

**A G H**

**AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE**  
**Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej**

## Praca dyplomowa

*System wspierający budowę i formalną weryfikację modeli  
inżynierii wymagań metodami logicznymi*  
*System supporting the construction and formal verification  
of requirements engineering models*

Autor:

inż. Szymon Lepianka

Kierunek studiów:

Informatyka i Systemy Inteligentne

Opiekun pracy:

dr hab. inż. Radosław Klimek prof. uczelni

Kraków, 2023



# **Spis treści**

<b>1. Wprowadzenie .....</b>	9
1.1. Cele i założenia pracy .....	10
1.2. Znaczenie logiki w inżynierii oprogramowania .....	11
1.2.1. Rola logiki i algebry w inżynierii oprogramowania .....	11
1.2.2. Weryfikacja modelowa, a wnioskowanie dedukcyjne.....	14
1.2.3. Logika w historii i filozofii inżynierii oprogramowania .....	15
1.2.4. Rozwój inżynierii oprogramowania.....	15
1.2.5. Zastosowanie logiki w analizie wymagań systemowych.....	16
1.3. Struktura pracy.....	17
<b>2. Przegląd istniejących rozwiązań .....</b>	19
2.1. Zintegrowane środowiska programistyczne.....	19
2.1.1. Warsztaty na temat Formal IDE.....	19
2.1.2. Narzędzie AutoFOCUS .....	24
2.2. Wzorce przepływu pracy .....	29
2.2.1. Właściwości wzorców przepływu pracy.....	33
2.2.2. Wyrażenie wzorcowe .....	34
2.2.3. Generacja specyfikacji logicznej.....	36
2.3. Modelowanie diagramu przypadków użycia .....	36
2.4. Systemy dowodzenia twierdzeń.....	41
2.4.1. Dowodzenie formuł logicznych.....	41
2.4.2. Charakterystyka Prover9.....	43
2.4.3. Charakterystyka SPASS Prover .....	45
2.4.4. Charakterystyka InKreSAT .....	48
<b>3. Opis rozwiązania i przyjętej metodologii .....</b>	51
3.1. Opis systemu.....	51
3.2. Funkcjonalności systemu.....	53

3.3. Metodologia rozwiązania.....	55
<b>4. Prezentacja systemu.....</b>	<b>59</b>
4.1. Graficzny interfejs użytkownika .....	59
4.2. Schemat działania systemu .....	60
4.3. Diagramy przypadków użycia .....	61
4.4. Scenariusze przypadków użycia .....	63
4.5. Modelowanie diagramu aktywności .....	65
4.6. Relacje pomiędzy przypadkami użycia .....	69
4.6.1. Relacja zawierania .....	70
4.6.2. Relacja generalizacji .....	72
4.6.3. Relacja rozszerzania.....	75
4.7. Wyrażenie wzorcowe i specyfikacja logiczna.....	79
4.8. Generacja kodu źródłowego .....	81
4.9. Definiowanie dodatkowych wymagań.....	83
4.10. Proces weryfikacji.....	84
4.11. Przykłady zastosowania systemu .....	89
4.11.1. System obsługi pasażerów na lotnisku .....	89
4.11.2. System kontroli wersji kodu .....	113
4.11.3. Wnioski .....	141
<b>5. Implementacja systemu i stos technologiczny .....</b>	<b>143</b>
5.1. Encje w systemie .....	143
5.2. Komponenty systemu .....	145
5.3. Architektura systemu.....	147
5.4. Algorytm generowania specyfikacji logicznej.....	149
5.4.1. Etykietowanie wyrażenia wzorcowego.....	152
5.4.2. Obliczanie skonsolidowanego wyrażenia .....	153
5.5. Reguły logiczne dla wzorców.....	155
5.6. Integracja z systemami dowodzenia twierdzeń.....	157
5.6.1. Konteneryzacja.....	158
5.6.2. Rozwiązane problemy z instalacją.....	160
5.7. Generacja kodu .....	162
5.8. Wykorzystywane technologie .....	167
5.8.1. Konteneryzacja.....	167

5.8.2. Kompilacja języka wzorców do kodu źródłowego .....	168
5.8.3. Edytor diagramów aktywności .....	168
5.8.4. Interfejs użytkownika.....	169
5.8.5. Zapis pracy w systemie .....	169
5.8.6. Język programowania .....	169
5.8.7. Tworzenie logów .....	170
<b>6. Podsumowanie.....</b>	<b>171</b>
6.1. Metryka projektu .....	172
6.2. Możliwości rozbudowy aplikacji.....	172
<b>A. Dokumentacja techniczna systemu dla użytkownika .....</b>	<b>179</b>



# 1. Wprowadzenie

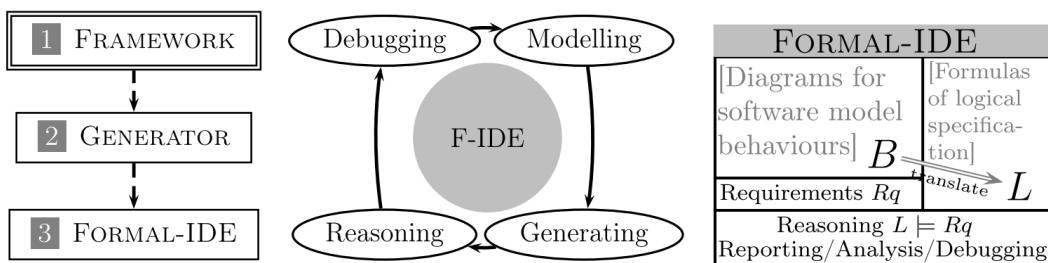
Faza inżynierii wymagań ma fundamentalne znaczenie dla poprawności wytwarzanego oprogramowania. Poprawne zrozumienie potrzeb użytkowników oraz właściwe zdefiniowanie i specyfikowanie wymagań są niezbędne dla sukcesu projektu. Jednakże, niezależnie od staranności w procesie zbierania i dokumentowania wymagań, wciąż istnieje ryzyko popełnienia błędów i niejednoznaczności, które mogą prowadzić do powstania wadliwego oprogramowania.

Współczesne projekty informatyczne coraz częściej wymagają formalnej specyfikacji i weryfikacji wymagań, aby zapewnić jakość, niezawodność i zgodność systemu z oczekiwaniami użytkowników. Wymagania stanowią podstawę do projektowania i implementacji systemu, dla tego ich dokładne i jednoznaczne określenie jest kluczowym aspektem sukcesu projektu. Jednakże, wraz ze wzrostem złożoności projektów, pojawiają się trudności w precyzyjnym i kompletnym określaniu wymagań.

Istotną przeszkodą dla inżynierów tworzących oprogramowanie jest konieczność ręcznego budowania logicznych specyfikacji zachowania modelu oprogramowania. Podczas tego procesu łatwo jest popełnić błędy lub wprowadzić niejednoznaczności. Dlatego istnieje potrzeba automatyzacji tego procesu, aby zapewnić większą precyzję i skuteczność.

W odpowiedzi na te wyzwania, w ramach artykułu [1] zaproponowano rozwiązanie mające na celu wspomaganie procesu tworzenia modeli inżynierii wymagań oraz ich formalnej weryfikacji. Autor skoncentrował się na logice temporalnej oraz wykorzystaniu przepływów pracy (ang. *workflows*) i wzorców (ang. *patterns*) do modelowania zachowania systemów. Został zaproponowany ujednolicony framework i algorytm do automatycznego generowania specyfikacji logicznych dla modeli behawioralnych opartych na przepływach pracy i wzorcach.

Specyfikacja logiczna odgrywa kluczową rolę w procesie tworzenia modeli inżynierii wymagań. Może służyć do przeprowadzania formalnej weryfikacji i analizy, a także dostarczać wskazówek dotyczących rozwiązywania ewentualnych problemów. W artykule [1] zaproponowano narzędzie o nazwie *Formal Specification IDE*, które integruje proces tworzenia modeli, generowania specyfikacji logicznych i weryfikacji formalnej. Narzędzie to zapewnia interfejs użytkownika i menu umożliwiające dostęp do wszystkich funkcji.



**Rys. 1.1. Po lewej:** trzy kolejne fazy powstawania projektu. **Pośrodku:** ilustracja etapów procesu powstawania projektu; punktem wejściowym jest modelowanie. **Po prawej:** zaplanowany wygląd okna środowiska ( $B$  - model zachowań systemu,  $L$  - tłumaczenie modelu na język logiki,  $Rq$  - utworzona formuła logiczna, która ma być poddana testom spełnialności). Źródło: [1]

Celem systemu *Formal Specification IDE* jest generowanie specyfikacji logicznych na podstawie zdefiniowanych i uzupełnionych scenariuszy przypadków użycia oraz diagramów aktywności. Wykorzystanie formalnej logiki do specyfikacji i weryfikacji wymagań pozwala na bardziej precyzyjne i jednoznaczne określenie oczekiwanią dotyczących systemu oraz wykrywanie potencjalnych problemów i sprzeczności już na etapie projektowania.

Metoda zaproponowana w artykule polega na wprowadzeniu wzorców jako prymitywów do modelowania logicznego, co umożliwia skalowanie rozwiązania do rzeczywistych problemów i wykorzystanie kompozycyjności. Zdefiniowane wzorce mogą być szeroko stosowane przez osoby o podstawowych umiejętnościach logicznych. Transformacje modelu i formuły zachowują logiczną spełnialność, co zapewnia naturalność i ufność podejścia. Dzięki zastosowaniu narzędzia *Formal Specification IDE* możliwe jest precyzyjne i jednoznaczne określenie modeli inżynierii wymagań.

## 1.1. Cele i założenia pracy

Celem pracy jest stworzenie systemu wspierającego budowę modeli inżynierii wymagań RE (ang. *requirements engineering*) w oparciu o diagramy przypadków użycia, scenariusze i diagramy aktywności. System będzie integrować znane metody i algorytmy, umożliwiając weryfikację tych modeli za pomocą dedukcyjnej metody wnioskowania przy użyciu istniejących silników wnioskowania logicznego.

Główne cele pracy to:

1. Stworzenie systemu *Formal Specification IDE*, należącego do klasy IDE (ang. *integrated development environment*) operujących na formalnej specyfikacji, który umożliwia tworzenie modeli behawioralnych na podstawie diagramów przypadków użycia, scenariuszy i diagramów aktywności.

2. Implementacja algorytmu automatycznej generacji specyfikacji logicznej, w oparciu o określone wzorce przepływu pracy oraz predefiniowane reguły logiczne wzorców.
3. Integracja różnych istniejących systemów wnioskowania logicznego, zarówno logiki pierwszego rzędu (Prover9, SPASS Prover), jak i logiki temporalnej (InKreSAT), z systemem *Formal Specification IDE*, umożliwiając ich wykorzystanie do weryfikacji modeli.
4. Przeprowadzenie eksperymentów prezentujących skuteczność stworzonego systemu w procesie inżynierii wymagań.

W ramach pracy zakłada się stworzenie nowego systemu *Formal Specification IDE*. System ten będzie rozwinięty od podstaw, aby umożliwić użytkownikom tworzenie modeli behawioralnych na podstawie diagramów przypadków użycia, scenariuszy i diagramów aktywności. Wymaga to implementacji nowych modułów odpowiedzialnych za obsługę tych modeli oraz stworzenia odpowiednich interfejsów użytkownika.

Weryfikacja formalna modeli behawioralnych będzie opierać się na dedukcyjnej metodzie wnioskowania. W tym celu zostaną wykorzystane istniejące silniki wnioskowania logicznego, takie jak Prover9, SPASS Prover i InKreSAT. Integracja tych silników z systemem *Formal Specification IDE* będzie obejmować odpowiednie interfejsy komunikacyjne oraz przekształcenie modeli behawioralnych do formatów akceptowanych przez silniki.

## 1.2. Znaczenie logiki w inżynierii oprogramowania

Złożoność współczesnych systemów informatycznych rośnie w szybkim tempie, co niesie ze sobą wyzwania związane z zapewnieniem ich niezawodności, bezpieczeństwa i zgodności z oczekiwaniami użytkowników. Jednym z kluczowych aspektów, który pozwala na osiągnięcie tych celów, jest inżynieria wymagań. Określenie precyzyjnych i spójnych wymagań funkcjonalnych i niefunkcjonalnych systemu stanowi fundament procesu tworzenia oprogramowania. W tym kontekście, znaczenie logiki w inżynierii oprogramowania nie może być niedoceniane. Wprowadzenie formalnej specyfikacji i metod analizy, opartych na logice formalnej, umożliwia inżynierom tworzenie bardziej niezawodnych i bezpiecznych systemów.

Zagadnienia dotyczące znaczenia logiki w inżynierii oprogramowania zostały omówione w wielu artykułach, których analiza pozwala lepiej zrozumieć rolę logiki i jej wpływ na proces inżynierii wymagań.

### 1.2.1. Rola logiki i algebry w inżynierii oprogramowania

Artykuł „On the Role of Logic and Algebra in Software Engineering”[2], autorstwa Manfreda Broya, omawia fundamentalne znaczenie logiki i algebry w inżynierii oprogramowania,

ze szczególnym skupieniem na inżynierii wymagań. Autor wskazuje na rosnącą złożoność systemów informatycznych i konieczność ich formalnego opisu. Inżynieria wymagań odgrywa kluczową rolę w tym procesie, pozwalając na precyzyjne określenie funkcjonalności i niefunkcjonalności systemu.

Wymagania (ang. *requirements*) to specyfikacja funkcjonalnych i niefunkcjonalnych cech systemu, które muszą zostać spełnione, aby system był użyteczny i zgodny z oczekiwaniami użytkowników. Wymagania są kluczowe dla procesu inżynierii oprogramowania, ponieważ stanowią podstawę dla projektowania, implementacji i testowania systemu.

Proces zbierania wymagań nazywa się inżynierią wymagań (ang. *requirements engineering*). Wymagania mogą być zbierane na różne sposoby, w zależności od charakteru projektu i potrzeb użytkowników. Mogą być zbierane poprzez wywiady z użytkownikami, analizę dokumentów, obserwacje użytkowników w trakcie pracy, itp. Ważne jest, aby wymagania były zbierane w sposób systematyczny i dokładny, aby uniknąć późniejszych problemów związanych z niekompletnymi lub nieprecyzyjnymi wymaganiami.

Po zebraniu wymagań, następuje ich przetwarzanie i analiza. Wymagania muszą być dokładnie przeanalizowane, aby upewnić się, że są one spójne, kompleksowe i precyzyjne. Wymagania muszą być również priorytetyzowane, aby określić, które wymagania są najważniejsze dla użytkowników i które mogą być zrealizowane w późniejszych fazach projektu.

Wymagania są kluczowe dla sukcesu projektu inżynierii oprogramowania, ponieważ stanowią podstawę dla projektowania, implementacji i testowania systemu. Dobre wymagania pozwalają na stworzenie systemu, który spełnia oczekiwania użytkowników, jest użyteczny i działa zgodnie z założeniami. Z drugiej strony, niekompletne lub nieprecyzyjne wymagania mogą prowadzić do poważnych problemów w trakcie projektowania i implementacji systemu, a także do niezadowolenia użytkowników i kosztownych poprawek. Dlatego ważne jest, aby proces inżynierii wymagań był przeprowadzany w sposób systematyczny i dokładny, a wymagania były dokładnie przeanalizowane i spójne.

Logiczna specyfikacja (ang. *logical specification*) to proces opisu systemu za pomocą formalnego języka logicznego, takiego jak logika temporalna. Logiczna specyfikacja pozwala na precyzyjne określenie wymagań systemu i jego funkcjonalności, co ułatwia projektowanie, implementację i testowanie systemu.

W artykule autor podkreśla, że opis systemu za pomocą formalnego języka logicznego pozwala na wykorzystanie narzędzi i technik z dziedziny logiki, takich jak kompletność, spójność i poprawność, co ułatwia proces projektowania i testowania systemu. Logiczna specyfikacja pozwala również na wykorzystanie technik generowania kodu z opisu logicznego, takich jak programowanie logiczne, co może przyspieszyć proces implementacji systemu.

W artykule autor podkreśla również, że logiczna specyfikacja nie jest łatwym procesem i wymaga specjalistycznej wiedzy z dziedziny logiki i matematyki. Jednakże, jeśli jest przeprowadzona w sposób dokładny i systematyczny, może przynieść wiele korzyści w procesie inżynierii oprogramowania.

Wykorzystanie narzędzi i technik z dziedziny logiki, takich jak formalna weryfikacja modeli, logika temporalna, czy teoria automatów, umożliwia matematyczne dowodzenie, czy dany system spełnia określone wymagania. Autor zwraca uwagę na to, że logika i algebra mogą pomóc w formalnym opisie wymagań, co pozwala na uniknięcie niejednoznaczności i sprzeczności w specyfikacji systemu.

W artykule przedstawione są różne metody formalnej weryfikacji, takie jak logika temporalna, teoria automatów i model checking. Formalna weryfikacja polega na matematycznym dowodzeniu, czy dany system spełnia określone wymagania. Metoda ta pozwala na automatyczne sprawdzenie, czy dany model spełnia określone wymagania, co pozwala na uniknięcie błędów i zapewnienie niezawodności systemu.

Model checking jest jedną z najpopularniejszych metod formalnej weryfikacji. Polega ona na przeszukaniu przestrzeni stanów systemu, w celu znalezienia błędów lub potwierdzenia poprawności modelu. Metoda ta jest szczególnie przydatna w przypadku systemów o dużej liczbie stanów, ponieważ pozwala na automatyczne sprawdzenie wszystkich możliwych stanów systemu.

Inne metody formalnej weryfikacji, takie jak logika temporalna i logika pierwszego rzędu, polegają na matematycznym opisie systemu i jego własności. Metody te pozwalają na formalne dowodzenie, czy dany system spełnia określone wymagania, co pozwala na uniknięcie błędów i zapewnienie niezawodności systemu.

W każdym przypadku, formalna weryfikacja wymaga matematycznego opisu systemu i jego wymagań, co może być czasochłonne i wymagać specjalistycznej wiedzy. Jednakże, wykorzystanie metod formalnej weryfikacji może znacznie zwiększyć niezawodność i bezpieczeństwo systemu, co jest szczególnie ważne w dziedzinach, w których błędy w systemach mogą mieć poważne konsekwencje dla ludzkiego życia i zdrowia.

Autor zwraca również uwagę na to, że logika odgrywa kluczową rolę w dziedzinach, w których bezpieczeństwo i niezawodność systemów są szczególnie ważne, takich jak przemysł lotniczy, kosmiczny, czy medyczny. W tych dziedzinach, błędy w systemach mogą mieć poważne konsekwencje dla ludzkiego życia i zdrowia, dlatego formalna weryfikacja modeli i logika są niezbędne do zapewnienia bezpieczeństwa i niezawodności systemów.

Podsumowując, artykuł podkreśla znaczenie logiki i algebry w inżynierii oprogramowania, szczególnie w kontekście inżynierii wymagań. Autor zwraca uwagę na to, że wykorzystanie metod logicznych i narzędzi, takich jak model checking, pozwala na formalne dowodzenie własności systemów, co jest kluczowe dla zapewnienia ich niezawodności i bezpieczeństwa.

### 1.2.2. Weryfikacja modelowa, a wnioskowanie dedukcyjne

Celem metod formalnych jest poprawa jakości i niezawodności systemów komputerowych. Wnioskowanie dedukcyjne (ang. *deductive reasoning*) oraz weryfikacja modelowa (ang. *model checking*) to dwie główne kategorie sposobów formalnej weryfikacji właściwości systemów, służących do ustalenia, czy projekt lub implementacja spełnia swoją specyfikację.

W artykule [3] autorzy porównują oba formalne podejścia w procesie tworzenia oprogramowania.

W podejściu dedukcyjnym specyfikacja systemu zawiera abstrakcyjny opis istotnego zachowania wymaganego systemu. Specyfikacja ta jest sformułowana przy użyciu logiki formalnej, takiej jak logika pierwszego rzędu. Wnioskowanie dedukcyjne polega na dowodzeniu twierdzeń w oparciu o tę specyfikację.

W przypadku weryfikacji modelowej, system jest przedstawiany jako automat o skończonym zbiorze stanów oraz przejść między nimi. Specyfikacja właściwości systemu jest opisana w odpowiedniej logice formalnej, takiej jak logika temporalna. Weryfikacja modelowa polega na eksploracji przestrzeni stanów i sprawdzaniu, czy spełnione są określone właściwości na tym skończonym modelu. Model checking jest bardziej ograniczony, ponieważ może być stosowany tylko do systemów, które można opisać za pomocą skończonej liczby stanów.

	Weryfikacja modelowa	Podejście dedukcyjne
System	Automat skończony	Logika formalna
Wymagania	Logika formalna	Logika formalna

**Tabela 1.1.** Porównanie kategorii sposobów formalnej weryfikacji właściwości systemów.

W przypadku weryfikacji modelowej, system przed wyspecyfikowaniem w danym języku programowania, może mieć reprezentację skończenie stanową – automat z stanami oraz przejściami. A właściwości systemu (ang. *requirements*) są opisane w wybranej logice formalnej, takiej jak logika pierwszego rzędu lub logika temporalna. W przypadku podejścia dedukcyjnego, zarówno właściwości, jak i system jest wyspecyfikowany w wybranej logice.

Wadą weryfikacji modelowej jest symulowanie przestrzeni stanów. Brak jest możliwości przebadania ścieżek nieskończonych. Wnioskowanie w podejściu dedukcyjnym uwzględnia ścieżki nieskończone.

W artykule [1] została opracowana metoda automatycznego generowania specyfikacji logicznej, co pozwala na wyspecyfikowanie systemu. W systemie *Formal Specification IDE* zastosowano podejście dedukcyjne, gdzie system oraz wymagania są wyspecyfikowane w logice formalnej.

### 1.2.3. Logika w historii i filozofii inżynierii oprogramowania

Artykuł „When Logic Meets Engineering: Introduction to Logical Issues in the History and Philosophy of Computer Science” [4], autorstwa Liesbeth De Mol i Giuseppe Primiero, zapewnia szeroki wgląd w związek między logiką, a inżynierią oprogramowania z perspektywy historii i filozofii. Wskazują oni, że logika formalna odgrywa kluczową rolę jako narzędzie analizy i opisu systemów informatycznych. Omawiają również historię logiki formalnej i jej pierwotne zastosowania, co pozwala zrozumieć ewolucję i znaczenie logiki w dziedzinie inżynierii oprogramowania. Artykuł ten podkreśla, że zrozumienie filozofii logiki i jej podstawowych pojęć pomaga inżynierom w lepszym wykorzystaniu logiki jako narzędzia do analizy i projektowania systemów informatycznych.

Historia logiki pozwala na zrozumienie, jakie były początki logiki formalnej i jakie były jej pierwotne zastosowania. Filozofia logiki natomiast pozwala na zrozumienie, jakie są podstawowe pojęcia i zasady logiki, a także jakie są jej ograniczenia. Dzięki temu, zrozumienie historii i filozofii logiki pozwala na lepsze zrozumienie znaczenia logiki w informatyce i na to, jakie są jej zastosowania w dziedzinie inżynierii oprogramowania.

Autorzy podają wiele przykładów, jak logika i inżynieria łączą się w informatyce. Jednym z przykładów jest zastosowanie logiki w projektowaniu i weryfikacji systemów informatycznych. Innym przykładem jest zastosowanie logiki w projektowaniu algorytmów i struktur danych.

Logika odgrywa kluczową rolę w informatyce, a jej zastosowanie pozwala na formalną weryfikację modeli inżynierii wymagań. Jednocześnie, zrozumienie historii i filozofii logiki pomaga w zrozumieniu znaczenia logiki w informatyce.

### 1.2.4. Rozwój inżynierii oprogramowania

Artykuł „Yesterday, Today, and Tomorrow: 50 Years of Software Engineering” [5], autorstwa Manfreda Broya, poświęcony jest obchodom 50. rocznicy narodzin inżynierii oprogramowania jako dyscypliny naukowej. Autor przedstawia historię inżynierii oprogramowania, począwszy od konferencji NATO w 1968 roku, która zainicjowała powstanie tej dyscypliny. Artykuł analizuje, jak inżynieria oprogramowania ewoluowała na przestrzeni ostatnich pięćdziesięciu lat, wskazując na rosnące znaczenie tej dziedziny i jej wpływ na rozwój technologii.

Autor podkreśla, że inżynieria oprogramowania jest dzisiaj niezwykle istotna i powszechna, a jej rozwój jest niezwykle szybki. Zwraca uwagę, że wykorzystanie logiki formalnej, w tym formalnej analizy wymagań, pozwala na projektowanie bardziej zaawansowanych i bezpiecznych systemów informatycznych. W artykule przedstawiono również, jakie wyzwania stoją przed inżynierią oprogramowania w przyszłości, takie jak rozwój sztucznej inteligencji, internetu rzeczy i systemów cyber-fizycznych.

W artykule zostają omówione również osiągnięcia, jakie zostały uzyskane w dziedzinie inżynierii oprogramowania w ciągu ostatnich 50 lat. W szczególności, autor odnosi się do formalnych metod inżynierii oprogramowania, takich jak matematyczna teoria Hoare'a i Dijkstry, które otworzyły drogę do formalnej weryfikacji systemów. Artykuł zawiera również informacje na temat wykorzystania modelowania i symulacji w inżynierii oprogramowania.

W artykule autor podkreśla, że inżynieria oprogramowania jest dzisiaj niezwykle ważna i rozległa, a jej rozwój jest niezwykle szybki. Wraz z rozwojem technologii, inżynieria oprogramowania musi dostosować się do nowych wyzwań, takich jak rozwój sztucznej inteligencji, internetu rzeczy i systemów cyber-fizycznych. Autor artykułu podkreśla, że inżynieria oprogramowania musi nadal rozwijać się i dostosowywać do zmieniających się potrzeb i wyzwań, aby sprostać wymaganiom dzisiejszego świata.

### 1.2.5. Zastosowanie logiki w analizie wymagań systemowych

Artykuł „A Case Study In Formal Analysis Of System Requirements” [6], autorstwa Dimitri Belli i Franco Mazzanti, prezentuje praktyczne zastosowanie logiki w formalnej analizie wymagań systemowych w sektorze kolejowym. Opisuje on wykorzystanie metody formalnej analizy wymagań przy użyciu narzędzia ProB i języka Event-B do modelowania i weryfikacji systemu sterowania ruchem kolejowym. Przeprowadzona formalna analiza pozwoliła na weryfikację spełnienia określonych właściwości systemu, co jest istotne w sektorze, gdzie bezpieczeństwo i niezawodność systemów są kluczowe.

Autorzy skupili się na modelowaniu i weryfikacji systemów z wykorzystaniem weryfikacji modelowej, który jest jednym z podejść opartych na modelach. W szczególności, skupili się na technikach weryfikacji modelowej dostępnych w narzędziu ProB dla specyfikacji Event-B.

W artykule autorzy opisują podejście oparte na formalnej analizie wymagań systemowych. W tym podejściu wykorzystywane są formalne metody i narzędzia do analizy wymagań systemowych, takie jak modelowanie formalne, weryfikacja formalna, dowodzenie twierdzeń, model checking. Autorzy opisują również różne techniki i narzędzia wykorzystywane w formalnej analizie wymagań systemowych, takie jak Event-B, ProB, UML-B, UPPAAL, Spin.

Event-B jest to język formalny do modelowania systemów, który opiera się na teorii zbiorów i logice predykatów. Event-B pozwala na modelowanie systemów w sposób formalny i precyzyjny, co ułatwia weryfikację formalną.

ProB jest to narzędzie do weryfikacji modelowej systemów opartych na Event-B. ProB pozwala na automatyczną weryfikację modeli systemów pod kątem spełnienia określonych właściwości, takich jak poprawność oraz bezpieczeństwo.

UML-B jest to język formalny do modelowania systemów, który łączy w sobie notację UML z formalnymi metodami opartymi na teorii zbiorów i logice predykatów. UML-B pozwala na modelowanie systemów w sposób graficzny i formalny, co ułatwia weryfikację formalną.

UPPAAL jest to narzędzie do modelowania i weryfikacji systemów czasu rzeczywistego opartych na automatach czasowych. UPPAAL pozwala na modelowanie systemów w sposób graficzny i formalny, co ułatwia weryfikację formalną.

Spin jest to narzędzie do weryfikacji modelowej systemów opartych na automatach skończonych. Spin pozwala na automatyczną weryfikację modeli systemów pod kątem spełnienia określonych właściwości, takich jak poprawność oraz bezpieczeństwo.

W artykule autorzy przedstawiają również swoją własną metodologię formalnej analizy wymagań systemowych, która składa się z kilku etapów, takich jak analiza wymagań, modelowanie formalne, weryfikacja formalna. W ramach tej metodologii autorzy przeprowadzili analizę wymagań systemowych dla systemu sterowania ruchem kolejowym oraz wykonali formalną weryfikację modelu systemu z wykorzystaniem narzędzia ProB.

Podsumowując, analiza tych czterech artykułów wyraźnie ukazuje, że logika formalna odgrywa kluczową rolę w inżynierii oprogramowania, szczególnie w inżynierii wymagań. Wykorzystanie narzędzi i technik opartych na logice pozwala na precyzyjne określenie wymagań systemu i zapewnienie jego niezawodności oraz zgodności z oczekiwaniami użytkowników. Pozwala to na tworzenie bardziej zaawansowanych i bezpiecznych systemów informatycznych, co jest niezwykle istotne w dzisiejszym rozwijającym się świecie technologii. Dlatego inżynierowie oprogramowania powinni zdobywać wiedzę i umiejętności z zakresu logiki, aby skutecznie projektować, implementować i testować systemy informatyczne.

## 1.3. Struktura pracy

Praca składa się z sześciu rozdziałów, które przedstawiają projekt, analizę i implementację systemu wspierającego budowę i formalną weryfikację modeli inżynierii wymagań za pomocą metod logicznych. Struktura pracy jest następująca:

Rozdział 1 - Wprowadzenie: Rozdział wprowadza w tematykę pracy, określa cele i założenia pracy oraz przedstawia strukturę kolejnych rozdziałów.

Rozdział 2 - Przegląd istniejących rozwiązań: W tym rozdziale przeprowadzono przegląd istniejących rozwiązań związanych z inżynierią wymagań, weryfikacją formalną i wnioskowaniem logicznym. Zaprezentowane są istniejące elementy, które mogą zostać wykorzystane w tworzonym systemie, ze wskazaniem na ich użyteczność.

Rozdział 3 - Opis zastosowanego rozwiązania i metodologii: Rozdział ten opisuje zastosowane rozwiązanie i metodologię pracy. Przedstawia się w nim projekt systemu oraz opisuje przyjęte podejście do budowy modeli behawioralnych i ich formalnej weryfikacji.

Rozdział 4 - Prezentacja systemu: W tym rozdziale przedstawiony jest stworzony system, nazwany *Formal Specification IDE*. Zaprezentowane są wszystkie jego elementy w kolejności, a także wykonano pełne przykłady użycia systemu na rozbudowanych przykładach.

Rozdział 5 - Implementacja systemu i stos technologiczny: W tym rozdziale omówiona jest implementacja systemu. Przedstawiona jest zastosowana architektura, opisane są dostępne komponenty, szczegóły dotyczące implementacji algorytmów oraz integracji z systemami dowodzenia twierdzeń. Wymienione są również wykorzystane technologie podczas implementacji systemu.

Rozdział 6 - Podsumowanie i dalsze kierunki rozwoju: W tym rozdziale dokonuje się podsumowanie pracy, podkreślając osiągnięte cele i wnioski. Przedstawione są również dalsze możliwości rozwoju systemu oraz kierunki, w których można kontynuować badania i rozwijać tematykę pracy.

Dodatkowo, do pracy dołączony jest dodatek zawierający dokumentację techniczną systemu dla użytkownika. Dokumentacja ta służy jako pomoc przy uruchamianiu aplikacji oraz edytowaniu kodu źródłowego projektu.

## **2. Przegląd istniejących rozwiązań**

W tym rozdziale przedstawiono przegląd istniejących rozwiązań związanych z projektowaniem i weryfikacją modeli inżynierii wymagań. Zbadano różne dziedziny, takie jak wzorce przepływu pracy, modelowanie diagramu przypadków użycia oraz systemy dowodzenia twierdzeń. Jednym z ważnych obszarów, który został poddany analizie, są narzędzia typu IDE (ang. *Integrated Development Environment*), które oferują kompleksowe środowisko wspierające proces tworzenia oprogramowania. Celem tego przeglądu jest zapoznanie się z istniejącymi rozwiązaniami, które mogą być wykorzystane w tworzonym systemie wspierającym budowę i formalną weryfikację modeli inżynierii wymagań.

### **2.1. Zintegrowane środowiska programistyczne**

Wykorzystanie zintegrowanych środowisk programistycznych (IDE) w tworzeniu oprogramowania rewolucjonizuje sposób pracy programistów, dostarczając różnorodne narzędzia i funkcje, które zwiększają produktywność i usprawniają proces tworzenia aplikacji. IDE są powszechnie stosowane w różnych dziedzinach, w tym w modelowaniu i weryfikacji systemów oprogramowania.

#### **2.1.1. Warsztaty na temat Formal IDE**

„Workshop on Formal Integrated Development Environment”[7] to coroczne spotkanie naukowe, które skupia się na zagadnieniach związanych z narzędziami typu IDE do projektowania i weryfikacji systemów oprogramowania. Celem warsztatów jest przedstawienie najnowszych osiągnięć w tej dziedzinie oraz wymiana doświadczeń i idei między badaczami, praktykami i użytkownikami narzędzi IDE.

Podczas warsztatów omawiane są różne tematy związane z narzędziami IDE, takie jak:

- Integracja różnych technologii i narzędzi w ramach środowisk IDE.
- Automatyzacja procesu tworzenia oprogramowania i weryfikacji.
- Wsparcie dla różnych języków programowania i modelowania.

- Rozszerzalność i konfigurowalność narzędzi IDE.
- Wprowadzanie nowych funkcjonalności i ulepszeń w narzędziach IDE.

Warsztaty „Workshop on Formal Integrated Development Environment” są ważnym forum dla naukowców i praktyków, którzy zajmują się projektowaniem, implementacją i weryfikacją systemów oprogramowania. Uczestnicy mają możliwość przedstawienia swoich badań i pomysłów oraz spotkania się z innymi ekspertami w tej dziedzinie, co ma na celu uczynienie metod formalnych bardziej dostępnymi zarówno dla specjalistów, jak i niespecjalistów.

Przykłady istniejących narzędzi związanych z projektowaniem i weryfikacją modeli inżynierii wymagań zostały zaprezentowane podczas szóstej edycji warsztatów [7], która odbyła się w maju 2021 roku jako część trzynastej konferencji „NASA Formal Methods”. W ramach tej edycji przedstawiono rozwiązania, takie jak mCRL2, Edukera oraz VeriFly. Te narzędzia są wykorzystywane do specyfikacji zachowania systemów komunikacyjnych, budowy matematycznych dowodów oraz sprawdzania poprawności programów.

Podczas trzeciej edycji warsztatów [8], która odbyła się w listopadzie 2016 roku jako część konferencji FM (International Conference on Formal Methods), zaprezentowano nowe narzędzia i rozszerzenia związane z programowaniem i weryfikacją. Przedstawiono rozwiązania, takie jak UPPAAL, Tinker oraz Dafny IDE, które dotyczą weryfikacji systemów cyberfizycznych, tworzenia dowodów matematycznych oraz analizy poprawności programów.

Systemy IDE różnią się między sobą pod względem funkcji, dostępnych narzędzi oraz obsługiwanych języków programowania. Te środowiska mogą być zastosowane w wielu dziedzinach, ze względu na przeznaczenie tych środowisk oraz ich zastosowanie w kontekście formalnych metod projektowania i weryfikacji systemów.

### 2.1.1.1. Omówienie wybranych narzędzi IDE

**mCRL2** jest językiem algebraicznym procesów, służącym do specyfikowania zachowania komunikujących się systemów. Ten język specyfikacji może być używany do określania oraz analizowania zachowania rozproszonych systemów i protokołów. Narzędzie to udostępnia różne algorytmy redukcji behawioralnej oraz wizualizacji, które pomagają zrozumieć zachowanie systemu.

mCRL2 korzysta z rachunku mu-calculus z danymi i czasem do określenia właściwości. Mu-calculus jest jedną z najważniejszych logik w weryfikacji modelowej, charakteryzuje się ona wyjątkową równowagą między wyrazistością, a właściwościami algorytmicznymi. mCRL2 oraz jego logika modalna są bardzo skuteczne w modelowaniu i analizowaniu podstawowych protokołów i algorytmów rozproszonych. Jednak dzięki swoim potężnym algorytmom, są one również wykorzystywane jako zaplecze weryfikacyjne dla różnych przemysłowych środowisk programistycznych.

Podstawową koncepcją w mCRL2 jest proces. Procesy mogą wykonywać działania i mogą być składane w celu tworzenia nowych procesów za pomocą operatorów algebraicznych. System zwykle składa się z kilku równoległych procesów (lub komponentów).

Proces może zawierać dane jako swoje parametry. Stan procesu jest określony przez kombinację wartości parametrów. Możliwe działania procesu zależą od jego stanu. Każdy proces ma swoją przestrzeń stanów, która zawiera wszystkie możliwe do osiągnięcia stany oraz możliwe przejścia między nimi.

Centralnym pojęciem w mCRL2 jest proces liniowy. Jest to proces, z którego usunięto wszystkie zrównoleglenia. Pojedynczy proces liniowy może odpowiadać tysiącom procesów. W przypadku systemów z nieskończoną przestrzenią stanów, proces liniowy jest skończony.

Na podstawie procesu liniowego oraz formuły określającej zachowanie systemu, można przeprowadzić model checking. Odbywa się to za pomocą PBES (Parameterised Boolean Equation Systems). Rozwiążanie PBES wskazuje, czy formuła zachowuje się poprawnie w danym procesie, czy nie.

**VeriFly** to narzędzie, które integruje framework CiaoPP w zintegrowanym środowisku programistycznym (IDE), umożliwiając analizę i weryfikację programów w czasie rzeczywistym.

CiaoPP to platforma programistyczna, która wykonuje kombinację statycznej i dynamicznej analizy programu, sprawdzanie asercji oraz transformacje programu. Można ją zastosować do programów wyrażonych jako ograniczone klauzule Horna, a także w innych językach wysokiego i niskiego poziomu, przy użyciu techniki semantycznego tłumaczenia na reprezentację opartą na klauzulach Horna. Dzięki tej niezależnej reprezentacji, możliwa jest obsługa różnych języków wejściowych.

Jednym z głównych zastosowań CiaoPP jest pomoc programistom podczas tworzenia programu poprzez wychwytywanie błędów semantycznych znacznie wyższego poziomu niż wykrywane przez klasyczne kompilatory.

Wszystkie wyniki analizy są bezpośrednio odzwierciedlane w tekście programu za pomocą kolorowania i podpowiedzi. Dzięki temu programiści otrzymują szybką i precyzyjną informację zwrotną na temat poprawności programu w trakcie jego tworzenia.

Wymagania dotyczące szybkiego czasu reakcji wynikające z interaktywnego użytkowania są realizowane za pomocą szeregu technik, w szczególności wydajnego silnika punktu stałego, który wykonuje stopniową analizę między procedurami, zależną od kontekstu i ścieżki. Dzięki temu unika się ponownych analiz tam, gdzie to możliwe, zarówno w ramach modułów, jak i poprzez modułową organizację kodu w oddzielne jednostki komplikacji.

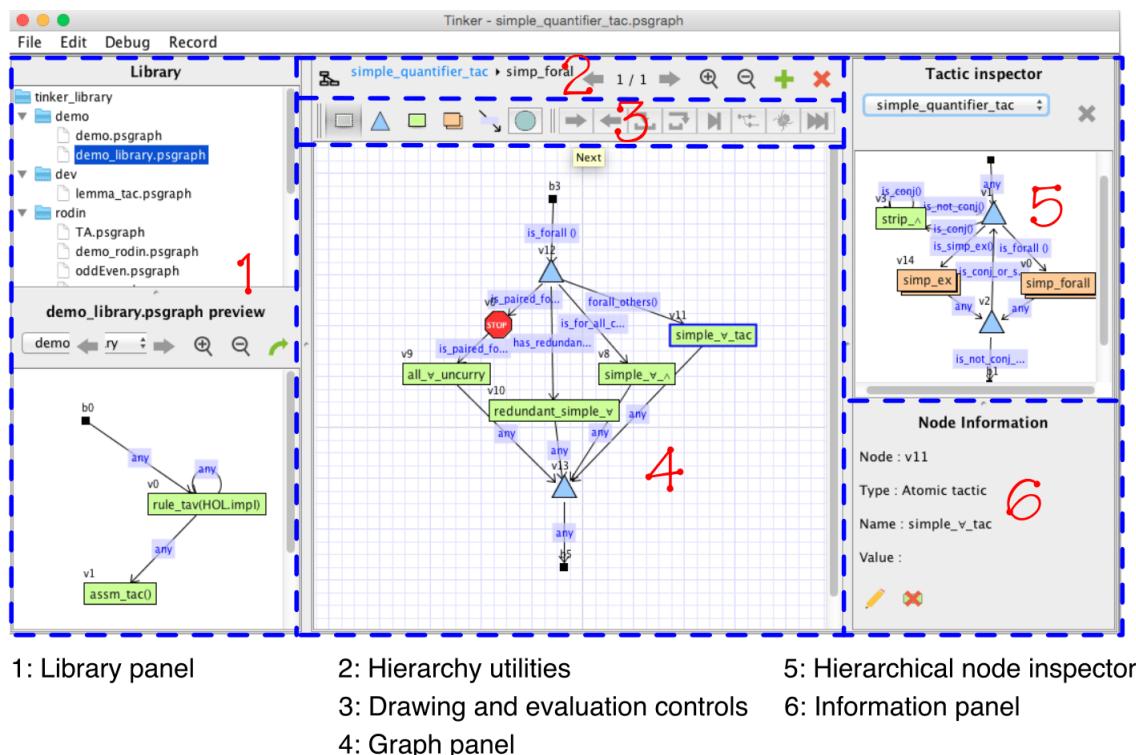
Ważnym elementem CiaoPP jest język asercji (*assertion language*). Asercje pozwalają wyrazić szeroki zakres właściwości, w tym właściwości funkcjonalne oraz niefunkcjonalne, takie jak wykorzystanie zasobów, niezawodność lub liczność. Wykorzystuje się je do wielu celów,

między innymi do pisania specyfikacji, raportowania wyników analizy statycznej i opisywania nieznanego kodu.

Podczas analizy statycznej, CiaoPP buduje wynik w postaci abstrakcyjnego grafu (*abstract graph*), obrazującego sposób wykonania klauzul. Węzły na grafie reprezentują wywołania predykatów. Predykat może mieć kilka węzłów, jeśli istnieją różne sytuacje wywołań. Krawędzie grafu odzwierciedlają sposób, w jaki predykaty wywołują się nawzajem. Dla każdej sytuacji wywołania wywnioskowane są właściwości, które zachowują się, jeśli predykat się powiedzie. Analityczny graf zawiera dwie różne domeny. Sam graf jest regularnym przybliżeniem ścieżek w programie. Oddziennie, wartości abstrakcyjne zawarte w węzłach grafu są skończonymi reprezentacjami stanów występujących w każdym punkcie tych ścieżek.

Framework CiaoPP działa w tle jako demon. Gdy plik jest otwierany, modyfikowany lub zapisywany, wyzwalane jest zdarzenie sprawdzania edytora. Zdarzenia edycji powiadamiają CiaoPP o zmianach w programie. Wyniki z analizy statycznej i dynamicznej są zwracane do IDE i prezentowane jako kolory i podpowiedzi bezpośrednio w tekście programu.

**Tinker** to narzędzie, które umożliwia programistom i matematykom tworzenie i wizualizację strategii dowodowych w sposób graficzny. Dzięki Tinkerowi, dowody matematyczne można zwięzle zakodować za pomocą grafów i węzłów, co ułatwia interaktywny proces dowodzenia twierdzeń.



