DOI: 10.20998/2079-0023.2023.01.16
UDC 004.053:004.428.2:519.688

D. M. NIKITIN, Postgraduate Student of the Department of Software Engineering, Kharkiv National University of Radio Electronics, Kharkiv, Ukraine, e-mail: nikitin27959@gmail.com, ORCID: https://orcid.org/0000-0003-4388-4996

SPECIFICATION FORMALIZATION OF STATE CHARTS FOR COMPLEX SYSTEM MANAGEMENT

This article presents a formalization approach for the requirements of object-oriented programs with state machines, using a spacecraft control system as a case study. It proposes a state pattern implementation, where each state is represented as a class with clearly defined responsibilities, and the transitions between states are controlled by the state objects themselves. Additionally, the application of model checking, theorem proving, and code generation techniques are discussed. The effectiveness of the proposed approach in ensuring compliance with the specified requirements is demonstrated, while also identifying potential drawbacks and limitations of the approach. The implementation is validated using a range of formal verification techniques, including model checking and theorem proving. The article also discusses how the approach can be extended and applied to other complex systems. Overall, the valuable insights into the formalization of requirements for object-oriented programs with state machines are provided, offering a practical and effective approach for verifying the correctness and completeness of such implementations. The results of this work have important implications for the development of safety-critical systems and can potentially improve the quality and reliability of software systems in various domains. By using mathematical models and rigorous formal methods, it is possible to detect and eliminate errors early in the development process, leading to higher confidence in the correctness of the final product. Future research in this area could explore the use of more advanced techniques, such as model-driven development and automatic code synthesis, to further streamline the software development process. Additionally, the development of more efficient and user-friendly tools could make these techniques more accessible to a wider range of developers and organizations. Altogether, the combination of formal methods and software engineering has the potential to revolutionize the way software systems are designed, developed, and verified, leading to safer and more reliable software for critical applications.

Keywords: formal methods, automated programming, state machines, model checking, theorem proving, code generation, object-oriented programming, spacecraft control, requirements formalization, verification and validation.

Introduction. Formalization of requirements for automated object-oriented programs involves the process of translating natural language requirements into a precise and unambiguous specification that can be used to guide the development of software systems. Object-oriented programming (OOP) is a popular approach to software development that emphasizes modular design, code reuse, and encapsulation of data and functionality within objects.

To formalize requirements for OOP programs, developers use a combination of textual descriptions, graphical models, and formal languages such as Unified Modeling Language (UML) or Object Constraint Language (OCL). UML provides a standard notation for modeling software systems, including class diagrams, sequence diagrams, and state machine diagrams, which can be used to visualize the structure and behavior of software components. OCL is a formal language for specifying constraints and operations on objects in an OOP system.

One key benefit of formalizing requirements for OOP programs is that it helps to minimize ambiguity and inconsistency in the software development process. By using a formal language to express requirements, developers can identify potential issues or conflicts early in the development cycle and ensure that the resulting software system meets the desired functional and non-functional requirements. Formalization of requirements also facilitates collaboration among developers, stakeholders, and end-users by providing a common language for discussing and refining requirements.

However, formalization of requirements for OOP programs can also be a challenging and time-consuming process. It requires a deep understanding of both the application domain and the OOP paradigm, as well as expertise in modeling and formal languages. Additionally, there is a risk of over-specifying requirements, which can lead to inflexibility and difficulties in adapting to changing user needs or system requirements. Thus, it is important to strike a balance between formalization and flexibility, and to involve all relevant stakeholders in the requirements engineering process.

Specification Formalization of State Charts. Specification formalization of state charts involves defining the behavior of a system using a graphical notation that represents states, transitions, and actions in a structured and systematic way [1]. State Charts can be used to model complex systems and provide a clear and concise way to specify the behavior of a system [2]. Such formalization involves creating a precise and unambiguous specification that can be used to verify the correctness of the system.

There are several formal methods that can be used to specify and analyze state charts, including model checking, theorem proving, and code generation. These methods can help to detect errors in the system design and ensure that the system meets its requirements.

Formalization of state charts is particularly important for safety-critical systems, where errors in the system design could have serious consequences. In these systems, formal methods can be used to verify that the system meets safety requirements and that it behaves correctly under all possible conditions.

An automated state machine can be used to control the behavior of a vehicle during its mission. The state machine can be defined using a set of mathematical formulas that describe the transition between states and the actions to be taken in each state [3].

