамічні моделі еволюції станів протягом SDLC на основі стохастичної функції переходів. Для побудови моделей використовуються атрибути якості такі як функціональна відповідність, надійність, вразливість, придатність до тестування та інші у комбінації з подієво-орієнтованими, автоматними та стано-орієнтованими моделями. Також наведено різні типи ФСПЗ та їх зв'язок із SDLC.

Результати. Отримані результати дослідження охоплюють формалізацію ФСПЗ, розробку оціночних метрик та практичні рекомендації для аналітики станів ПЗ на всіх етапах SDLC, що дозволяє забезпечити проактивний контроль якості, надійності, безпеки та відповідності програмних систем.

Висновки. Формалізація поняття функціональних станів ПЗ, включно з їх типами, властивостями та параметрами, дозволила встановити обґрунтований зв'язок із фазами SDLC. Запропоновані метрики та рекомендації сприяють розвитку аналітики станів програмних систем, забезпечуючи як теоретичну цілісність підходу, так і його

практичну застосовність у завданнях моніторингу, аналізу та прогнозування стану ПЗ. Ця методологія створює нову основу для самонавчальних SDLC-орієнтованих екосистем, у яких ФСПЗ прогнозуються, оцінюються та керуються автоматично в реальному часі.

Ключові слова: життєвий цикл програмного забезпечення, функціональний стан програмного забезпечення, функціональна придатність, прогнозування стану програмного забезпечення, аналітика функціонального стану програмного забезпечення, характеристики стану програмного забезпечення

Received / Одержано	Revised / Доопрацьовано	Accepted / Прийнято	Published / Опубліковано
14 November, 2025	01 December, 2026	01 December, 2025	25 December, 2025