Rys. 2.1. Układ graficznego interfejsu narzędzia Tinker. Źródło: [8]

Tinker implementuje PSGraph, graficzny język strategii dowodzenia, w którym złożone taktyki (zbiory funkcji) są reprezentowane jako skierowane hierarchiczne grafy. Narzędzie Tinker wspiera Isabelle/HOL, ProofPower i Rodin. W ramach narzędzia ProofPower badamy również przemysłowe zastosowanie PSGraph w Tinkerze, który może być używany jako normalna metoda w IDE. Jeśli jednak wykonanie się nie powiedzie, można uruchomić PSGraph w trybie interaktywnym, gdzie graficzny interfejs użytkownika Tinker służy do wizualizacji, prowadzenia dowodu oraz identyfikowania, gdzie proces dowodzenia się nie powiodł.

W PSGraph, węzły zawierają taktyki (*tactics*) lub zagnieździone grafy i składają się z etykietowanych krawędzi. Etykiety te nazywane są typami celu i opisują oczekiwane właściwości celów. Proces dowodzenia odbywa się przez zastosowanie taktyki do docelowego węzła, który znajduje się na jednym z jego krawędzi wejściowych. Wynikowe cele są wysyłane do krawędzi wyjściowych węzła taktycznego.

Użytkownik korzystający z narzędzia Tinker może narysować PSGraph, dodając wybrane węzły oraz łącząc je przeciągając między nimi linie. Panel sterowania rysowaniem i ocenami zapewnia użytkownikom dużą elastyczność podczas opracowywania i stosowania strategii dowodowych.

**Dafny** to język programowania, który umożliwia weryfikację poprzez specyfikację pożądanych właściwości wraz z ich implementacją w tekście programu. Wykorzystuje zautomatyzowany system dowodzenia twierdzeń, aby udowodnić, że specyfikacja jest spełniona przez program. Specyfikacja pełni funkcję dokumentacji programu i określa właściwości, które mają zostać potwierdzone.

Dafny oferuje wtyczkę do środowiska Visual Studio IDE, co pozwala na wygodne stosowanie weryfikacji w tle. Podczas edycji programu, rozszerzenie Visual Studio przesyła aktualny stan programu do weryfikatora Dafny, który koduje formuły jako translację do języka Boogie. Boogie jest językiem pośrednim wykorzystywanym do weryfikacji programów. Programy w języku Boogie składają się z kilku deklaracji najwyższego poziomu, takich jak aksjomaty, zmienne i procedury, które służą do sformalizowania programów napisanych w języku wyższego poziomu, np. Dafny.

Każda funkcja napisana w Dafny jest tłumaczona na procedury Boogie, które przechwytyują zdefiniowane warunki specyfikacji funkcji. Następnie treść funkcji jest sprawdzana pod kątem spełnienia specyfikacji. Wynikowy program Boogie jest weryfikowany za pomocą automatycznego mechanizmu wnioskowania (SMT-solvera Z3). Błędy weryfikacji, które zostaną ujawnione w tym procesie, są propagowane do rozszerzenia w środowisku Visual Studio i wyświetlane użytkownikowi.

### 2.1.2. Narzędzie AutoFOCUS

AutoFOCUS 3 [9] to zaawansowane narzędzie, które wspiera rozwój projektowania systemów wbudowanych. Jest to zintegrowane narzędzie oparte na modelach (ang. *model-based*), które obejmuje cały proces rozwoju oprogramowania: począwszy od pozyskiwania wymagań, przez wdrażanie, modelowanie platformy sprzętowej, aż po generowanie kodu.

Narzędzie opiera się na modelu systemowym opartym na teorii FOCUS, który pozwala precyjnie opisać system, jego interfejs, zachowanie oraz rozmieszczenie komponentów na różnych poziomach abstrakcji. Ze względu na złożoność opisania takich systemów, stosuje się różne perspektywy, które zapewniają dedykowane techniki opisu dla różnych aspektów strukturalnych (np. platforma sprzętowa, na której wdrażany jest system) oraz behawioralnych (np. wykonanie systemu). Łączenie perspektyw umożliwia dodatkową integrację, np. odwzorowanie zachowania komponentu na element sprzętowy.

Modele systemowe umożliwiają różne mechanizmy analizy i syntezy, np. analizę zgodności częściowego lub pełnego zachowania, syntezę wdrażania komponentów na platformie sprzętowej.

W celu wspierania różnych zadań w procesie tworzenia oprogramowania, perspektywy są zorganizowane w punkty widzenia (ang. *viewpoint*). Punkt widzenia służy jako struktura do zarządzania artefaktami związanymi z procesem rozwoju oprogramowania. Punkty widzenia koncentrują się na definiowaniu wymagań systemowych w analizie wymagań, projektowaniu architektury oprogramowania (jako sieć komunikujących się komponentów) oraz realizacji systemu jako zaplanowanych zadań wykonywanych na procesorach w postaci architektury sprzętowej.

Narzędzie to zostało stworzone przez zespół badawczy z Technische Universität Braunschweig (Uniwersytet Techniczny w Brunszwiku) w Niemczech. Pierwsza wersja narzędzia powstała w latach 90. XX wieku i od tamtej pory było ono rozwijane, co zaowocowało obecną wersją - AutoFOCUS 3. Celem AutoFOCUS 3 jest pokazanie, że podejście oparte na zintegrowanych modelach, perspektywach i punktach widzenia jest rzeczywiście wykonalne.

AutoFOCUS 3 nie jest powiązany z konkretnym procesem tworzenia oprogramowania, jednak zazwyczaj prace przebiegają zgodnie z następującym procesem:

1. **Analiza wymagań:** Wymagania są pozyskiwane, dokumentowane w postaci ustrukturyzowanego tekstu, analizowane, udoskonalane oraz stopniowo sformalizowane. Ze specyfikacji wysokiego poziomu można generować zestawy testów.
2. **Architektura oprogramowania:** System jest projektowany z użyciem języka opartego na komponentach, określającym architekturę oprogramowania aplikacji oraz zachowanie systemu. Projekt może być weryfikowany za pomocą symulacji, testowania (które może być oparte na wygenerowanych testach wysokiego poziomu) oraz formalnej weryfikacji (opartej na weryfikacji modelowej).

**3. Architektura sprzętowa:** Komponenty oprogramowania są wdrażane na platformie, z uwzględnieniem określonych wymagań systemowych. Harmonogramy optymalizujące kryteria są generowane za pomocą solverów SMT (Satisfiability modulo theories).

Kod jest generowany na podstawie modeli, zgodnie z wdrożeniem i wybranym harmonogramem. Można również modelować Safety Cases (ang. *przypadki bezpieczeństwa*), czyli udokumentowane dowody, które dostarczają argumenty, że system jest bezpieczny dla danej aplikacji w danym środowisku.

W narzędziu AutoFOCUS 3 wymagania są dokumentowane z użyciem szablonów z nazwanymi polami do określania m.in. tytułu wymagania, autora, opisu, uzasadnienia, statusu. Dodatkowo można definiować źródła wymagań i słowniki. W przypadku użycia ich w tekście opisu wymagania, wpisy te są automatycznie podświetlane, a definicja jest dostępna w wyskakującym okienku. Wymagania mogą być hierarchicznie pogrupowane według pakietów i zorganizowane według powiązań (ang. *trace links*).

Requirement	
ID	1.2
Type	Requirement
Title	Safety requirement: accidents prevent
Description	Pedestrians and cars should NOT b
Rationale	Prevent traffic accidents.
Author	Dan
Source	System architect Christopher Pike
Document Reference	
Status	Analyzed
Priority	Normal - Satisfier

Rys. 2.2. Przykład ustrukturyzowanego wymagania. Źródło: [9]

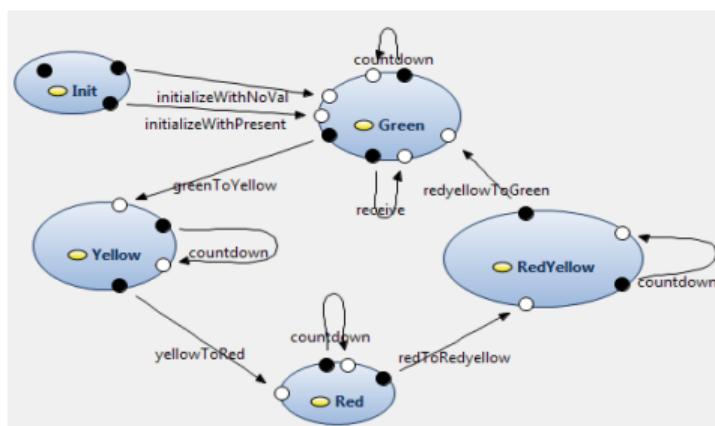
Wymagania mogą być dokumentowane nie tylko jako tekst, ale również z użyciem modeli, które można przetwarzać maszynowo. Wykresy sekwencji komunikatów (ang. *Message sequence charts*) są wykorzystywane do opisania pożądanych lub niepożądanych interakcji aktorów.

Wyrażenia logiki temporalnej można wykorzystać do wyrażenia pożądanego i niepożądanego zachowania opracowywanego systemu. AUTOFOCUS 3 zapewnia szablony do określania wyrażeń logiki temporalnej.

Gdy analiza wymagań jest wystarczająco zaawansowana, możliwe jest wyrażenie sformalizowanych zachowań w postaci automatów stanów. Można przeprowadzać symulacje z użyciem automatów stanowych, co jest używane zarówno w analizie, jak i walidacji wymagań.

W AUTOFOCUS 3 dostępne są raporty o statusach i statystykach podczas analizy wymagań. Możliwa jest identyfikacja pustych pól, duplikatów, niespójnych statusów wymagań i ich powiązań (*trace links*).

Architektura oprogramowania systemu jest opisywana za pomocą klasycznego języka opartego na komponentach z semantyką opartą na teorii FOCUS. Komponenty wykonują się równolegle zgodnie z globalnym, synchronicznym i dyskretnym zegarem czasu. Możliwa jest symulacja architektury oprogramowania na wszystkich poziomach, zarówno pojedynczego automatu stanowego, jak i komponentów złożonych.



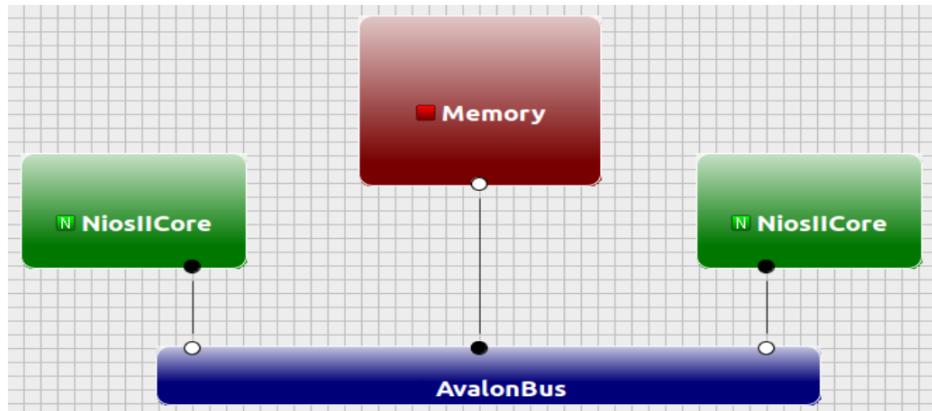
Rys. 2.3. Przykład automatu skończonego stanowego definiującego zachowanie atomicznych komponentów. Źródło: [9]

Zachowanie atomicznych komponentów może być zdefiniowane za pomocą automatów stanowych, tablic lub prostego kodu imperatywnego. Komponenty wchodzą ze sobą w interakcję za pomocą określonych portów wejściowych i wyjściowych.

Dostępne są analizy formalne, takie jak analiza osiągalności, sprawdzanie niedeterminizmu, sprawdzanie granic zmiennych oraz weryfikacja wzorców logiki temporalnej. Architektura oprogramowania jest tłumaczona na język NuSMV/nuXmv (model checker). Tłumaczenie i wywołanie weryfikatora modelowego odbywa się w tle, aby zapewnić przyjazność dla użytkownika. Wyniki sprawdzania modelu są również tłumaczone z powrotem w ten sam sposób.

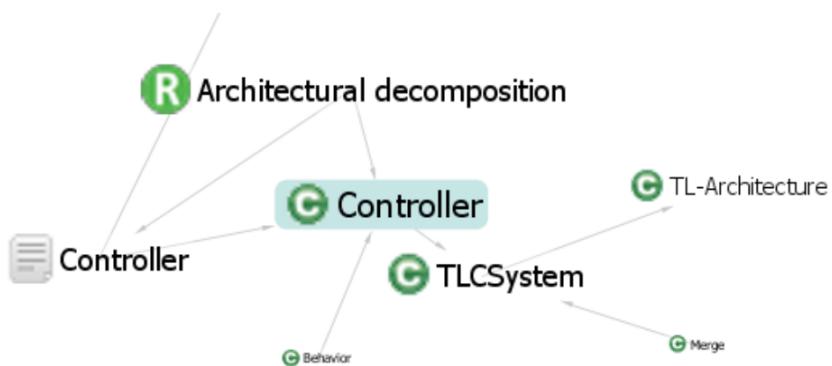
AUTOFOCUS 3 umożliwia modelowanie sprzętu, takiego jak procesory, magistrale, elementy wykonawcze i czujniki, a także połączenia między nimi. Modele architektury sprzętowej dotyczą nie tylko wyłącznie sprzętu, lecz obejmują architekturę platformy, która zawiera środowiska wykonawcze — od samego sprzętu metalowego, poprzez środowiska systemu operacyjnego, aż po środowiska uruchomieniowe wyższego poziomu.

Integracja analizy wymagań, architektury oprogramowania i architektury sprzętowej ułatwia proces integracji narzędzi (odrębne interfejsy, semantyka i standardy). AUTOFOCUS 3 umożliwia integrację modeli, co prowadzi do spójnego, zintegrowanego projektu systemu.



**Rys. 2.4.** Przykład wielordzeniowej architektury sprzętowej z pamięcią wspólnie dzieloną. Źródło: [9]

Integracja wymagań i architektury oprogramowania odbywa się przy użyciu nieformalnych śladów (ang. *informal traces*). Do danego wymagania można dodać powiązanie (trace) do komponentów architektury oprogramowania. Oznacza to, że dany komponent powiązany z wymaganiem powinien spełniać to wymaganie. Powiązania są widoczne na poziomie architektury komponentu. Dostępne są wizualizacje zarówno między wymaganiami, jak i między wymaganiami, a elementami architektury oprogramowania, co pozwala użytkownikowi na ogólnygląd relacji wewnętrz i między punktami widzenia.



**Rys. 2.5.** Wizualizacja powiązań między wymaganiami, a elementami architektury oprogramowania. Źródło: [9]

Połączenia pomiędzy elementami modelu są tworzone za pomocą śladów i są one nieformalne, ponieważ większość wymagań na początku procesu jest nieformalna. W późniejszym etapie uzyskuje się bardziej sformalizowane zachowanie. W takich przypadkach śladы można rozszerzyć do udoskonaleń (ang. *refinements*), które opisują nie tylko zwykłe połączenie, lecz

wyrażają formalną relację między opisem zachowania na poziomie wymagań, a jego implementacją na poziomie oprogramowania. Udoskonalenia definiują sposób, w jaki wartości na poziomie wymagań powinny zostać przekształcone w wartości na poziomie oprogramowania. Udoskonalenia można wykorzystać do weryfikacji modelowej, by potwierdzić, czy komponent rzeczywiście implementuje daną funkcjonalność.

Wymagania wraz z architekturą oprogramowania można połączyć z wykorzystaniem wykresów sekwencji komunikatów (MSC). Jednostki MSC mogą reprezentować aktorów określonych w wymaganiach lub odnosić się do komponentów. W edytorze MSC można dodać odniesienie do danego komponentu. Analogicznie odpowiednie komunikaty można połączyć z portami danych komponentów. Po zapewnieniu tych połączeń, AUTOFOCUS 3 umożliwia weryfikację, tłumacząc MSC na formułę logiki temporalnej i stosując weryfikację modelową, by potwierdzić, że dany MSC jest rzeczywiście wykonalny w architekturze oprogramowania z danymi komponentami i portami.

Integracja oprogramowania i architektury sprzętowej odbywa się za pomocą modeli wdrożenia, które opisują integrację między różnymi punktami widzenia (*viewpoints*). Punkty widzenia oprogramowania są odwzorowywane na punkty widzenia sprzętu. Model wdrożenia obejmuje również przypisanie portów logicznych komponentu oprogramowania do odpowiadających im portów sprzętowych.

Po zdefiniowaniu wdrożenia można zastosować metody Eksploracji przestrzeni projektowej (Design Space Exploration), aby zdefiniować harmonogram komponentów oprogramowania na odpowiednich komponentach sprzętowych. Proces ten polega na automatycznym badaniu różnych konfiguracji komponentów w celu znalezienia optymalnych rozwiązań. Przy użyciu symbolicznego schematu kodowania i solvera teorii spełnialności modulo (SMT), możliwe jest generowanie modeli oraz automatyczne syntezowanie wdrożeń i harmonogramów. Wizualizacje są wykorzystywane do prezentacji wyników, a proponowane podejście jest skalowalne i umożliwia badanie różnorodnych rozwiązań projektowych.

Po zdefiniowaniu architektury oprogramowania, architektury platformy i modelu wdrożenia, AUTOFOCUS 3 umożliwia generowanie kodu. Wejściem dla generatora są zmapowane komponenty oprogramowania na jednostki wykonawcze platformy. Wynikiem jest implementacja modelu systemu wraz z plikami konfiguracyjnymi. Generator kodu składa się z dwóch części: Pierwsza część tłumaczy specyfikację zachowania oprogramowania na kod pośredni. Następnie z tej reprezentacji pośredniej jest generowany ostateczny kod systemu, specyficzny dla danej jednostki. W konkretnych przypadkach pośrednia implementacja może zostać pominięta, gdy wydajniej będzie transformować oryginalną specyfikację do języka docelowego.

W celu dowodzenia o bezpieczeństwie systemów wykorzystuje się technikę przypadków bezpieczeństwa (ang. *Safety Cases*), która umożliwia systematyczne argumentowanie. Przypadki bezpieczeństwa mogą zawierać złożone argumenty, które można rozłożyć na artefakty

systemu modułowego, które zazwyczaj zależą od artefaktów z różnych punktów widzenia. Przykładowo, potrzeba redundancji wpływa zarówno na architekturę oprogramowania, jak i sprzętu. Dzięki wykorzystaniu zintegrowanego modelu systemu, zapewniana jest powtarzalna argumentacja oraz dowody na poprawność argumentacji. W AutoFOCUS 3 przypadki bezpieczeństwa są modelowane w oparciu o *Goal structuring notation*. Jest to graficzna notacja diagramu używana do przedstawiania elementów argumentu i relacji między tymi elementami. Przykładowo, można połączyć cel bezpieczeństwa z danymi wymaganiem.

Podsumowując, AutoFOCUS 3 ułatwia proces rozwoju oprogramowania, dostarczając narzędzi do modelowania, generowania kodu, weryfikacji modeli, planowania wdrożenia oraz zarządzania projektem. Oferuje on również różnorodne techniki i narzędzia, które pozwalają na automatyzację procesów i zmniejszenie ryzyka błędów ludzkich. Dzięki swojej otwartej strukturze jest dostępny dla szerokiego grona użytkowników i może być dostosowany do indywidualnych potrzeb oraz wymagań projektów systemów wbudowanych.

## 2.2. Wzorce przepływu pracy

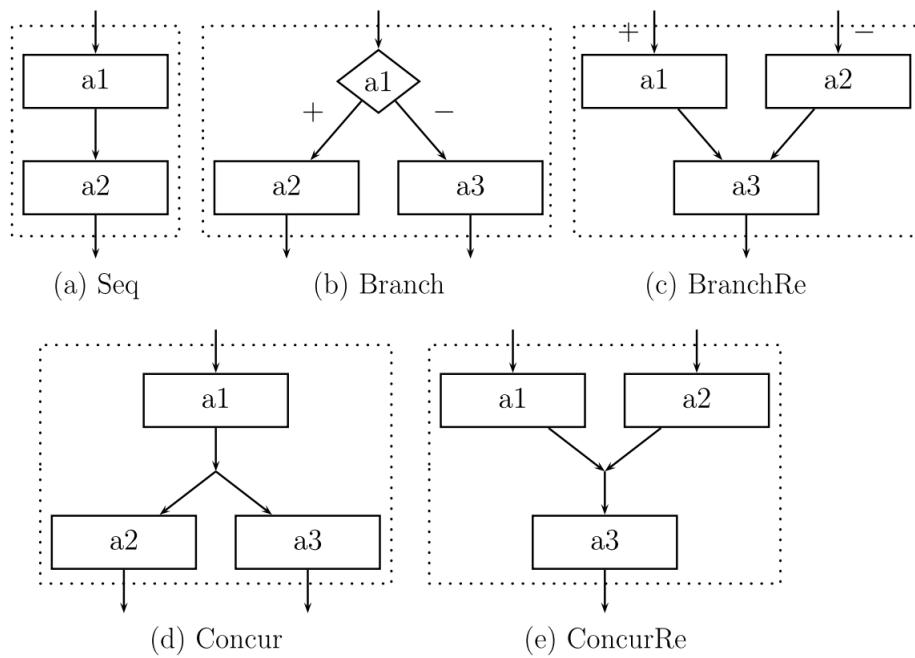
Wzorzec przepływu pracy (ang. *workflow pattern*, lub *control flow meta-pattern*, lub krótko *pattern*) jest ogólnym opisem struktury obliczeń i nie ogranicza możliwości modelowania dowolnie złożonych modeli behawioralnych. Wybrane wzorce przepływu pracy, zwane zatwierdzonymi wzorcami (ang. *approved patterns*), to zbiór, który wylicza wybrane wzorce dla danego procesu deweloperskiego. Wzorce stanowią ważną część procesów tworzenia i modelowania oprogramowania i są powszechnie akceptowane przez społeczność IT. Wzorce przyspieszają procesy projektowe, dzięki zastosowaniu powtarzalności. W artykule [1] zaproponowano następujący zestaw wzorców zachowań oprogramowania:  $\Pi_0 = \{Sequence, Branching, BranchingReturn, Concurrency, ConcurrencyReturn\}$  lub używając ich skróconych nazw

$$\Pi_0 = \{Seq, Branch, BranchRe, Concur, ConcurRe\} \quad (2.1)$$

Wzorce te modelują sekwencję, rozgałęzienie, łączenie rozgałęzienia, współbieżność i synchronizację współbieżności, odpowiednio dla określonych działań. Rysunek 2.6 przedstawia graficzną reprezentację tych podstawowych wzorców przepływu pracy  $\Pi_0$ .

Wzorce mogą uwzględniać większą liczbę aktywności, co umożliwia wygodniejsze tworzenie wzorców. W artykule [1] przedstawiono również bardziej złożone wzorce przepływu pracy:  $\Pi_1 = \{Conditional, Parallel, LoopWhile\}$  lub używając ich skróconych nazw

$$\Pi_1 = \{Cond, Para, Loop\} \quad (2.2)$$



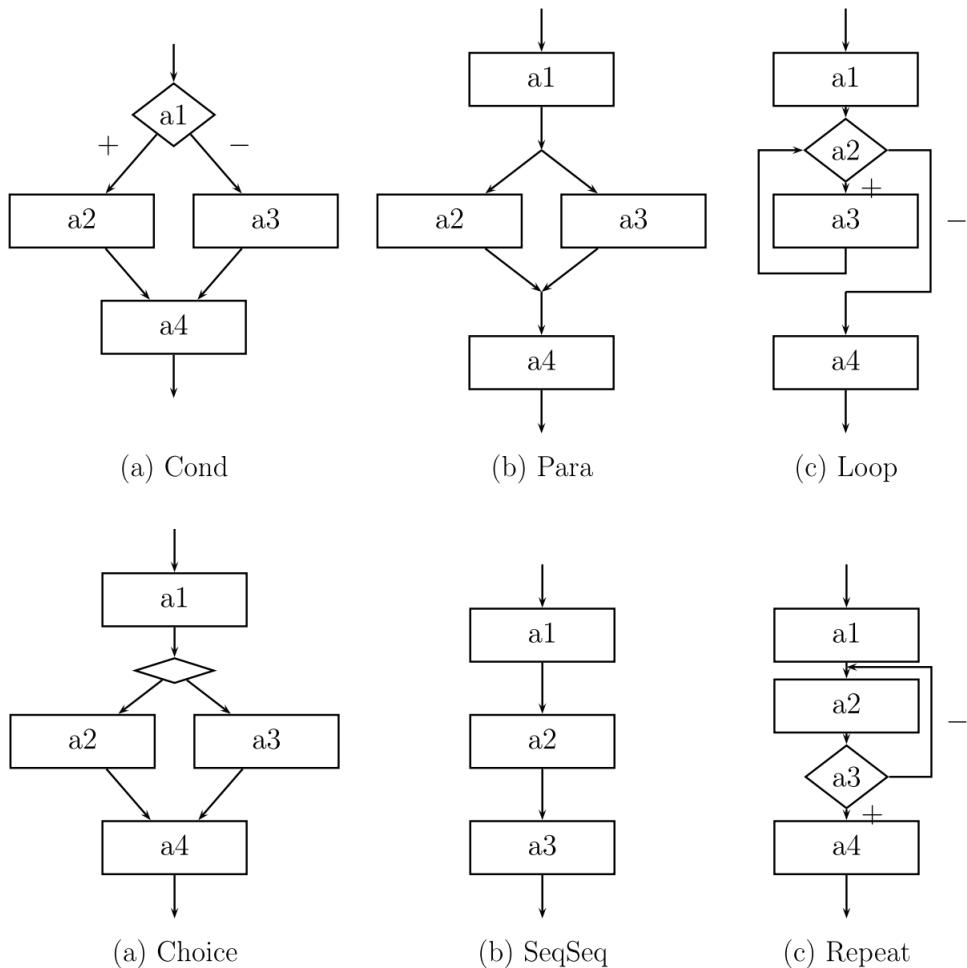
**Rys. 2.6.** Graficzna reprezentacja podstawowych wzorców przepływu pracy  
 $\Pi_0$  dla formuły 2.1. Źródło: [1]

Ich znaczenie jest następujące: cała struktura dla rozgałęzienia, cała struktura dla współbieżności oraz ogólna postać przetwarzania w pętli. Rysunek 2.7 przedstawia graficzną reprezentację tych bardziej złożonych wzorców przepływu pracy  $\Pi_1$ . Jeszcze innym zestawem wzorców, przedstawionych w artykule [1], pełniących rolę uzupełniającą jest:  $\Pi_2 = \{SimpleChoice, DoubleSequence, LoopRepeat\}$  lub używając ich skróconych nazw

$$\Pi_2 = \{Choice, SeqSeq, Repeat\} \quad (2.3)$$

Wyjaśnienie jest następujące: rozgałęzianie bez modelowania warunku logicznego, sekwenca sekwencji i przetwarzanie w pętli, z co najmniej jednym wykonaniem ciała. Rysunek 2.7 przedstawia graficzną reprezentację uzupełniających wzorców przepływu pracy  $\Pi_2$ .

Wzorce zostały zdefiniowane za pomocą modelu Kripkego (ang. *Kripke structure*). Model Kripkego jest używany do definiowania semantyki logiki modalnej i temporalnej [10]. Jednak model ten jest również użyteczny przy definiowaniu wzorców. Każdy wzorzec jest reprezentowany jako graf z krawędziami (kontrolującymi przepływ) oraz węzłami, które są osiągalne i posiadają etykiety (aktywności). W definicji wzorców używa się zbioru atomicznych aktywności  $AA = \{a, b, c, \dots\}$ , który zawiera najprostsze możliwe wyrażenia (operacje obliczeniowe, akcje) do opisania w rozważanym modelu. Tych aktywności nie da się rozłożyć na prostsze aktywności.



**Rys. 2.7.** Graficzna reprezentacja bardziej złożonych wzorców przepływu pracy  $\Pi_0$  dla formuły 2.1. Źródło: [1]

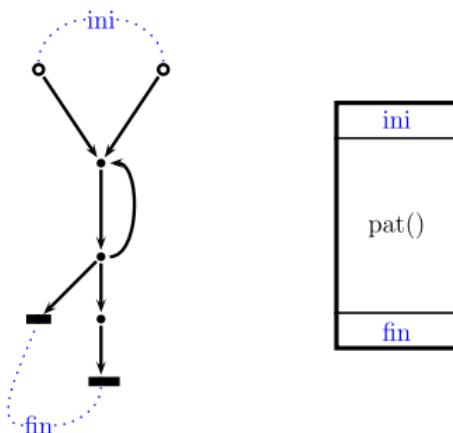
Niektóre atomiczne aktywności mogą odnosić się do ewaluacji wyrażeń boolowskich. W takim przypadku takie aktywności są wyrażeniami warunkowymi. Oznacza to, że wszystkie wyrażenia warunkowe są podziobrem AA. Każda aktywność może mieć maksymalnie jeden warunek. Innymi słowy, pewna aktywność może być wyrażeniem warunkowym, które posiada powiązany warunek w postaci prostego wyrażenia boolowskiego. Gdy warunek jest spełniony, pozostaje on niezmienny podczas wykonywania wyrażenia warunkowego, dopóki aktywność nie zostanie zakończona.

Wzorzec jest strukturą  $Pattern = \langle ini, fin, pat(a_1, a_2, \dots) \rangle$ , gdzie:

- *ini* i *fin* to wyrażenia logiczne klasycznej logiki zdaniowej, które opisują logiczne okoliczności rozpoczęcia i zakończenia wykonania całego wzorca.
  - *pat()* to zestaw formuł logicznych opisujących zachowanie danego wzorca. Jest to koniunkcja wszystkich wymagań, które muszą być spełnione.

Argumenty wzorca  $a_1, a_2, \dots$  zostały podzielone na trzy rozłączne podzbiory:

- Argumenty *ini* – Przynajmniej jeden element, tworzący *ini-expression*.
- Zwykłe argumenty – Ten podzbiór może być pusty.
- Argumenty *fin* – Przynajmniej jeden element, tworzący *fin-expression*.



Rys. 2.8. Graficzna reprezentacja wzorców oraz wyrażeń *ini*- i *fin*- . Źródło: [1]

Rysunek 2.8 przedstawia wyrażenia *ini*- i *fin*- . Wyrażenia te opisują sytuacje, w których przepływ jest, odpowiednio, już i nadal wewnątrz wzorca. Są one wyrażane w klasycznej logice zdań, ponieważ są sprawdzane w określonym momencie.

Struktura każdego wzorca składa się z przynajmniej dwóch atomicznych aktywności. Wyrażenia *ini*- i *fin*- opisują różne warunki dla dwóch różnych argumentów, przynajmniej jeden dla wyrażeń *ini*- i przynajmniej jeden dla wyrażeń *fin*- .

Wzorce przepływu pracy zostaną wykorzystane w celu modelowania diagramów aktywności w ramach omawianego systemu. Dzięki nim możliwe będzie stworzenie wyrażenia wzorcowego, które posłuży jako wejście dla algorytmu generującego specyfikację logiczną 5.4.

Proces modelowania diagramów aktywności oparty na wzorcach przepływu pracy pozwoli na abstrakcyjne przedstawienie struktury i zachowania systemu. Poprzez wykorzystanie odpowiednich wzorców, będzie możliwa reprezentowanie różnych działań. Dzięki temu możliwe będzie opisanie i zrozumienie różnych scenariuszy i przypadków użycia systemu.

Stworzone wyrażenie wzorcowe, oparte na zdefiniowanych wzorcach przepływu pracy, będzie stanowiło podstawę dla algorytmu generującego specyfikację logiczną. Algorytm ten będzie analizował wyrażenie wzorcowe i na jego podstawie generował odpowiednie reguły, które będą miały wpływ na zachowanie systemu. Specyfikacja logiczna będzie służyć do precyzyjnego opisania oczekiwanej funkcjonalności oraz reguł działania systemu.

Przy użyciu wzorców przepływu pracy oraz algorytmu generującego specyfikację logiczną możliwe będzie skuteczne projektowanie, analizowanie i implementowanie systemów o złożonym zachowaniu. Wzorce przepływu pracy stanowią uniwersalne narzędzie do opisu i modelowania różnych aspektów działania systemu, a ich wykorzystanie w przedmiotowym systemie ma na celu zwiększenie czytelności, elastyczności i użyteczności generowanej specyfikacji logicznej.

### 2.2.1. Właściwości wzorców przepływu pracy

Ważnym aspektem wzorców przepływu pracy są ich właściwości, które określają charakterystyczne cechy i zachowanie wzorców. Zbiór ustalonych właściwości  $\Sigma$  dla zatwierdzonych wzorców przepływu pracy  $\Pi$  definiuje zbiór logicznych wzorców  $\Sigma = \{Pattern_1, Pattern_2, \dots\}$ .[1]

$$\Sigma_0 = \{ \begin{aligned} \mathbf{Seq}(a1, a2) &= \langle a1, a2, \diamond a1, \square(a1 \Rightarrow \diamond a2), \square \neg(a1 \wedge a2) \rangle, \\ \mathbf{Branch}(a1, a2, a3) &= \langle a1, a2 \vee a3, \diamond a1, \square(a1 \Rightarrow (\diamond a2 \wedge \neg \diamond a3) \vee (\neg \diamond a2 \wedge \diamond a3)), \square(a1^+ \Rightarrow \diamond a2), \square(a1^- \Rightarrow \diamond a3), \square \neg(a1 \wedge a2), \square \neg(a1 \wedge a3) \rangle, \\ \mathbf{BranchRe}(a1, a2, a3) &= \langle a1 \vee a2, a3, (\diamond a1 \wedge \neg \diamond a2) \vee (\neg \diamond a1 \wedge \diamond a2), \square(a1 \vee a2 \Rightarrow \diamond a3), \square \neg(a1 \wedge a3), \square \neg(a2 \wedge a3) \rangle, \\ \mathbf{Concur}(a1, a2, a3) &= \langle a1, a2 \vee a3, \diamond a1, \square(a1 \Rightarrow \diamond a2 \wedge \diamond a3), \square \neg(a1 \wedge a2), \square \neg(a1 \wedge a3) \rangle, \\ \mathbf{ConcurRe}(a1, a2, a3) &= \langle a1 \vee a2, a3, \diamond a1, \diamond a2, \square(a1 \Rightarrow \diamond a3), \square(a2 \Rightarrow \diamond a3), \square \neg(a1 \wedge a3), \square \neg(a2 \wedge a3) \rangle \end{aligned} \}$$

Rys. 2.9. Zbiór właściwości dla zatwierdzonych wzorców dla formuły 2.1.

Źródło: [1]

Przykładowy zbiór właściwości wzorców reprezentowanych przez formułę 2.1 został przedstawiony w artykule [1] i ilustruje rysunek 2.9. Zbiór właściwości jest ustalony i nie ulega zmianom w trakcie procesu tworzenia modeli oprogramowania. W oznaczeniach użyto  $a1, a2, \dots$  jako formalnych argumentów dla wzorca, reprezentujących atomiczne formuły. Operator przecinka między formułami logiki temporalnej oznacza koniunkcję logiczną.

Również dla bardziej skomplikowanych wzorców przepływu pracy, reprezentowanych przez formułę 2.2, zostały przedstawione odpowiednie definicje właściwości, które ilustruje rysunek 2.10. Właściwości te są zdefiniowane w kontekście logiki temporalnej.

W celu wykorzystania prezentowanych reguł wzorców w systemie, konieczne będzie ich przepisanie na format, który będzie łatwiejszy do edycji i użytkowania w ramach tego systemu. Ponadto, będzie wymagane określenie reguł dla wzorców w kontekście logiki pierwszego rzędu, gdyż zaprezentowane reguły w artykule [1] są zdefiniowane w liniowej logice temporalnej.

$$\Sigma_1 = \{ \begin{aligned} \mathbf{Cond}(a1, a2, a3, a4) &= \langle a1, a4, \diamond a1, \square(a1 \Rightarrow (\diamond a2 \wedge \neg \diamond a3) \vee \\ &\quad (\neg \diamond a2 \wedge \diamond a3)), \square(a1^+ \Rightarrow \diamond a2), \square(a1^- \Rightarrow \diamond a3), \square(a2 \vee a3 \Rightarrow \\ &\quad \diamond a4), \square\neg(a1 \wedge (a2 \vee a3)), \square\neg((a2 \vee a3) \wedge a4)), \\ \mathbf{Para}(a1, a2, a3, a4) &= \langle a1, a4, \diamond a1, \square(a1 \Rightarrow \diamond a2 \wedge \\ &\quad \diamond a3), \square(a2 \Rightarrow \diamond a4), \square(a3 \Rightarrow \diamond a4), \square\neg(a1 \wedge (a2 \vee \\ &\quad a3)), \square\neg((a2 \vee a3) \wedge a4)), \\ \mathbf{Loop}(a1, a2, a3, a4) &= \langle a1, a4, \diamond a1, \square(a1 \Rightarrow \diamond a2), \square(a2 \Rightarrow \\ &\quad (\diamond a3 \wedge \diamond a4) \vee (\neg \diamond a3 \wedge \diamond a4)), \square(a2 \wedge a2^+ \Rightarrow \\ &\quad \diamond a3), \square(a2 \wedge a2^- \Rightarrow \neg \diamond a3 \wedge \diamond a4), \square(a3 \Rightarrow \diamond a2), \square(a4 \Rightarrow \\ &\quad \neg \diamond a2 \wedge \neg \diamond a3), \square\neg(a1 \wedge (a2 \vee a3 \vee a4)), \square\neg(a2 \wedge (a1 \vee a3 \vee \\ &\quad a4)), \square\neg(a3 \wedge (a1 \vee a2 \vee a4)), \square\neg(a4 \wedge (a1 \vee a2 \vee a3))) \} \end{aligned} \}$$

Rys. 2.10. Zbiór właściwości dla zatwierdzonych wzorców dla formuły 2.2.

Źródło: [1]

Właściwości wzorców przepływu pracy stanowią kluczowy element umożliwiający opisanie ich charakterystyki i zachowania. Odpowiednie zdefiniowanie właściwości wzorców przyczynia się do precyzyjnego modelowania różnych aspektów systemu oraz generowania specyfikacji logicznej, co jest istotne dla projektowania, analizowania i implementowania systemów oprogramowania o złożonym zachowaniu.

## 2.2.2. Wyrażenie wzorcowe

W celu precyzyjnej i zwięzlej reprezentacji struktur przepływu pracy o dowolnej złożoności, w tym wzorców zagnieżdżonych, w artykule [1] ustalono notację dla wyrażeń wzorcowych.

Wyrażenie wzorcowe (ang. *pattern expression* lub *well-formed expression*) jest strukturą, która powstaje w oparciu o skończony i niepusty alfabet symboli wzorców *Wzorzec1*, *Wzorzec2*, .... Te symbole wzorców są nazwami zatwierdzonych wzorców ze zbioru właściwości. Wyrażenie wzorcowe jest konstruowane zgodnie z następującymi zasadami:

1. Jeśli każdy argument wzorca jest atomiczną formułą, wzorzec ten jest wyrażeniem wzorcowym.
2. Dla każdego wzorca, każdy jego argument jest albo atomiczną formułą, albo innym wyrażeniem wzorcowym.

Każde wyrażenie wzorcowe jest interpretowane na podstawie zestawu ustalonych właściwości wzorców.

W przedmiotowym systemie wyrażenie wzorcowe zostanie zamodelowane przy użyciu diagramu aktywności. Na tej podstawie możliwe będzie wygenerowanie specyfikacji logicznej.

W artykule [1] przedstawiono również etykietowane wyrażenie wzorcowe  $w'$  (ang. *Labelled pattern expression*), które jest uzyskiwane poprzez wprowadzenie etykiet numerycznych do wyrażenia wzorcowego  $w$ . Proces etykietowania odbywa się zgodnie z następującymi regułami:

- Każdy lewy nawias okrągły jest zamknięty z prawej strony etykietą numeryczną i prawym nawiasem kwadratowym.
- Każdy prawy nawias okrągły zamknięty jest z lewej strony prawym nawiasem kwadratowym i etykietą numeryczną.
- Wszystkie sparowane nawiasy okrągłe mają tę samą etykietę numeryczną.
- Wyrażenie wzorcowe jest przetwarzane od lewej do prawej strony, a najmniejszą dostępną etykietą jest 1.
- Każda liczba jest wprowadzana jako najmniejsza możliwa etykieta, a dopiero użycie liczby z prawym nawiasem okrągłym zwalnia ją do dalszego etykietowania.

Przykładem wyrażenia wzorcowego jest:

$$w = Seq(a, Seq(Concur(b, c, d), ConcurRe(e, f, g)) \quad (2.4)$$

Wtedy etykietowane wyrażenie wzorcowe dla 2.4 będzie:

$$w' = Seq(1)a, Seq(2)Concur(3)b, c, d[3], ConcurRe(3)e, f, g[3][2][1] \quad (2.5)$$

W artykule [1] przedstawiono algorytm etykietowania wyrażeń wzorcowych, który został zaimplementowany i wykorzystany w opracowanym systemie do generacji specyfikacji logicznej.

Skonsolidowane wyrażenie wzorcowe (ang. *consolidated expression*) to zjednoczony zbiór argumentów należących do wyrażenia *ini*- lub *fin*- . Wyróżnia się skonsolidowane wyrażenie *ini*- oraz *fin*- . Obliczanie skonsolidowanego wyrażenia wzorcowego odbywa się zgodnie z następującymi regułami:

- Jeśli nie ma wzorca w miejscu dowolnego argumentu atomowego, który składniowo należy do wyrażenia *ini*- (lub odpowiednio wyrażenia *fin*-), to skonsolidowane wyrażenie wzorcowe równe jest argumentom *ini* danego wzorca.
- Jeśli istnieje wzorzec, zamiast dowolnego argumentu atomowego, który składniowo należy do wyrażenia *ini*- (lub odpowiednio wyrażenia *fin*-), to ten argument jest zastępowany przez istniejący wzorzec.

W artykule [1] przedstawiono algorytm obliczania skonsolidowanego wyrażenia wzorcowego, który został zaimplementowany i wykorzystany w opracowanym systemie do generacji specyfikacji logicznej.

### 2.2.3. Generacja specyfikacji logicznej

Algorytm generowania specyfikacji logicznej został zaprezentowany i szczegółowo opisany w artykule [1].

Generowanie specyfikacji logicznej ma na celu konwersję wyrażenia wzorca na konkretne formuły logiczne, które stanowią specyfikację modelu oprogramowania. Specyfikacja logiczna (ang. *software model logical specification*) składa się z wyprowadzonych formuł, które są rezultatem przetwarzania wyrażenia wzorcowego za pomocą wspomnianego algorytmu.

Algorytm generowania specyfikacji logicznej przyjmuje dwa wejścia. Pierwszym z nich jest wyrażenie wzorcowe, które jest zmienną reprezentującą model przepływu pracy. Drugim wejściem jest predefiniowany zestaw reguł, który jest stały dla konkretnej klasy modeli oprogramowania. Wyjściem algorytmu jest wygenerowana specyfikacja logiczna, która składa się z zbioru formuł logicznych.

Specyfikacja logiczna jest zazwyczaj sumą specyfikacji uzyskanych dla poszczególnych wzorców. Oznacza to, że specyfikacja logiczna  $L$  jest równa sumie specyfikacji  $L_2_1 \cup L_2_2 \cup \dots$ , gdzie każde  $L_2_i$  jest specyfikacją logiczną dla wzorca  $W_{zorzec_i}$ .

W ramach stworzonego systemu, zaimplementowano wspomniany algorytm generowania specyfikacji logicznej. Implementacja tego algorytmu umożliwia wygenerowanie specyfikacji logicznej na podstawie zadanych wyrażeń wzorcowych i zestawu reguł.