For example, let us consider a spacecraft that is designed to perform a series of maneuvers, including attitude control, trajectory correction, and payload deployment. The state machine for this spacecraft could be defined using a set of differential equations that govern the spacecraft's motion and the forces acting on it.

The state variables in this case could include the spacecraft's position, velocity, and attitude, as well as the state of its propulsion and control systems. The state machine could be defined using a set of mathematical equations that specify the transition between states and the actions to be taken in each state.

For instance, the state machine could transition from the attitude control state to the payload deployment state when certain conditions are met, such as reaching a certain altitude or orientation. The mathematical formula for this transition could involve calculating the spacecraft's position and velocity relative to the payload and adjusting the attitude and thruster firing accordingly.

Let's present a spacecraft that needs to perform an attitude control maneuver to align its sensors with a target object in space. The spacecraft's attitude can be described by its orientation relative to a reference frame, such as the Earth-centered inertial (ECI) frame.

The spacecraft's attitude is described by a quaternion

$$q = [q_0, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3], \tag{1}$$

where q_0 represents the scalar component and $\boldsymbol{q}_1$, $\boldsymbol{q}_2$ and $\boldsymbol{q}_3$ represent the vector component of the quaternion. The goal of the attitude control maneuver is to adjust the quaternion to a desired value that corresponds to the desired orientation of the spacecraft. The equations could be represented using matrix algebra, as follows:

$$\boldsymbol{I}\dot{\mathrm{w}} + \boldsymbol{WIw} = \boldsymbol{BU}, \tag{2}$$

where $\boldsymbol{I}$ – spacecraft's inertia matrix of the spacecraft, which is a 3×3 matrix representing the distribution of mass in the spacecraft about its center of mass;

$\boldsymbol{w}$ – spacecraft's angular velocity vector;

$\boldsymbol{W}$ – skew-symmetric matrix of $\boldsymbol{w}$;

$\boldsymbol{B}$ – spacecraft's control torque matrix, which represents the external torque applied to the spacecraft by the control system. It is a 3×3 matrix that is determined by the control law used to adjust the spacecraft's attitude;

$\boldsymbol{U}$ – control input vector, which represents the control commands issued by the spacecraft's control system. It is a 3-dimensional vector that is determined by the control law.

The dot notation denotes the time derivative of a variable, representing its rate of change over time.

$\boldsymbol{W}$ is a matrix representation of the cross product of $\boldsymbol{w}$ with itself, defined as follows:

$$\boldsymbol{W} = \begin{pmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{pmatrix}. \tag{3}$$

The equation (1) states that the rate of change of the spacecraft's angular momentum (***Iw***) is equal to the external torque applied to it (***BU***), with the skew-symmetric matrix of ***w*** (i.e., ***WIw***) representing the Coriolis and centrifugal forces acting on the spacecraft. The equation is a second-order ordinary differential equation and can be solved numerically to obtain the angular velocity vector ***w*** as a function of time.

The external torque is generated by the spacecraft's control system, which adjusts the angular velocity of the spacecraft in response to the error between the desired and actual quaternion values.

The control torque matrix ***B*** and the control input vector ***U*** can be derived using a control law that minimizes the error between the desired and actual quaternion values. This control law could be represented using a formula such as:

$$\begin{cases} \boldsymbol{B} = -\boldsymbol{k}_p \boldsymbol{Q} - \boldsymbol{k}_d \boldsymbol{W}, \\ \boldsymbol{U} = \boldsymbol{k}_p q + \boldsymbol{k}_d \boldsymbol{w}, \end{cases} \tag{4}$$

where q is the error quaternion between the desired and actual orientations, $\boldsymbol{Q}$ is the skew-symmetric matrix of q, and $\boldsymbol{k}_p$ and $\boldsymbol{k}_d$ are the proportional and derivative gain matrices, respectively.

The control law uses the error quaternion and the angular velocity of the spacecraft to calculate the control torque and input vectors that adjust the spacecraft's attitude. The proportional and derivative gains, $\boldsymbol{k}_p$ and $\boldsymbol{k}_d$ are tuning parameters that determine the response of the control system to changes in the error and velocity.

These equations form the basis of a closed-loop control system that adjusts the spacecraft's attitude to the desired orientation. The state machine can transition to the next state once the desired orientation is achieved, such as when the spacecraft's sensors are aligned with the target object in space.

Here is an example Python code listing that implements the formulas described earlier:

```
    import numpy as np
    def spacecraft_dynamics(I, w, B, U):
      """
      Computes the derivative of angular
velocity vector w
      Args:
        I: 3x3 inertia matrix
        w: 3-dim angular velocity vector
        B: 3x3 control torque matrix
        U: 3-dim control input vector
      Returns:
        3-dimensional array representing the
time derivative of the angular velocity
vector w
      """
      w_dot = np.linalg.inv(I).dot(-
np.cross(w, I.dot(w), axisa=0, axisb=0) +
B.dot(U))
      return w_dot
```

This code defines a function called spacecraft_dynamics that takes as input the spacecraft's inertia matrix ***I***, angular velocity vector ***w***, control torque matrix ***B***, and control input vector ***U***, and computes the time derivative of ***w*** using the spacecraft dynamics equation. The numpy module is used to perform the necessary matrix operations, such as matrix inversion and cross products.

State Charts in OOP Domain. In the "State" object-oriented design pattern, each component of the model would have its own state object that encapsulates the behavior and data specific to that state [4]. Here's how the state objects could be defined and described for each component of the spacecraft dynamics model:

1. Angular Velocity State: represents the current angular velocity of the spacecraft. It has a single variable, w, that stores the current angular velocity vector. The state object provides methods to update the angular velocity vector and to compute its time derivative using the spacecraft dynamics equation.
2. Inertia State: represents the current inertia matrix of the spacecraft. It has a single variable, I, that stores the current inertia matrix. The state object provides methods to update the inertia matrix.
3. Control Torque State: represents the current control torque applied to the spacecraft. It has a single variable, B, which stores the current control torque matrix. The state object provides methods to update the control torque matrix.
4. Control Input State: represents the current control input commands issued by the spacecraft's control system. It has a single variable, U, which stores the current control input vector. The state object provides methods to update the control input vector.

Each state object has its own set of methods that allow it to interact with other states and components in the spacecraft dynamics model. For example, the Angular Velocity State might have a method that computes the Coriolis and centrifugal forces acting on the spacecraft, given the current inertia matrix, and control torque and input states. Similarly, the Control Input State might have a method that generates control input commands based on the current spacecraft state.

Using the "State" design pattern can help to modularize the spacecraft dynamics model, making it easier to modify and extend in the future. By encapsulating the behavior and data specific to each state in its own object, the overall complexity of the model can be reduced, and its overall structure made more maintainable [5].

In the provided spacecraft example, the states can be replaced with each other through a process of state transitions. This means that as the spacecraft system runs, each state object can update its own internal state and then transition to a new state object, which will take over control of the system.

The process of state transition can be controlled by the spacecraft control software, which can determine when a state object should transition to a new state based on certain conditions. For example, the control software may trigger a state transition when a certain time has elapsed, when a certain event occurs, or when certain sensor readings meet certain thresholds [6].

To implement state transitions in the spacecraft control system, each state class should implement an update() method that updates its own internal state, and then returns a new instance of a state class that represents the next state of the system. The control software can then update the current state object with the new state object, allowing the system to transition to the new state.

For example, the AngularVelocityState class might implement an update() method that reads sensor data to calculate the current angular velocity of the spacecraft, and then returns a new instance of a state class that represents the next state of the system based on that velocity. This new state object might be an instance of the InertiaState class, which would update the system's internal state based on the current inertia of the spacecraft.

Overall, by implementing state transitions in this way, the spacecraft control software can dynamically switch between different state objects to control the spacecraft system in a safe and reliable manner.

Formal Methods of State Chart Analysis. To use model checking, theorem proving, and code generation with the spacecraft example, we can first start by creating a formal specification of the system using a modeling language such as Statecharts or Mermaid. This formal specification will represent the desired behavior of the system, including the states and transitions between them, as well as any constraints or requirements that must be satisfied [7].

Once the formal specification is created, we can use model checking and theorem proving techniques to verify that the specification is correct and satisfies the desired requirements. Model checking involves automatically verifying that a model of a system satisfies a given set of properties [8]. Theorem proving, on the other hand, involves manually proving that a model satisfies a set of logical properties using formal logic and mathematical reasoning [9].