## 2.3. Modelowanie diagramu przypadków użycia

Rozpoczynając pracę z aplikacją, użytkownik ma możliwość importowania plików diagramu przypadku użycia w formacie .xml do systemu. Plik .xml zawiera informacje o przypadkach użycia oraz relacjach pomiędzy nimi, takich jak *include*, *extend* oraz *generalization*. Dodatkowo, istnieje opcja importu plików graficznych w formacie .jpg, .jpeg lub .png, które zawierają wizualizację diagramu przypadków użycia.

Na rynku dostępnych jest wiele narzędzi umożliwiających tworzenie i eksportowanie diagramów przypadków użycia. Z tego powodu, w stworzonym systemie nie zapewniono bezpośrednich funkcji modelowania diagramu przypadków użycia, lecz umożliwiono import plików źródłowych z zewnętrznych programów.

Pliki projektów z gwarantowanym wyodrębnieniem przypadków użycia pochodzących z programów takich jak:

- Visual Paradigm

Visual Paradigm[11] to zaawansowane narzędzie, w formie aplikacji desktopowej, do modelowania diagramów przypadków użycia, które jest szeroko stosowane w dziedzinie inżynierii oprogramowania. Program oferuje szeroki zakres funkcji i możliwości, które

umożliwiają tworzenie profesjonalnych i szczegółowych diagramów. Użytkownicy mogą tworzyć, edytować i eksportować diagramy przypadków użycia w różnych formatach, takich jak XML, PNG, PDF itp. Visual Paradigm posiada intuicyjny interfejs użytkownika, który ułatwia pracę i zapewnia płynne doświadczenie użytkownika. Wraz z tworzeniem diagramów przypadków użycia, program oferuje również wiele innych funkcji, takich jak zarządzanie wymaganiami, generowanie dokumentacji, zarządzanie projektem i wiele innych.

- Enterprise Architect

Enterprise Architect[12] to kompleksowe narzędzie do modelowania i analizy systemów, w formie aplikacji desktopowej, które znajduje zastosowanie w różnych branżach, w tym w dziedzinie inżynierii oprogramowania. Program oferuje bogaty zestaw funkcji, w tym narzędzia do tworzenia diagramów przypadków użycia. Użytkownicy mogą importować i eksportować diagramy przypadków użycia w różnych formatach, co umożliwia łatwą wymianę informacji z innymi narzędziami i systemami. Enterprise Architect umożliwia również generowanie dokumentacji na podstawie diagramów, co ułatwia tworzenie szczegółowych specyfikacji i raportów.

- Eclipse Papyrus

Eclipse Papyrus[13] to zaawansowane narzędzie do modelowania, w formie aplikacji desktopowej, które jest częścią projektu Eclipse. Program oferuje wiele funkcji do tworzenia różnych rodzajów diagramów, w tym diagramów przypadków użycia. Użytkownicy mogą tworzyć diagramy przy użyciu intuicyjnego interfejsu użytkownika i wykorzystywać różne elementy, takie jak aktorzy, przypadki użycia, relacje itp. Eclipse Papyrus umożliwia importowanie i eksportowanie diagramów w różnych formatach, co zapewnia elastyczność w wymianie danych z innymi narzędziami i środowiskami.

- Sinvas UML

Sinvas UML[14] to prosty i intuicyjny program desktopowy do tworzenia diagramów UML, w tym diagramów przypadków użycia. Program oferuje podstawowe funkcje modelowania, które umożliwiają użytkownikom definiowanie aktorów, przypadków użycia i relacji pomiędzy nimi. Sinvas UML zapewnia prosty interfejs użytkownika, który ułatwia tworzenie i edycję diagramów. Użytkownicy mogą eksportować diagramy w formacie graficznym, takim jak JPEG, PNG lub SVG, co pozwala na łatwe udostępnianie i prezentację diagramów.

- GenMyModel

GenMyModel[15] to zaawansowane narzędzie online do modelowania UML, które umożliwia tworzenie diagramów przypadków użycia. Program oferuje wiele funkcji i możliwości, które umożliwiają użytkownikom projektowanie i tworzenie profesjonalnych diagramów. GenMyModel umożliwia edycję diagramów w trybie współpracy, co umożliwia użytkownikom pracę zespołową nad tworzeniem i udoskonalaniem diagramów. Narzędzie oferuje również funkcję generowania kodu z diagramów, co przyspiesza proces implementacji. Użytkownicy mogą eksportować diagramy w różnych formatach, takich jak PDF, PNG, SVG itp., co zapewnia elastyczność w udostępnianiu i prezentowaniu diagramów.

Wymienione programy służą do modelowania i projektowania systemów. Oferują one różnorodne funkcje, które umożliwiają użytkownikom tworzenie, edytowanie, udostępnianie i eksportowanie diagramów przypadków użycia w sposób efektywny i elastyczny.

W przypadku badania formatów plików używanych przez narzędzia do modelowania diagramów przypadków użycia, warto zauważyć pewne ograniczenia w zakresie pobierania informacji o relacjach (*include*, *extend* i *generalization*). Okazuje się, że formaty używane przez Enterprise Architect i Sinvias UML nie zapewniają możliwości pobierania tych relacji z wyeksportowanych plików. Z drugiej strony, pozostałe programy (Visual Paradigm, Eclipse Papyrus i GenMyModel), oferują pewne funkcje związane z tymi relacjami. W przypadku formatów plików eksportowanych przez Enterprise Architect i Sinvias UML, brak możliwości pobrania informacji o relacjach może wpływać na dokładność analizy diagramów przypadków użycia. Bez tych istotnych informacji, analiza może być mniej kompletna, a wyniki mogą być mniej precyzyjne.

W kontekście analizy narzędzi do modelowania diagramów przypadków użycia, warto zauważyć, że GenMyModel wyróżnia się spośród innych programów ze względu na swoją charakterystykę online. Oferuje on platformę internetową, która umożliwia użytkownikom tworzenie diagramów przypadków użycia bez konieczności instalacji dodatkowego oprogramowania na swoim komputerze. Dzięki temu, użytkownicy mają elastyczność i wygodę dostępu do narzędzia z dowolnego miejsca i urządzenia, o ile mają połączenie internetowe.

GenMyModel jest dostępny jako platforma wspomagająca modelowanie diagramów przypadków użycia oraz innych typów diagramów UML. Choć inne narzędzia, takie jak Visual Paradigm, Enterprise Architect, Eclipse Papyrus i Sinvias UML, również oferują zaawansowane funkcje modelowania, jednak wymagają instalacji na lokalnym komputerze. To oznacza, że użytkownicy muszą zainstalować oprogramowanie i mieć dostęp do odpowiedniego środowiska pracy. W przypadku GenMyModel, wystarczy dostęp do przeglądarki internetowej, co czyni go bardziej dostępnym dla szerokiego spektrum użytkowników.

Dzięki importowi tych plików, użytkownik może skorzystać z istniejących diagramów przypadków użycia i wykorzystać je w *Formal Specification IDE*. Aplikacja analizuje importowany

plik i wyodrębnia informacje dotyczące przypadków użycia oraz relacji, co umożliwia dalszą pracę nad specyfikacją. W przypadku, gdy użytkownik chce dokonać zmian w diagramie przypadków użycia, powinien wprowadzić je w zewnętrznym programie, a następnie ponownie wykonać import do systemu w celu uwzględnienia aktualizacji.

Położenie informacji o przypadkach użycia w programach może się różnić w zależności od konfiguracji eksportu. Położenia informacji o przypadkach użycia dla programów:

- Visual Paradigm

- Xml\_structure=„simple”:

- \* / Project / Models / UseCase
    - \* / Project / Models / Model[@Abstract='false'] / ModelChildren / UseCase
    - \* / Project / Models / Model[@Abstract='false'] / ModelChildren / System / ModelChildren / UseCase

- Xml\_structure=„traditional”:

- \* / Project / Models / Model[@composite='false'] / ChildModels / Model / ChildModels / Model[@modelType='UseCase']

- Enterprise Architect

- XMI = 1.0, UML = 1.3:

- \* / XMI / XMI.content / Model\_Management.Model / Foundation.Core.Namespace.ownedElement / Model\_Management.Package / Foundation.Core.Namespace.ownedElement / Behavioral\_Elements.Use\_Cases.UseCase

- XMI = 1.1, UML = 1.3:

- \* / XMI / XMI.content / UML:Model / UML:Namespace.ownedElement / UML:Package / UML:Namespace.ownedElement / UML:UseCase

- XMI = 1.2, UML = 1.4:

- \* / XMI / XMI.content / UML:Model / UML:Namespace.ownedElement / UML:UseCase

- XMI  $\geq$  2.1, UML  $\geq$  2.1:

- \* / xmi:XMI / uml:Model /packagedElement[@type='uml:Package'] / packageElement[@type='uml:UseCase']

- Eclipse Papyrus

- brak konfiguracji eksportu:
  - \* / uml:Model / packagedElement[@type='uml:UseCase']
- Sinvias UML
  - exporterVersion="1.0":
    - \* / xmi:XMI / UMLProject / UMLModel / UMLUseCase
- GenMyModel
  - XMI = 2.1:
    - \* / xmi:XMI / uml:Model / packagedElement[@type='uml:UseCase']
    - \* / xmi:XMI / uml:Model / packagedElement[@type='uml:Package'] / packageElement[@type='uml:UseCase']

Położenie informacji o relacjach w przypadkach użycia jest następujące:

- Visual Paradigm
  - / Project / Models / ModelRelationshipContainer / ModelChildren / ModelRelationshipContainer / ModelChildren / Include
  - / Project / Models / ModelRelationshipContainer / ModelChildren / ModelRelationshipContainer / ModelChildren / Extend
  - / Project / Models / ModelRelationshipContainer / ModelChildren / ModelRelationshipContainer / ModelChildren / Generalization
- Eclipse Papyrus
  - / uml:Model / packagedElement / include
  - / uml:Model / packagedElement / extend
  - / uml:Model / packagedElement / generalization
- GenMyModel
  - / xmi:XMI / uml:Model / packagedElement / include
  - / xmi:XMI / uml:Model / packagedElement / extend
  - / xmi:XMI / uml:Model / packagedElement / generalization

W stworzonym systemie udostępniono plik *config.json*, który zawiera wymienione wcześniej ścieżki do informacji dotyczących przypadków użycia. Plik ten umożliwia łatwe konfigurowanie systemu i dostosowywanie go do różnych programów o różnych strukturach. W przypadku, gdy istnieje potrzeba skorzystania z innego programu, który posiada inną strukturę danych, istnieje możliwość dodania dodatkowych ścieżek w pliku *config.json*. Te dodatkowe ścieżki zostaną uwzględnione podczas procesu pobierania informacji o diagramie przypadków użycia. Dzięki temu system jest elastyczny i można go łatwo dostosować do różnych narzędzi i formatów.

## 2.4. Systemy dowodzenia twierdzeń

W dziedzinie inżynierii wymagań, systemy dowodzenia twierdzeń odgrywają istotną rolę. Są to programy komputerowe, które automatycznie przeprowadzają dowody logiczne na podstawie zdefiniowanych aksjomatów i reguł wnioskowania. Wykorzystanie tych narzędzi znajduje szerokie zastosowanie w różnych dziedzinach i umożliwia przeprowadzenie procesu formalnej weryfikacji, co stanowi część systemu *Formal Specification IDE*.

W tej sekcji omówione zostaną trzy popularne systemy dowodzenia twierdzeń: Prover9, SPASS Prover oraz InKreSAT. Dwa pierwsze operują na logice pierwszego rzędu, co oznacza, że mogą dowodzić twierdzenia wzbogacone o predykaty, zmienne, kwantyfikatory i relacje. Trzeci system opiera się na zadaniowej logice temporalnej (ang. *Propositional Linear Temporal Logic*). Jest to odmiana logiki LTL, która jest oparta na rachunku zdań.

W kolejnych podsekcjach przedstawiona zostanie charakterystyka każdego z tych systemów oraz opisany będzie sposób ich integracji z narzędziem *Formal Specification IDE*.

### 2.4.1. Dowodzenie formuł logicznych

Dowodzenie formuł logicznych odgrywa kluczową rolę w analizie i weryfikacji systemów informatycznych. Pozwala na matematyczne potwierdzenie prawdziwości bądź fałszywości danego zdania logicznego poprzez logiczne wnioskowanie. W dziedzinie inżynierii oprogramowania, formalna weryfikacja jest niezwykle ważna, ponieważ umożliwia dowiedzenie poprawności i bezpieczeństwa systemu przed jego wdrożeniem.

Książka „Logic in Computer Science: Modelling and Reasoning about Systems” [16] autorstwa Michaela Hutha i Marka Ryana omawia techniki dowodzenia formuł logicznych oraz metody stosowane w systemach dowodzenia twierdzeń, zwłaszcza w kontekście informatyki i modelowania systemów. W ramach dowodzenia formuł logicznych autorzy opierają się na formalnych zasadach logiki matematycznej, które umożliwiają logiczne wnioskowanie, a także dowodzenie poprawności i prawdziwości zdaniowej wyrażeń.

Ważnym elementem dowodzenia formuł logicznych jest wykorzystanie aksjomatów i reguł wnioskowania. Aksjomaty są podstawowymi założeniami logicznymi, które są akceptowane jako prawdziwe bez dowodu. Reguły wnioskowania natomiast pozwalają na wyprowadzanie nowych twierdzeń na podstawie istniejących założeń i aksjomatów.

Przykładami reguł wnioskowania są *modus ponens* (jeśli A jest prawdziwe i z A wynika B, to B jest prawdziwe) i *modus tollens* (jeśli z A wynika B, a B nie jest prawdziwe, to nie jest prawdziwe A).

Przykłady dowodzenia formuł logicznych mogą być bardzo różnorodne. Poniżej zostaną przedstawione dwa podstawowe przypadki z logiki pierwszego rzędu. W omawianych przykładach zdania logiczne zostaną dowodzone przy użyciu aksjomatów i reguł wnioskowania.

**Przykład 1:** Dowód prostej implikacji

Rozważmy zdanie logiczne: „Jeśli pada deszcz, to ulice są mokre.” Oznaczmy je jako  $p \rightarrow q$ , gdzie  $p$  oznacza zdanie „pada deszcz”, a  $q$  oznacza zdanie „ulice są mokre”.

Dowodem implikacji  $p \rightarrow q$  będzie pokazanie, że jeśli zostanie założona prawdziwość zdania  $p$  (czyli założenie, że pada deszcz), to można wnioskować, że zdanie  $q$  (czyli „ulice są mokre”) również jest prawdziwe. Wykorzystany zostanie dowód za pomocą dedukcji naturalnej.

**Dowód:**

1. Założmy, że  $p$  jest prawdziwe. (Założenie)
2. Z reguły implikacji (*modus ponens*): Jeśli  $p$  jest prawdziwe, to  $q$  jest prawdziwe. (Reguła wnioskowania)
3. Wnioskujemy, że  $q$  jest prawdziwe. (Wnioskowanie z kroku 1 i 2)

W ten sposób został przeprowadzony dowód implikacji  $p \rightarrow q$ . Założono prawdziwość zdania  $p$  (pada deszcz) i wykorzystując regułę wnioskowania, stwierdzono, że zdanie  $q$  (ulice są mokre) musi być również prawdziwe.

**Przykład 2:** Na podstawie przykładu z książki [16]

Rozważmy zdanie logiczne: Jeżeli pociąg przybył późno i na stacji nie było żadnych taksówek, to spowodowało to spóźnienie Johna na spotkanie. Jednakże John nie spóźnił się na spotkanie. Dodatkowo, pociąg rzeczywiście przyjechał późno. W związku z tym, na stacji były taksówki.

Dowodzenie tego przykładu można przeprowadzić przy użyciu reguł wnioskowania *modus ponens* oraz negacji.

**Dowód:**

1. Założenie: Jeżeli pociąg przyjedzie późno i nie będzie żadnych taksówek na stacji, to John spóźni się na spotkanie. (formuła:  $(p \wedge \neg q) \rightarrow r$ )

2. Założenie: John nie spóźnił się na spotkanie. (formuła:  $\neg r$ )
3. Założenie: Pociąg rzeczywiście przyjechał późno. (formuła:  $p$ )
4. Założenie:  $\neg q$  (brak taksówek na stacji)
5. Z reguły wnioskowania modus ponens (1, 4):  $r$  (John spóźni się na spotkanie)
6. Z reguły wnioskowania negacji (2, 5): sprzeczność
7. Ze sprzeczności (6) wynika, że założenie  $\neg q$  (brak taksówek na stacji) było błędne.
8. Z reguły wnioskowania negacji (7):  $\neg\neg q$  (na stacji były taksówki, czyli  $q$ ).

Wynika stąd, że na stacji były taksówki (formuła  $q$ ), co zostało udowodnione na podstawie danych założeń.

Dowodzenie formuł logicznych można przeprowadzać ręcznie, ale w dzisiejszych czasach coraz częściej stosuje się do tego celu specjalne oprogramowanie, takie jak systemy Prover9, SPASS Prover oraz InKreSAT, które zostaną dokładnie omówione w tej sekcji. Te programy automatycznie przeprowadzają dowody logiczne na podstawie zadanych aksjomatów i reguł wnioskowania. Wykorzystanie tych narzędzi jest szczególnie przydatne w przypadku skomplikowanych i rozległych systemów, gdzie ręczne dowodzenie może być bardzo czasochłonne i podatne na błędy.

Ważną cechą systemów dowodzenia twierdzeń jest również ich zdolność do obsługi logik różnych rzędów oraz różnych odmian, takich jak logika temporalna czy logika pierwszego rzędu. Pozwala to na elastyczne przeprowadzanie dowodów dla różnych typów systemów i wymagań.

Dowodzenie formuł logicznych jest jednym z kluczowych narzędzi w formalnej weryfikacji systemów informatycznych. Daje pewność, że system spełnia określone założenia i wymagania, co jest niezwykle istotne w dziedzinie inżynierii oprogramowania. Wykorzystanie zaawansowanych narzędzi i technik, takich jak systemy dowodzenia twierdzeń, pozwala inżynierom tworzyć bardziej niezawodne i bezpieczne systemy informatyczne.

## 2.4.2. Charakterystyka Prover9

Prover9[17] to zautomatyzowane narzędzie do dowodzenia twierdzeń w logice pierwszego rzędu, stworzone przez Williama McCune'a. Jest on następcą innego narzędzia do dowodzenia o nazwie Otter, również autorstwa McCune'a. Prover9 został opracowany wraz z systemem Mace4, który służy do poszukiwania skończonych modeli formuł logicznych pierwszego rzędu. Oba programy mogą działać jednocześnie na tych samych danych wejściowych. Mace4 próbuje znaleźć kontrprzykłady dla zadanych tez, podczas gdy Prover9 stara się znaleźć dowód ich

prawdziwości. Zarówno Prover9, jak i wiele innych narzędzi do dowodzenia twierdzeń, zostały zbudowane na bazie biblioteki LADR, co ułatwiło ich implementację. Prover9 jest ceniony za generowanie czytelnych dowodów oraz udzielanie przydatnych wskazówek.

W przeszłości Prover9 był dostępny do instalacji w systemie Windows, jednak obecnie nie udostępnia się oficjalnie archiwów umożliwiających taką instalację. Obecnie jedyną opcją jest uruchomienie Prover9 poprzez użycie linuksowych archiwów, dostępnych na oficjalnej stronie narzędzia. Te archiwa są kompatybilne z każdą dystrybucją systemu Linux. Z kolei narzędzie *Formal Specification IDE* zostało stworzone z myślą o systemie Windows. Problem integracji z Prover9 został rozwiązany poprzez użycie konteneryzacji Docker, co zostanie szczegółowo opisane w dalszej części pracy (sekcja 5.6).

Przykładem prostego zastosowania Prover9 jest dowodzenie tezy: „Wszyscy ludzie są śmiertelni, a Sokrates jest człowiekiem, więc Sokrates jest śmiertelny”. Ten przykład przedstawiony w formacie Prover9 prezentuje się następująco:

```
formulas(assumptions).
    man(x) -> mortal(x).
    man(socrates).
end_of_list.

formulas(goals).
    mortal(socrates).
end_of_list.
```

Format Prover9 jest bardzo zbliżony do prostego matematycznego zapisu formuł logicznych. Aby Prover9 działał poprawnie, dane wejściowe muszą być odpowiednio przygotowane, aby program mógł je odczytać.

Znaczenie	Symbol	Przykład
negacja	-	(-p)
alternatywa		(p   q   r)
koniunkcja	&	(p & q & r)
implikacja	->	(p -> q)
implikacja wsteczna	<-	(p <- q)
równoważność	<->	(p <-> q)
dla każdego	all	(all x all y p(x, y))
istnieje	exists	(exists x exists y p(x,y))

**Tabela 2.1.** Wykaz kwantyfikatorów akceptowanych przez Prover9.

Prover9 po otrzymaniu odpowiednio przygotowanych danych rozpoczyna działanie i poszukuje dowodu. Po zakończeniu pracy udostępnia statystyki dotyczące wykonania. Prover9 próbuje znaleźć dowód poprzez negację wcześniej wprowadzonych założeń. Jeśli znajdzie sprzeczność, udowadnia to, co zostało wprowadzone jako cel.

W celu korzystania z Prover9 w narzędziu *Formal Specification IDE* konieczne było rozwiązywanie problemów związanych z instalacją. Szczegóły dotyczące rozwiązywanych problemów z instalacją Prover9 zostaną opisane w dalszej części pracy (sekcja 5.6.2). W celu ułatwienia integracji narzędzia z Prover9 zdecydowano się na wykorzystanie konteneryzacji Docker, co zostanie szczegółowo opisane w sekcji 5.6.

### 2.4.3. Charakterystyka SPASS Prover

SPASS Theorem Prover[18] jest narzędziem do automatycznego dowodzenia twierdzeń w rachunku predykatów pierwszego rzędu. Jego celem jest rozstrzygnięcie, czy dane twierdzenie jest dowodliwe. Każdy krok dowodu realizowany przez SPASS Prover musi wynikać jasno z poprzedniego kroku lub przyjętego przez użytkownika aksjomatu.

Narzędzie to zostało stworzone i udostępnione przez naukowców z Max-Planck Institute for Informatics w Niemczech. Obecnie najnowszą dostępną wersją provera jest wersja 3.9.

Wcześniejsze wersje SPASS Prover były dostępne do instalacji w systemie Windows, jednak najnowsza wersja jest dostępna wyłącznie na systemy uniksowe. W celu integracji SPASS Prover z systemem *Formal Specification IDE*, użyto konteneryzacji Docker, co zostało szczegółowo opisane w dalszej części pracy (sekcja 5.6.1).

Wejściem SPASS Prover są odpowiednio przekształcone formuły logiki pierwszego rzędu. Proces dowodzenia rozpoczyna się od podania kilku informacji na temat problemu, takich jak nazwa, autor, status i opis.

```
begin_problem(Driver).  
  
list_of_descriptions.  
    name({*Driver and car*}).  
    author({*FP & ML*}).  
    status(unsatisfiable).  
    description({* Car will never move unless driver is inside *}).  
end_of_list.
```

Następnie w liście symboli należy wpisać wszystkie znane niestandardowe predykaty i funkcje. W tym miejscu można zadeklarować symbole, takie jak kwantyfikatory czy operatory, które nie są predefiniowane i będą używane w aksjomatach. Zadeklarowane symbole nie mogą się powtarzać. W tym miejscu można sformułować predykaty umożliwiające logiczny opis problemu.

```

list_of_symbols.

functions[ (t1,0) , (t2,0) ].

predicates[ (Autostoi,1) , (Kierowcawaucie,1) , (Greaterequal,2) ].

end_of_list.

```

Korzystając z wcześniej zadeklarowanych symboli oraz standardowych kwantyfikatorów i operatorów, można podać listę znanych aksjomatów, które są prawdziwe.

```

list_of_formulae(axioms).

formula(Greaterequal(t2,t1)).

formula(not(Kierowcawaucie(t1))).

formula(forall([d,t],implies(and(Greaterequal(d,t),not(Kierowcawaucie(t
))),Autostoi(d)))).

end_of_list.

```

Następnie, w postaci formuł, podaje się wszystkie przypuszczenia, które mają być udowodnione.

```

list_of_formulae(conjectures).

formula(Autostoi(t2)).

end_of_list.

```

Po przekazaniu odpowiednio przygotowanego wejścia SPASS Prover rozpoczyna proces wnioskowania. Celem tego procesu jest wykazanie, że koniunkcja wszystkich aksjomatów implikuje dysjunkcję wszystkich przypuszczeń.

Format SPASS Provera został wprowadzony przez członków DFG-Schwerpunktprogramm Deduktion[19]. Jest to format, który można łatwo analizować i stanowi kompromis między potrzebami różnych grup. Pozwala na formuły bezklauzulowe i posortowane, kilka formatów dowodowych, a także na definiowanie operatorów przez użytkownika. Pełny opis wszystkich możliwości jest dostępny na stronie SPASS Prover[20]. Aby SPASS Prover działał poprawnie, dane wejściowe muszą być odpowiednio przygotowane, tak aby program mógł je odczytać.

Znaczenie	Symbol	Przykład
negacja	not	not(p)
alternatywa	or	or(p, q, r)
koniunkcja	and	and(p, q, r)
implikacja	implies	implies(p, q)
implikacja wsteczna	implied	implied(p, q)
równoważność	equiv	equiv(p, q)
dla każdego	forall	forall([T], and(p(T), q(T)))
istnieje	exists	exists([T], p(T))

**Tabela 2.2.** Wykaz kwantyfikatorów akceptowanych przez SPASS Prover.

W trakcie działania SPASS Prover parsuje dane wejściowe i sprowadza je do koniunkcyjnej postaci normalnej, czyli takiej, która jest równoważna formule wejściowej, ale zapisana tylko i wyłącznie jako koniunkcja klauzul (usuwane są wszystkie implikacje, równoważności itp.). Koniunkcje te są następnie negowane, ponieważ SPASS dowodzi poprzez negację.

Następnie SPASS analizuje przekształcony problem i dobiera odpowiednie opcje, które umożliwiają analizę problemu. W zależności od charakterystyki dostarczonego problemu, SPASS może zastosować różne strategie dowodzenia, sortować klauzule, redukować lub upraszczać formuły itp.

Na wyjściu programu użytkownik otrzymuje wynik analizy, który zawiera charakterystykę problemu przez SPASS oraz listę wybranych przez prover opcji. SPASS dobiera strategię dowodzenia w zależności od charakterystyki dostarczonego problemu.

Dowodzenie problemu może zakończyć się na trzy różne sposoby:

- Znaleziono dowód. W takim przypadku na wyjściu pojawia się komunikat „Proof found”.
- Jeśli formuła zawiera błędy semantyczne lub nie można udowodnić poprawności przypuszczeń, na wyjściu pojawia się komunikat „Completion found”.
- W przypadku błędu składniowego, informowany jest użytkownik o linii, w której wystąpił błąd.

Instalacja SPASS Prover przebiegła bezproblemowo. Szczegóły tej instalacji zostały opisane w dalszej części pracy (sekcja 5.6).

#### **2.4.3.1. Formaty i różnice między Prover9, a SPASS Prover**

Oba narzędzia, Prover9 i SPASS Prover, operują na logice pierwszego rzędu, ale różnią się w sposobie zapisu formuł logicznych oraz w nazwach i składni używanych do reprezentowania aksjomatów i celów.

Format Prover9 jest oparty na formacie TPTP (ang. *Thousands of Problems for Theorem Provers*). Jest to powszechnie używany format danych wejściowych dla wielu narzędzi do dowodzenia twierdzeń. Pozwala na zapisywanie aksjomatów, celów, symboli, kwantyfikatorów i operatorów logicznych w sposób łatwy do analizy i przetwarzania przez narzędzia automatycznego dowodzenia. Prover9 oczekuje, że dane wejściowe zostaną przygotowane zgodnie z określonymi zasadami i wykorzystuje własną składnię do reprezentacji formuł logicznych.

Z kolei SPASS Prover używa własnego formatu zapisu problemu logicznego. Format ten został zaprojektowany przez członków DFG-Schwerpunktprogramm Deduktion. Zapis formuł logicznych w SPASS Prover również jest zrozumiały dla użytkowników, ale nie jest zgodny z TPTP.

Różnice w zapisie formuł logicznych między Prover9, a SPASS Prover wymagają odpowiedniego przekształcenia danych wejściowych przed przekazaniem ich do odpowiedniego

narzędzia. Narzędzie *Formal Specification IDE* musi wykonać konwersję formuł między formatami, aby możliwe było użycie zarówno Prover9, jak i SPASS Prover do przeprowadzania dowodów logicznych.

W dystrybucji SPASS Prover znajduje się narzędzie o nazwie dfg2otter, które umożliwia konwersję formuł logicznych ze składni SPASS Prover na format zrozumiały dla Prover9. Dzięki temu narzędziu użytkownicy, którzy mają swoje problemy logiczne zapisane w formacie SPASS Prover, mogą łatwo skonwertować je na format akceptowany przez Prover9 i przeprowadzić dowód logiczny za pomocą tego narzędzia. Po przekonwertowaniu formuł, użytkownik otrzymuje plik wyjściowy zawierający formuły w formacie Prover9, gotowe do przeprowadzenia dowodu.

#### 2.4.4. Charakterystyka InKreSAT

InKreSAT[21] jest proverem dla logik modalnych K, T, K4, S4. Jego głównym celem jest rozwiązywanie problemu spełnialności formuł modalnych poprzez redukcję ich do problemu spełnialności logicznej (SAT) i wykorzystanie solvera SAT do rozwiązywania tego problemu. InKreSAT wykorzystuje translację i wywołania solvera SAT w sposób zintegrowany, korzystając z informacji zwrotnych dostarczanych przez solver SAT w celu kierowania procesem translacji. Dzięki temu osiąga się lepszą wydajność i możliwość integracji z mechanizmami blokującymi znanych z modalnych programów typu tableau.

Główna idea działania InKreSAT polega na analizie wyników solvera SAT. Jeśli solver SAT zwraca wynik niezadowalający, oznacza to, że początkowy problem jest niezadowalający, więc dalsze przetwarzanie nie jest konieczne. Natomiast jeśli solver SAT zwraca model zdaniowy częściowego kodowania formuły modalnej, jest on wykorzystywany przez InKreSAT do kierowania dalszymi krokami translacji.

InKreSAT został zaimplementowany w języku OCaml i korzysta z solvera SAT o nazwie MiniSat[22] w wersji v2.2.0. Kod źródłowy InKreSAT jest dostępny na stronie <https://www.ps.uni-saarland.de/kaminski/inkresat/>. W trakcie procesu budowania programu InKreSAT, zostały napotkane pewne problemy, wynikające z faktu, że system ten został stworzony ponad 10 lat temu. Rozwiązywanie programów instalacyjnych zostało szczegółowo opisane w sekcji 5.6.2.

InKreSAT jest używany w systemie *Formal Specification IDE* do przeprowadzania formalnej weryfikacji formuł liniowej logiki temporalnej. Jednakże należy zauważyć, że system InKreSAT jest dostępny wyłącznie na systemy Linux. W celu integracji InKreSAT z *Formal Specification IDE* zastosowano technikę konteneryzacji, co jest dokładnie opisane w sekcji 5.6.1.

Znaczenie	Symbol	Przykład
negacja	$\sim$	$\sim p$
alternatywa	$ $	$p   q$
koniunkcja	$\&$	$p \& q$
implikacja	$\rightarrow$	$p \rightarrow q$
równoważność	$\leftrightarrow$	$p \leftrightarrow q$
dla każdego	$[]$	$[](a \& b)$
istnieje	$\langle\rangle$	$\langle\rangle p$

**Tabela 2.3.** Wykaz kwantyfikatorów akceptowanych przez InKreSAT.

Wejście dla InKreSAT stanowi lista formuł liniowej logiki temporalnej, która ma zostać zweryfikowana, i jest zawarta między słowami kluczowymi *begin*, a *end*. Poniżej przedstawiamy przykładowe wejście dla InKreSAT:

```
begin
[]<>~p2
& (~[]<>p2 | <>(p2 & <>~p3) )
& [] (~p2 | ~<>(~ p2 | []p3))
& []~[] (~ p2 | []p3)
end
```

Format wejścia dla InKreSAT został wprowadzony przez autorów oprogramowania. Pełny opis wszystkich możliwości znajduje się na stronie InKreSAT: <https://www.ps.uni-saarland.de/kaminski/inkresat/>. Aby InKreSAT działał poprawnie, dane wejściowe muszą być odpowiednio przygotowane, aby program mógł je odczytać.

Wynikiem działania InKreSAT jest odpowiedź na pytanie o spełnialność formuł liniowej logiki temporalnej. Użytkownik otrzymuje wynik analizy, który zawiera statystyki problemu oraz informację, czy formuła jest *satisfiable* (spełnialna) lub *unsatisfiable* (niespełnialna).



## **3. Opis rozwiązania i przyjętej metodologii**

W niniejszym rozdziale przedstawione zostanie rozwiązanie w postaci aplikacji o nazwie *Formal Specification IDE*, które ma na celu wspomaganie procesu tworzenia modeli inżynierii wymagań oraz ich formalnej weryfikacji. Aplikacja umożliwia generowanie formuł logicznych w logikach FOL (*First-Order Logic*) i LTL (*Linear Temporal Logic*) na podstawie zdefiniowanych i uzupełnionych scenariuszy przypadków użycia oraz diagramów aktywności. Wygenerowane formuły logiczne mogą być następnie poddane formalnej weryfikacji przy użyciu zintegrowanych proverów, takich jak *Prover9*, *SPASS Prover* i *InKreSAT*. Zastosowanie formalnej logiki do specyfikacji i weryfikacji wymagań pozwala na bardziej precyzyjne i jednoznaczne opisanie oczekiwania dotyczących systemu oraz wykrywanie potencjalnych problemów i sprzeczności już na etapie projektowania.

W ramach tego rozdziału zostaną przedstawione opisy systemu oraz przyjętej metodologii, które składają się z kilku etapów: modelowanie diagramów przypadków użycia, tworzenie scenariuszy przypadków użycia, modelowanie diagramów aktywności, generowanie specyfikacji logicznej, generowanie kodu oraz formalna weryfikacja formuł logicznych. Każdy z tych etapów ma na celu zapewnienie precyzyjnego i jednoznacznego opisu wymagań oraz możliwość ich weryfikacji pod kątem poprawności logicznej.

### **3.1. Opis systemu**

System *Formal Specification IDE* jest zaawansowanym środowiskiem programistycznym, które umożliwia inżynierom oprogramowania i projektantom pracę nad specyfikacją formalną systemu. Głównym celem systemu jest wspomaganie procesu tworzenia i weryfikacji wymagań, zarówno funkcjonalnych, jak i niefunkcjonalnych, oraz generowanie kodu na podstawie zdefiniowanych specyfikacji.

System oferuje interfejs graficzny, który umożliwia tworzenie projektów, do których można importować pliki w formacie .xml, reprezentujące diagramy przypadków użycia. Pliki takie mogą być wcześniej stworzone przy użyciu jednego z dostępnych programów do modelowania

diagramów przypadków użycia, takich jak: *Visual Paradigm*, *Enterprise Architect*, *Eclipse Parapyrus*, *Sinvas UML*, *GenMyModel*. Szczegółowy opis dostępnych programów został zawarty w sekcji 2.3.

Aplikacja przetwarza wskazany plik .xml w celu wyodrębnienia informacji dotyczących przypadków użycia oraz relacji, takich jak zawieranie (ang. *include*), rozszerzanie (ang. *extend*) oraz generalizacja (ang. *generalization*). Istnieje również możliwość dołączenia pliku z grafiką przedstawiającą diagram, co umożliwia jego wizualizację. Następnie użytkownik może tworzyć scenariusze dla poszczególnych przypadków użycia. Przynajmniej jeden scenariusz główny jest obowiązkowy, ale można również tworzyć scenariusze alternatywne. Scenariusze są wypełniane tekstem, który ma na celu precyzyjne określenie odpowiednich czasowników reprezentujących atomiczne aktywności.

System umożliwia użytkownikom tworzenie i edycję scenariuszy przypadków użycia oraz diagramów aktywności. Diagramy aktywności pozwalają na przedstawienie różnych scenariuszy użycia systemu za pomocą zidentyfikowanych aktywności oraz zależności między nimi. Tworzenie diagramów aktywności odbywa się przy użyciu zdefiniowanych aktywności i predefiniowanych wzorców. Diagramy aktywności służą do przedstawienia przepływu sterowania i operacji w systemie, opisując konkretne kroki i akcje, które są wykonywane podczas realizacji przypadków użycia.

Na podstawie diagramów aktywności system generuje wyrażenie wzorcowe, które przekazywane jest do generatora formuł logicznych w celu wygenerowania specyfikacji logicznej w logikach FOL (*First-Order Logic*) i LTL (*Linear Temporal Logic*).

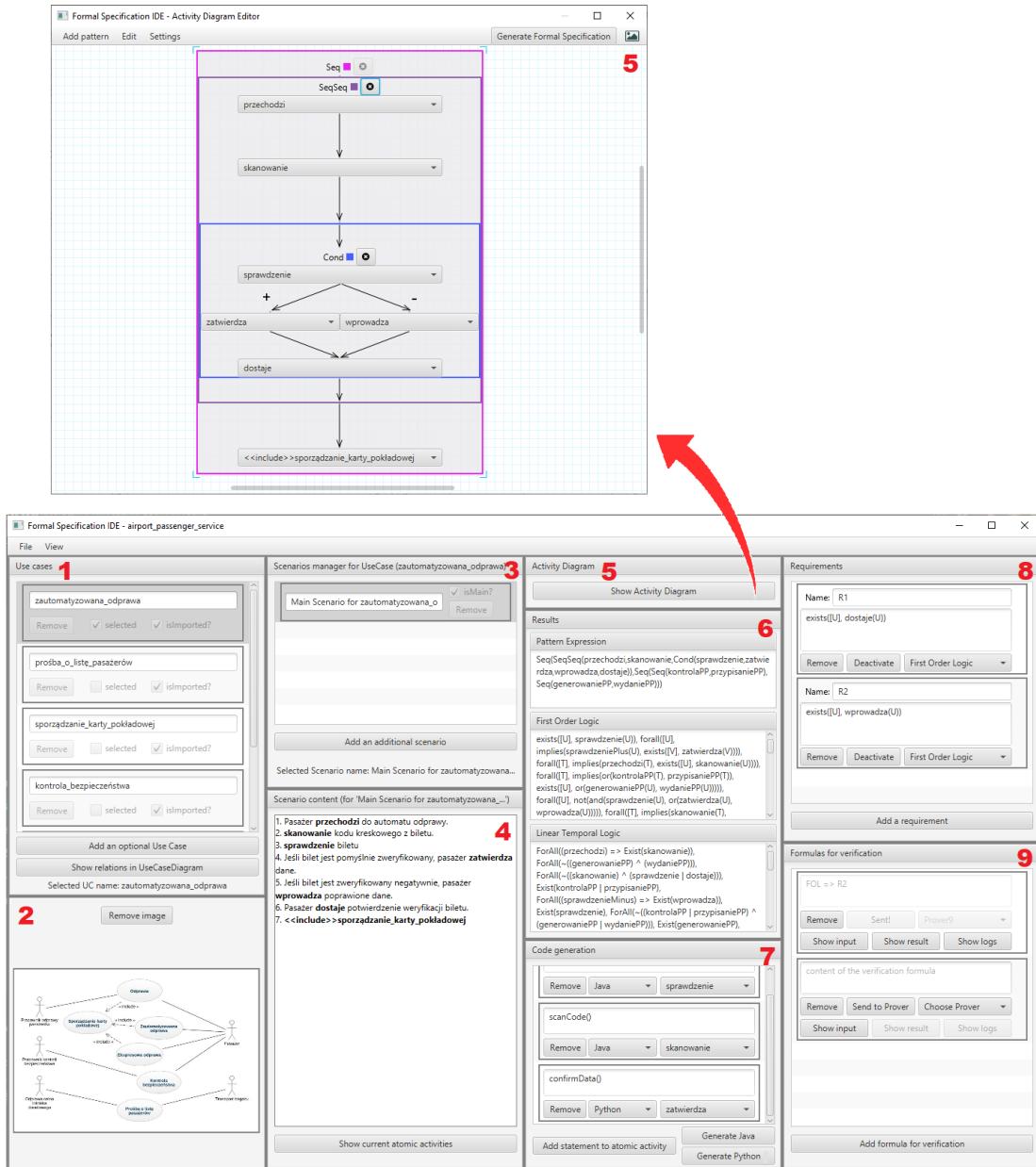
System umożliwia także formalną weryfikację wymagań przy użyciu zintegrowanych proverów. Zostały one zintegrowane z systemem *Formal Specification IDE* i obejmują provery: *Prover9*, *SPASS* i *InKreSAT*. Pozwalają one sprawdzić, czy określone wymagania są spełnione oraz czy nie naruszają określonych ograniczeń i reguł. Wyniki weryfikacji są prezentowane użytkownikowi w czytelnej formie, informującej o ewentualnych błędach i problemach w specyfikacji. Użytkownik ma również możliwość dodawania dodatkowych wymagań, które mogą zostać połączone z wygenerowanymi formułami logicznymi i poddane formalnej weryfikacji.

Dodatkowo, system obsługuje generowanie kodu w dwóch językach programowania na podstawie specyfikacji. Wystarczy wybrać docelowy język programowania, a system automatycznie generuje odpowiedni kod implementujący dane wyrażenie. Dostępne języki programowania to Java oraz Python.

System *Formal Specification IDE* dostarcza inżynierom oprogramowania narzędzia i funkcjonalności niezbędne do skutecznego tworzenia i weryfikacji specyfikacji formalnych. Dzięki temu możliwe jest poprawienie jakości tworzonego oprogramowania, minimalizacja błędów i ryzyka oraz zwiększenie efektywności procesu tworzenia systemów informatycznych.

## 3.2. Funkcjonalności systemu

Aplikacja *Formal Specification IDE* oferuje szeroki zakres funkcjonalności, umożliwiających efektywne tworzenie modeli inżynierii wymagań oraz ich formalną weryfikację.



Rys. 3.1. Układ graficznego interfejsu aplikacji *Formal Specification IDE*.

1. Tworzenie, zapisywanie i wczytywanie projektów: Użytkownicy mogą tworzyć nowe projekty, zapisywać ich stan oraz wczytywać wcześniej utworzone projekty. Ta funkcjonalność umożliwia łatwe przełączanie się między różnymi zagadnieniami i kontynuowanie pracy nad nimi.

2. Automatyczne wyodrębnianie przypadków użycia: System automatycznie analizuje pliki w formacie .xml, które zawierają diagramy przypadków użycia. Na podstawie tych plików aplikacja wyodrębnia przypadki użycia oraz identyfikuje relacje między nimi: *include*, *extend* i *generalization*. Panel z przypadkami użycia został oznaczony numerem 1 na rysunku 3.1. Dodatkowo istnieje możliwość importowania wizualizacji diagramu przypadków użycia, co zostało zaznaczone numerem 2 na rysunku 3.1.
3. Tworzenie scenariuszy przypadków użycia: Użytkownicy mogą tworzyć scenariusze dla poszczególnych przypadków użycia. Można dodawać kroki do scenariuszy oraz zaznaczać atomiczne aktywności. Scenariusze są edytowalne, umożliwiając elastyczne dostosowywanie ich treści. Panel z dostępnymi scenariuszami danego przypadku użycia został oznaczony numerem 3 na rysunku 3.1. Panel służący do uzupełniania kroków scenariuszy został oznaczony numerem 4 na rysunku 3.1.
4. Tworzenie diagramów aktywności: Aplikacja umożliwia tworzenie diagramów aktywności na podstawie zdefiniowanych atomicznych aktywności i predefiniowanych wzorców. Edytor diagramów jest interaktywny, umożliwiając użytkownikom modelowanie diagramów zgodnie z ich potrzebami. Wzorce mogą być wypełniane aktywnościami i mogą być zagnieździone w innych wzorcach. Panel służący do modelowania diagramów aktywności został oznaczony numerem 5 na rysunku 3.1. Po kliknięciu przycisku, zostaje wyświetcone dodatkowe okno do edycji.
5. Generowanie specyfikacji logicznej i formuł logicznych: System generuje specyfikację logiczną oraz formuły logiczne w logikach FOL (*First-Order Logic*) i LTL (*Linear Temporal Logic*) na podstawie utworzonych diagramów aktywności oraz dostarczonych plików z regułami FOL i LTL. Panel z wynikami generacji został oznaczony numerem 6 na rysunku 3.1.
6. Obsługa relacji *include*, *extend* i *generalization*: Podczas tworzenia scenariuszy, diagramów aktywności oraz generowania specyfikacji logicznej, aplikacja obsługuje i uwzględnia relacje *include*, *extend* i *generalization* między przypadkami użycia.
7. Generacja kodu: Na podstawie wyrażenia wzorcowego, system może generować kod w dwóch popularnych językach programowania: Java i Python. Istnieje również możliwość definiowania wyrażeń, które zastąpią określone aktywności podczas procesu generacji kodu. Panel generatora kodu został oznaczony numerem 7 na rysunku 3.1.
8. Definiowanie dodatkowych wymagań: Użytkownicy mogą zdefiniować dodatkowe wymagania, które zostaną uwzględnione w procesie weryfikacji. Te wymagania można połączyć z wygenerowanymi formułami logicznymi (FOL i LTL) i poddać weryfikacji za

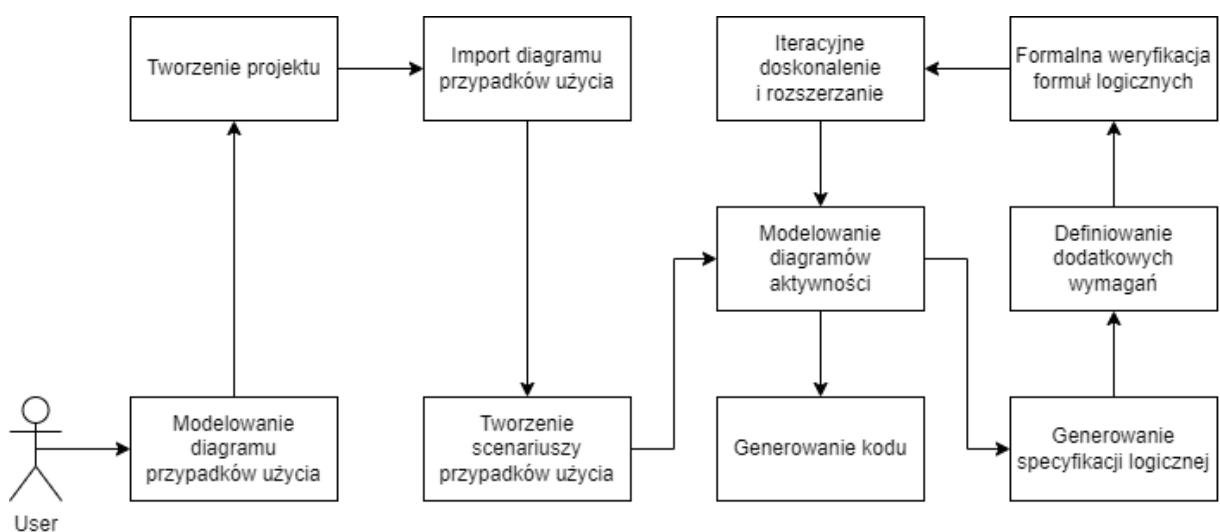
pomocą zintegrowanych proverów. Panel do definiowania dodatkowych wymagań został oznaczony numerem 8 na rysunku 3.1.

9. Formalna weryfikacja formuł logicznych: Wygenerowane formuły logiczne oraz dodatkowe wymagania mogą być poddane formalnej weryfikacji za pomocą istniejących proverów, takich jak *Prover9*, *SPASS* i *InKreSAT*. Ta funkcjonalność umożliwia sprawdzenie poprawności logicznej stworzonych modeli i wymagań. Panel do przeprowadzania formalnej weryfikacji został oznaczony numerem 9 na rysunku 3.1.

Dzięki wymienionym funkcjonalnościom, aplikacja *Formal Specification IDE* stanowi kompleksowe narzędzie wspomagające proces tworzenia i weryfikacji modeli inżynierii wymagań.

### 3.3. Metodologia rozwiązania

Metodologia rozwiązania oparta jest na iteracyjnym podejściu do tworzenia modeli inżynierii wymagań oraz formalnej weryfikacji. Ogólny przepływ pracy przedstawiony jest na rysunku 3.2.



Rys. 3.2. Graficzna reprezentacja przyjętej metodologii rozwiązania.

Szczegółowy opis poszczególnych etapów przedstawia się następująco:

1. **Modelowanie diagramu przypadków użycia:** Na początku procesu tworzenia modeli inżynierii wymagań, diagram przypadków użycia jest tworzony przy użyciu jednego z dostępnych programów do modelowania diagramów przypadków użycia (opisanych w sekcji 2.3).

2. **Tworzenie projektu:** Na etapie inicjalizacji korzystania z aplikacji *Formal Specification IDE*, tworzony jest nowy projekt. Użytkownik definiuje nazwę projektu oraz określa ustalenia projektu.
3. **Import diagramu przypadków użycia:** Następnie użytkownik importuje plik w formacie .xml, który zawiera diagram przypadków użycia. Plik ten powinien zostać wcześniej utworzony przy użyciu jednego z dostępnych programów do modelowania diagramów przypadków użycia. Aplikacja analizuje ten plik i wyodrębnia przypadki użycia oraz relacje między nimi, takie jak *include*, *extend* i *generalization*.
4. **Tworzenie scenariuszy przypadków użycia:** Użytkownik tworzy scenariusze dla poszczególnych przypadków użycia. Scenariusze składają się z kroków w postaci tekstu naturalnego. W ramach scenariuszy można zdefiniować zarówno scenariusz główny, jak i scenariusze alternatywne. Scenariusze są wypełniane tekstem, który ma na celu wyodrębnienie odpowiednich czasowników reprezentujących atomiczne aktywności.
5. **Modelowanie diagramów aktywności:** Na podstawie scenariuszy oraz dostępnych atomicznych aktywności i predefiniowanych wzorców, użytkownik tworzy diagramy aktywności. Edytor diagramów aktywności umożliwia interaktywne tworzenie diagramów i modelowanie zależności między aktywnościami. Wzorce mogą być zagnieżdżone w innych wzorach, co umożliwia tworzenie bardziej skomplikowanych diagramów.
6. **Generowanie specyfikacji logicznej:** Na podstawie utworzonych diagramów aktywności oraz reguł FOL i LTL, aplikacja generuje specyfikację logiczną. Specyfikacja logiczna opisuje model i wymagania w języku formalnym. Wygenerowane formuły logiczne są zgodne z predefiniowanymi wzorcami i odpowiadają opisanym scenariuszom i diagramom.
7. **Generowanie kodu:** Na podstawie wyrażenia wzorcowego, system generuje kod w wybranym języku programowania (Java lub Python). Wygenerowany kod implementuje opisane w modelu aktywności i jest gotowy do dalszego rozwoju i testowania.
8. **Definiowanie dodatkowych wymagań:** Użytkownik ma możliwość definiowania dodatkowych wymagań, które będą uwzględnione w procesie weryfikacji. Te wymagania mogą dotyczyć zarówno logiki FOL, jak i LTL.
9. **Formalna weryfikacja formuł logicznych:** Wygenerowane formuły logiczne oraz dodatkowe wymagania mogą być poddane formalnej weryfikacji za pomocą zintegrowanych proverów, takich jak *Prover9*, *SPASS* i *InKreSAT*. Prover sprawdza poprawność logiczną stworzonych modeli i wymagań, dostarczając odpowiedzi na pytania logiczne dotyczące spełniania określonych warunków.

10. **Iteracyjne doskonalenie i rozszerzanie:** Po przeprowadzeniu weryfikacji, użytkownik może analizować wyniki, dokonywać zmian w modelu, poprawiać błędy i rozwijać model dalej. Proces ten jest iteracyjny i może być powtarzany, aż do osiągnięcia oczekiwanych rezultatów.

Metodologia rozwiązania opisana powyżej zapewnia strukturalny i iteracyjny proces tworzenia modeli inżynierii wymagań oraz formalnej weryfikacji. Pozwala to na skuteczne zarządzanie wymaganiami i zapewnienie poprawności logicznej stworzonych modeli.



## 4. Prezentacja systemu

W tym rozdziale przedstawiona zostanie prezentacja systemu, z uwzględnieniem graficznego interfejsu użytkownika oraz ogólnego schematu działania systemu. Ponadto, przedstawione są również możliwości modelowania diagramu aktywności, generacji kodu źródłowego oraz procesu weryfikacji. W końcu, w rozdziale przedstawiono przykład zastosowania systemu na rozbudowanym scenariuszu.

### 4.1. Graficzny interfejs użytkownika

Graficzny interfejs użytkownika (GUI) jest kluczowym elementem systemu, umożliwiającym interakcję użytkownika z aplikacją. GUI systemu zostało zaprojektowane w sposób intuicyjny i przyjazny dla użytkownika, umożliwiając skuteczne wykorzystanie funkcji i narzędzi oferowanych przez system.

W graficznym interfejsie użytkownika można wyróżnić następujące panele:

- Panel z przypadkami użycia:** Ten panel prezentuje listę dostępnych przypadków użycia, umożliwiając użytkownikowi wybór konkretnego przypadku użycia do dalszej edycji i analizy.
- Panel z wizualizacją diagramu przypadków użycia:** Ten panel wyświetla graficzną reprezentację diagramu przypadków użycia wybranego przypadku użycia. Użytkownik może oglądać i analizować zależności między przypadkami użycia oraz ich strukturę.
- Panel do zarządzania scenariuszami danego przypadku użycia:** Ten panel umożliwia użytkownikowi zarządzanie scenariuszami dla danego przypadku użycia. Użytkownik może tworzyć, edytować, usuwać scenariusze oraz przełączać się między różnymi scenariuszami.
- Panel do tworzenia zawartości danego scenariusza oraz wskazywania atomicznych aktywności:** Ten panel umożliwia użytkownikowi tworzenie zawartości dla wybranego scenariusza przypadku użycia. Użytkownik może wprowadzać tekstowe opisy kroków oraz definiować atomiczne aktywności.

5. **Panel do modelowania diagramu aktywności:** Ten panel umożliwia użytkownikowi modelowanie diagramu aktywności na podstawie wybranego scenariusza. Użytkownik może dodawać, usuwać i modyfikować aktywności oraz ustalać ich kolejność i zależności.
6. **Panel do generowania kodu źródłowego:** Ten panel pozwala użytkownikowi na generowanie kodu źródłowego na podstawie modelowanego diagramu aktywności. Użytkownik może wybrać preferowany język programowania (Java lub Python) i wygenerować kod, który implementuje opisane w modelu aktywności.
7. **Panel do definiowania dodatkowych wymagań:** Ten panel umożliwia użytkownikowi definiowanie dodatkowych wymagań, które mogą zostać uwzględnione podczas procesu weryfikacji. Użytkownik może wprowadzać formuły w logice FOL (First Order Logic) lub LTL (Linear Temporal Logic).
8. **Panel do procesu weryfikacji:** Ten panel umożliwia użytkownikowi przeprowadzanie procesu formalnej weryfikacji na wygenerowanych formułach logicznych i dodatkowych wymaganiach. Użytkownik może wykorzystać zintegrowane prover, takie jak *Prover9*, *SPASS* i *InKreSAT*, aby sprawdzić poprawność logiczną modeli i wymagań.

Przepływ informacji między poszczególnymi panelami jest realizowany w określonej kolejności, umożliwiając użytkownikowi płynne przechodzenie między różnymi funkcjami systemu. Szczegółowy opis wszystkich części interfejsu graficznego znajduje się w kolejnych sekcjach tego rozdziału.

## 4.2. Schemat działania systemu

Schemat działania systemu opiera się na przyjętej metodologii rozwiązania, opisanej wcześniej w sekcji 3.3. Proces tworzenia modeli inżynierii wymagań oraz formalnej weryfikacji jest realizowany w sposób strukturalny i iteracyjny, zapewniając skuteczne zarządzanie wymaganiami i zapewnienie poprawności logicznej stworzonych modeli.

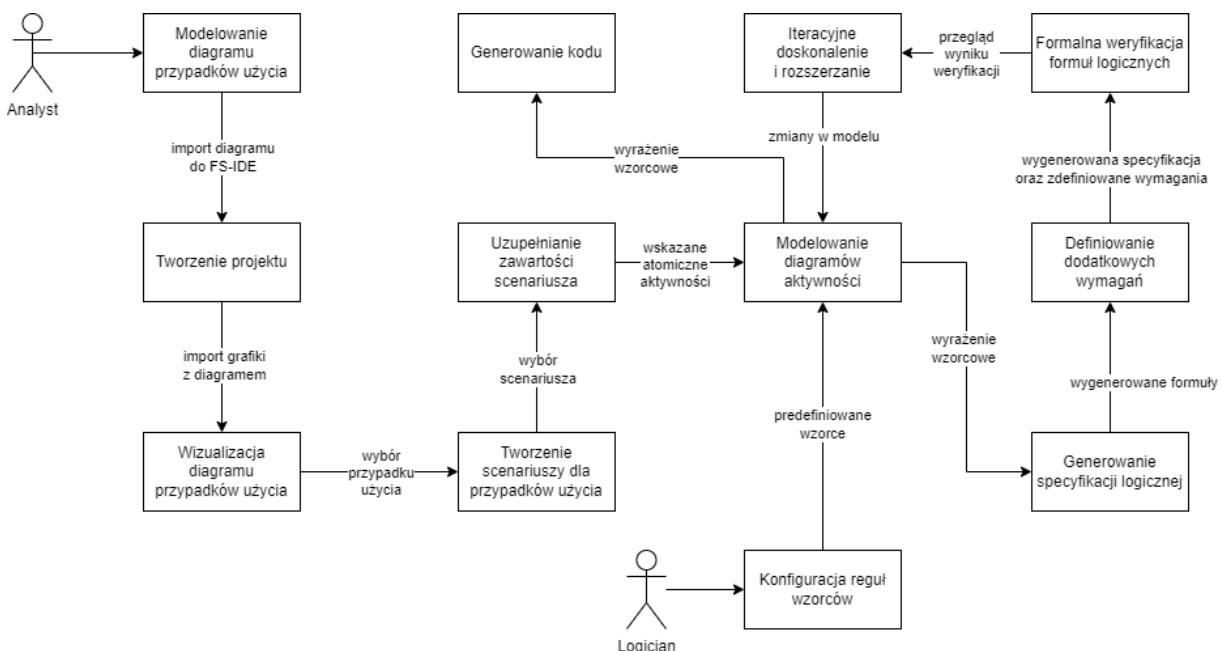
Początkowo użytkownik tworzy diagram przypadków użycia za pomocą jednego z dostępnych programów do modelowania diagramów przypadków użycia. Następnie tworzony jest nowy projekt w aplikacji *Formal Specification IDE*, gdzie użytkownik importuje wcześniej utworzony plik .xml zawierający diagram przypadków użycia.

Kolejnym krokiem jest tworzenie scenariuszy dla poszczególnych przypadków użycia oraz modelowanie diagramów aktywności na podstawie tych scenariuszy. Użytkownik ma możliwość definiowania dodatkowych wymagań, które będą uwzględnione podczas procesu weryfikacji.

Po zdefiniowaniu modelu i wymagań, system generuje specyfikację logiczną opisującą model i wymagania w języku formalnym. Na podstawie tej specyfikacji, system generuje kod źródłowy w wybranym języku programowania (Java lub Python).

Wygenerowane formuły logiczne oraz dodatkowe wymagania mogą być poddane formalnej weryfikacji za pomocą zintegrowanych proverów, które sprawdzają poprawność logiczną i spełnienie wymagań.

W przypadku wykrycia błędów lub niezgodności, użytkownik ma możliwość powrotu do wcześniejszych etapów procesu, aby wprowadzić odpowiednie zmiany i poprawić model lub wymagania.



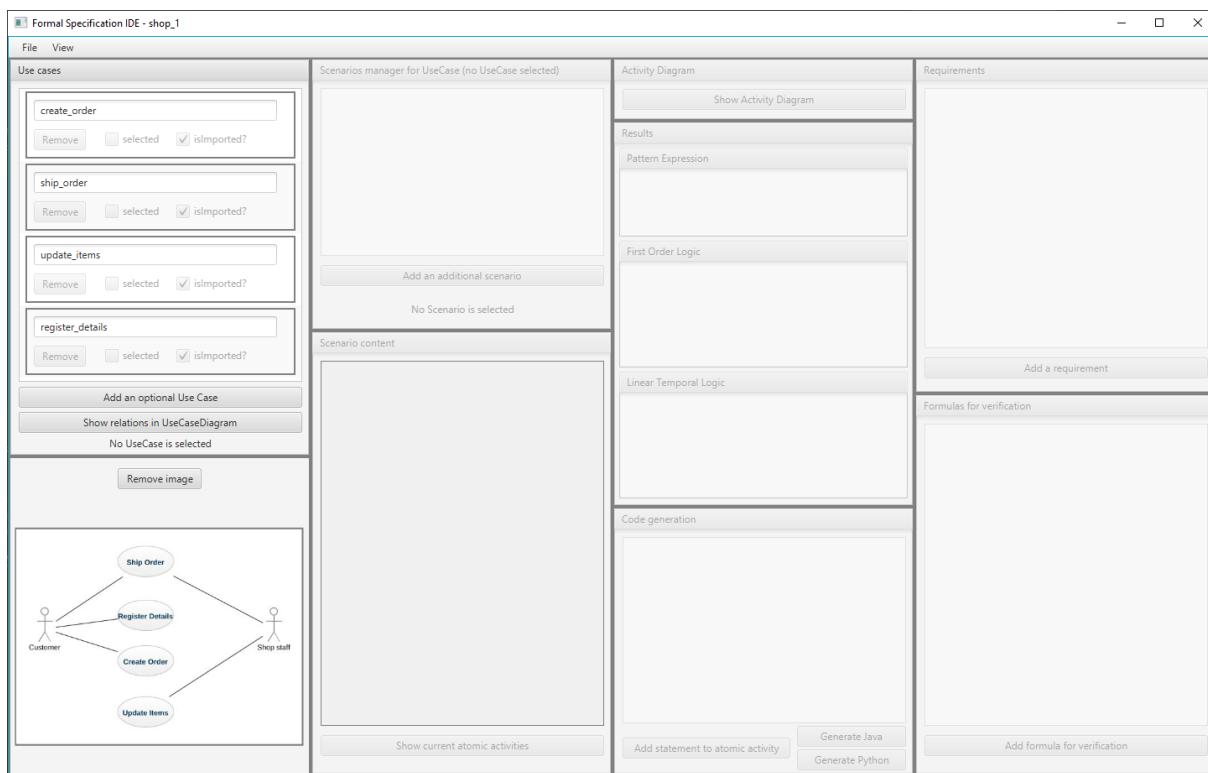
**Rys. 4.1.** Idea działania stworzonej aplikacji. Analityk modeluje diagram przypadków użycia, scenariusze, diagramy przypadków użycia. Na ich podstawie możliwa jest generacja specyfikacji logicznej oraz jej formalna weryfikacja.

### 4.3. Diagramy przypadków użycia

Po uruchomieniu aplikacji *Formal Specification IDE* użytkownik ma możliwość tworzenia nowego projektu. Aby to zrobić, należy wybrać opcję „File” w menu, a następnie „New”. Po tym kroku otwiera się okno wyboru pliku w formacie .xml, który zawiera diagramy przypadków użycia.

Diagram przypadków użycia może zostać stworzony przy użyciu jednego z dostępnych programów do modelowania takich diagramów, o których więcej informacji można znaleźć

w sekcji 2.3. Po wybraniu pliku do zainportowania, aplikacja prosi o podanie nazwy nowego projektu.



**Rys. 4.2.** Panel przypadków użycia znajduje się w skrajnie lewej części głównego okna aplikacji. Został stworzony projekt „shop\_1”, zimportowany plik .xml oraz grafika z wizualizacją diagramu.

Po pomyślnym zainportowaniu plików, aplikacja wyświetla listę przypadków użycia znalezionych w pliku .xml, gotowych do uzupełnienia. Panel z przypadkami użycia stanowi jedną z głównych części graficznego interfejsu użytkownika. Po zainportowaniu diagramów przypadków użycia, w tym panelu wyświetlane są wszystkie dostępne przypadki użycia wraz z ich nazwami.

Użytkownik może wybierać i edytować poszczególne przypadki użycia poprzez kliknięcie na ich nazwy w panelu. Dzięki temu panelowi użytkownik ma łatwy dostęp do wszystkich przypadków użycia w systemie i może nimi zarządzać w prosty sposób.

Dodatkowo, użytkownik ma możliwość zainportowania grafiki przedstawiającej diagram przypadków użycia za pomocą panelu do wizualizacji diagramu aktywności. Taka wizualizacja może pomóc w lepszym zrozumieniu i prezentacji struktury systemu oraz ułatwia zrozumienie struktury i relacji między poszczególnymi przypadkami użycia.

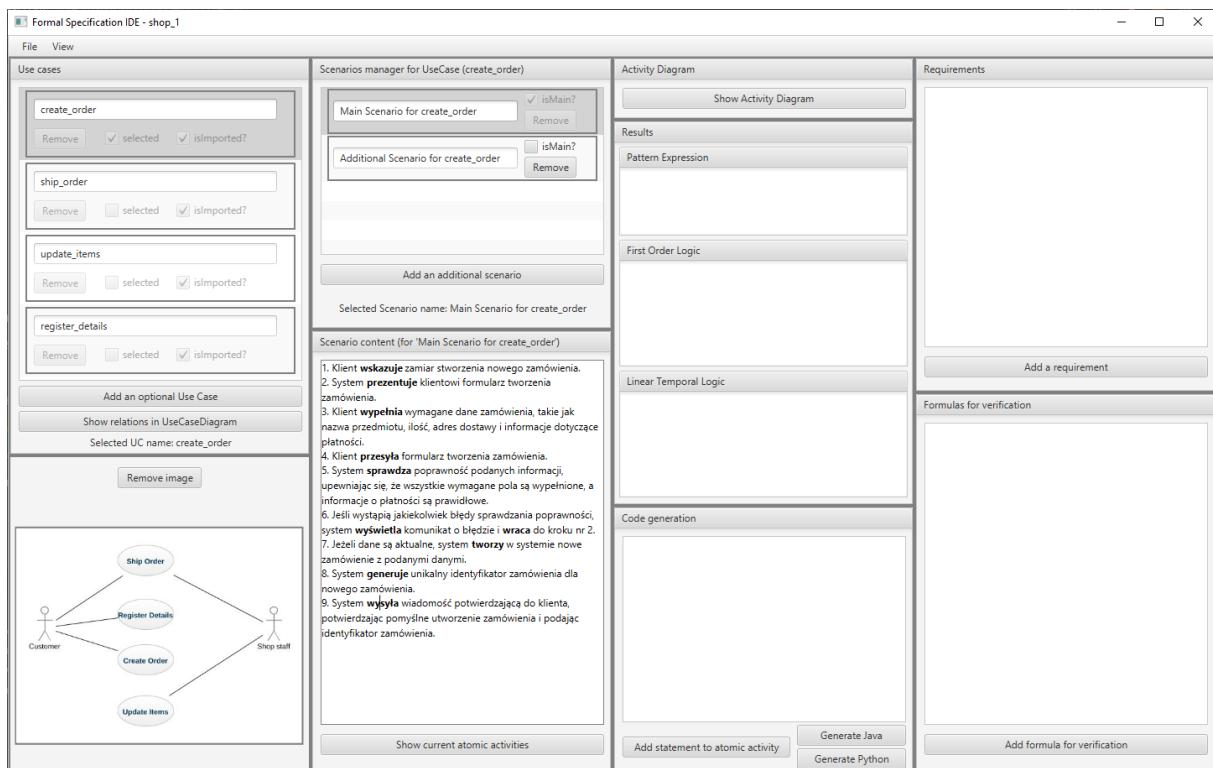
Zainportowana grafika zostaje przypisana do stworzonego projektu. Po kliknięciu na diagram, wyświetla się dodatkowe okno z możliwością przesuwania oraz powiększania/oddalania, aby lepiej zobrazować poszczególne elementy.

Panel z wizualizacją diagramu przypadków użycia umożliwia interaktywną nawigację po diagramie oraz dostarcza wizualnej reprezentacji struktury systemu.

## 4.4. Scenariusze przypadków użycia

Panel do zarządzania scenariuszami danego przypadku użycia pozwala użytkownikowi na tworzenie, edytowanie i usuwanie scenariuszy dla poszczególnych przypadków użycia.

Po wybraniu przypadku użycia z panelu z przypadkami użycia, w tym panelu wyświetlane są wszystkie dostępne scenariusze dla danego przypadku użycia wraz z ich nazwami.



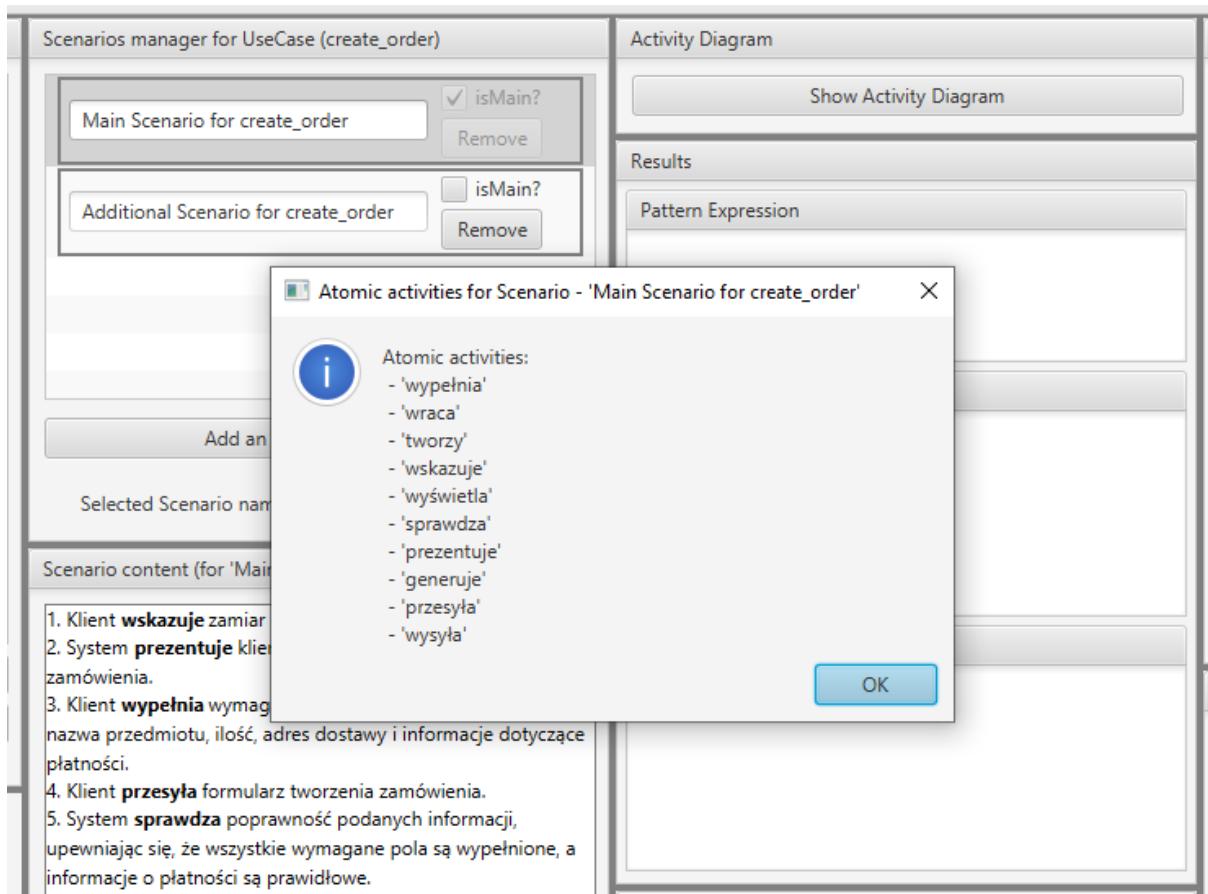
**Rys. 4.3.** Panel scenariuszy przypadków użycia znajduje się w drugiej kolumnie od lewej strony głównego okna aplikacji. Został stworzony przykładowy scenariusz dla „Create order” oraz uzupełniony krokami w postaci tekstu naturalnego. W scenariuszu zostały wskazane atomiczne aktywności.

Użytkownik może wybrać i edytować poszczególne scenariusze poprzez kliknięcie na ich nazwy w panelu. Panel umożliwia również dodawanie nowych scenariuszy oraz usuwanie istniejących.

Dla danego przypadku użycia domyślnie dodany jest jeden scenariusz główny, który jest obligatoryjny. Dodatkowo istnieje możliwość dodania dowolnej liczby scenariuszy alternatywnych. Scenariusza głównego nie można usunąć.

Dzięki temu panelowi użytkownik może tworzyć różne scenariusze dla danego przypadku użycia, modelować różne możliwe ścieżki działania i analizować zachowanie systemu w różnych sytuacjach.

Panel do tworzenia zawartości scenariusza oraz wskazywania atomicznych aktywności jest ważnym elementem graficznego interfejsu użytkownika. Pozwala użytkownikowi na definiowanie kroków scenariusza i powiązanie ich z odpowiednimi atomicznymi aktywnościami.



**Rys. 4.4.** Po kliknięciu przycisku „Show current atomic activities”, zostaje wyświetlone okno, w którym wyświetlona jest lista wszystkich atomicznych aktywności dla danego scenariusza.

Po wybraniu scenariusza z panelu do zarządzania scenariuszami, w tym panelu można edytować zawartość scenariusza poprzez dodawanie, usuwanie i modyfikowanie kroków. Każdy krok scenariusza może być opisany za pomocą tekstu naturalnego.

W panelu użytkownik ma również możliwość wskazania atomicznych aktywności, które są związane z danym krokiem scenariusza. Atomiczne aktywności reprezentują podstawowe, niepodzielne czynności w systemie. Z założenia atomiczna aktywność to czasownik.

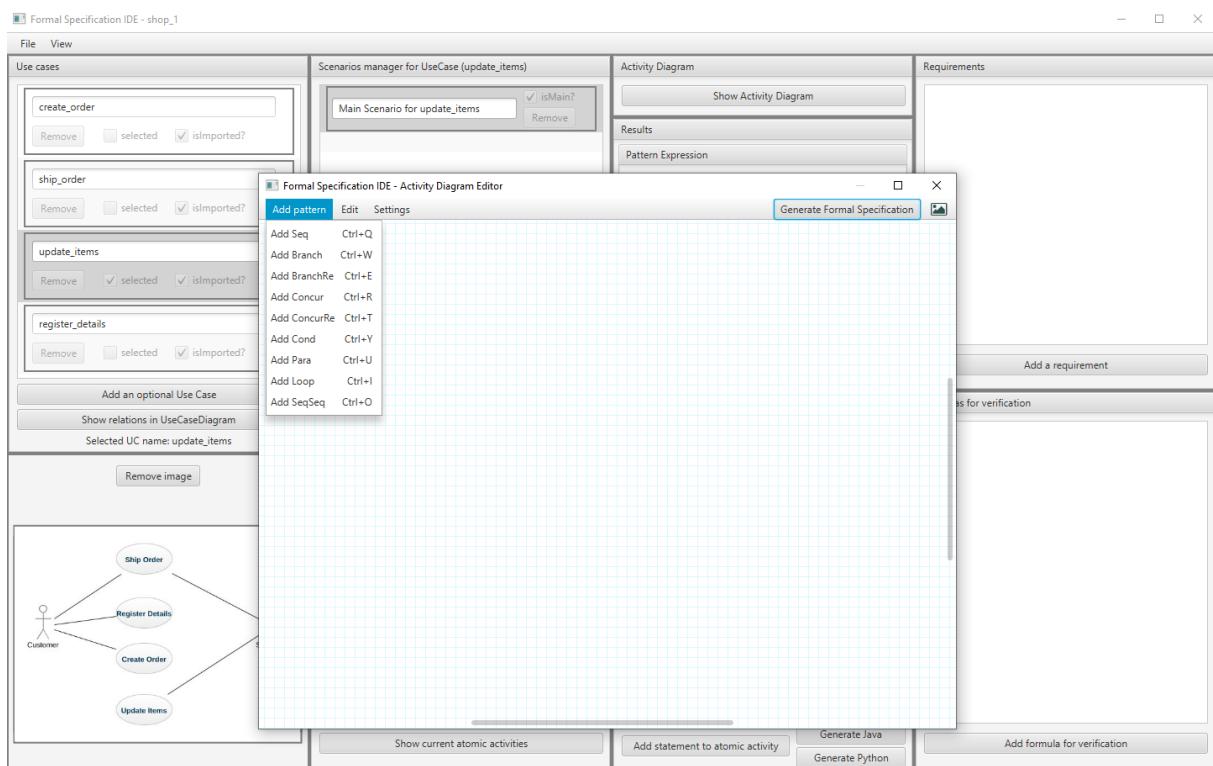
Aktywność w napisanym tekście naturalnym wskazuje się poprzez kliknięcie prawym klawiszem myszki, w miejscu, w którym znajduje się kurSOR. W takim przypadku dane słowo zostaje pogrubione, co świadczy o fakcie, że zostało ono zaznaczone jako atomiczna aktywność.

Dodatkowo, w celu sprawdzenia poprawności zaznaczenia aktywności, po kliknięciu przycisku „Show current atomic activities”, zostaje wyświetlane okno, w którym wylistowane są wszystkie obecne aktywności dla danego scenariusza.

Panel do tworzenia zawartości scenariusza umożliwia użytkownikowi precyzyjne modelowanie scenariuszy i definiowanie ich związku z atomicznymi aktywnościami.

## 4.5. Modelowanie diagramu aktywności

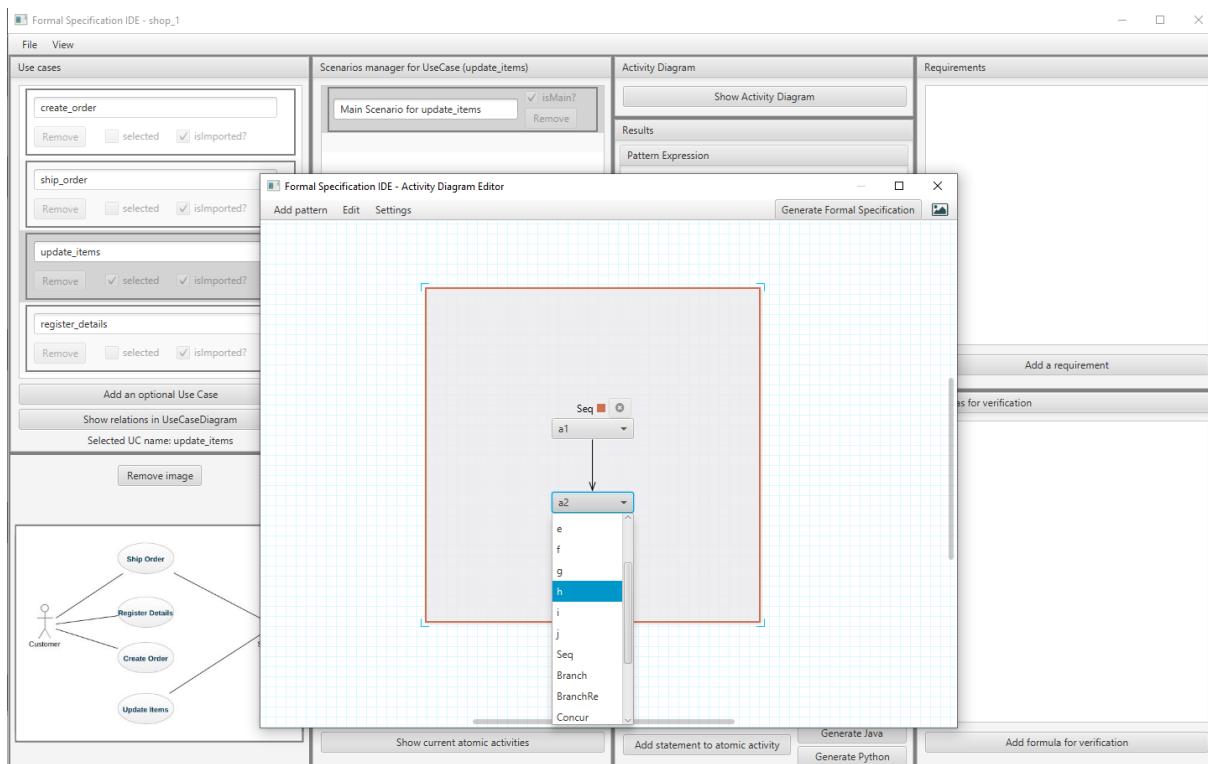
Kolejnym kluczowym aspektem interfejsu graficznego aplikacji *Formal Specification IDE* jest panel do modelowania diagramu aktywności. Ten panel pełni ważną rolę w procesie tworzenia modeli behawioralnych, umożliwiając użytkownikowi rozbudowane tworzenie i edycję diagramów aktywności na podstawie atomicznych aktywności oraz predefiniowanych wzorców przepływu pracy, które zostały opisane w sekcji 2.2.



**Rys. 4.5.** Po kliknięciu przycisku „Show Activity Diagram” użytkownik ma możliwość modelowania diagramu aktywności.

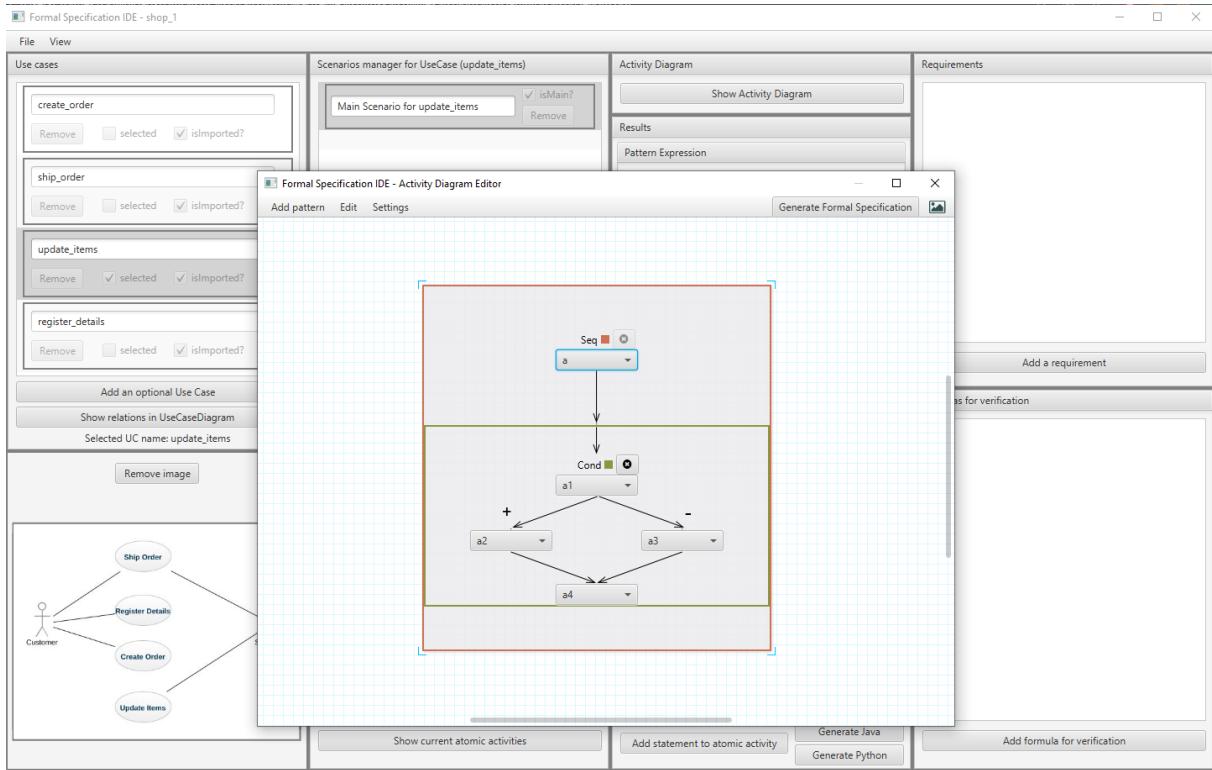
Diagramy aktywności stanowią skuteczne narzędzie do graficznej reprezentacji przepływu sterowania i danych w systemie. Są one używane do przedstawiania akcji, decyzji, warunków, pętli i innych elementów, które oddają dynamikę działania systemu. Diagramy aktywności szczególnie przydatne są w modelowaniu procesów biznesowych, umożliwiając ukazanie sekwencji czynności, zależności oraz podejmowanych decyzji.[23].

W ramach aplikacji *Formal Specification IDE*, użytkownik ma możliwość tworzenia diagramów aktywności na bazie atomicznych aktywności oraz predefiniowanych wzorców przepływu pracy. Po uzupełnieniu scenariusza przypadku użycia oraz określeniu atomicznych aktywności, możliwe jest przystąpić do kreowania diagramu aktywności. Ten proces rozpoczyna się poprzez kliknięcie przycisku „Show Activity Diagram”, co otwiera nowe okno, gdzie użytkownik może rozbudowywać diagram.



**Rys. 4.6.** Wybór elementów wzorca w panelu modelowania diagramu aktywności. Każdy element wzorca należy uzupełnić za pomocą listy wyboru, za pomocą określonych atomicznych aktywności oraz nazw wzorców, które zostaną zagnieżdżone.

Na początku procesu tworzenia diagramu aktywności przedstawiającego przepływ pracy w systemie, użytkownik ma do dyspozycji puste okno, jak pokazano na rysunku 4.5. W tym oknie można dodawać i edytować elementy diagramu, takie jak akcje, decyzje, warunki, pętle, itp. W menu, pod opcją „Add pattern” zawarte są wszystkie dostępne wzorce. Po wyborze pojawia



**Rys. 4.7.** Zagnieżdżanie wzorców w diagramie aktywności. W przypadku wyboru jednej z nazw wzorców, wskazany wzorzec zostanie zagnieżdżone w określonym miejscu.