Finally, once we have verified that the formal specification is correct, we can use code generation techniques to automatically generate code that implements the desired behavior of the system. This code can be written in a programming language such as C or Python and can be used to control the behavior of the spacecraft in accordance with the formal specification as illustrated in fig. 1.

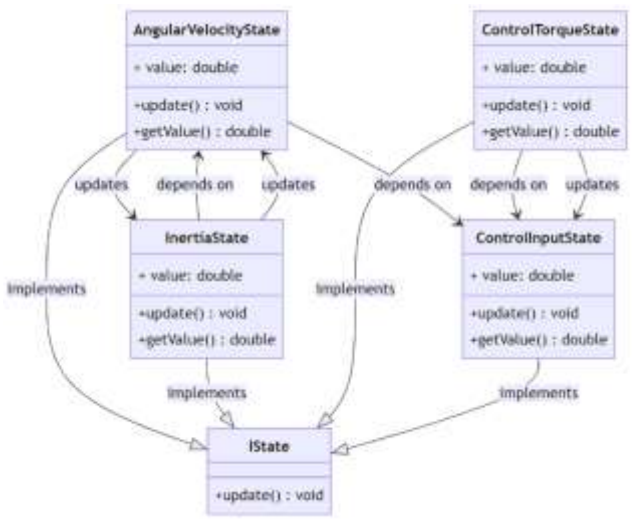

Fig. 1. State chart implementation for the spacecraft

In fig. 1, each state class (AngularVelocityState, InertiaState, ControlTorqueState, and ControlInputState) contains a value property, update() and getValue() methods. The update() method changes the state of the object, while the getValue() method retrieves the current value of the object's state.

The IState interface contains only an update() method, which is implemented by each of the state classes.

This diagram represents the relationships between the four state classes and the IState interface that they all implement. It also shows the dependencies between the states (for example, AngularVelocityState depends on ControlInputState) and the state updates (for example, InertiaState updates AngularVelocityState).

Overall, the use of model checking, theorem proving, and code generation can help ensure the correctness and reliability of the spacecraft's control system and can help mitigate the risks associated with space missions [10].

First, model checking can be used to verify that the state transitions in the spacecraft control software are correct and satisfy the system requirements. A model of the spacecraft control software can be created, and the model checker can explore all system states to check that the software behaves as intended under all possible conditions.

Second, theorem proving can be used to formally prove that the state transitions in the spacecraft control software are correct and satisfy the system requirements. A formal specification of the spacecraft control software can be created, and the theorem prover can use mathematical logic to prove that the software satisfies the specified requirements.

Finally, code generation can be used to automatically generate executable code from the formal specification of the spacecraft control software. The formal specification can serve as a precise and unambiguous description of the software behavior, and the code generator can automatically generate code that faithfully implements the specified behavior.

By applying these formal methods to the spacecraft control software, we can ensure that the software is correct and reliable, and that it satisfies the system requirements. This can help to reduce the risk of errors and malfunctions in the software.

Model Checking. To verify that the spacecraft control software behaves correctly under all conditions, we can model the software using a state machine and use a model checker to explore all possible system states and check that the software behaves as intended [11]. Here is an example of a state machine model in Promela, a modeling language for the Spin model checker:

```
    mtype     =     {IDLE,     ACCELERATING,
DECELERATING};
    mtype state = IDLE;
    active proctype spacecraft() {
      do
      :: state == IDLE ->
           state = ACCELERATING;
      :: state == ACCELERATING ->
           state = DECELERATING;
      :: state == DECELERATING ->
           state = IDLE;
      od
    }
```

In this model, the spacecraft can be in one of three states: IDLE, ACCELERATING, or DECELERATING. The spacecraft transitions between states based on certain conditions, and the model checker can explore all system states to check that the software behaves correctly under all conditions.

Theorem Proving. If we want to prove that the spacecraft operates as expected we can use a theorem prover to create a formal specification of the software, and then use mathematical logic to prove that the software satisfies the requirements [12]. Here is an example of a specification of the spacecraft software in Z notation:

```
    state   ::=   IDLE   |   ACCELERATING   |
DECELERATING
    SPC ::= [state: state]
    InitSPC == state = IDLE
    AccelerateSPC == state = ACCELERATING
    DecelerateSPC == state = DECELERATING
    NextSPC     ==     AccelerateSPC     \/
DecelerateSPC \/ (state = IDLE /\ state' /=
IDLE)
```

This specification defines the initial state, the state transitions, and the constraints on the possible state transitions. We can use a theorem prover to prove that the specification is correct, and that the software satisfies the specified requirements.