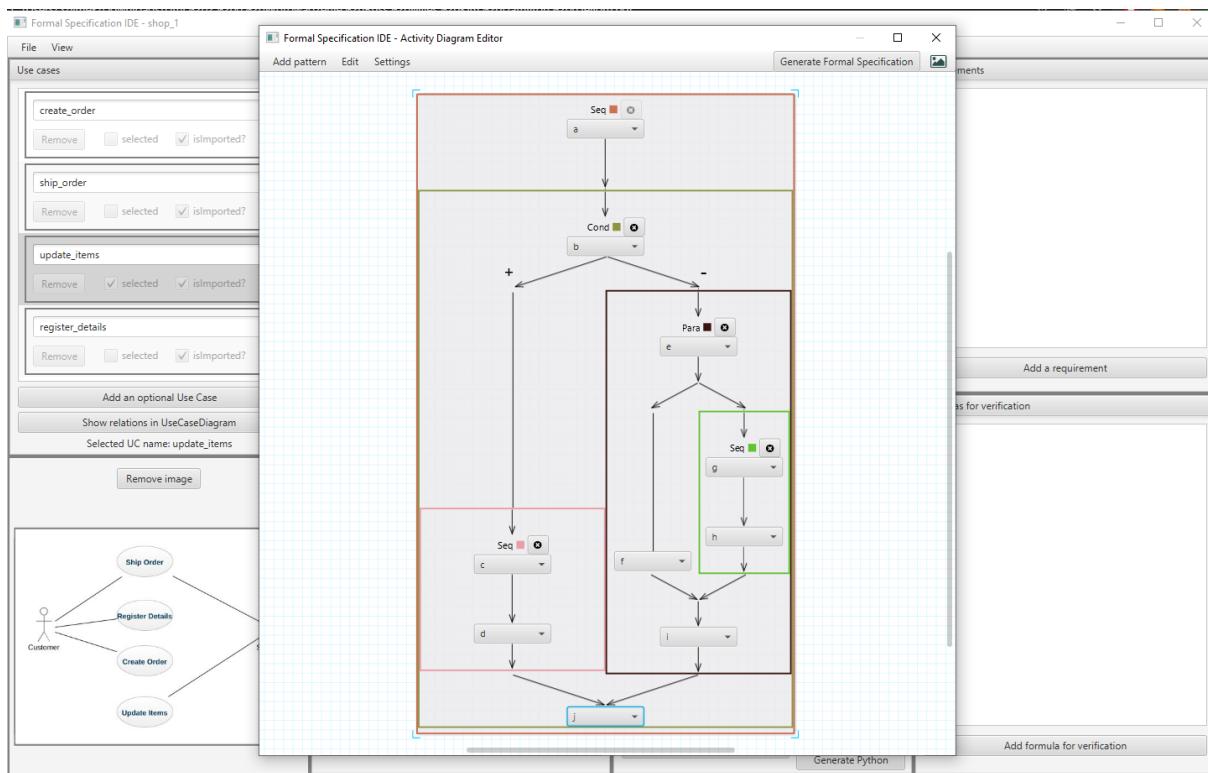
się nowy element. Następnie należy uzupełnić wszystkie listy wyboru, za pomocą dostępnych wzorców, albo atomicznych aktywności.

Podstawowymi elementami diagramu aktywności są atomiczne aktywności. Są to najmniejsze jednostki działania, które reprezentują konkretne czynności wykonywane przez system. Atomiczne aktywności są definiowane w scenariuszach przypadków użycia i mogą być używane w różnych miejscach diagramu aktywności.

W celu ułatwienia tworzenia diagramów aktywności, system *Formal Specification IDE* udostępnia predefiniowane wzorce przepływu pracy. Wzorce te reprezentują często spotykane schematy przepływu sterowania w systemach informatycznych. Przykłady wzorców to „Seq”, „Cond”, „Para”. Szczegóły dotyczące wzorców zostały opisane w sekcji 2.2. Użytkownik może korzystać z tych wzorców, aby modelować bardziej skomplikowane zachowania systemu.

Podczas tworzenia diagramu aktywności, użytkownik może dodawać i edytować elementy diagramu. Elementy te mogą się połączyć za pomocą strzałek, które reprezentują przepływ sterowania między akcjami. Użytkownik może także stosować wzorce przepływu pracy, które umożliwiają uporządkowane i zrozumiałe przedstawienie przepływu sterowania w systemie.

Na rysunku 4.8 przedstawiono przykładowy diagram aktywności, który przedstawia przepływ pracy w systemie. Diagram ten składa się z różnych elementów, które reprezentują kolejne



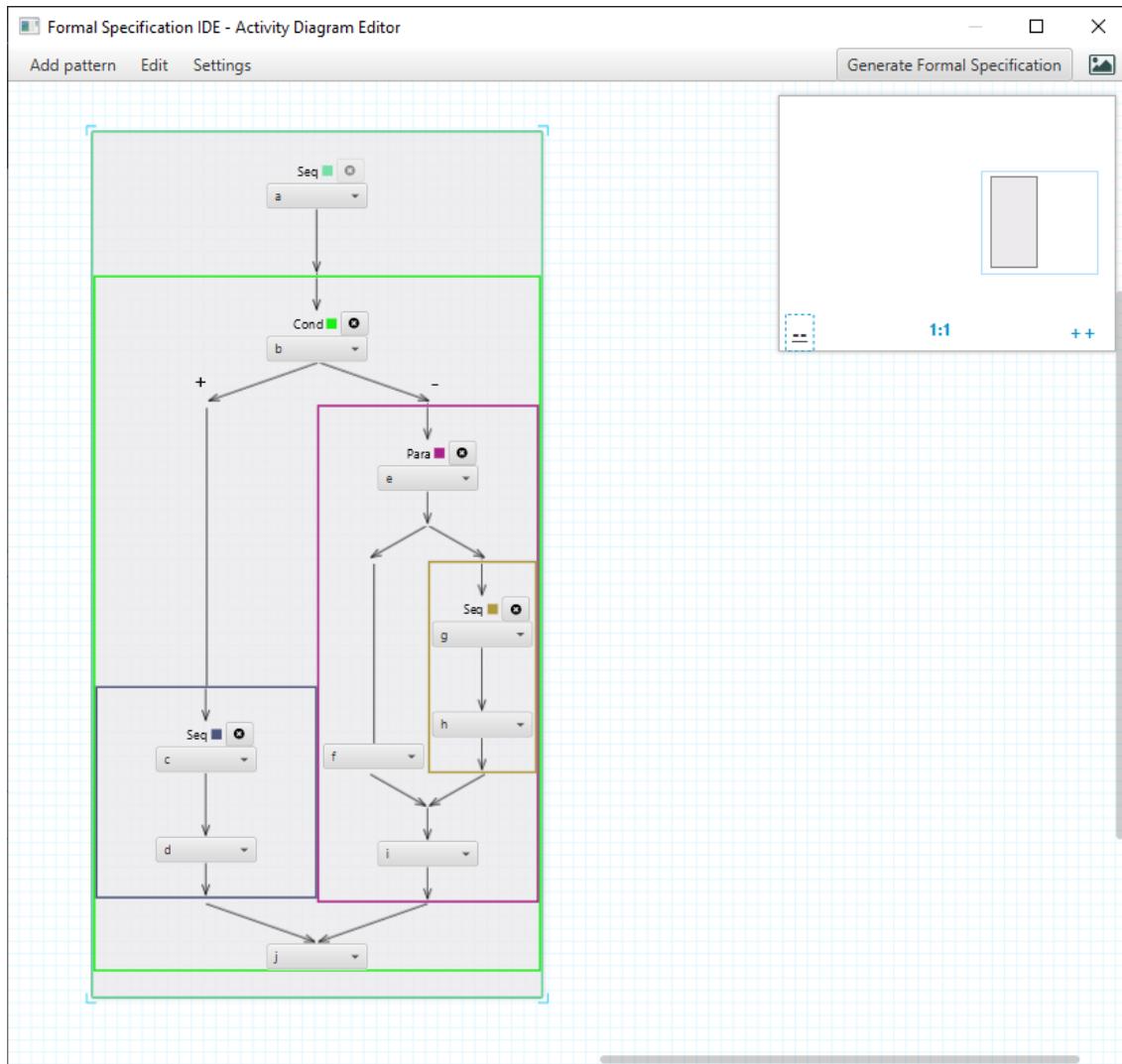
**Rys. 4.8.** Wzorce można dowolnie w sobie zagnieżdzać. Docelowo każdy z dostępnych elementów musi być wypełniony za pomocą atomicznej aktywności lub zagnieżdżonego wzorca.

czynności i zależności między nimi. Poprzez odpowiednie połączenie tych elementów za pomocą strzałek, można przedstawić logiczny przebieg działania systemu.

Okno do modelowania diagramów aktywności można dowolnie zmieniać w celu dostosowania rozmiaru do potrzeb. Dodatkowo, samo pole robocze można powiększać i pomniejszać, dostosowując się do rozmiaru diagramu. Powiększanie i pomniejszanie polega na przytrzymywaniu klawisza Ctrl i przewijaniu myszką. W prawym górnym rogu znajduje się ikona miniatury, umożliwiająca przegląd całego obszaru roboczego oraz nawigację, jak pokazano na rysunku 4.9.

Modelowanie diagramów aktywności ma wiele zalet. Przede wszystkim, umożliwia ono wizualne przedstawienie przepływu sterowania i danych w systemie, co ułatwia zrozumienie i analizę procesów biznesowych. Ponadto, diagramy aktywności są łatwe w odczytaniu i interpretacji, co ułatwia komunikację. Dodatkowo, modelowanie diagramów aktywności umożliwia wykrywanie ewentualnych błędów i niejednoznaczności w projekcie systemu już na etapie analizy i projektowania.

Podsumowując, panel do modelowania diagramu aktywności w systemie *Formal Specification IDE* dostarcza intuicyjnych narzędzi i funkcji, które umożliwiają tworzenie czytelnych,



Rys. 4.9. W oknie do modelowania diagramów aktywności jest dostępny podgląd (minimap) całej przestrzeni roboczej, z możliwością przesuwania, przybliżania, oddalania.

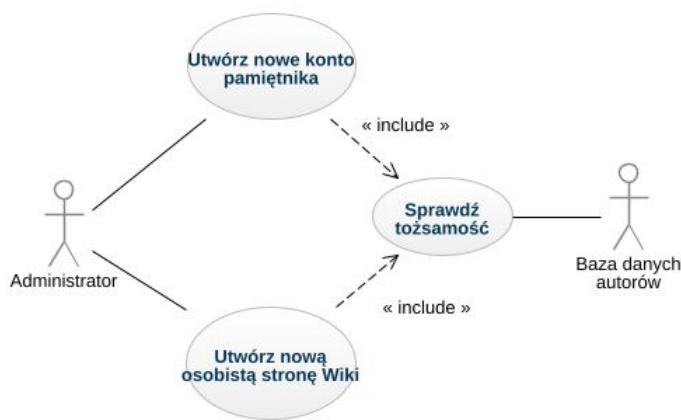
spójnych i zrozumiałych diagramów aktywności. Dzięki temu użytkownik może skutecznie modelować przepływ pracy w systemie i precyzyjnie przedstawiać zachowanie systemu w formie graficznej.

## 4.6. Relacje pomiędzy przypadkami użycia

Diagramy przypadków użycia służą do graficznej reprezentacji przypadków użycia, aktorów oraz relacji między nimi. W takich diagramach występują trzy główne typy relacji: relacje zawierania, generalizacji oraz rozszerzania. W tej sekcji zostaną omówione te relacje oraz ich interpretacja i praktyczne zastosowanie w kontekście narzędzia *Formal Specification IDE*.

### 4.6.1. Relacja zawierania

W przypadkach użycia często występuje potrzeba współdzielenia pewnych fragmentów zachowań między różnymi przypadkami. Aby skutecznie zarządzać tym współdzieleniem i uniknąć powtarzania kodu, można wyodrębnić wspólne fragmenty zachowań do oddzielnego przypadku użycia. Następnie, za pomocą relacji zawierania («`include`»), można wykorzystać ten wyodrębniony przypadek użycia w innych przypadkach.

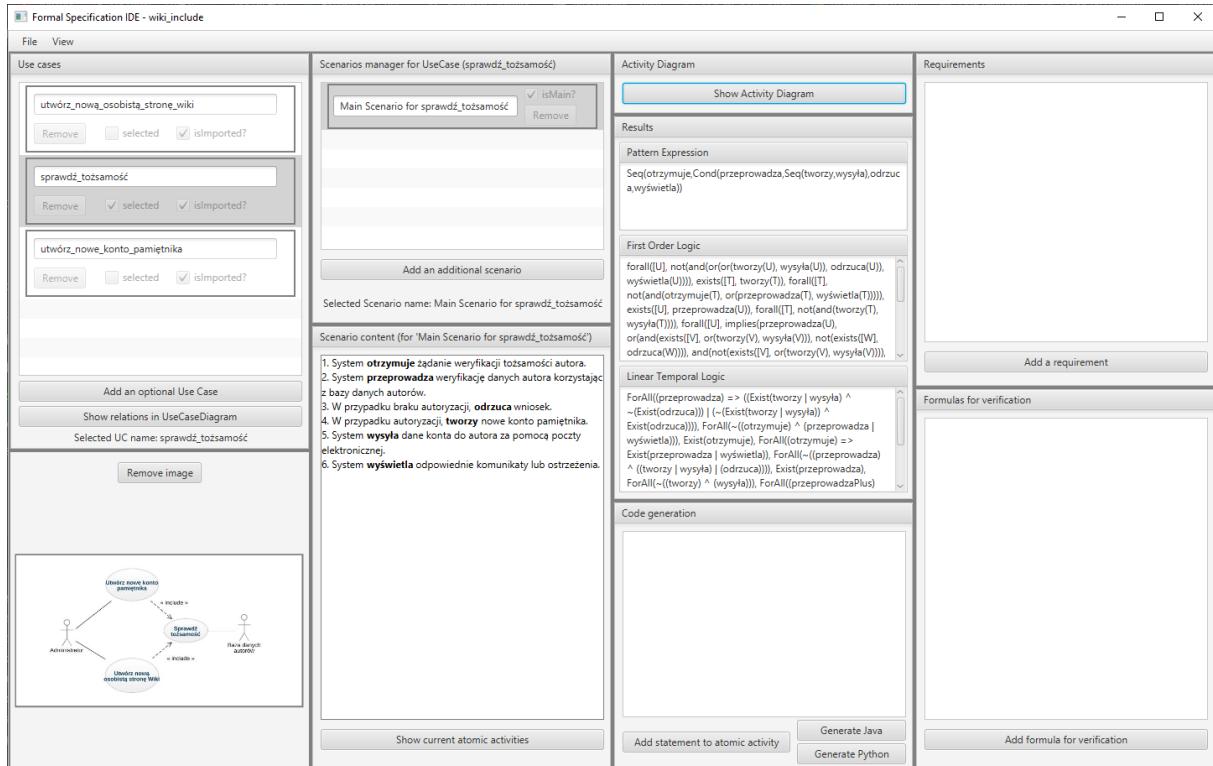


**Rys. 4.10.** Przykład zastosowania relacji zawierania «`include`». Diagram został stworzony na bazie przykładu zawartego w książce [23]. Diagram został zamodelowany z wykorzystaniem aplikacji GenMyModel (opisanej w sekcji 2.3).

W przypadku zależności «`include`», przypadek użycia zawierający reprezentuje ogólną sekwencję kroków, które muszą być wykonane. Przypadek użycia wyodrębniony reprezentuje specyficzne kroki, które są włączane do sekwencji wykonywanych kroków w przypadku użycia zawierającego.

Na rysunku 4.10 przedstawiono przykład zastosowania relacji zawierania («`include`»). Przypadek użycia „Utwórz nowe konto pamiętnika” jest przypadkiem użycia zawierającym, który obejmuje ogólny proces tworzenia nowego konta. Wyodrębniono przypadek użycia „Sprawdź tożsamość”, który reprezentuje specyficzne kroki związane z dostarczaniem zamówienia. Za pomocą relacji zawierania («`include`»), przypadek użycia „Sprawdź tożsamość” może być wielokrotnie wykorzystywany w różnych przypadkach użycia, takich jak „Utwórz nowe konto pamiętnika” czy „Utwórz nową osobistą stronę Wiki”.

Jak pokazano na rysunku 4.10, przy użyciu zależności «`include`» przypadek użycia „Sprawdź tożsamość” został dwukrotnie użyty przez przypadki „Utwórz nowe konto pamiętnika” oraz „Utwórz nową osobistą stronę Wiki”.

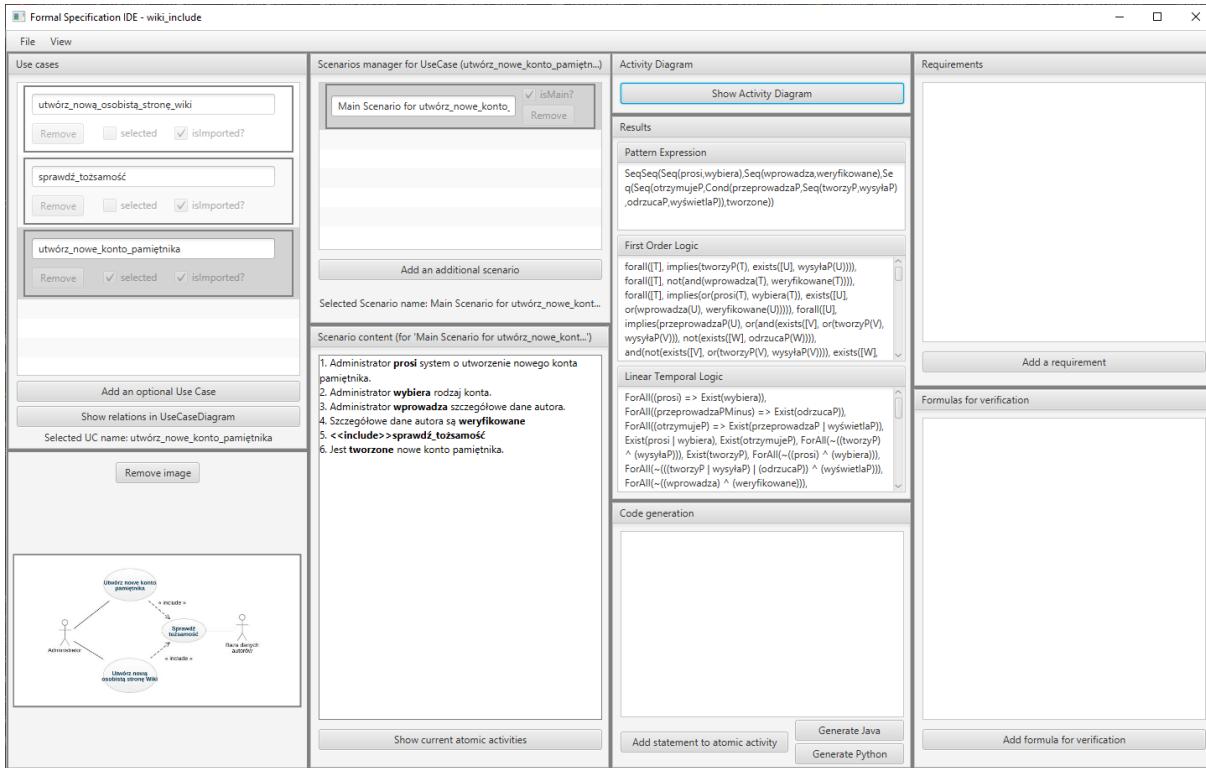


**Rys. 4.11.** Dla wyodrębnionego przypadku użycia „Sprawdź tożsamość” został uzupełniony scenariusz oraz zamodelowany diagram aktywności, w wyniku którego zostało wygenerowane wyrażenie wzorcowe. Podczas modelowania diagramu aktywności dla zawierającego przypadku użycia, stworzone wyrażenie zostanie wstrzyknięte w wskazanym miejscu.

Podczas tworzenia scenariuszy przypadków użycia, zależność «`include`» jest uwzględniana w treści scenariuszy. W miejscach, w których występują kroki związane z przypadkiem użycia wyodrębnionym, stosuje się składnię «`include`» [nazwa\_przypadku\_użycia], gdzie nazwa\_przypadku\_użycia to nazwa przypadku użycia wyodrębnionego. Przykładowo: «`include`» sprawdz\_tozsamosc.

Podczas generowania specyfikacji logicznej na podstawie diagramów aktywności i scenariuszy, zależności «`include`» są również uwzględniane. Wygenerowane wyrażenie wzorcowe przypadku zawieranego zostaje podczas tworzenia diagramu aktywności wstrzyknięte do wyrażenia wzorcowego przypadku użycia zawierającego.

Relacja zawierania («`include`») w przypadkach użycia umożliwia efektywne zarządzanie wspólnymi fragmentami zachowań i zapewnia modularność i reużywalność w projektowaniu systemów informatycznych.

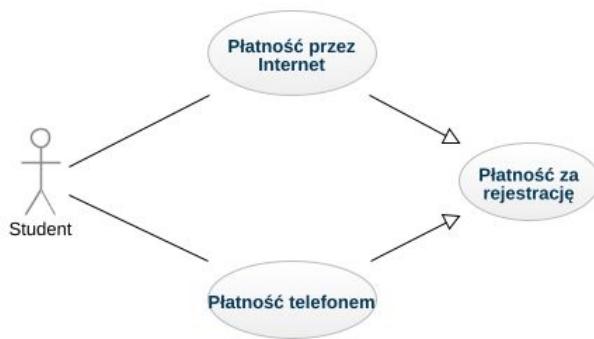


**Rys. 4.12.** Dla zawierającego przypadku użycia „Utwórz nowe konto pamiętnika” został uzupełniony scenariusz. W jednym z punktów scenariusza należało obligatoryjnie zawrzeć «*include*» sprawdż\_tożsamość. Następnie podczas zapisywania diagramu aktywności, w wyrażeniu wzorcowym , aktywność zawierania została zastąpiona przez wyrażenie wzorcowe zawieranego przypadku użycia.

## 4.6.2. Relacja generalizacji

Relacja generalizacji jest jednym z kluczowych elementów modelowania przypadków użycia, umożliwiającym opisanie hierarchii przypadków użycia. Ta relacja pozwala na reprezentację przypadku użycia ogólniejszego, który posiada podtypy, reprezentujące bardziej szczegółowe zachowania. W odróżnieniu od relacji «*include*», gdzie określone zachowanie jest współdzielone, relacja generalizacji dotyczy zastosowania całego przypadku użycia i wprowadzenia zmian w zachowaniu.

Relacja generalizacji znajduje zastosowanie w przypadkach, gdy istnieje pewna wspólna struktura lub zachowanie między przypadkami użycia, ale jednocześnie istnieją różnice w ich szczegółach. Przypadek użycia ogólny zawiera ogólne zachowanie, natomiast podtypy wprowadzają specyficzne zmiany lub rozszerzenia, dostosowując się do szczególnych potrzeb lub warunków.



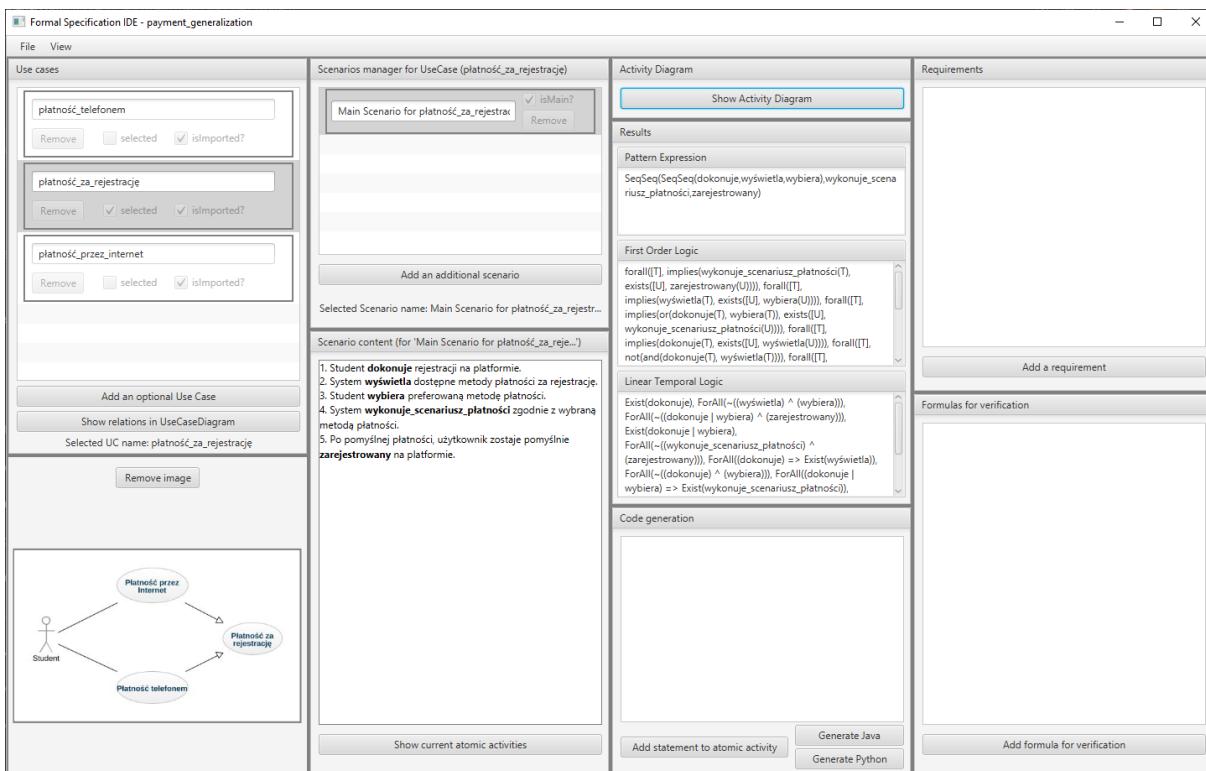
**Rys. 4.13.** Przykład zastosowania relacji generalizacji. Diagram został zamodelowany z wykorzystaniem aplikacji GenMyModel (opisanej w sekcji 2.3).

W celu zobrazowania relacji generalizacji, można posłużyć się diagramem przypadków użycia. Przykład takiego diagramu przedstawiono na rysunku 4.13. Przypadek użycia „Płatność za rejestrację” stanowi ogólniejszy przypadek, a „Płatność przez internet” i „Płatność telefonem” są jego podtypami. W tym scenariuszu, system płatności w firmie szkoleniowej umożliwia dokonywanie płatności za kursy zarówno online, jak i telefonicznie. Oba przypadki mają wiele cech wspólnych, takich jak podanie danych osobowych czy informacji o płatności. Jednak istnieją również różnice między tymi przypadkami. Dlatego najlepszym rozwiązaniem jest zastosowanie ogólnego przypadku „Płatność za rejestrację” jako punktu wyjścia, który zawiera ogólne zachowanie, a następnie tworzenie wyspecjalizowanych przypadków użycia, w których zostaną wprowadzone charakterystyczne zmiany.

Rysunek 4.14 ilustruje jak można rozpoczęć proces tworzenia przypadku użycia szczegółowego, opierając się na przypadku ogólnym. W przypadku ogólnym zostaje uzupełniony scenariusz, stworzony diagram aktywności oraz wyrażenie wzorcowe, które stanowią punkt wyjścia dla tworzenia podtypów. Następnie, jak pokazano na rysunkach 4.15 i 4.16, można tworzyć przypadki szczegółowe, które dostosowują ogólne zachowanie do specyficznych potrzeb. W tym procesie, diagram aktywności dla przypadku ogólnego jest automatycznie dostępny w przypadku szczegółowym, i można go edytować, wprowadzając specyficzne aktywności lub zmieniając istniejące.

Dana zmiana w szczegółowym przypadku użycia może dotyczyć zamiany dane aktywności na inną – specyficzną lub zamianę danych aktywności na zagnieżdżony wzorzec i wypełnienie go aktywnościami.

Podczas tworzenia scenariusza dla przypadku użycia ogólniejszego, należy uwzględnić, że może on być realizowany zarówno przez przypadki użycia ogólniejsze, jak i przez ich podtypy.

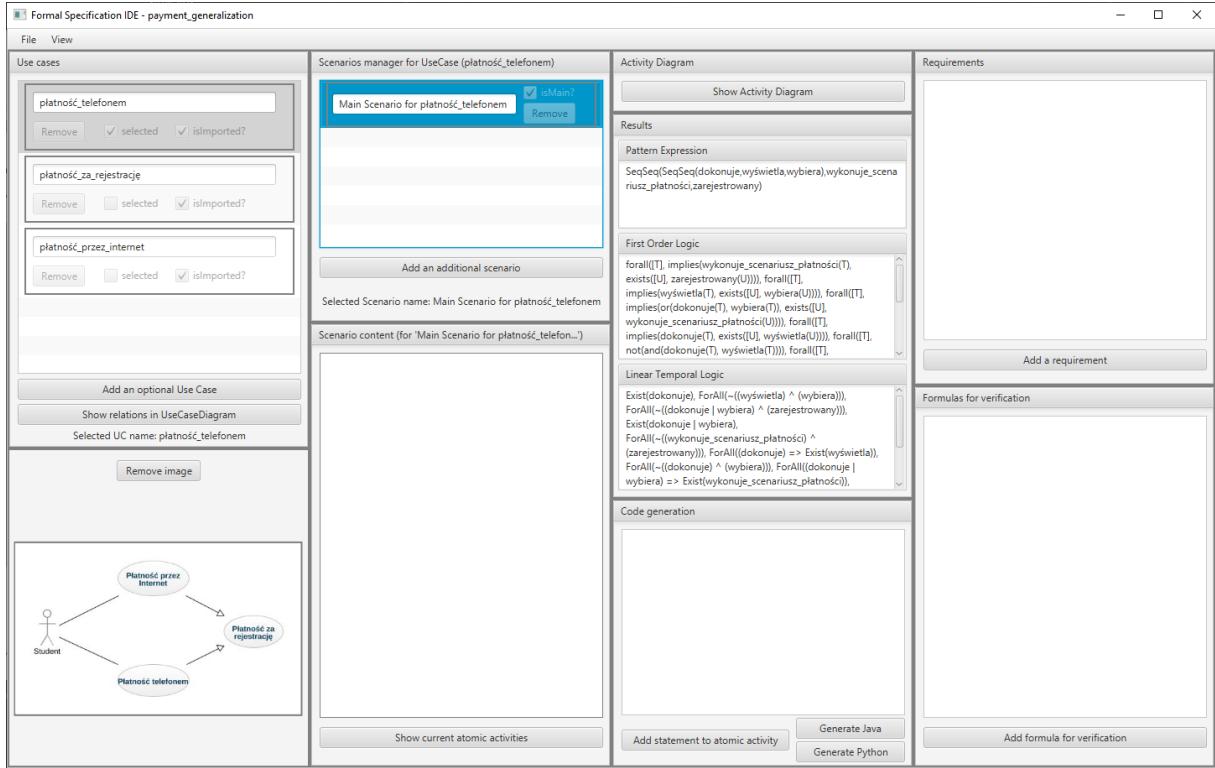


**Rys. 4.14.** Dla ogólnego przypadku użycia „Płatność za rejestrację” został uzupełniony scenariusz, stworzony diagram aktywności oraz wyrażenie wzorcowe. Posłuży to jako punkt startowy dla szczegółowych przypadków użycia.

W systemie *Formal Specification IDE*, w celu rozpoczęcia tworzenia diagramu aktywności dla przypadku szczególnego, konieczne jest w pierwszej kolejności uzupełnienie scenariusza dla ogólnego przypadku użycia i na jego podstawie zamodelowanie diagramu aktywności. Podczas edycji szczególnego przypadku użycia, punktem wyjścia będzie diagram uzyskany dla ogólnego przypadku, który następnie będzie można edytować, aby wprowadzić charakterystyczne zmiany.

Wykorzystanie relacji generalizacji w przypadkach użycia przynosi wiele korzyści. Po pierwsze, umożliwia tworzenie bardziej elastycznych i modułowych modeli, gdzie ogólne zachowanie jest opisane w przypadku użycia ogólnym, a bardziej szczegółowe zachowania są opisane w przypadkach użycia podtypów. Przypadek użycia ogólny może zawierać kroki lub scenariusze, które są wspólne dla wszystkich podtypów, a podtypy mogą rozszerzać lub modyfikować te kroki lub scenariusze, aby odzwierciedlić swoje szczegółowe zachowanie.

Podczas wprowadzania zmian dla szczególnych przypadków użycia, należy mieć na uwadze fakt, że w przypadku ponownej edycji ogólnego przypadku użycia, zmiany wprowadzone w szczególnym przypadku zostaną nadpisane przez obecny stan ogólnego przypadku użycia. Użytkownik systemu *Formal Specification IDE* zostanie o tym fakcie poinformowany ostrzeżeniem.



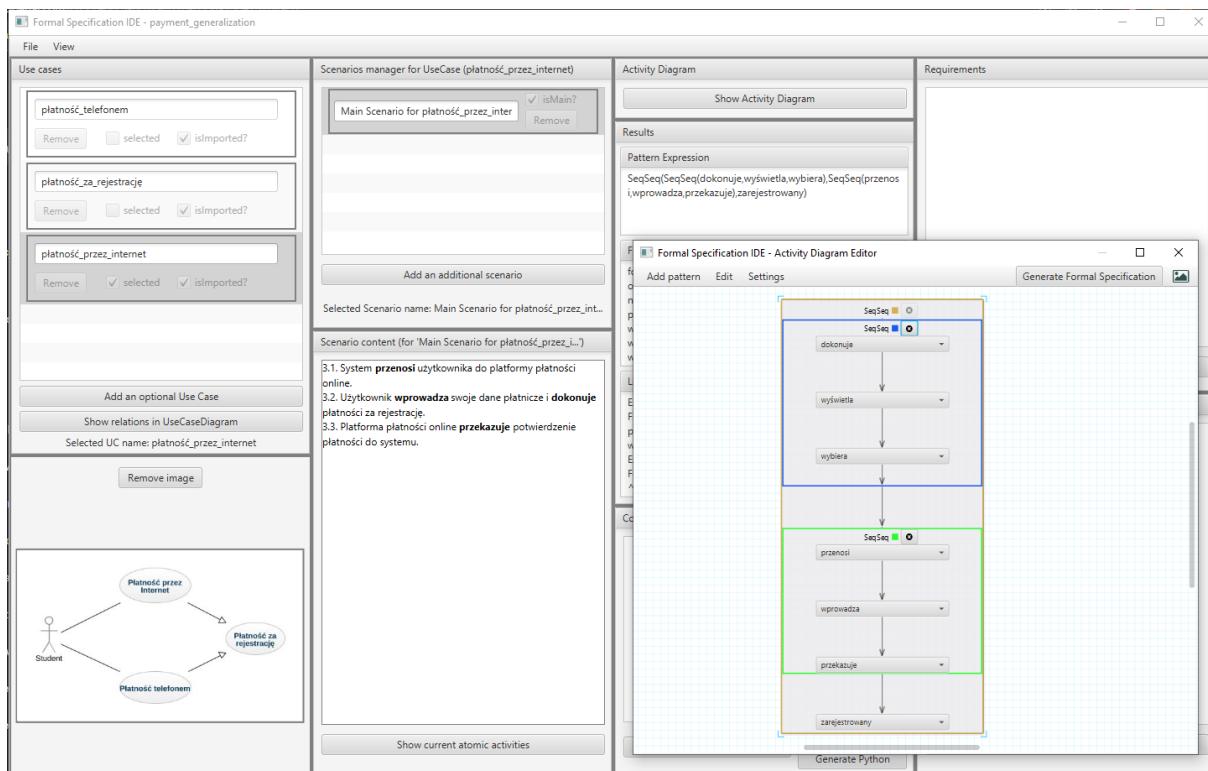
**Rys. 4.15.** Po wejściu w szczegółowy przypadek użycia, automatycznie zostaje uzupełnione wyrażenie wzorcowe. Powielony został ogólny przypadek użycia. Następnie można uzupełnić scenariusz i zaadaptować diagram aktywności.

Podsumowując, relacja generalizacji w przypadkach użycia pozwala na hierarchiczną organizację przypadków, umożliwiając wykorzystanie wspólnego zachowania w przypadkach użycia ogólnych, które są dziedziczone przez podtypy. Dzięki temu, model przypadków użycia staje się bardziej elastyczny, modułowy i łatwy w utrzymaniu.

### 4.6.3. Relacja rozszerzania

Relacja rozszerzania «extend» umożliwia opisanie alternatywnego zachowania przypadku użycia, które może zostać wykonane tylko w określonych warunkach. Przypadek użycia rozszerzający zostaje wykonany tylko wtedy, gdy spełniony jest warunek zdefiniowany w relacji rozszerzania.

Relacja rozszerzania pozwala na modularyzację przypadków użycia, umożliwiając dodawanie opcjonalnych funkcjonalności do istniejących scenariuszy w zależności od warunków. Jest szczególnie użyteczna w przypadku, gdy pewne czynności lub scenariusze są wspólne dla wielu przypadków użycia, ale nie są one zawsze wykonywane.

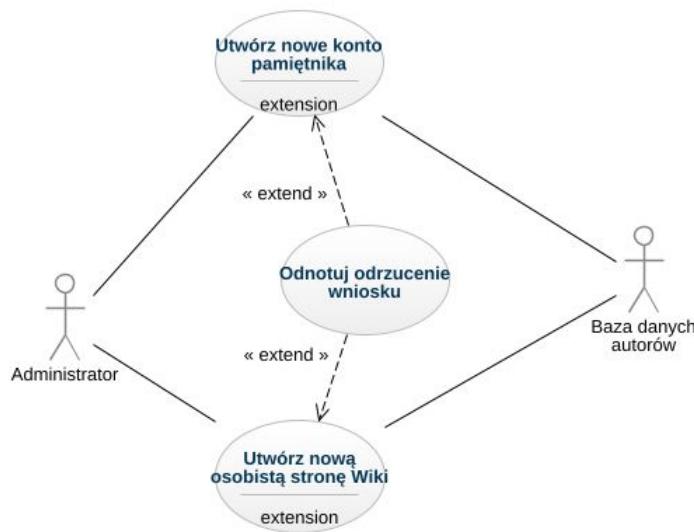


**Rys. 4.16.** Diagram aktywności został zaadaptowany względem ogólnego przypadku użycia. Zostały wprowadzone aktywności specyficzne dla szczegółowego przypadku użycia.

W przypadku zależności «extend», zachowanie danego przypadku użycia może być całkowicie wykorzystane w innym przypadku użycia (podobnie jak w przypadku zależności «include»), niemniej jednak to wykorzystywanie jest opcjonalne i zależy od decyzji w trakcie wykonywania programu lub implementacji systemu.

Na rysunku 4.17 przedstawiono przykład zastosowania relacji rozszerzania («extend»). Przypadek użycia „Utwórz nowe konto pamiętnika” może zawierać informację, że wniosek autora tworzącego nowe konto został odrzucony, i należy odnotować tę próbę w historii wniosków danej osoby. W związku z tym, w scenariuszu przypadku „Utwórz nowe konto pamiętnika” mogą zostać dodane kroki opisujące opcjonalne zachowanie. Podobne zachowanie może wystąpić w przypadku użycia „Utwórz nową osobistą stronę Wiki”. Powstaje więc opcjonalne zachowanie systemu w obu przypadkach.

Zostaje wprowadzony nowy przypadek „Odnotuj odrzucenie wniosku”, który odnotowuje fakt odrzucenia wniosku autora, niezależnie od tego, czy dotyczył on osobistej strony Wiki, czy określonego rodzaju konta pamiętnika. Dzięki wykorzystaniu relacji «extend», ten przypadek użycia jest opcjonalnie współdzielony przez „Utwórz nowe konto pamiętnika” oraz „Utwórz nową osobistą stronę Wiki”.



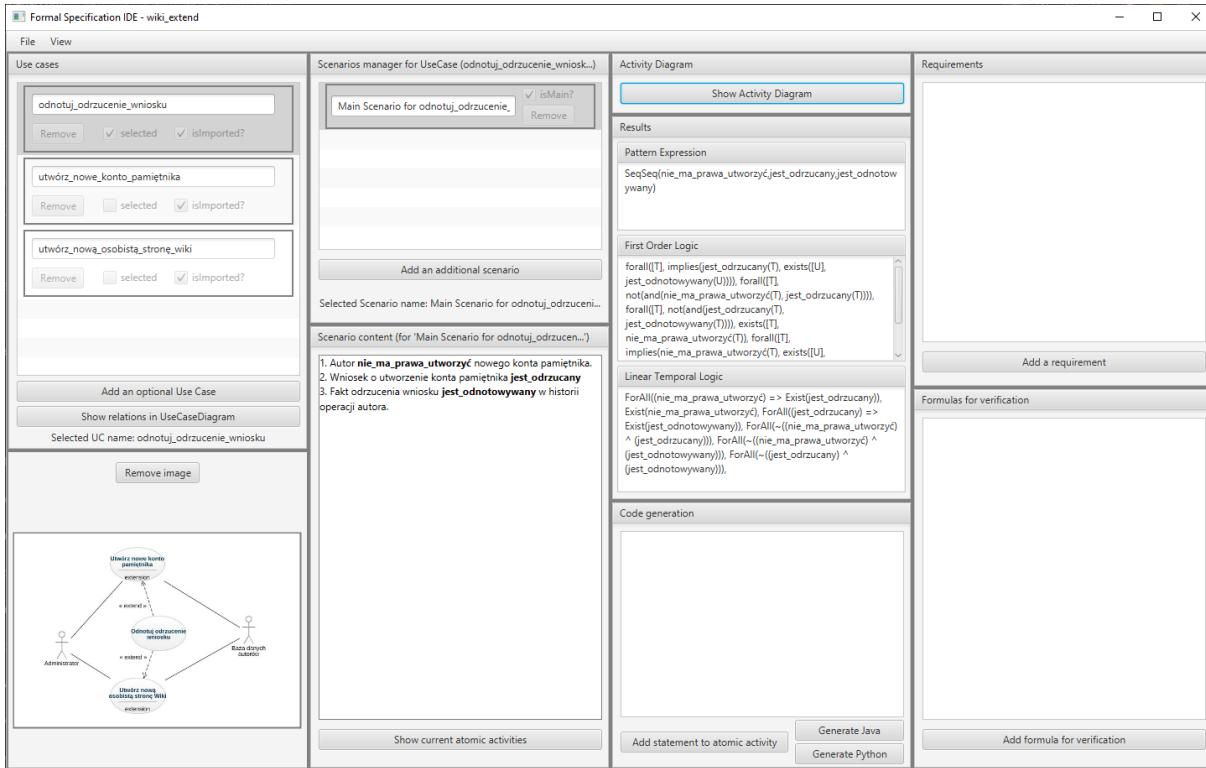
**Rys. 4.17.** Przykład zastosowania relacji rozszerzania. Diagram został stworzony na bazie przykładu zawartego w książce [23]. Diagram został zamodelowany z wykorzystaniem aplikacji GenMyModel (opisanej w sekcji 2.3).

Na rysunku 4.18 przedstawiono przykład rozszerzającego przypadku użycia „Odnotuj odrzucenie wniosku”. Scenariusz tego przypadku użycia zawiera dodatkowe kroki, które są wykonywane tylko w przypadku odrzucenia wniosku. Podczas modelowania diagramu aktywności dla rozszerzanego przypadku użycia, stworzone wyrażenie zostanie opcjonalnie wstrzyknięte w wskazanym miejscu.

Podczas tworzenia scenariusza dla przypadku użycia rozszerzającego, należy uwzględnić warunek rozszerzenia i opisać alternatywne zachowanie w przypadku spełnienia tego warunku.

Podobnie, w przypadku użycia rozszerzanego, należy uwzględnić miejsce, w którym rozszerzenie może zostać włączone. Może to być osiągnięte poprzez dodanie punktu rozszerzenia w scenariuszu, jak pokazano na rysunku 4.19. W tym przykładzie, rozszerzany przypadek użycia „Utwórz nowe konto pamiętnika” zawiera punkt rozszerzenia z warunkiem «extend»WARUNEK?odnotuj\_odezwanie\_wniosku. Podczas zapisywania diagramu aktywności, w wyrażeniu wzorcowym, aktywność rozszerzenia zostaje zastąpiona przez wyrażenie wzorcującego przypadku użycia.

Relacja rozszerzania ma istotne znaczenie w procesie modelowania diagramu aktywności. Przy tworzeniu wyrażenia wzorcowego na podstawie diagramów aktywności i scenariuszy, uwzględniane są zależności «extend». Dana aktywność zawierająca relację rozszerzenia zostanie zastąpiona opcjonalnym użyciem wzorcującego przypadku użycia.