Code Generation. When there is a formal specification of the spacecraft control software, and we need to automatically generate executable code that implements the specified behavior, we can use a code generator to automatically generate code from the formal specification [13]. Here is an example of how we could generate C# code from the Z specification above using the Zing code generator:

```
    public class SPC {
      private    enum    state    {    IDLE,
ACCELERATING, DECELERATING };
      private state _state = state.IDLE;
      public void AccelerateSPC() {
        _state = state.ACCELERATING;
      }
      public void DecelerateSPC() {
        _state = state.DECELERATING;
      }
      public void NextSPC() {
        if (_state == state.IDLE) {
          _state = state.ACCELERATING;
        }    else    if    (_state    ==
state.ACCELERATING) {
          _state = state.DECELERATING;
        }    else    if    (_state    ==
state.DECELERATING) {
          _state = state.IDLE;
        }
      }
    }
```

Contract Compliance. To ensure compliance with the contract of the states in the spacecraft example, we can use a combination of static code analysis tools and automated testing.

Static code analysis tools can help detect violations of coding standards and best practices, as well as potential errors and vulnerabilities in the code. These tools can analyze the code and flag any violations of the interface contract, such as missing or incorrect method signatures or non-compliant access modifiers.

Automated testing can help ensure that the code adheres to the interface contract by verifying that each state class behaves as expected [14]. Unit tests can be written to verify that the update() and getValue() methods of each state class perform their expected functions, and that the state transitions between the classes are correct. Integration tests can also be written to test the system as a whole, and to verify that the interactions between the state classes are correct.

By using both static code analysis tools and automated testing approach, we can ensure that the code adheres to the contract of the states and prevent bugs and errors in the system.

State Chart Implementation Concerns. The state pattern provides a way to encapsulate state-specific behavior in separate classes and allows for the object's behavior to change dynamically as its state changes [15]. While the state pattern can be a useful tool for designing complex software systems, there are potential drawbacks to its implementation:

1. Increased memory usage: as each state object maintains its own internal state and any associated data, this can lead to increased memory usage, which may be a concern in systems with limited memory resources;

2. Complex object interactions: as the state objects interact with each other to transition between states, the code can become more complex and difficult to follow. This may make it harder to debug and maintain the code over time;

3. Potential for errors in state transition logic: the transition logic between states is implemented in each state object's update method, which may lead to errors if not implemented correctly. For example, if the state transition conditions are not properly defined, the system may get stuck in a certain state or transition to the wrong state.

4. Increased development time: implementing the state pattern can be more time-consuming than other approaches to managing state, such as using a switch statement or if–else blocks, which may lead to longer development times and increased costs.

5. Potential for performance issues: as the state objects update and transition between states at run time, this may introduce performance overhead and impact the system's overall performance.

Conclusions. The use of formalization techniques such as model checking, theorem proving, and code generation can greatly enhance the reliability and safety of automated systems such as spacecraft control systems. By formalizing the system requirements and specifications, potential errors and bugs can be caught early in the development process, reducing the risk of catastrophic failures.

In the context of the discussed spacecraft example, it was demonstrated how the use of state machines and the state design pattern can provide a structured approach to modeling and implementing complex control systems.

The use of formal methods for developing reliable and correct software systems was explored, specifically in the context of state machines.

The results present a detailed example of using the state pattern to model the behavior of a spacecraft and how formal verification techniques can help ensure the correctness and completeness of the system design. The article demonstrates how model checking can be used to detect potential errors and violations of the system requirements and how theorem proving can be used to formally verify the accuracy of the system's behavior.

Despite the benefits of using formal methods for developing state machine-based systems, there are also some limitations and challenges that need to be addressed. Some of the drawbacks of using this approach, such as the complexity of the mathematical models and the high computational costs of verification techniques, have been discussed.

Overall, the use of formal methods is a promising approach for developing reliable and correct software systems based on state machines. By ensuring the correctness and robustness of state machine-based systems, we can increase their security and overall quality, which is especially important for safety-critical systems such as those used in aerospace and transportation.

References

1. Lodi S., Mesiti M., Orsi G. A state machine for relational databases. *In Proceedings of the «34th IEEE/ACM International Conference on Automated Software Engineering»*. 2019. P. 114–125.
2. Liggesmeyer P., Seib E., Prehofer C. A state machine approach for modeling and testing autonomous driving functions. *In Proceedings of the «2021 IEEE Intelligent Vehicles Symposium».* 2021. P. 2854–2859.
3. Giannakopoulou D., Pasareanu C., Rungta N. An integrated approach to analyzing and testing stateful systems. *In Proceedings of the «32nd IEEE/ACM International Conference on Automated Software Engineering».* 2017. P. 943–948.
4. Saenz J. C., Perez–Palacin D., d'Amorim M. Behavior–driven development of stateful systems: a case study of a medical information system. *In Proceedings of the «2019 IEEE/ACM International Conference on Automated Software Engineering».* 2019. P. 1011–1016.
5. Azevedo G., Ribeiro M., Medeiros F. Model–based test generation for stateful systems using an FSM language. *In Proceedings of the «2018 IEEE International Conference on Software Testing, Verification and Validation».* 2018. P. 237–247.
6. Daumke P., Laroche L., Graubner S. A state machine–based approach for the dynamic adaptation of software systems. *In Proceedings of the «14th International Conference on Software Technologies».* 2019. P. 466–475.
7. Lo D., Liu Y., Xie X., Wong S. Symbolic execution of stateful programs with abstract state machines. *In Proceedings of the «28th ACM SIGSOFT International Symposium on Software Testing and Analysis».* 2019. P. 285–296.
8. Li Y., Li Y., Dong J. S. Formally verifying the state machine–based software through the UPPAAL model checker. *Journal of Intelligent & Fuzzy Systems 36(4)*. 2019. P. 3657–3668.
9. Chen Q., Liu S., Wang S., Sun J. Efficient generation of state machine models from Java source code for vulnerability detection. *Journal of Systems and Software, 177*. 2021. 237 p.
10. Jovanovic J., Rackovic M., Milicic M. Analysis of the role of state machine diagrams in software development: An exploratory study. *Information and Software Technology, 103*. 2019. 132–146 p.
11. Chen X., Chen J., Wang J. Research on Formalization Method of State Transition Rules for Automated Vehicle Systems. *IEEE Access, 8*. 2020. P. 134116–134127.

12. Eltoweissy M., Alnemr R., Seliem M., Ali M. Formal specification and verification of state–based software systems: A systematic literature review. *Journal of Systems and Software, 163*. 2021. P. 1105–1128.
13. Gaber A., Elragal A. Formalizing requirements for automated driving systems: A systematic literature review. *Safety Science, 142*. 2021. P. 538–549.
14. Gu C., Li X., Liu Y. A Formal Method for Analyzing and Validating the Functionality of Statecharts. *IEEE Access, 7*. 2021. P. 135–154.
15. Taha I., Ahmed E., Al–Mamory S., Karama S. Formalizing Requirements for State Machine Models of Safety–Critical Systems: A Review. *IEEE Access, 9*. 2021. P. 315–333.

References (transliterated)