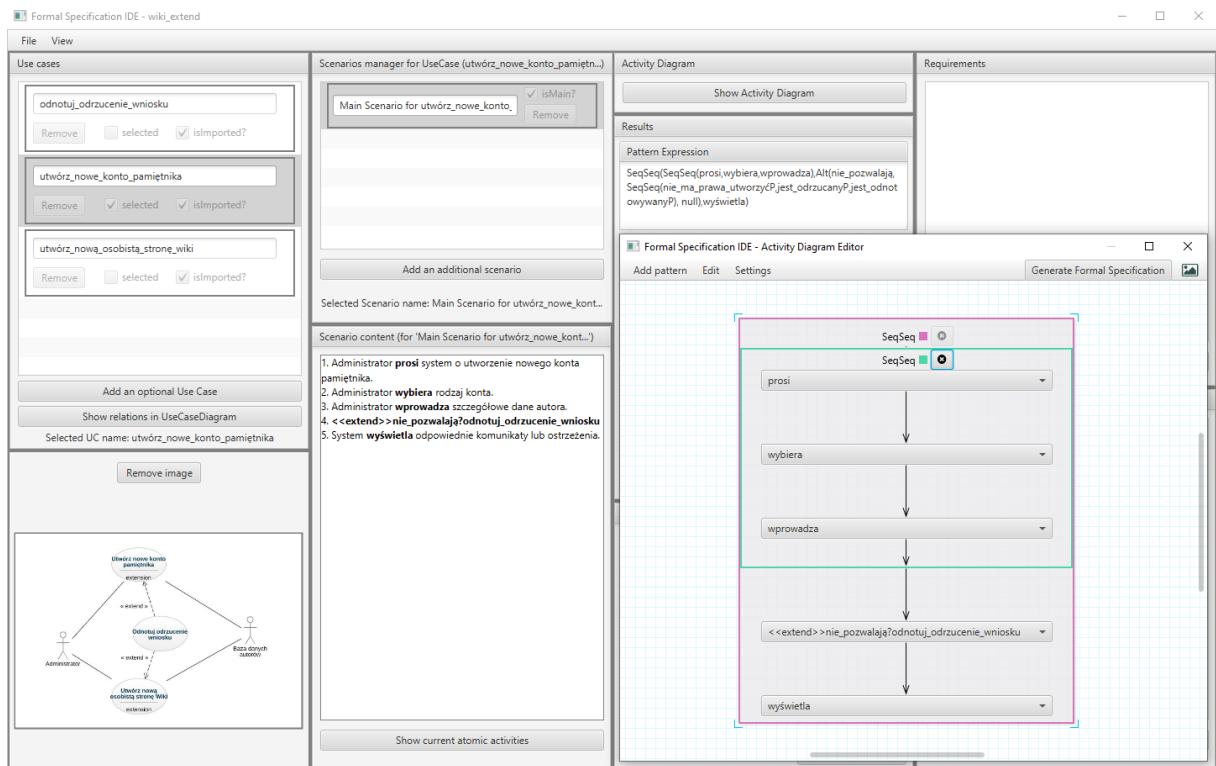


**Rys. 4.18.** Dla rozszerzającego przypadku użycia „Odnotuj odrzucenie wniosku” został uzupełniony scenariusz oraz zamodelowany diagram aktywności, w wyniku którego zostało wygenerowane wyrażenie wzorcowe. Podczas modelowania diagramu aktywności dla rozszerzanego przypadku użycia, stworzone wyrażenie zostanie opcjonalnie wstrzyknięte w wskazanym miejscu.

W systemie *Formal Specification IDE*, podczas tworzenia scenariusza dla przypadku użycia, który powinien zostać opcjonalnie rozszerzony, należy w jednym z kroków scenariusza zastosować składnię «extend»*condition?nazwa\_przypadku\_użycia*, gdzie *condition* oznacza warunek wystąpienia rozszerzenia, a *nazwa\_przypadku\_użycia* jest nazwą przypadku życia, który ma zostać włączony.

W wyrażeniu wzorcowym rozszerzanego przypadku użycia, aktywność rozszerzania zostanie zastąpiona przez wzorzec *Alt*, który odpowiada alternatywie. Przyjmuje on postać: *Alt(warunek, rozszerzające\_wyrażenie, null)*, gdzie warunek został zdefiniowany w scenariuszu w aktywności rozszerzającej, a *rozszerzające\_wyrażenie* jest uprzednio wygenerowanym wyrażeniem wzorcowym rozszerzającego przypadku użycia. Trzecim elementem wzorca jest *null* i oznacza to wyrażenie puste.

Jeżeli użytkownik nie wprowadzi wszystkich relacji w scenariuszu, zostanie poinformowany o błędzie i prawidłowej formie wprowadzanych danych.



**Rys. 4.19.** Dla rozszerzanego przypadku użycia „Utwórz nowe konto pamiętnika” został uzupełniony scenariusz. W jednym z punktów scenariusza należało obligatoryjnie zawrzeć «extend»WARUNEK?odnotuj\_odezwanie\_wniosku. Następnie podczas zapisywania diagramu aktywności, w wyrażeniu wzorcowym, aktywność rozszerzenia została zastąpiona przez wyrażenie wzorcowe rozszerzającego przypadku użycia.

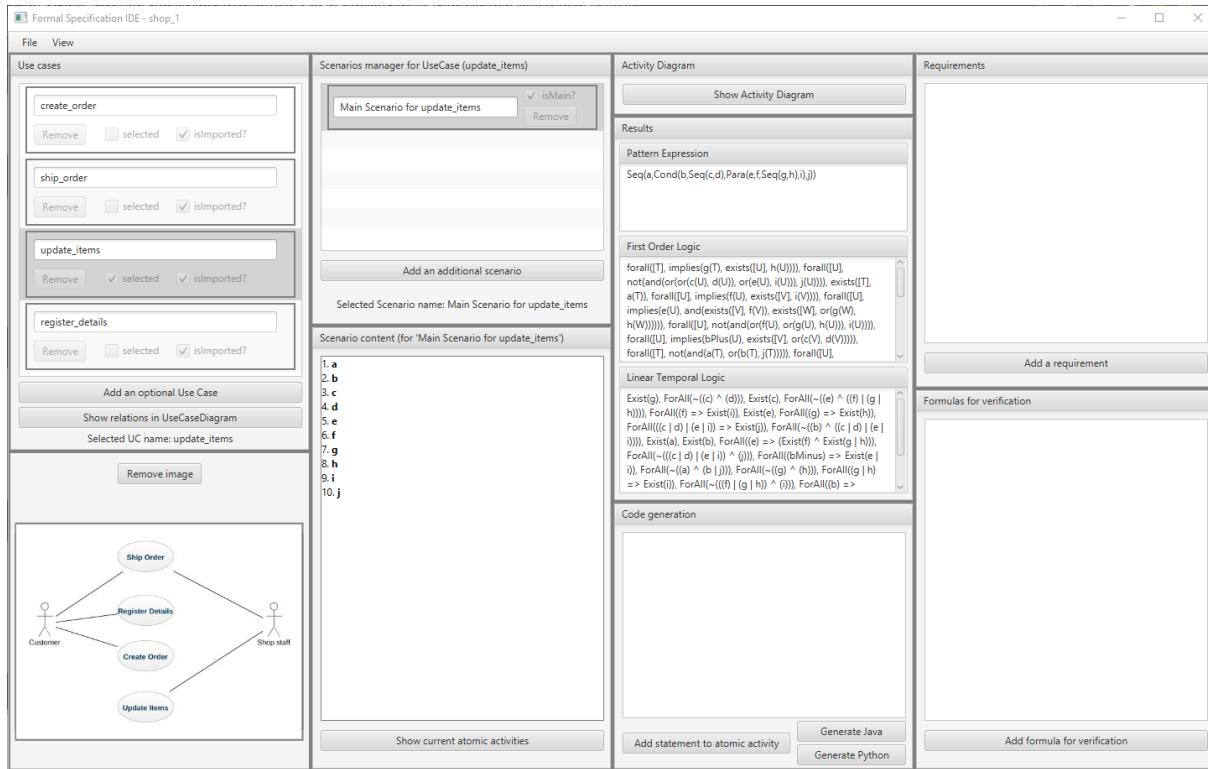
Zastosowanie relacji rozszerzania «extend» w modelowaniu przypadków użytkowania i generowaniu specyfikacji logicznej pozwala na precyzyjne określanie alternatywnych zachowań w zależności od warunków oraz umożliwia modularność i elastyczność w projektowaniu systemu.

## 4.7. Wyrażenie wzorcowe i specyfikacja logiczna

Po utworzeniu diagramu aktywności w panelu do modelowania diagramu aktywności, można przejść do generowania wyrażenia wzorcowego oraz specyfikacji logicznej. Klikając przycisk „Generate Formal Specification” w oknie edytora diagramu aktywności, użytkownik uzyskuje:

- Wygenerowane wyrażenie wzorcowe *Pattern Expression*.

- Wygenerowane formuły logiczne w logice FOL (*First Order Logic*).
- Wygenerowane formuły logiczne w logice LTL (*Linear Temporal Logic*).



**Rys. 4.20.** Stworzony diagram aktywności został przekształcony do postaci wyrażenia wzorcowego. Na jego podstawie została wygenerowana specyfikacja logiczna w dwóch logikach (FOL oraz LTL).

Wyrażenie wzorcowe jest literałem reprezentującym diagram aktywności w postaci tekstuowej. Jest ono generowane na podstawie struktury graficznej diagramu aktywności. Wyrażenie wzorcowe zawiera informacje o sekwenacji czynności, decyzjach, pętlach i innych elementach, które zostały zamodelowane w diagramie.

Szczegółowy opis wyrażenia wzorcowego oraz jego znaczenie został przedstawiony w sekcji 2.2.2, opierając się na artykule [1].

Specyfikacja logiczna jest generowana na podstawie wyrażenia wzorcowego i reguł predefiniowanych wzorców opisanych w sekcji 2.2. Proces generacji specyfikacji logicznej polega na przekształceniu struktury wyrażenia wzorcowego i reguł wzorców na formuły logiczne w logice FOL oraz LTL. Algorytm generacji specyfikacji logicznej został opisany w sekcji 2.2.3.

Wygenerowane wyrażenie wzorcowe oraz specyfikacja logiczna mają wiele zastosowań. Przede wszystkim, stanowią one formalny i precyzyjny opis zachowania systemu. Wyrażenie

wzorcowe może być wykorzystane jako punkt odniesienia do analizy, a także do automatycznego generowania kodu. Specyfikacja logiczna może posłużyć jako podstawa do analizy modelu systemu oraz formalnej weryfikacji systemu.

Podsumowując, generowanie wyrażenia wzorcowego i specyfikacji logicznej na podstawie diagramu aktywności w systemie *Formal Specification IDE* dostarcza formalnych i precyzyjnych opisów zachowania systemu. Oba te elementy stanowią ważne narzędzia w procesie analizy, weryfikacji i implementacji systemu.

## 4.8. Generacja kodu źródłowego

Panel do generowania kodu źródłowego w interfejsie graficznym aplikacji *Formal Specification IDE* stanowi ważny element, który umożliwia użytkownikowi automatyczne generowanie kodu na podstawie stworzonych wyrażeń wzorcowych, z przekształconych diagramów aktywności.

Po zakończeniu modelowania diagramów aktywności, użytkownik może skorzystać z panelu do generowania kodu źródłowego w celu automatycznego wygenerowania kodu w wybranym języku programowania, takim jak Java lub Python.

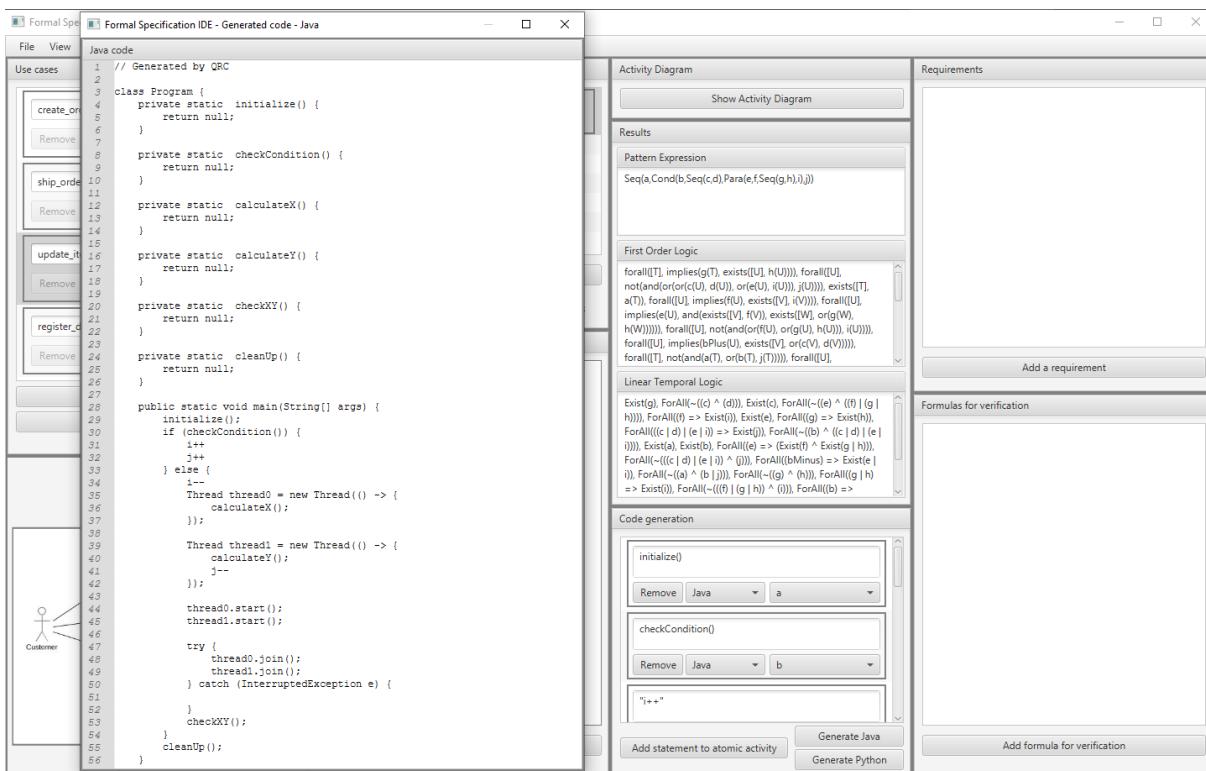
Panel ten umożliwia użytkownikowi wybór docelowego języka programowania oraz konfigurację innych parametrów generowania kodu. Po dokonaniu odpowiednich ustawień, użytkownik może uruchomić proces generowania kodu, który przetworzy stworzone wyrażenia wzorcowe na odpowiadający kod źródłowy.

Wygenerowany kod źródłowy jest oparty na wyrażeniach wzorcowych, które zostały stworzone w poprzednich etapach. Kod źródłowy jest generowany zgodnie z ustaloną konwencją dla danego języka programowania, co zapewnia spójność i czytelność wynikowego kodu.

Dostępne jest zdefiniowanie mapowania danych atomicznych aktywności znajdujących się w wyrażeniu wzorcowym na instrukcje języka programowania. W ten sposób wyrażenie wzorcowe może posłużyć jako wejście dla generatora kodu źródłowego, a także jako wejście dla generatora specyfikacji logicznej.

Przykład wygenerowanego kodu źródłowego w języku Java przedstawiono na rysunku 4.21. W kodzie można zauważyc zmapowane instrukcje języka programowania, które zostały przypisane do poszczególnych atomicznych aktywności z diagramu aktywności. Dzięki temu mapowaniu możliwe jest generowanie bardziej zaawansowanego kodu, który odzwierciedla specyfikację danego systemu.

W panelu do generowania kodu źródłowego dostępne są również funkcje umożliwiające dodawanie, edytowanie i usuwanie instrukcji języka programowania, które zastępują poszczególne atomiczne aktywności. To pozwala na dostosowanie generowanego kodu do konkretnych wymagań i preferencji użytkownika.



**Rys. 4.21.** Przykład wygenerowanego kodu źródłowego w języku Java. Zastosowano mapowanie nazw aktywności do instrukcji języka programowania, co zapewnia większe możliwości generacji.

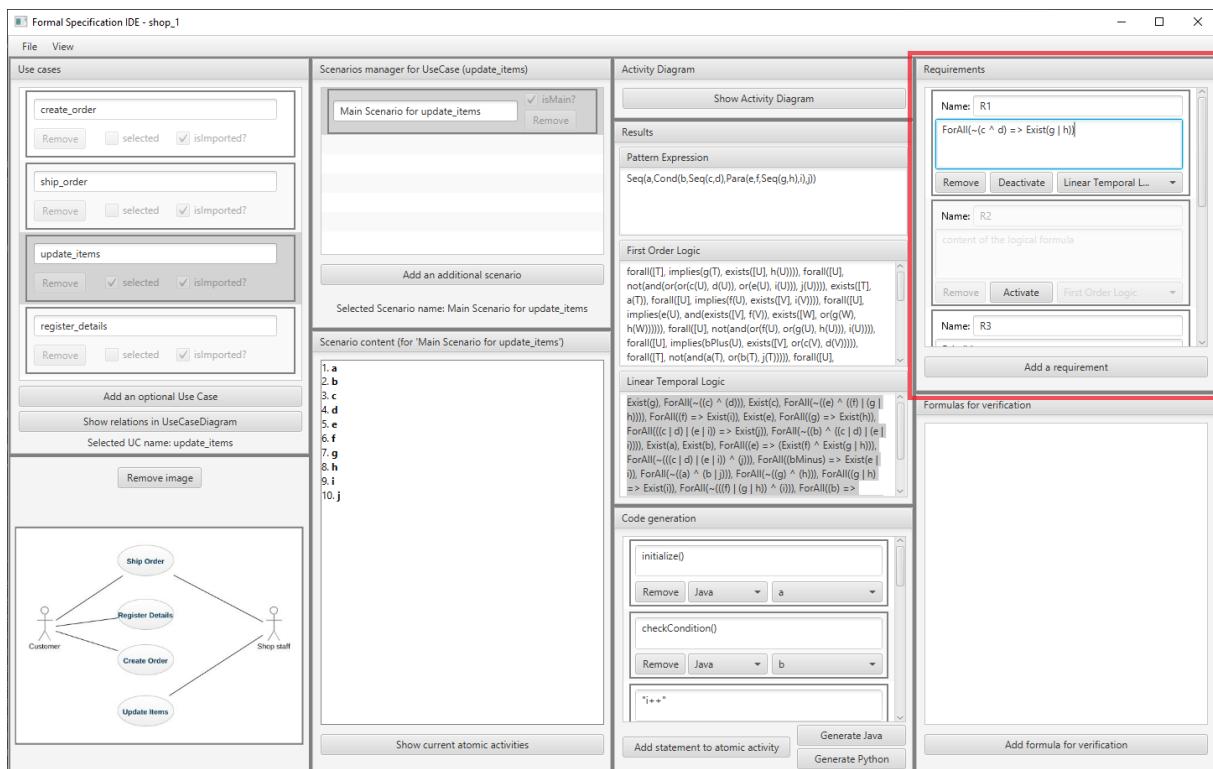
Za pomocą przycisku „Add statement to atomic activity” można dodać instrukcję języka programowania, która zastąpi wskazaną atomiczną aktywność podczas procesu generacji kodu. Należy wprowadzić daną instrukcję, język którego dotyczy oraz wybrać aktywność, która ma zostać zmapowana. Elementy można dodać, edytować oraz usuwać (za pomocą przycisku „Remove”).

Dostępne są dwa przyciski do generacji kodu: „Generate Java” oraz „Generate Python”. Szczegółowe możliwości generatora kodu źródłowego zostały opisane w sekcji 5.7

Panel do generowania kodu źródłowego znacznie usprawnia proces implementacji systemu, umożliwiając automatyczne wygenerowanie podstawowego kodu na podstawie zdefiniowanych modeli. Generator kodu źródłowego stanowi istotne narzędzie w procesie implementacji systemu. Automatyczne generowanie kodu na podstawie modeli i wyróżień wzorcowych znacznie przyspiesza proces tworzenia oprogramowania, jednocześnie zapewniając spójność i zgodność z wcześniej zdefiniowanymi specyfikacjami.

## 4.9. Definiowanie dodatkowych wymagań

W celu wsparcia procesu weryfikacji, analizy oraz kompleksowego modelowania stworzonych systemów, aplikacja *Formal Specification IDE* oferuje użytkownikom możliwość precyzyjnego definiowania dodatkowych wymagań. Panel *Requirements* pozwala użytkownikom wzbogacać specyfikację formalną o nowe aksjomaty, formuły logiczne i ograniczenia, które mogą wejść w skład analizy i weryfikacji systemu.



**Rys. 4.22.** Przykład definicji dodatkowego wymagania w postaci formuły w liniowej logice temporalnej, która może zostać wykorzystana w procesie formalnej weryfikacji systemu.

Definiowanie dodatkowych wymagań pozwala użytkownikami na bardziej szczegółową i kompleksową analizę systemu. Ten panel umożliwia dodawanie specyficznych warunków, oczekiwani i ograniczeń, które mogą nie być zawarte w głównej specyfikacji formalnej. Odpowiednio zdefiniowane dodatkowe wymagania pozwalają użytkownikom na przeprowadzanie analiz bardziej zbliżonych do rzeczywistości scenariuszy.

Użytkownicy mogą tworzyć różne rodzaje dodatkowych wymagań i formuły weryfikacyjnych, to jest formuły w logice pierwszego rzędu (FOL) oraz liniowej logice temporalnej (LTL). W panelu *Requirements* użytkownicy mają możliwość nadawania nazw swoim wymaganiom i formułom, co ułatwia późniejszą identyfikację i korzystanie z nich.

Jednym z przykładów może być dodatkowe wymaganie R1, przedstawione na rysunku 4.22, które informuje, że z negacji koniunkcji aktywności  $c \wedge d$  wynika istnienie aktywności  $g$  lub  $h$ .

Dodatkowe wymagania mogą być również weryfikowane niezależnie od głównej specyfikacji logicznej. Użytkownicy mogą przeprowadzać analizę tylko na podstawie zdefiniowanych dodatkowych wymagań lub łączyć je z główną specyfikacją, co daje im pełną kontrolę nad analizą i weryfikacją systemu.

Definiowanie dodatkowych wymagań stanowi istotny krok w procesie tworzenia modeli behawioralnych oraz analizy systemów. Pozwala na dokładniejszą i bardziej szczegółową analizę systemu, uwzględniając niuanse i specyficzne wymagania projektowe. Umożliwia to wykrywanie błędów i niejednoznaczności, a także spełnienie wszystkich aspektów projektowych i biznesowych.

## 4.10. Proces weryfikacji

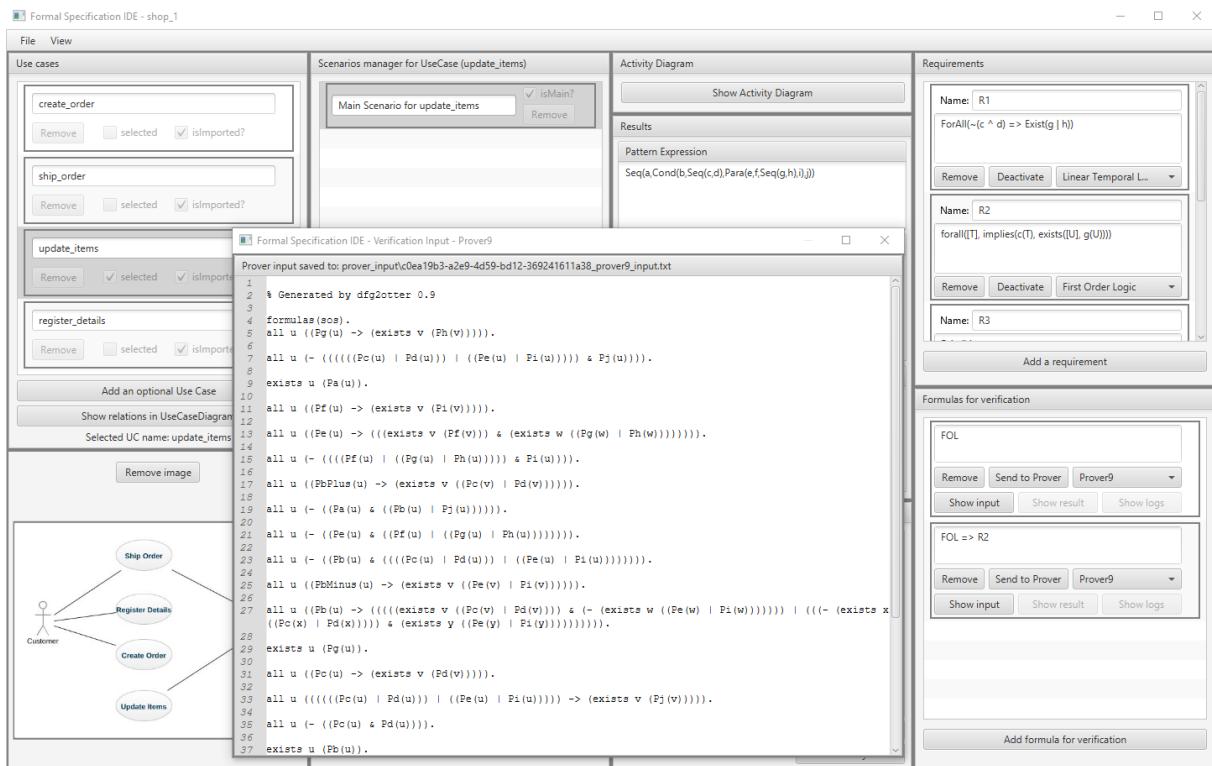
Ostatnim, niezwykle istotnym etapem w tworzeniu oraz analizie stworzonych modeli w aplikacji *Formal Specification IDE* jest panel do procesu weryfikacji. Ten kluczowy element interfejsu użytkownika oferuje kompleksowe narzędzia do formalnej weryfikacji i analizy modeli, celem upewnienia się o ich poprawności oraz zgodności z założeniami projektowymi.

Wewnątrz tego panelu, użytkownik ma dostęp do funkcji i narzędzi, które wspierają proces weryfikacji specyfikacji logicznej oraz prowadzenia dowodów twierdzeń. W kolejnych fragmentach sekcji, omówione zostaną szczegółowo poszczególne komponenty oraz funkcje, które wchodzą w skład tego panelu w aplikacji.

Zadaniem tego panelu jest umożliwienie użytkownikowi wprowadzania formuł logicznych do procesu weryfikacji. Po wcisnięciu przycisku „Add formula for verification”, użytkownik ma możliwość wprowadzenia treści formuły oraz wyboru jednego z dostępnych proverów, które przeprowadzą analizę i dowód formuły.

Użytkownik ma możliwość wyboru jednego z trzech systemów dowodzenia twierdzeń: Prover9, SPASS Prover oraz InKreSAT. Każdy z tych narzędzi ma unikalne zdolności oraz ograniczenia, które zostały wnikliwie opisane w sekcji 2.4. Prover9 oraz SPASS Prover operują na logice pierwszego rzędu (FOL), co pozwala na dowodzenie twierdzeń opartych na predykatach, zmiennych, kwantyfikatorach i relacjach. Natomiast InKreSAT specjalizuje się w logice temporalnej LTL (Propositional Linear Temporal Logic), bazującej na rachunku zdań. Rozwój integracji tych narzędzi z *Formal Specification IDE* oraz napotkane wyzwania związane z instalacją opisano w sekcji 5.6.2.

Użytkownik wprowadza treść formuły logicznej, która ma zostać zweryfikowana. W zależności od wybranego provera, formuła może być zapisana w logice pierwszego rzędu (FOL) lub



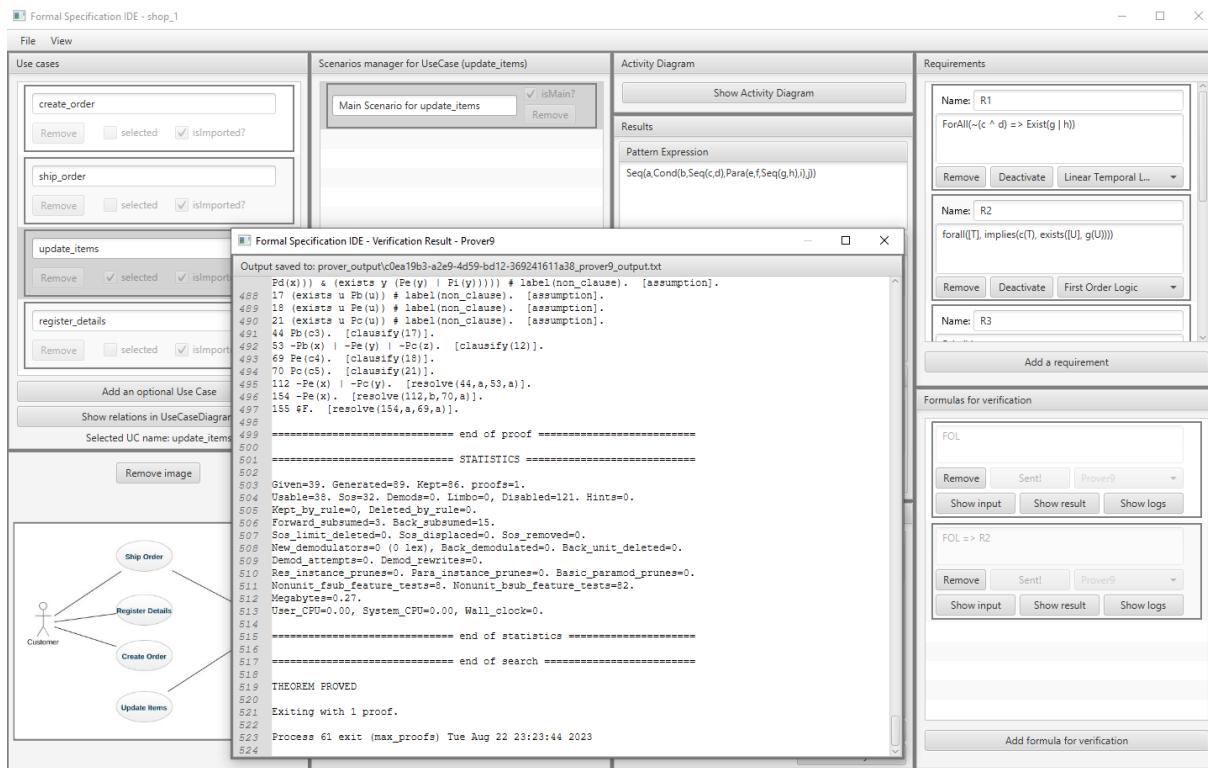
Rys. 4.23. Przykład wejścia dla systemu Prover9.

liniowej logice temporalnej (LTL). W celu weryfikacji specyfikacji logicznej w logice pierwszego rzędu należy wpisać w treści formuły „FOL”. Analogicznie w celu weryfikacji specyfikacji logicznej w liniowej logice temporalnej, należy wpisać w treści „LTL”.

Użytkownik może tworzyć dowolne formuły logiczne do weryfikacji, wykorzystując odpowiednie elementy składniowe:

- FOL (First-Order Logic) – W miejscu wystąpienia tego elementu w formule umieszczana jest specyfikacja logiczna w logice pierwszego rzędu (FOL).
- LTL (Linear Temporal Logic) – W miejscu wystąpienia tego elementu w formule umieszczana jest specyfikacja logiczna w liniowej logice temporalnej (LTL).
- $\Rightarrow$  – Operator implikacji logicznej, który zachodzi między dwoma zdaniem, z których pierwsze jest przesłanką, a drugie jej konsekwencją. Przykład: FOL  $\Rightarrow$  R2.
- $\wedge$  (symbol karety) – Operator koniunkcji, który odpowiada spójnikowi „i” w logice. Umożliwia elastyczne łączenie składników formuły. Przykład: LTL  $\Rightarrow$  R3  $\wedge$  R4.

Przykładem formuły do weryfikacji może być LTL  $\Rightarrow$  R1, która określa, że jeśli specyfikacja logiczna w liniowej logice temporalnej (LTL) jest prawdziwa, to wymaganie R1 musi zostać spełnione. Innym przykładem może być formuła FOL  $\Rightarrow$  R2, która określa, że jeśli



Rys. 4.24. Przykład wyniku weryfikacji dla systemu Prover9. Dowód został znaleziony.

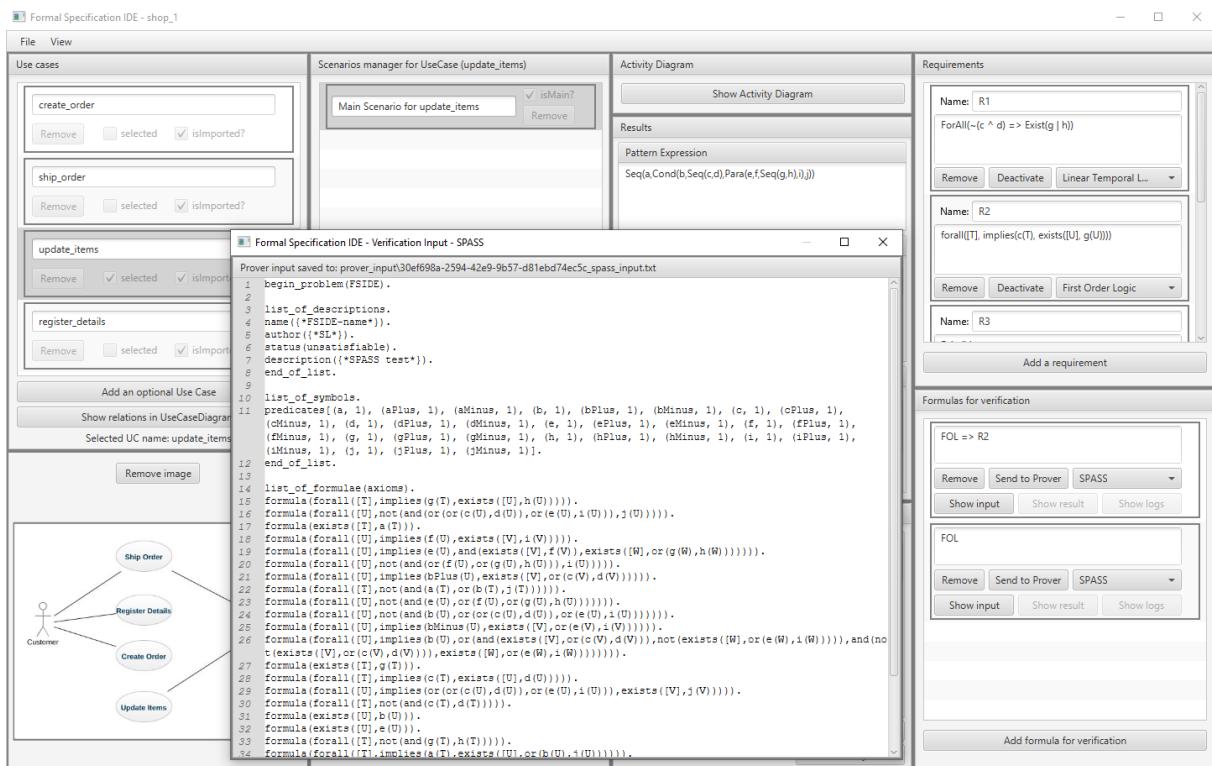
specyfikacji logiczna w logice pierwszego rzędu (FOL) jest spełniona, to wymaganie R2 musi zostać spełnione.

Po utworzeniu formuły i wyborze odpowiedniego provera, użytkownik może zapoznać się z generowanym wejściem dla tego narzędzia, używając przycisku „Show input”. Okno zostanie wyświetcone, prezentując treść wejścia, która zostanie przekazana do systemu dowodzenia.

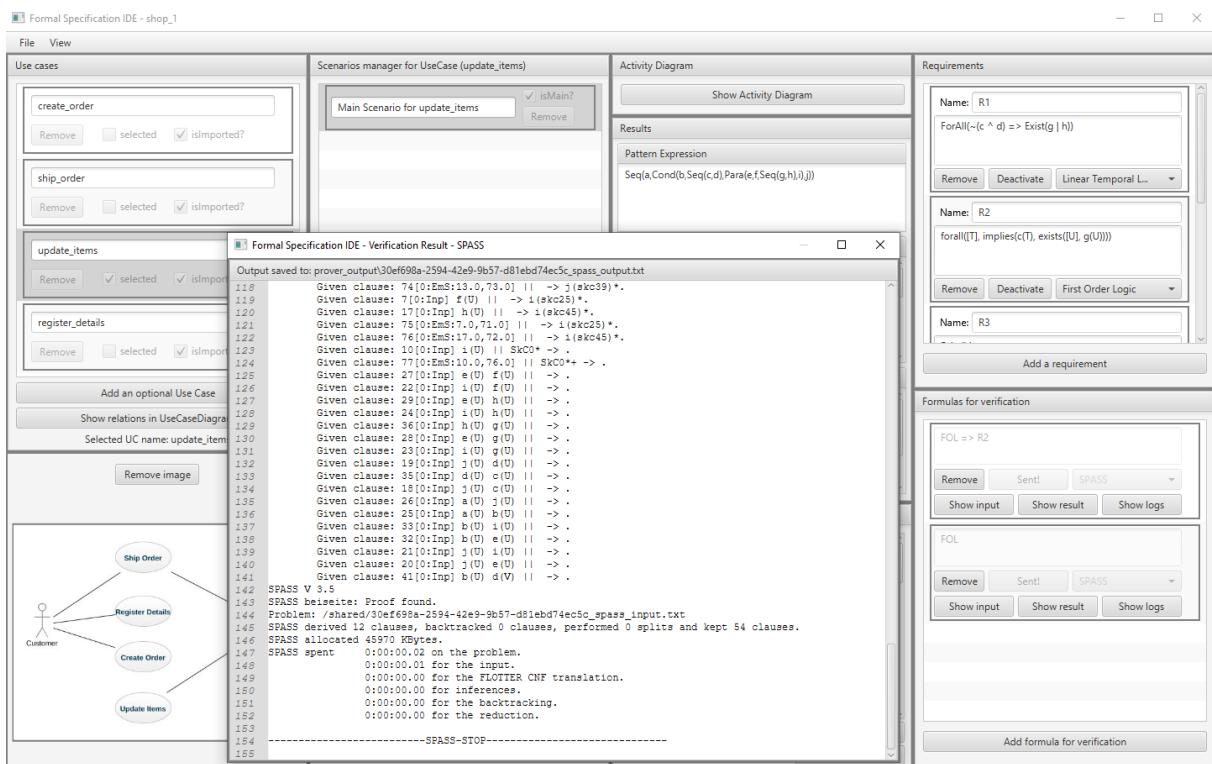
Po zapoznaniu się z treścią wejścia, użytkownik może inicjować proces weryfikacji, klikając „Send to Prover”. Aplikacja nawiązuje komunikację z wybranym narzędziem do dowodzenia, przekazując mu formułę do analizy. Dalsze kroki wykonywane są w sposób zautomatyzowany, a rezultat zostaje wyświetlony użytkownikowi.

Integracja z *Formal Specification IDE* została zrealizowana z użyciem konteneryzacji, co zostało dokładnie opisane w sekcji 5.6.1. Treść wejścia przekazywana jest jako plik do kontenera zainstalowanym proverem. Wykonywana jest odpowiednia komenda, której efektem jest plik wyjściowy, który dostępny jest do podglądu pod przyciskiem „Show result”.

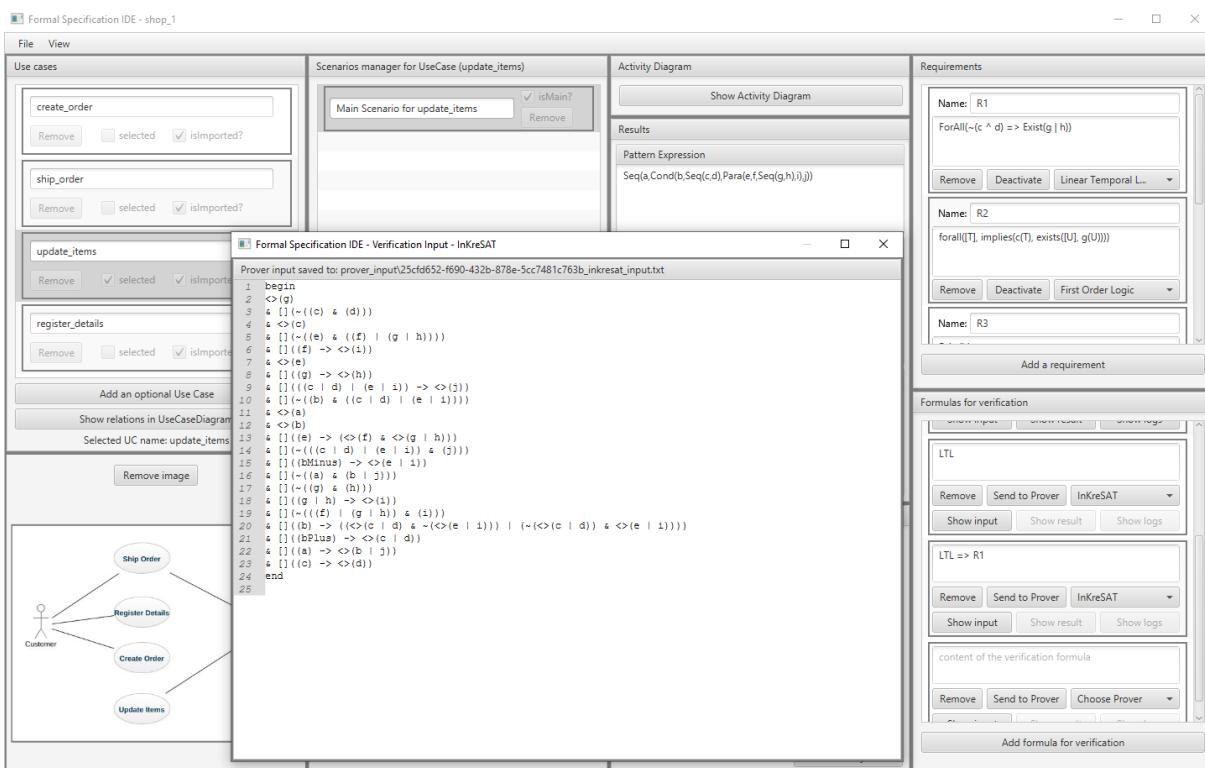
Jeśli dowód twierdzenia zostanie odnaleziony, aplikacja poinformuje o tym użytkownika, prezentując odpowiednie komunikaty oraz szczegóły dotyczące znalezionej dowody. W sytuacji, gdy proces weryfikacji zakończy się niepowodzeniem, użytkownik może przeanalizować logi provera, dostępne poprzez przycisk „Show logs”.



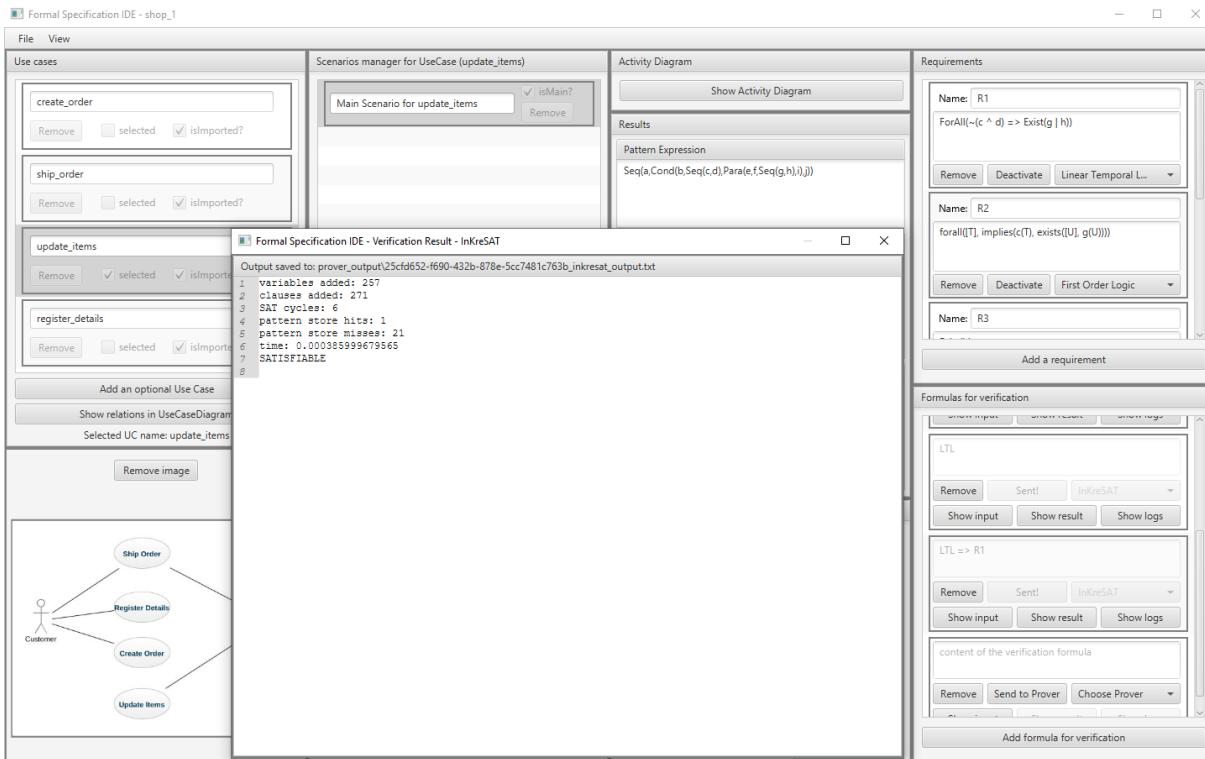
Rys. 4.25. Przykład wejścia dla systemu SPASS Prover.