1. Lodi S., Mesiti M., Orsi G. A state machine for relational databases. *In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 2019. P. 114–125.
2. Liggesmeyer P., Seib E., Prehofer C. A state machine approach for modeling and testing autonomous driving functions. *In Proceedings of the 2021 IEEE Intelligent Vehicles Symposium.* 2021. P. 2854–2859.
3. Giannakopoulou D., Pasareanu C., Rungta N. An integrated approach to analyzing and testing stateful systems. *In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering.* 2017. P. 943–948.
4. Saenz J. C., Perez–Palacin D., d'Amorim M. Behavior–driven development of stateful systems: a case study of a medical information system. *In Proceedings of the 2019 IEEE/ACM International Conference on Automated Software Engineering*. 2019. P. 1011–1016.
5. Azevedo G., Ribeiro M., Medeiros F. Model–based test generation for stateful systems using an FSM language. *In Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation.* 2018. P. 237–247.
6. Daumke P., Laroche L., Graubner S. A state machine–based approach for the dynamic adaptation of software systems. *In Proceedings of the 14th International Conference on Software Technologies.* 2019. P. 466–475.
7. Lo D., Liu Y., Xie X., Wong S. Symbolic execution of stateful programs with abstract state machines. *In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 2019. P. 285–296.
8. Li Y., Li Y., Dong J. S. Formally verifying the state machine–based software through the UPPAAL model checker. *Journal of Intelligent & Fuzzy Systems 36(4)*. 2019. P. 3657–3668.
9. Chen Q., Liu S., Wang S., Sun J. Efficient generation of state machine models from Java source code for vulnerability detection. *Journal of Systems and Software, 177*. 2021. 237 p.
10. Jovanovic J., Rackovic M., Milicic M. Analysis of the role of state machine diagrams in software development: An exploratory study. *Information and Software Technology, 103*. 2019. 132–146 p.
11. Chen X., Chen J., Wang J. Research on Formalization Method of State Transition Rules for Automated Vehicle Systems. *IEEE Access, 8*. 2020. P. 134116–134127.
12. Eltoweissy M., Alnemr R., Seliem M., Ali M. Formal specification and verification of state–based software systems: A systematic literature review. *Journal of Systems and Software, 163*. 2021. P. 1105–1128.
13. Gaber A., Elragal A. Formalizing requirements for automated driving systems: A systematic literature review. *Safety Science, 142.* 2021. P. 538–549.
14. Gu C., Li X., Liu Y. A Formal Method for Analyzing and Validating the Functionality of Statecharts. *IEEE Access, 7*. 2021. P. 135–154.
15. Taha I., Ahmed E., Al–Mamory S., Karama S. Formalizing Requirements for State Machine Models of Safety–Critical Systems: A Review. *IEEE Access, 9*. 2021. P. 315–333.

Received 12.05.2023

УДК 004.053:004.428.2:519.688

Д. М. НІКІТІН, аспірант кафедри програмної інженерії, Харківський національний університет радіоелектроніки м. Харків, Україна, e-mail: nikitin27959@gmail.com, ORCID: https://orcid.org/0000-0003-4388-4996

ФОРМАЛІЗАЦІЯ СПЕЦИФІКАЦІЇ СХЕМ СТАНУ ДЛЯ УПРАВЛІННЯ СКЛАДНИМИ СИСТЕМАМИ

У статті представлено підхід формалізації для вимог об’єктно-орієнтованих програм із кінцевими автоматами з використанням як прикладу системи керування космічним апаратом. Запропоновано реалізацію шаблону стану, де кожен стан представлено як клас із чітко визначеними обов’язками, а переходи між станами контролюються самими об’єктами стану. Крім того, обговорюється застосування методів перевірки моделі, доведення теорем і генерації коду. Продемонстровано ефективність запропонованого підходу щодо забезпечення відповідності зазначеним вимогам, а також виявлено потенційні недоліки та обмеження підходу. Реалізація перевіряється за допомогою низки формальних методів перевірки, включаючи перевірку моделі та доведення теорем. У статті також обговорюється, як цей підхід можна розширити та застосувати до інших складних систем. Загалом, надано детальну інформацію щодо формалізації вимог до об’єктно–орієнтованих програм із кінцевими автоматами, що пропонує практичний та ефективний підхід для перевірки правильності та повноти таких реалізацій. Результати цієї роботи мають важливе значення для розробки критично важливих для безпеки систем і потенційно можуть підвищити якість і надійність програмних систем у різних областях. За допомогою математичних моделей і строгих формальних методів можна виявити й усунути помилки на ранніх стадіях процесу розробки, що веде до більшої впевненості в правильності кінцевого продукту. Майбутні дослідження в цій галузі можуть вивчити використання більш передових методів, таких як розробка на основі моделі та автоматичний синтез коду, для подальшої оптимізації процесу розробки програмного забезпечення. Крім того, розробка більш ефективних і зручних інструментів може зробити ці методи більш доступними для широкого кола розробників і організацій. Загалом, поєднання формальних методів і розробки програмного забезпечення має потенціал революціонізувати спосіб проектування, розробки та перевірки систем програмного забезпечення, створюючи безпечніше та надійніше програмне забезпечення для критичних програм.

Ключові слова: формальні методи, автоматизоване програмування, кінцеві автомати, перевірка моделі, доведення теорем, генерація коду, об’єктно–орієнтоване програмування, управління космічним кораблем, формалізація вимог, верифікація та валідація.

Повне ім’я автора / Author's full name

Нікітін Дмитро Михайлович, Nikitin Dmytro Mykhailovych