Rys. 4.26. Przykład wyniku weryfikacji dla systemu SPASS Prover. Dowód został znaleziony.



Rys. 4.27. Przykład wejścia dla systemu InKreSAT.



Rys. 4.28. Przykład wyniku weryfikacji dla systemu InKreSAT. Dana formuła jest satysfakcjonująca (SATISFIABLE).

Proces weryfikacji, oferowany przez ten panel, stanowi niezwykle istotny etap w tworzeniu modeli behawioralnych. Pozwala na precyzyjne sprawdzenie ich poprawności, identyfikację potencjalnych błędów oraz ich naprawę. To narzędzie daje użytkownikom pewność co do jakości oraz zgodności stworzonych modeli z założeniami projektowymi.

Proces weryfikacji jest nieodzownym krokiem podczas projektowania systemów informacyjnych, gwarantującym, że stworzone modele są zgodne z oczekiwaniami oraz spełniają określone wymagania.

## 4.11. Przykłady zastosowania systemu

W tej sekcji przedstawione zostaną przykłady zastosowania systemu *Formal Specification IDE* na rozbudowanych przykładach.

### 4.11.1. System obsługi pasażerów na lotnisku

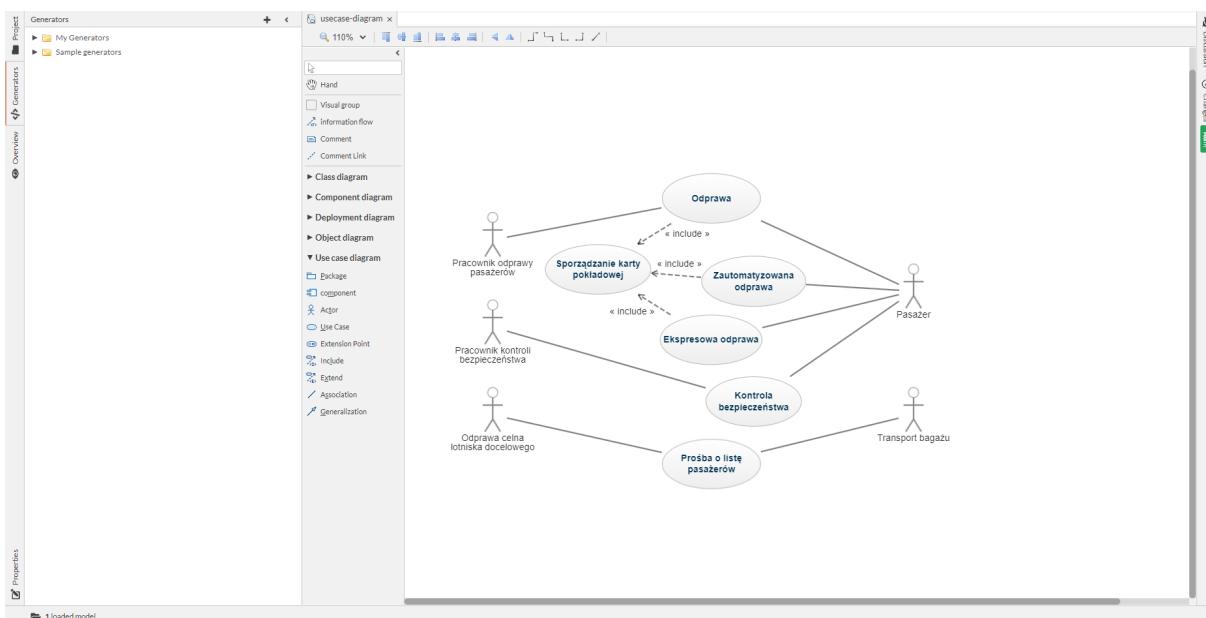
Omawianym systemem jest system obsługi pasażerów na lotnisku. Problem został opracowany na podstawie przykładu zawartego w książce [24]. Przykład został ograniczony do obszarów lotniska, z którymi pasażerowie stykają się podczas odlotu.

W przypadku obsługi pasażerów na lotnisku, pracownicy przy odprawie mają do czynienia z pasażerami, biletami lotniczymi i lotami, które są prawdziwe. Z drugiej strony, w systemie informacyjnym znajduje się reprezentacja lub obraz tych pasażerów, biletów lotniczych i lotów. Obrazy te składają się z informacji o pasażerach, biletach i lotach przechowywanych w systemie informatycznym, potrzebnych do procesów operacyjnych.

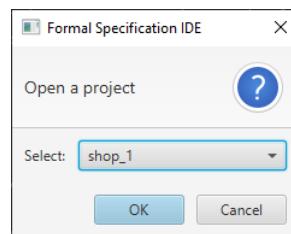
W tym przypadku system informatyczny ma zostać zintegrowany z usługą pasażerów. Dlatego system biznesowy został zawężony tylko do obsługi pasażerów. Pozostałe działy, takie jak transport bagażu czy catering, zostaną potraktowane jako zewnętrzne systemy biznesowe.

Diagramy przypadków użycia pokazują przypadki użycia, aktorów i relacje między nimi. Relacje między aktorami a przypadkami użycia stwierdzają, że aktor może korzystać z określonej funkcjonalności systemu biznesowego. Na rysunku 4.29 przedstawiono zamodelowany diagram przypadków użycia dla omawianego przykładu, wykonany przy użyciu zewnętrznego narzędzia GenMyModel [15]. Ten i kilka pozostałych przykładów systemów do modelowania diagramów przypadków użycia został opisany 2.3. Z aplikacji GenMyModel został wyeksportowany plik projektu .xml oraz grafika .jpeg. Następnie te elementy mogą zostać zimportowane do systemu *Formal Specification IDE*.

Na diagramie 4.29 przedstawiono aktorów: pasażera, pracownika odprawy, odprawę celną na lotnisku docelowym oraz transport bagażu. Wykazano również przypadki użycia związane



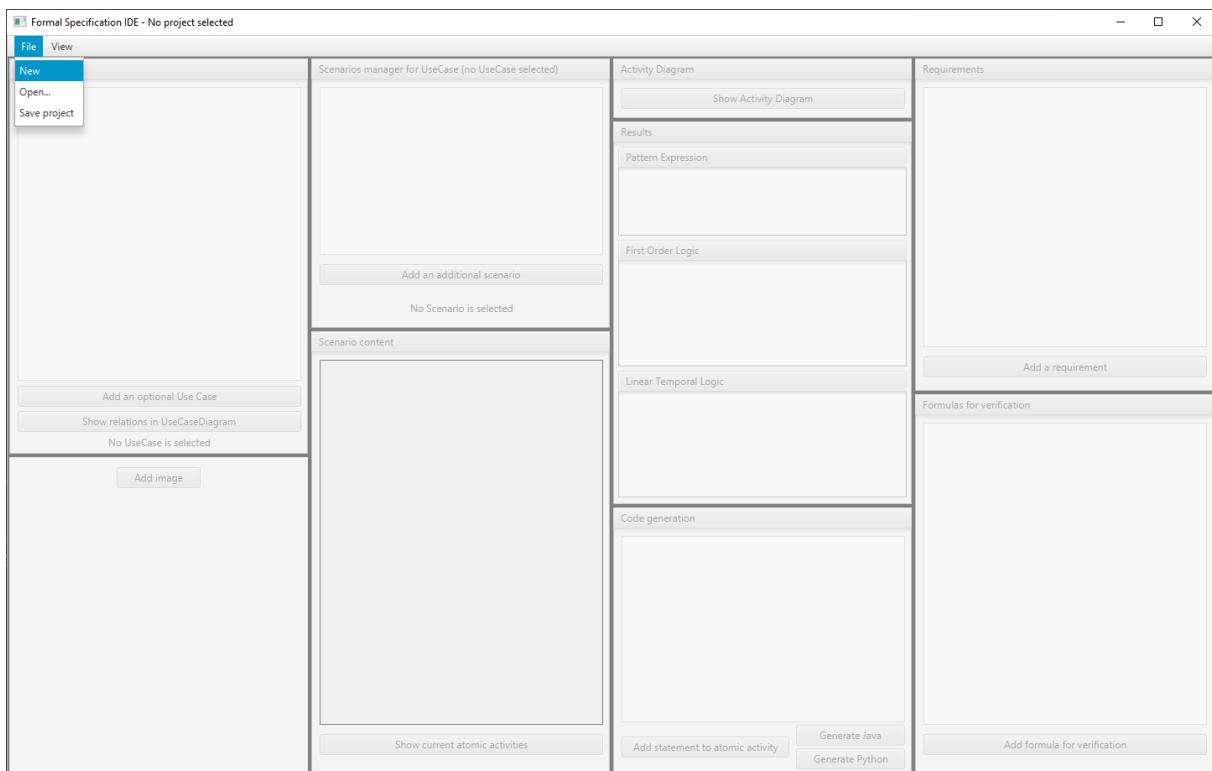
Rys. 4.29. Diagram przypadków użycia dla systemu obsługi pasażerów na lotnisku. Projekt został wykonany z użyciem aplikacji GenMyModel[15].



Rys. 4.30. Bezpośrednio po uruchomieniu aplikacji *FS-IDE* pojawia się okno z możliwością wyboru jednego z istniejących projektów. W przypadku wyboru „Cancel”, zostanie wyświetlane główne okno, z możliwością stworzenia nowego projektu.

z obsługą pasażerów, takie jak odprawa, ekspresowa odprawa, zautomatyzowana odprawa i kontrola bezpieczeństwa.

Zaczynając od pasażera, zawarte są powiązania z czterema biznesowymi przypadkami użycia, odprawą, ekspresową odprawą, zautomatyzowaną i kontrolą bezpieczeństwa. Oznacza to, że osoby występujące jako pasażerowie mogą przejść przykładowo przez odprawę lub odprawę ekspresową, którą można przeprowadzić bez bagażu. Pracownik odprawy pasażerów ma również powiązanie z odprawą. Oznacza to, że nie tylko pasażer, ale także ktoś, kto go reprezentuje, może się odprawić. Fakt, że pracownik odprawy jest powiązany tylko z przypadkiem użycia „Odprawa” oznacza, że na lotnisku przedstawiciel pasażera nie może dokonać ekspresowej odprawy.

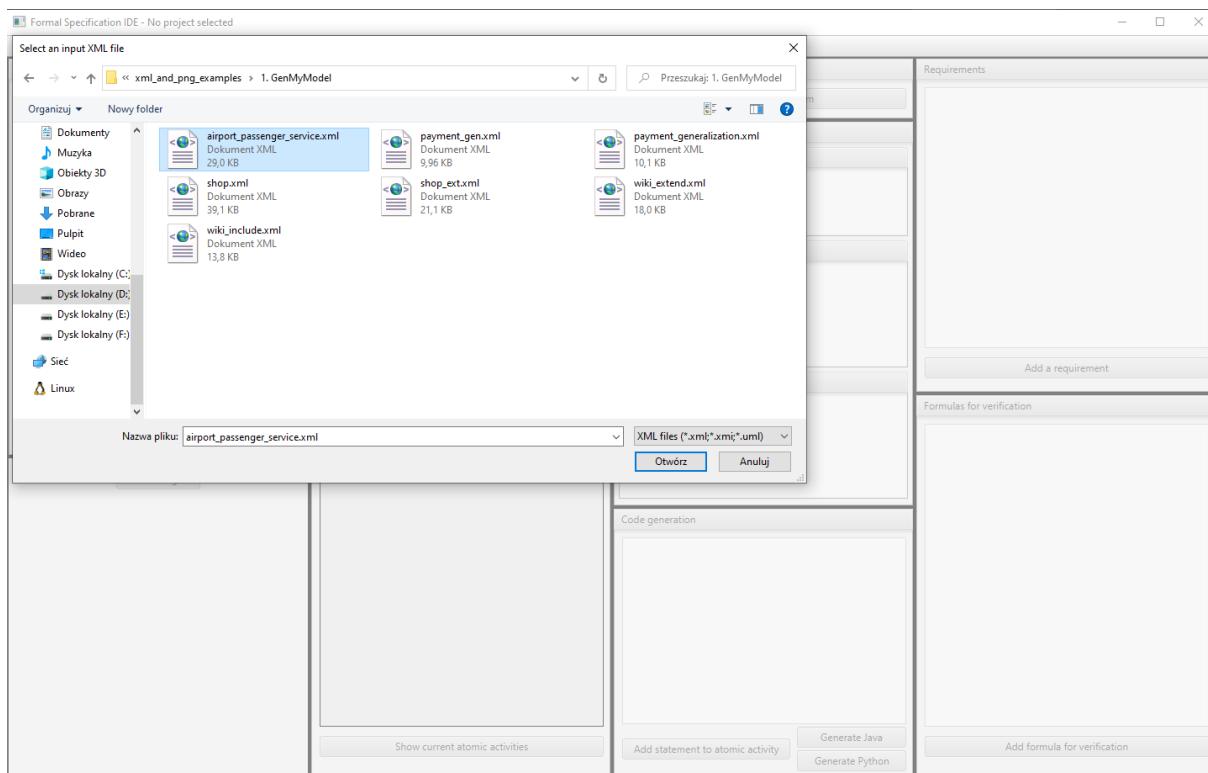


**Rys. 4.31.** W menu, pod opcją „File” znajduje się opcja stworzenia nowego projektu („New”).

Procedura odprawy obejmuje złożenie biletu, odprawę bagażu, rezerwację miejsca oraz wydanie i przekazanie karty pokładowej. Pasażerowie posiadający jedynie bagaż podręczny mogą skorzystać z ekspresowej odprawy. Nie jest przeprowadzana odprawa bagażu. Podczas wejścia na pokład karta pokładowa pasażera jest weryfikowana przy bramce. Automatyczna odprawa odbywa się bez pomocy pracownika odprawy, bezpośrednio przy maszynie. Bagażu nie można odprawić.

Pasażer często podróżuje z bagażem, który odprawia. Transport bagażu odpowiada za załadowanie bagażu do samolotu. Transport bagażu jest realizowany przez niezależną organizację, w związku z tym jest uważany za aktora, a dokładniej za zewnętrznego usługodawcę. Przykładowo dziesięć minut przed odlotem firma zajmująca się transportem bagażu żąda od służb pasażerskich listy pasażerów, która obejmuje każdego pasażera, który odprawił się, ale nie wszedł na pokład samolotu. Na podstawie tej listy cały bagaż, którego to dotyczy, zostanie ponownie wyładowany z samolotu. Jeżeli lot jest lotem międzynarodowym, organy celne kraju, w którym znajduje się lotnisko docelowe, żądają również listy pasażerów.

Karta pokładowa jest generowana i wydawana podczas odprawy. W pewnym momencie podczas odprawy, odprawy ekspresowej lub odprawy automatycznej wydawana jest karta pokładowa.



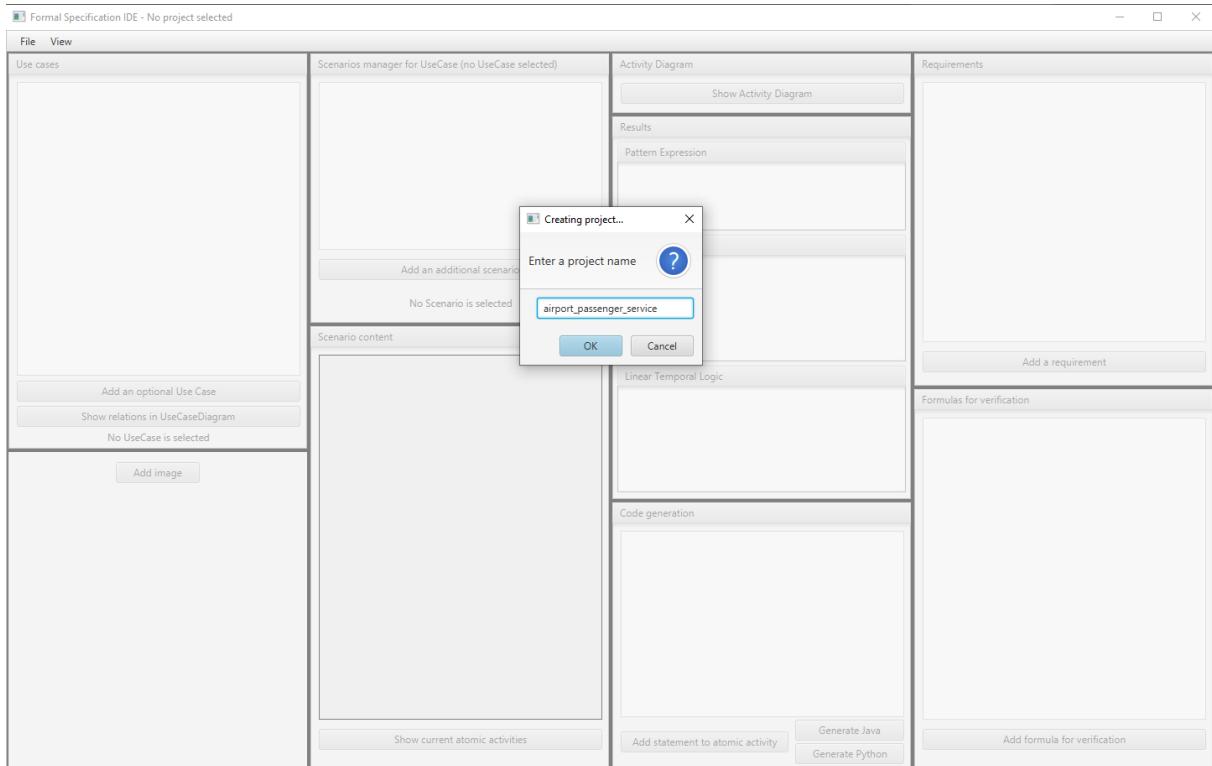
**Rys. 4.32.** Po wyborze opcji stworzenia nowego projektu, pojawia się okno z prośbą o wskazanie pliku .xml z diagramem przypadków użycia.

Proces modelowania przykładu składa się z kilku etapów. Rozpoczyna się od otwarcia aplikacji *Formal Specification IDE*, gdzie istnieje możliwość wyboru istniejącego projektu (rysunek 4.30) lub utworzenia nowego (rysunek 4.31). W przypadku tworzenia nowego projektu, należy wskazać plik .xml z diagramem przypadków użycia (rysunek 4.32). Następnie należy nadać nazwę projektowi (rysunek 4.33), wtedy zostają zaimportowane przypadki użycia, które są wyświetlane w panelu „Use Cases” (rysunek 4.34). Istnieje również opcja dodania grafiki diagramu przypadków użycia (rysunek 4.35) w celu lepszego wizualnego zrozumienia modelu (rysunek 4.36). W przypadku chęci powiększenia grafiki w systemie, po kliknięciu na nią, zostaje wyświetlone okno podglądu, z przybliżania oraz przesuwania grafiki (rysunek 4.37).

Kolejnym krokiem jest wypełnienie scenariuszy przypadków użycia, co odbywa się poprzez wybór konkretnego przypadku użycia z listy dostępnych i dodanie odpowiednich kroków scenariusza, w formie tekstu naturalnego. Scenariusze mogą być wypełniane w panelu „Scenario content” (rysunek 4.38).

Przykładowe treści scenariuszy, dla wszystkich rozważanych przypadków użycia, zostały przedstawione na rysunkach: 4.39, 4.40, 4.41, 4.42, 4.43, 4.44.

Podczas tworzenia scenariuszy, w każdym momencie dostępny jest podgląd aktualnych atomicznych aktywności, pod przyciskiem „Show current atomic activities”, co zostało pokazane na rysunku 4.45.



**Rys. 4.33.** Po wskazaniu pliku .xml pojawia się okno z prośbą o nadanie nazwy projektowej.

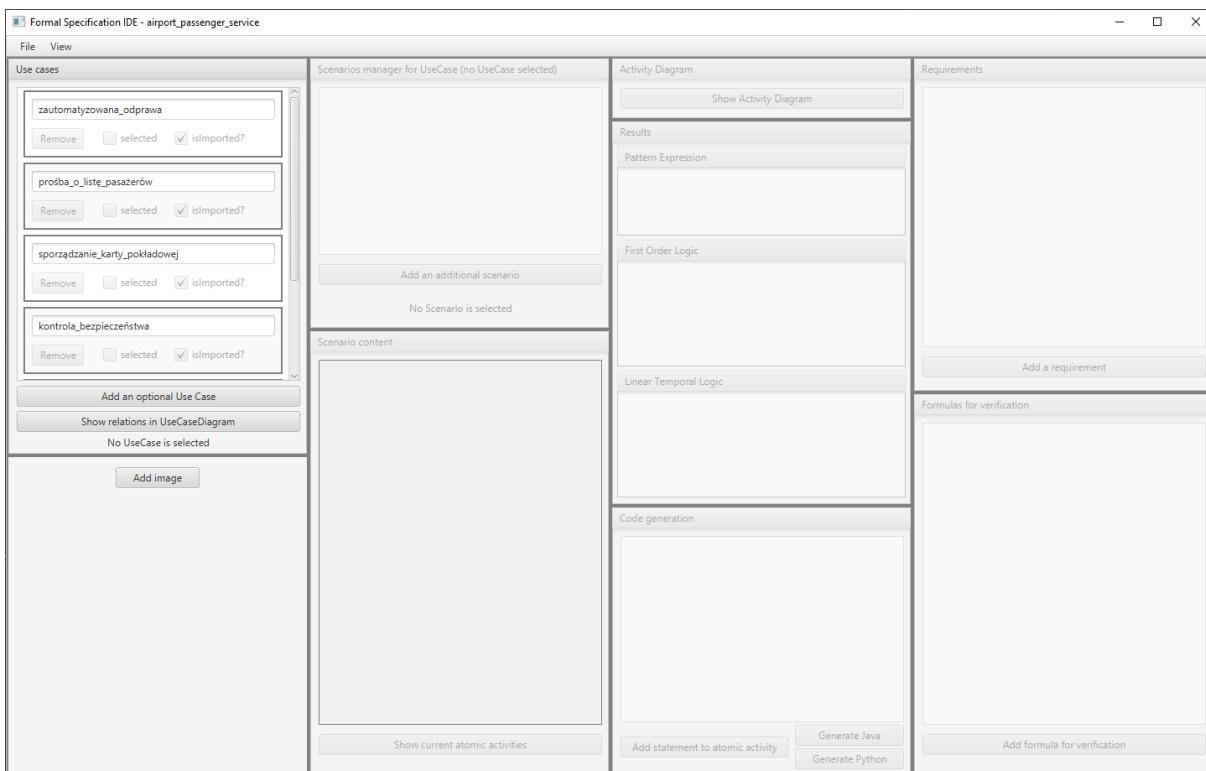
Po wypełnieniu scenariuszy można przejść do modelowania diagramu aktywności. Na podstawie scenariusza tworzony jest diagram aktywności, który ilustruje przebieg akcji i zachowań w systemie. Na diagramie można zobaczyć, jakie aktywności są wykonywane i w jakiej kolejności. Edytor diagramów aktywności można uruchomić klikając przycisk „Show Activity Diagram” (rysunek 4.46).

Przykładowe diagramy aktywności, dla wszystkich rozważanych przypadków użycia, zostały przedstawione na rysunkach: 4.47, 4.49, 4.51, 4.53, 4.55, 4.57.

Po zamodelowaniu diagramu aktywności można przystąpić do generowania specyfikacji. System generuje wyrażenia wzorcowe oraz specyfikacje logiczne w dwóch różnych logikach (logika temporalna LTL, logika pierwszego rzędu FOL). Wygenerowane specyfikacje można wykorzystać do formalnej weryfikacji.

Przykładowe stworzone wyrażenia wzorcowe oraz wygenerowane specyfikacje logiczne, dla wszystkich rozważanych przypadków użycia, zostały przedstawione na rysunkach: 4.48, 4.50, 4.52, 4.54, 4.56, 4.58.

System *Formal Specification IDE* umożliwia również przeprowadzenie formalnej weryfikacji wygenerowanych specyfikacji, co stanowi kluczową fazę analizy i sprawdzania poprawności modeli. Specyfikacje stworzone w aplikacji mogą być przekazywane do specjalistycznych narzędzi weryfikacyjnych, takich jak InKreSAT, SPASS Prover oraz Prover9 (rysunek 4.61). W



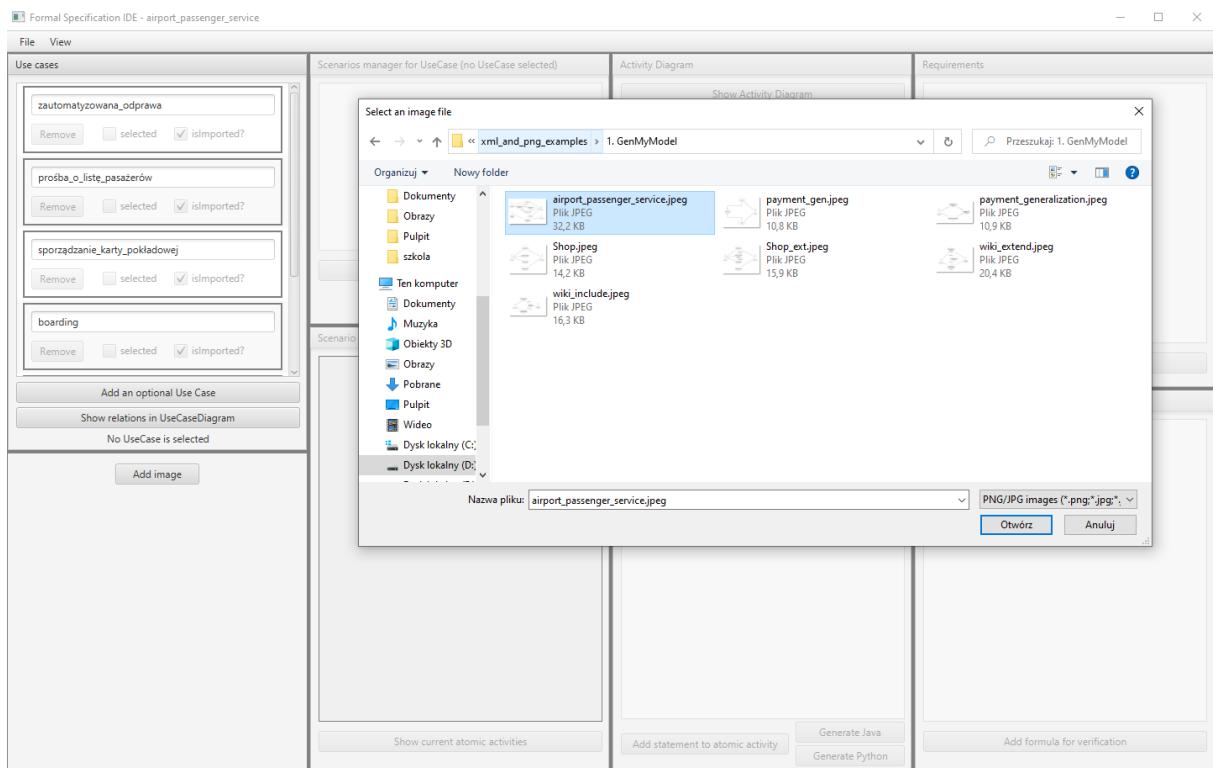
**Rys. 4.34.** Po wpisaniu nazwy projektu, przypadki użycia zostają zaimportowane i wyświetlane na liście w panelu „Use cases”.

wyniku tego procesu, użytkownik otrzymuje szczegółowe informacje dotyczące spełnienia lub niespełnienia określonych warunków lub własności systemu, co pozwala na dokładne zrozumienie zachowania modelu w kontekście spełniania wymagań.

Na przykład, dla przypadku użycia „Odprawa”, można przeprowadzić weryfikację w logice temporalnej LTL. Wynik weryfikacji może przyjąć formę pozytywną lub negatywną, w zależności od stopnia spełnienia określonych warunków. Analogicznie, dla innych przypadków użycia, takich jak „Kontrola bezpieczeństwa” czy „Zautomatyzowana odprawa”, można przeprowadzić weryfikację w logice pierwszego rzędu FOL.

Aby ilustrować wyniki weryfikacji, przedstawiono przykładowe przypadki dla różnych scenariuszy. Na rysunkach 4.62, 4.63, 4.66, 4.64 oraz 4.67 zaprezentowano wyniki weryfikacji dla poszczególnych przypadków użycia. Wyniki mogą mieć charakter pozytywny lub negatywny, co zależy od spełnienia określonych warunków logicznych.

W celu zobrazowania kompletnego procesu weryfikacji, w tabeli 4.1 przedstawiono zestawienie wyników weryfikacji przeprowadzonych dla projektowanego systemu obsługi pasażerów. W tabeli umieszczono przykładowe formuły, które mogą zostać poddane weryfikacji, oraz przedstawiono wyniki pozytywne i negatywne dla każdego zintegrowanego narzędzia weryfikacyjnego. W przypadkach, w których uzyskano negatywne wyniki, poprawiono formułę i uzyskano wynik pozytywny.

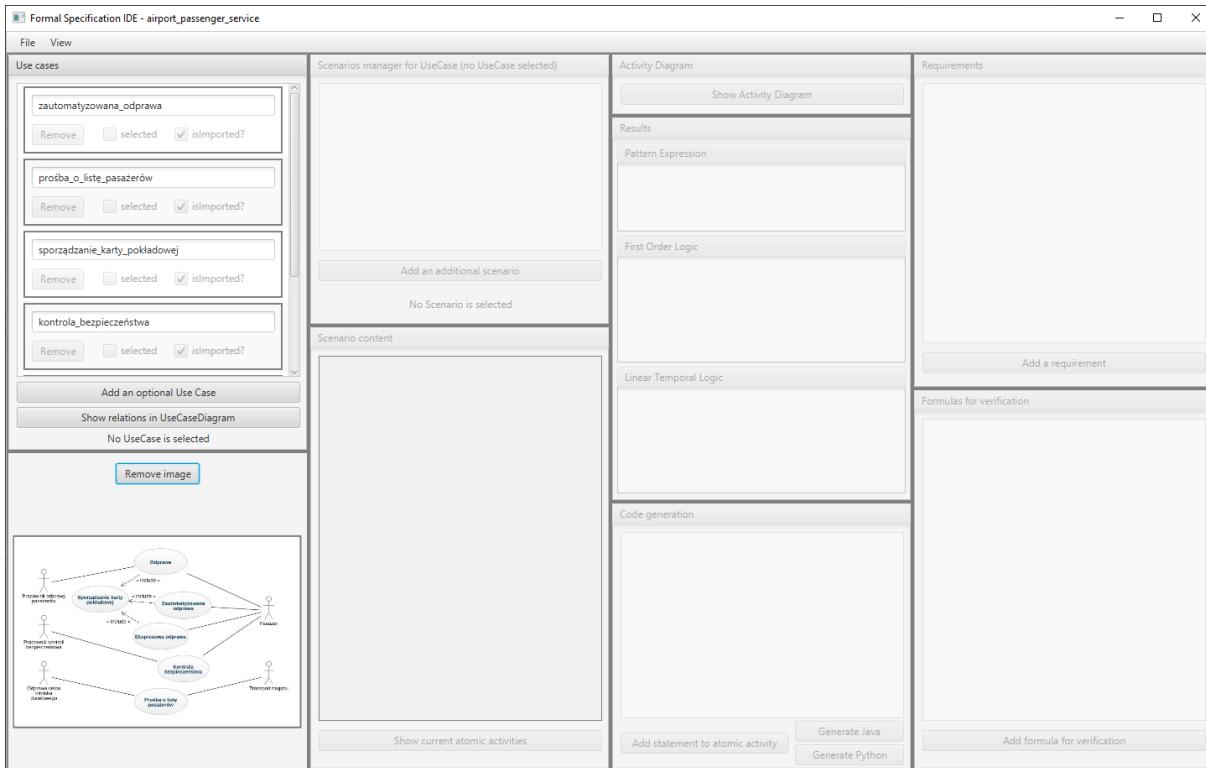


**Rys. 4.35.** Istnieje możliwość wskazania pliku („Add image”) z grafiką diagramu przypadków użycia, w celu wizualizacji.

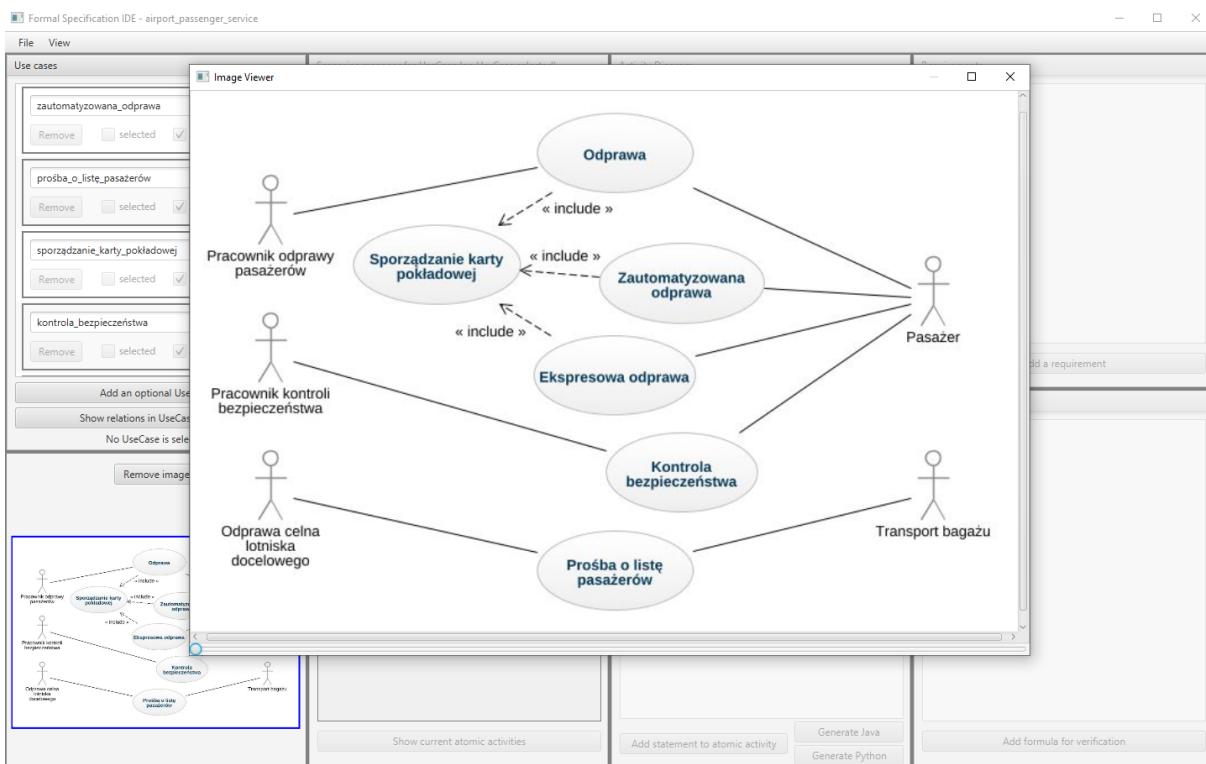
System *Formal Specification IDE* umożliwia również generowanie kodu źródłowego na podstawie zamodelowanego systemu. Wygenerowany kod może być w dwóch różnych językach programowania, takich jak Java, Python. Wykorzystuje się mapowanie aktywności na instrukcje języka programowania.

Przykłady wygenerowanego kodu źródłowego w języku Java i Python można zobaczyć na diagramach 4.59 i 4.60.

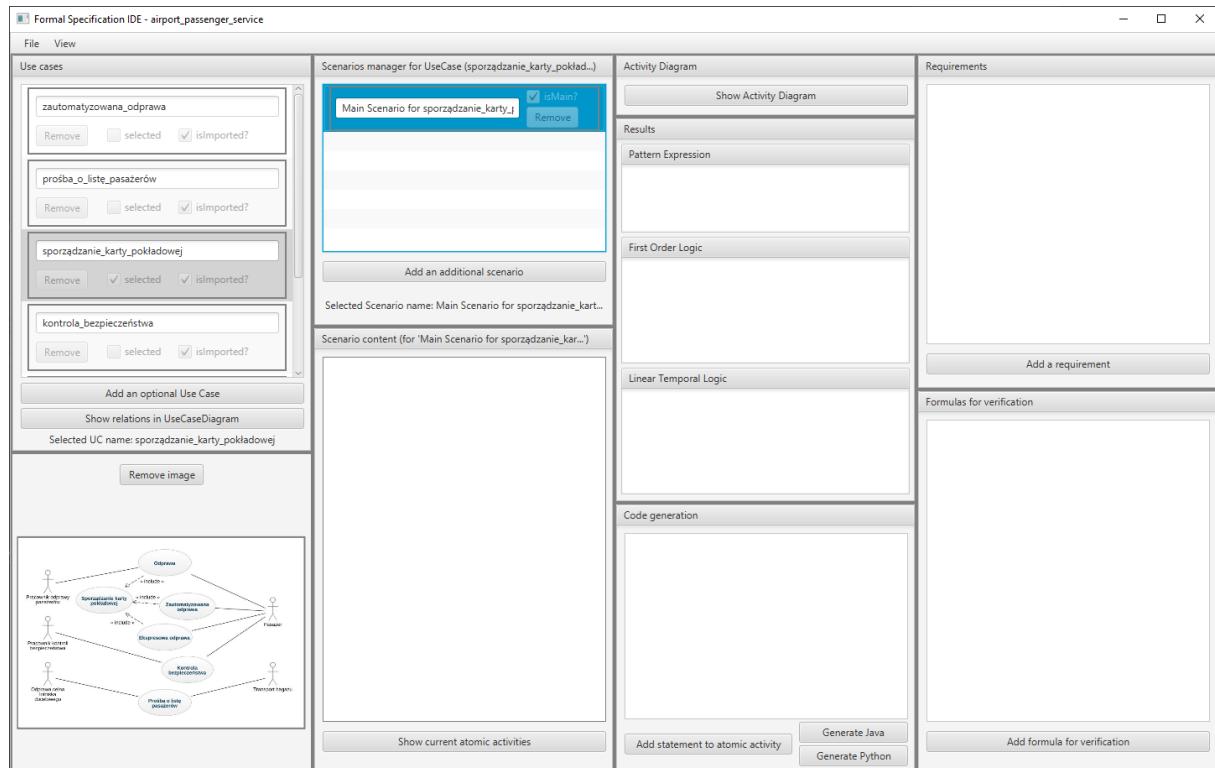
W ten sposób system *Formal Specification IDE* umożliwia modelowanie, generowanie specyfikacji, formalną weryfikację oraz generowanie kodu źródłowego na podstawie zamodelowanego systemu obsługi pasażerów na lotnisku.



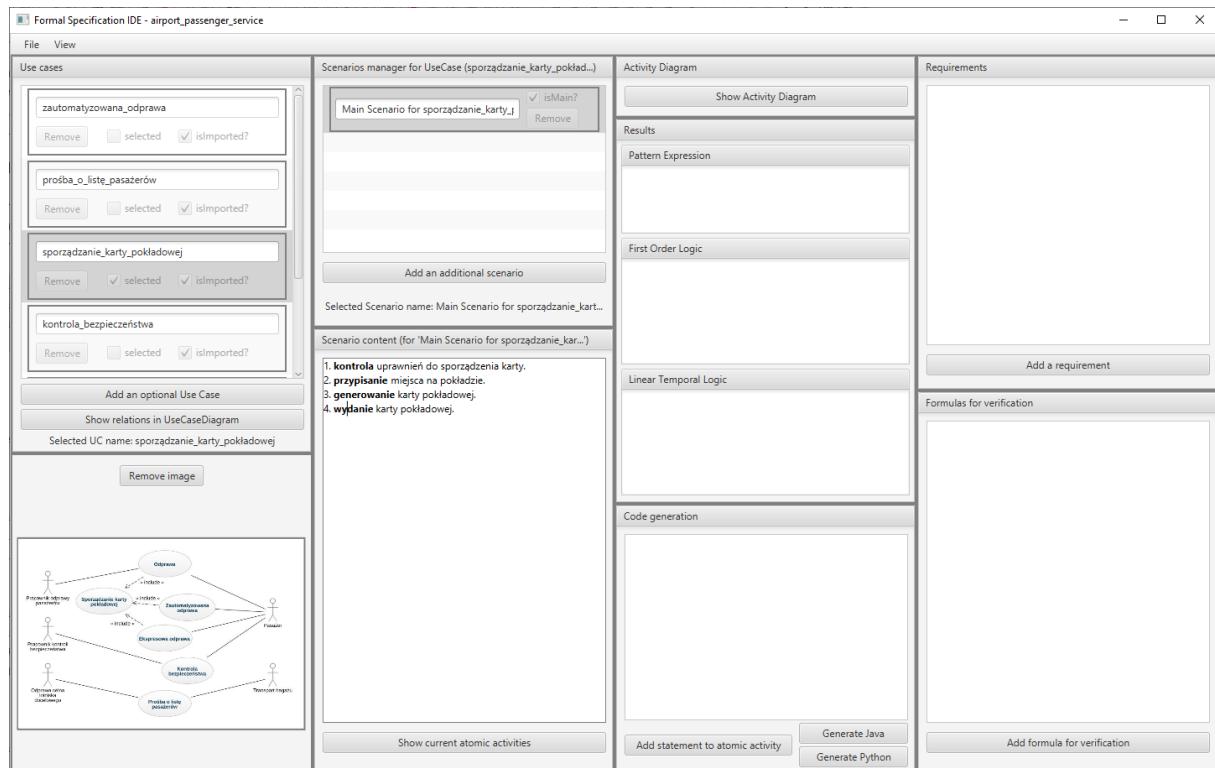
Rys. 4.36. Po wyborze pliku z grafiką, obraz zostaje wyświetlony w lewym dolnym rogu okna aplikacji.



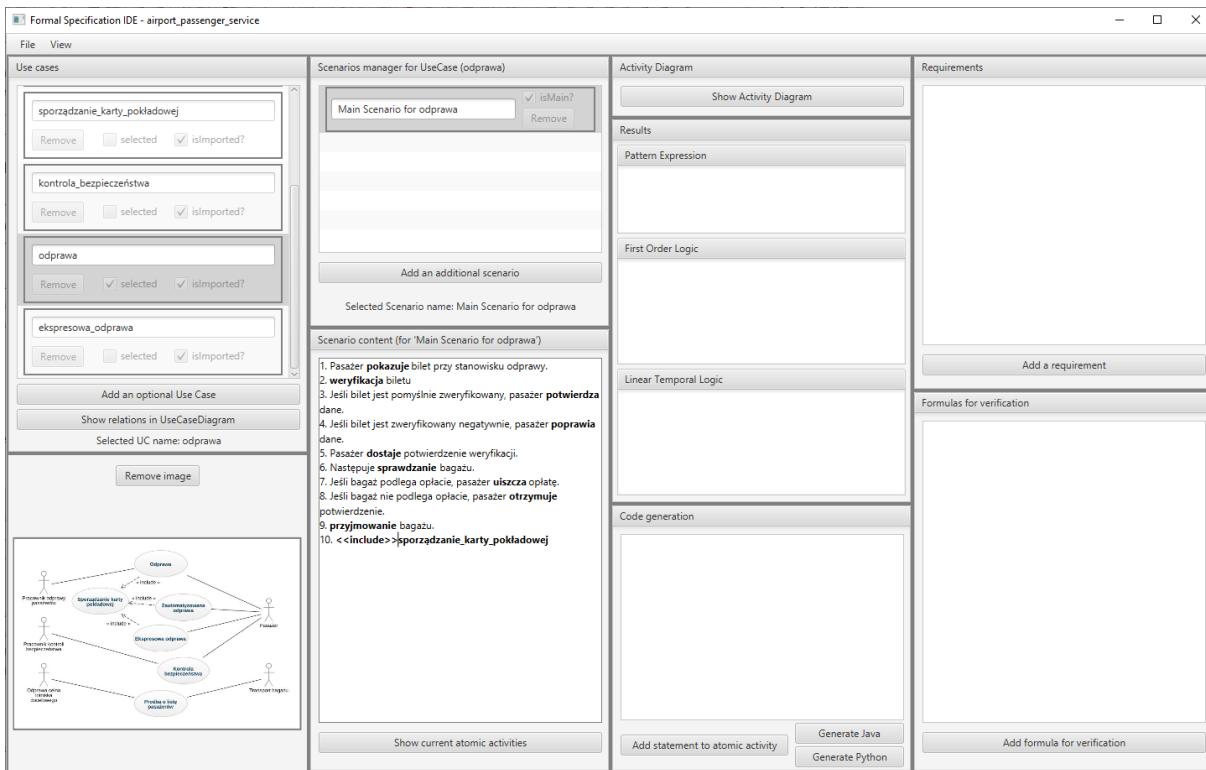
Rys. 4.37. W celu lepszego podglądu grafiki, po kliknięciu na obraz, zostaje wyświetlone nowe okno, z możliwością przesuwania i przybliżania.



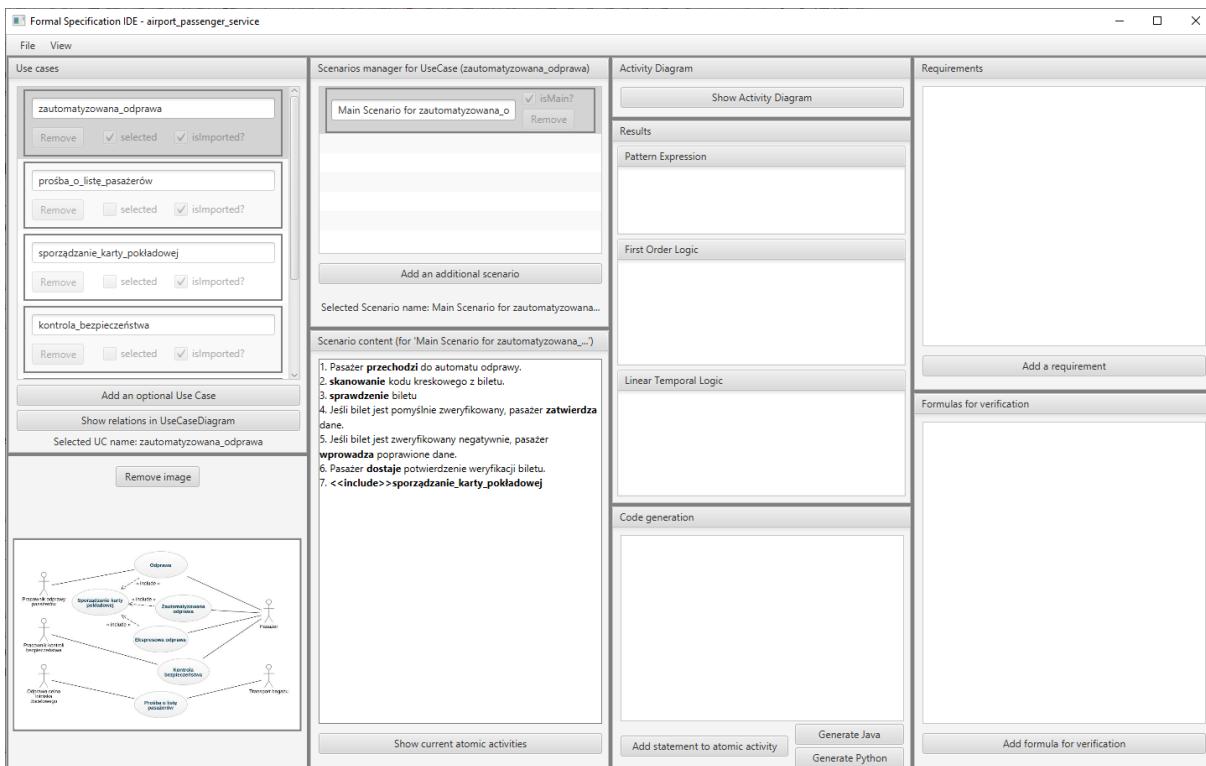
Rys. 4.38. Po kliknięciu na konkretny przypadek użycia, w panelu „Scenarios manager” wyświetlany jest główny scenariusz, który jest domyślnie dodany.



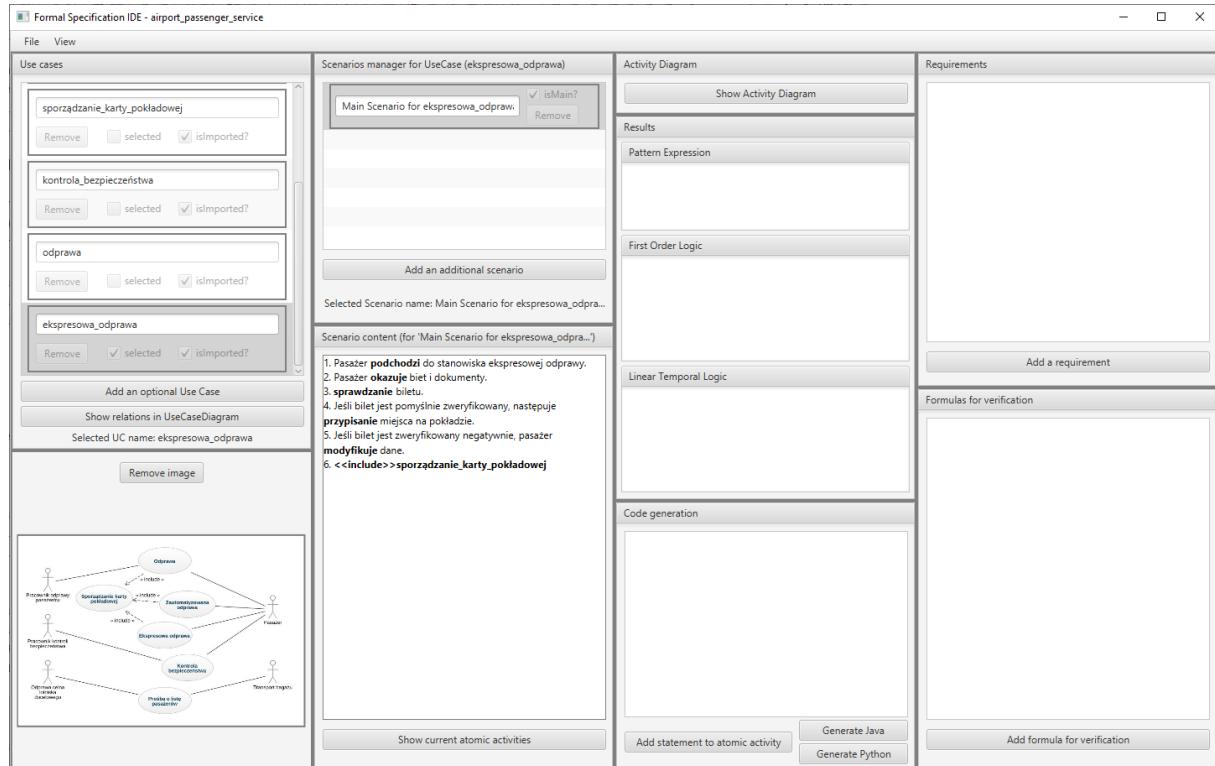
Rys. 4.39. Przykład wypełnienia scenariusza przypadku użycia „Sporządzanie karty pokładowej”.



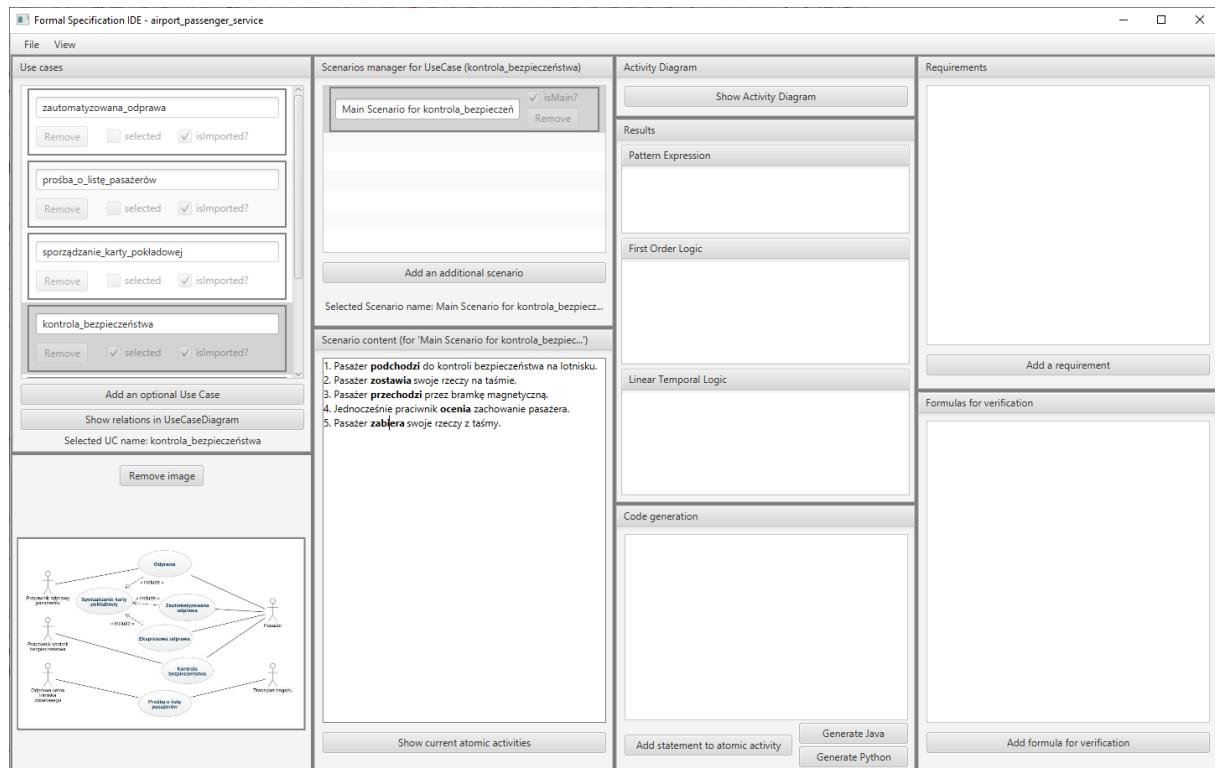
Rys. 4.40. Przykład wypełnienia scenariusza przypadku użycia „Odprawa”.



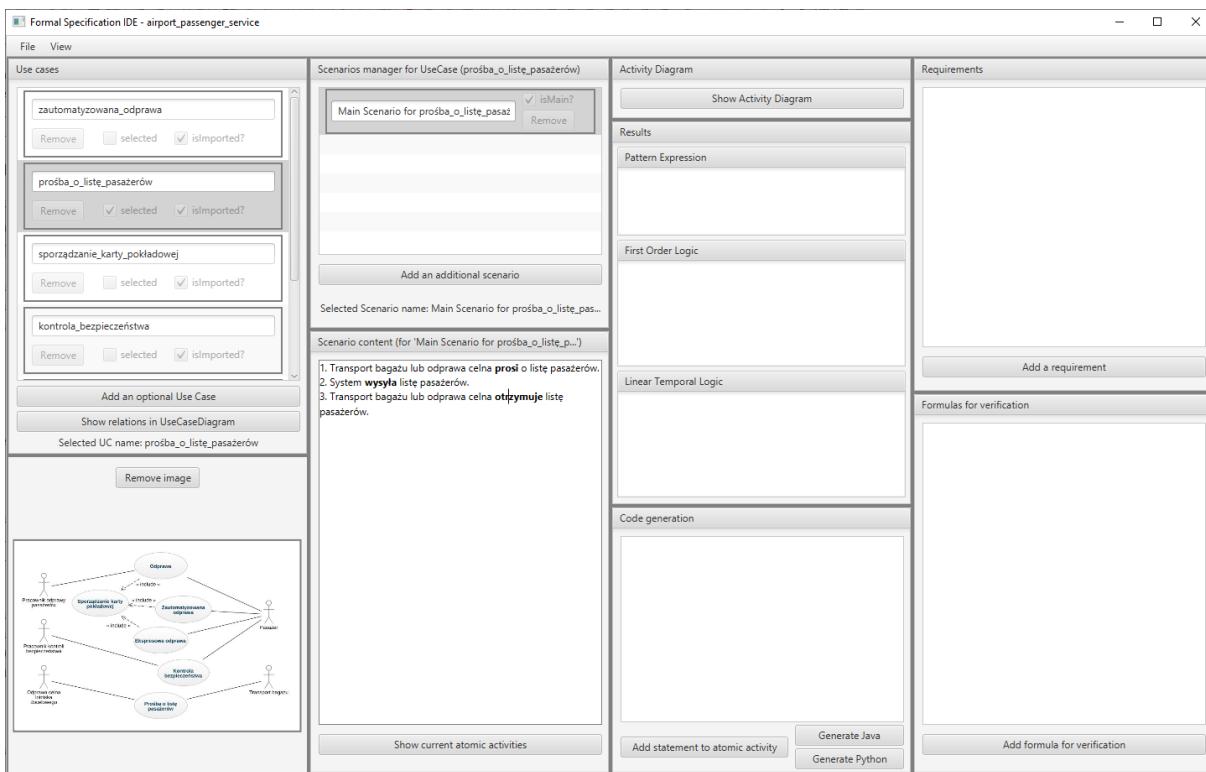
Rys. 4.41. Przykład wypełnienia scenariusza przypadku użycia „Zautomatyzowana odprawa”.



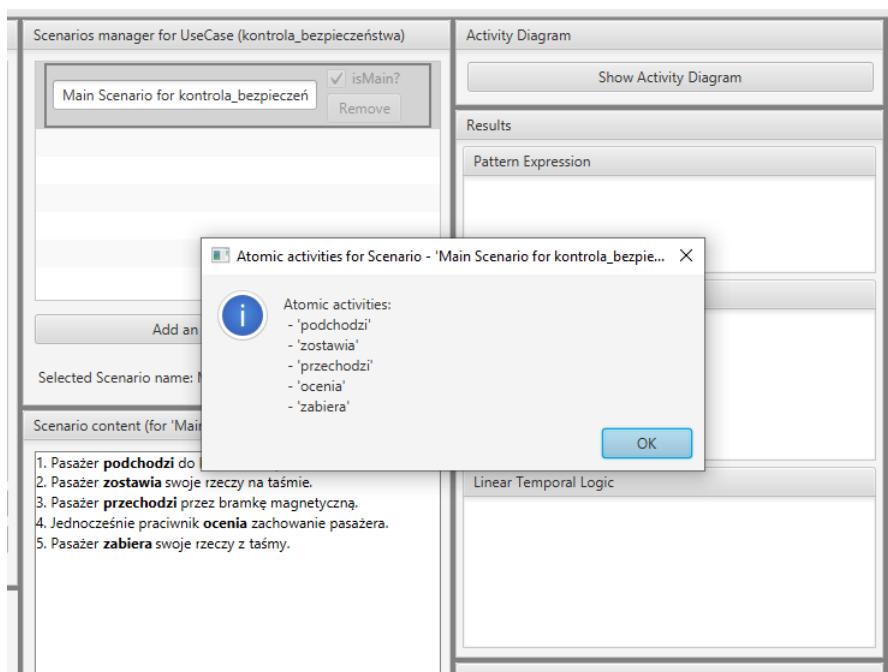
Rys. 4.42. Przykład wypełnienia scenariusza przypadku użycia „Ekspresowa odprawa”.



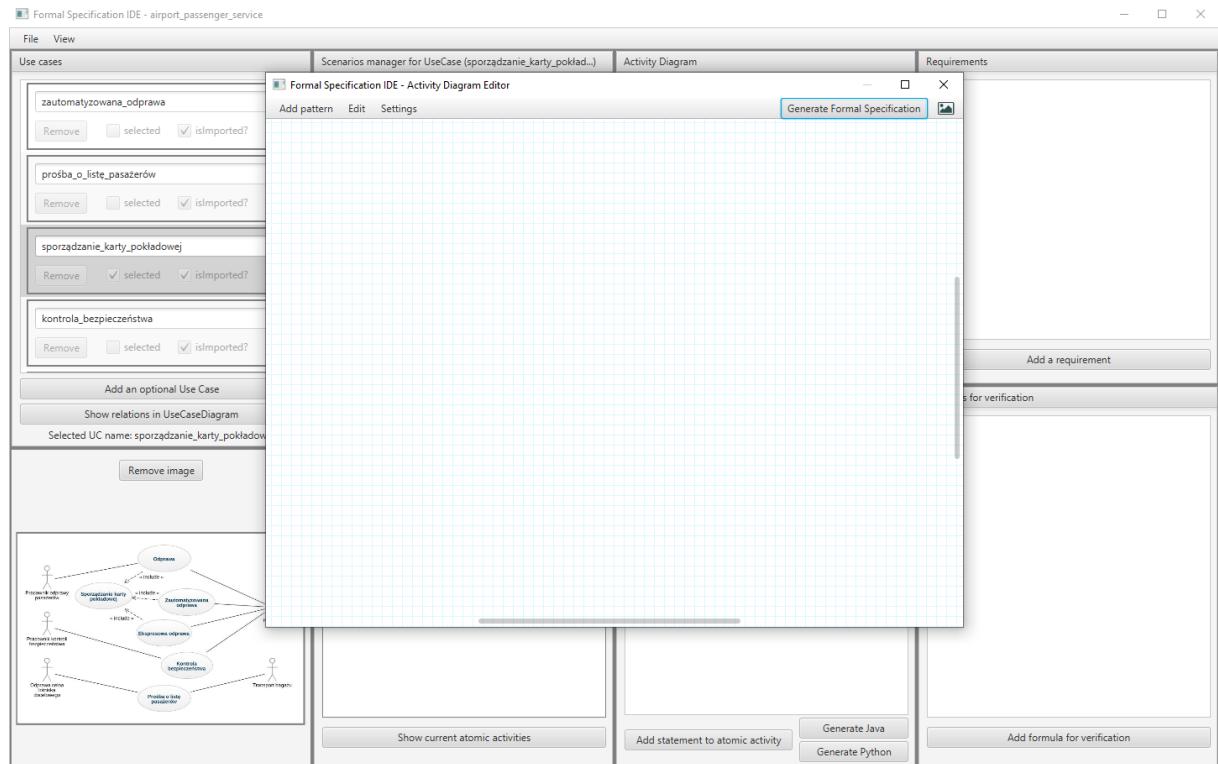
Rys. 4.43. Przykład wypełnienia scenariusza przypadku użycia „Kontrola bezpieczeństwa”.



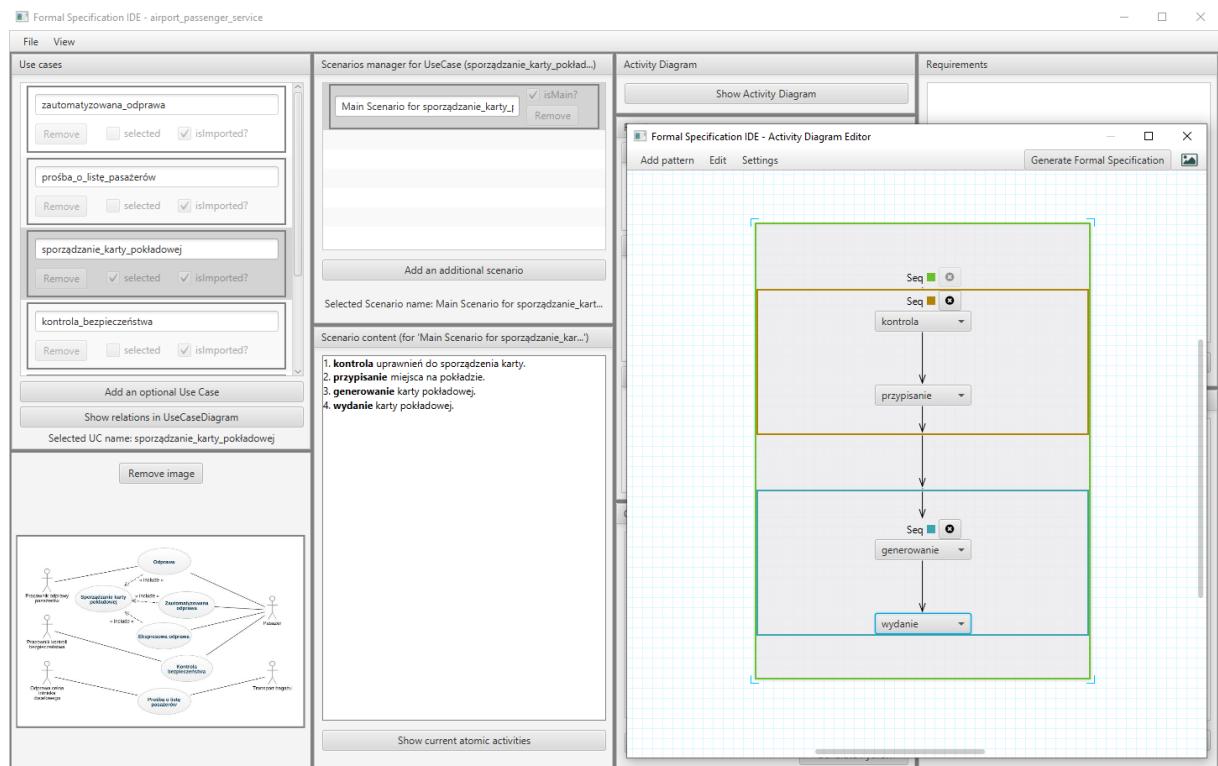
Rys. 4.44. Przykład wypełnienia scenariusza przypadku użycia „Prośba o listę pasażerów”.



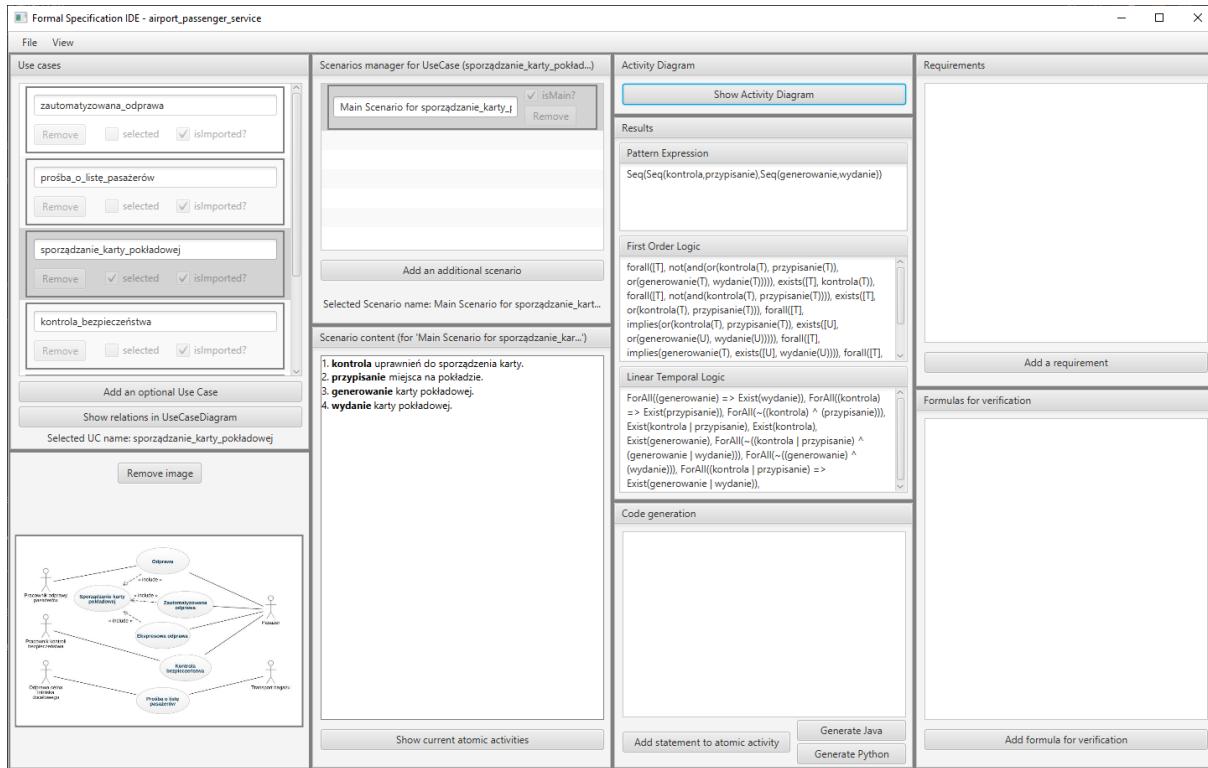
Rys. 4.45. Podczas tworzenia scenariuszy, w każdym momencie dostępny jest podgląd aktualnych atomicznych aktywności, pod przyciskiem „Show current atomic activities”.



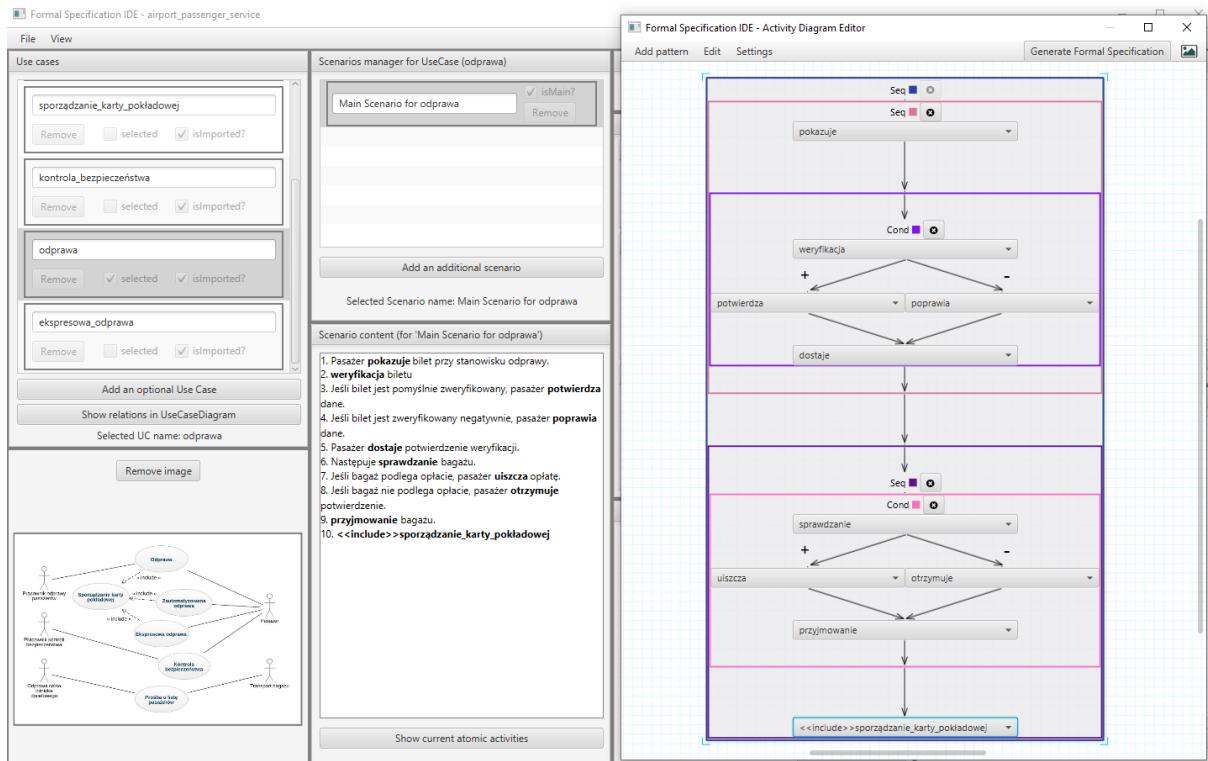
Rys. 4.46. Po uzupełnieniu treści scenariusza, następnym etapem jest modelowanie diagramu aktywności, poprzez kliknięcie „Show Activity Diagram”.



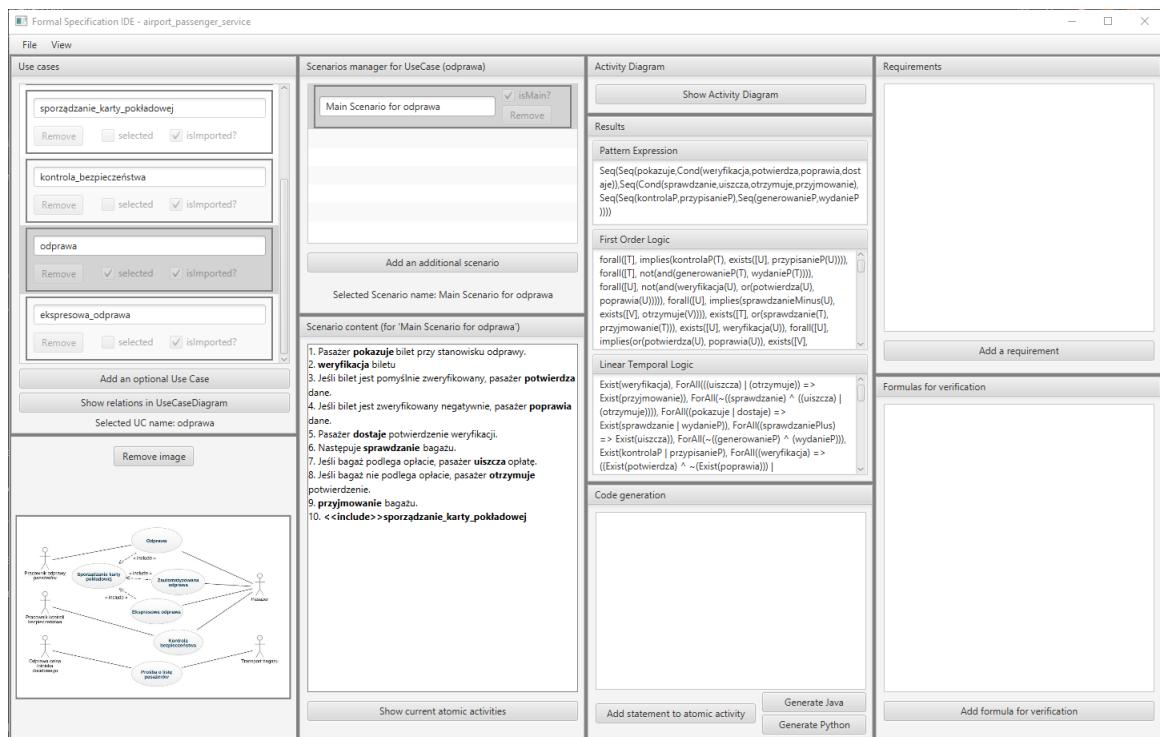
Rys. 4.47. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Sporządzanie karty pokładowej”.



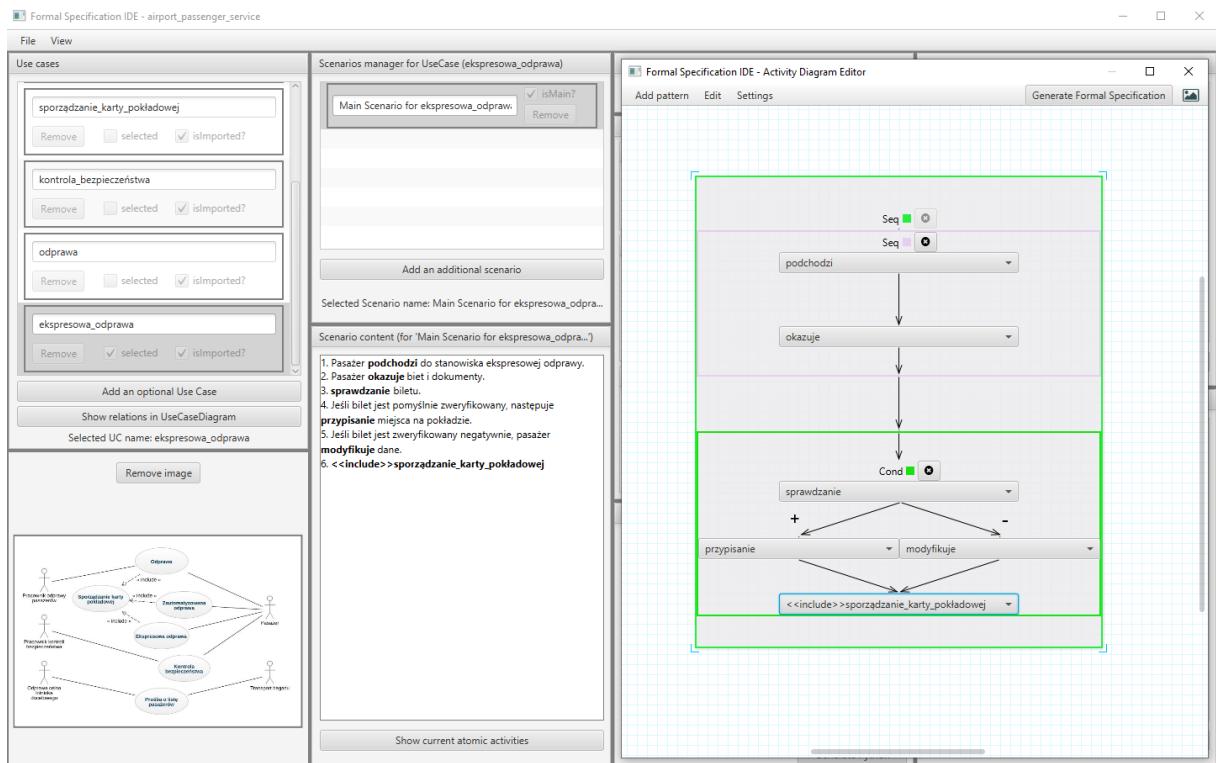
Rys. 4.48. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



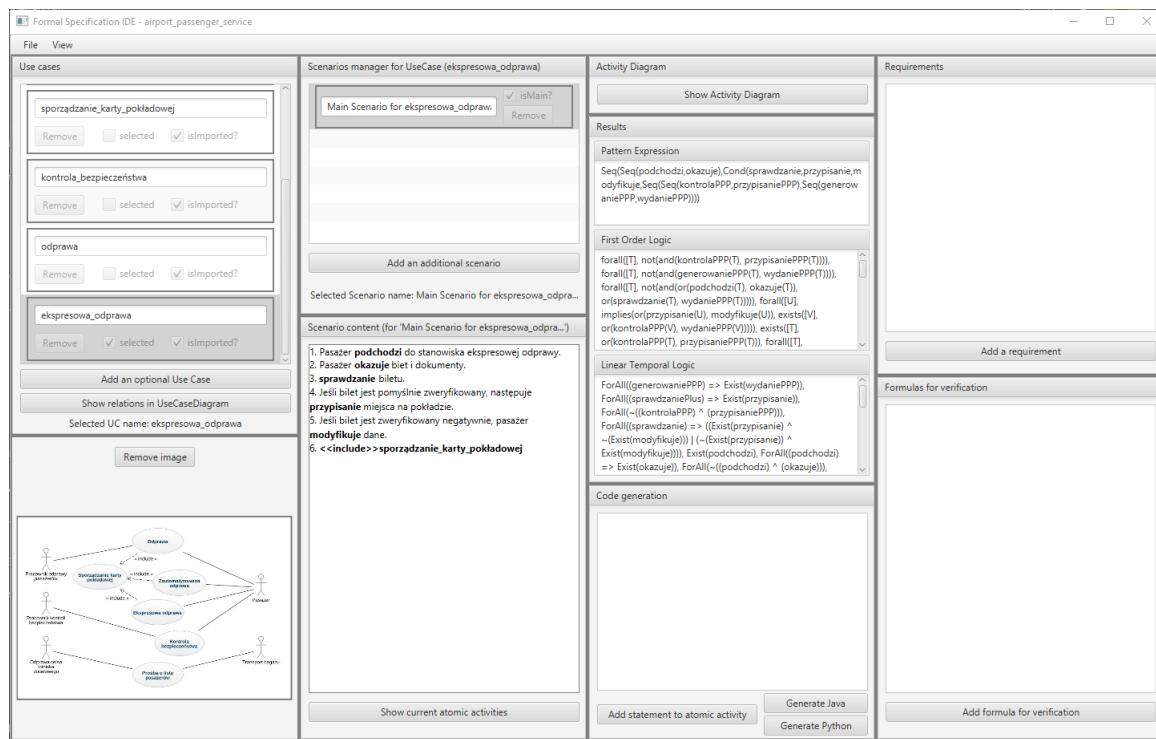
Rys. 4.49. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Odprawa”.



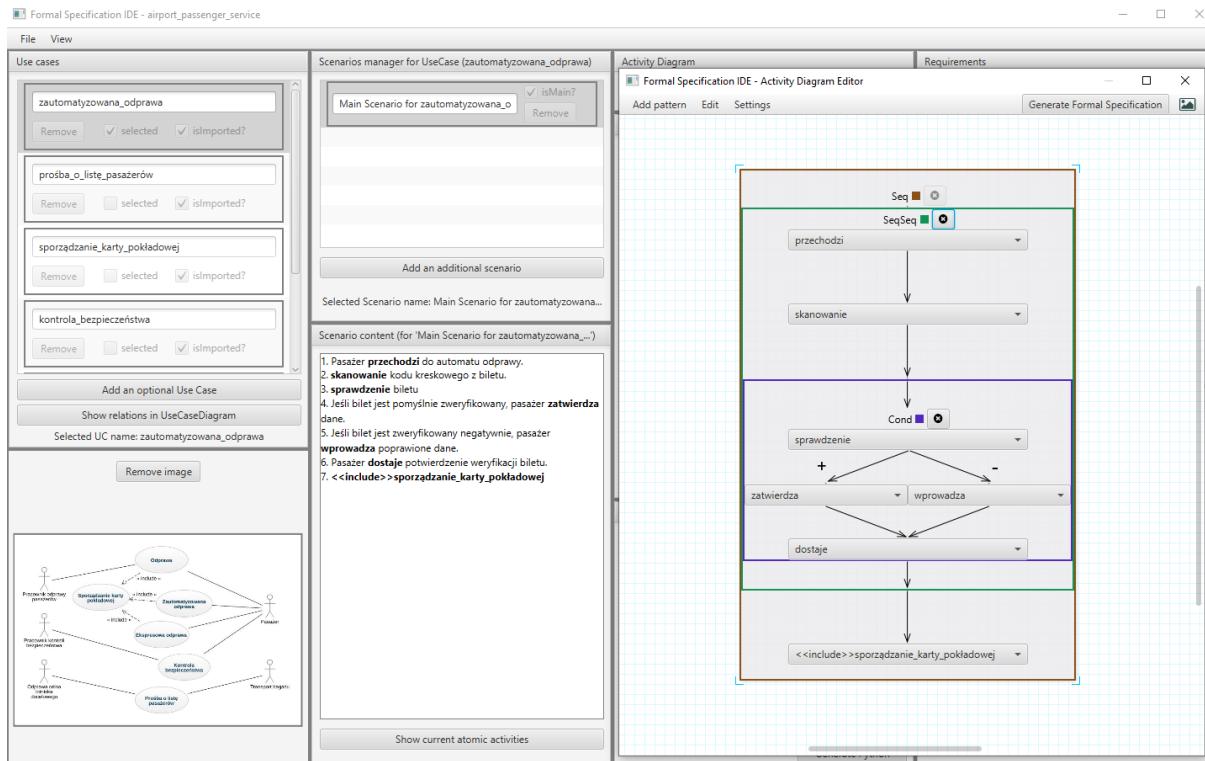
Rys. 4.50. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



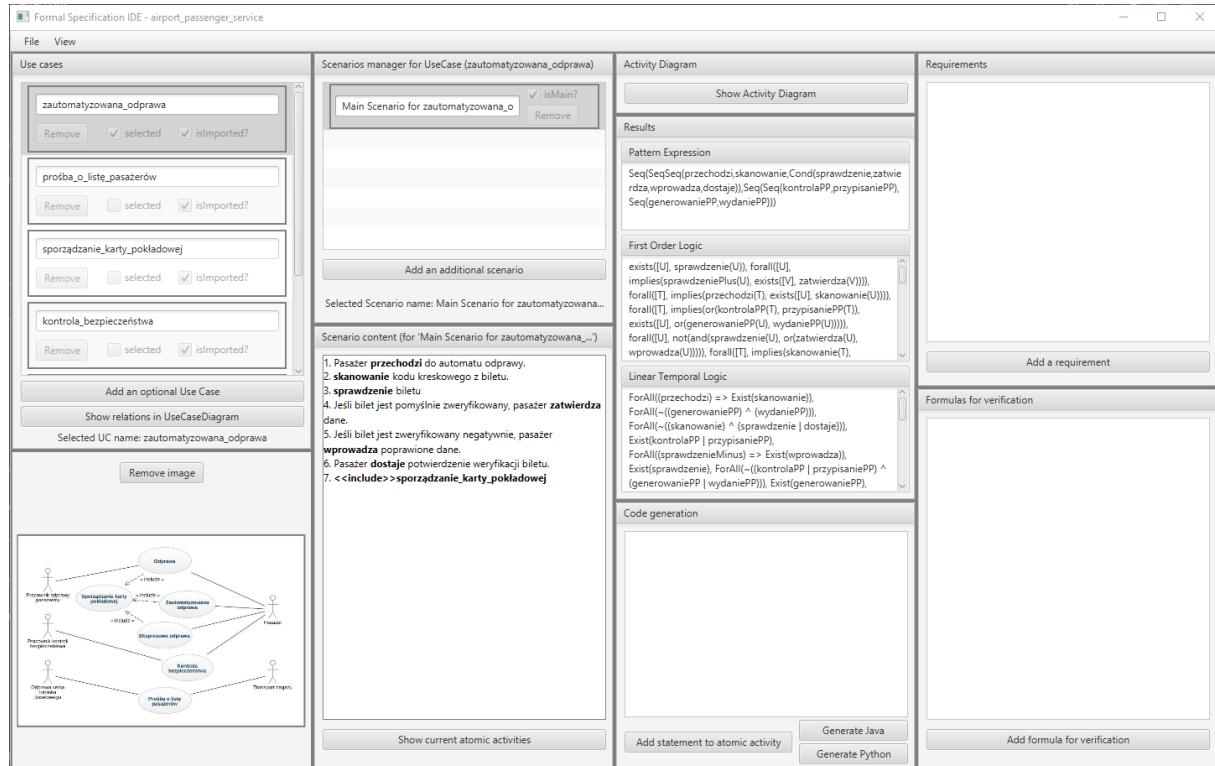
Rys. 4.51. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Ekspresowa odprawa”.



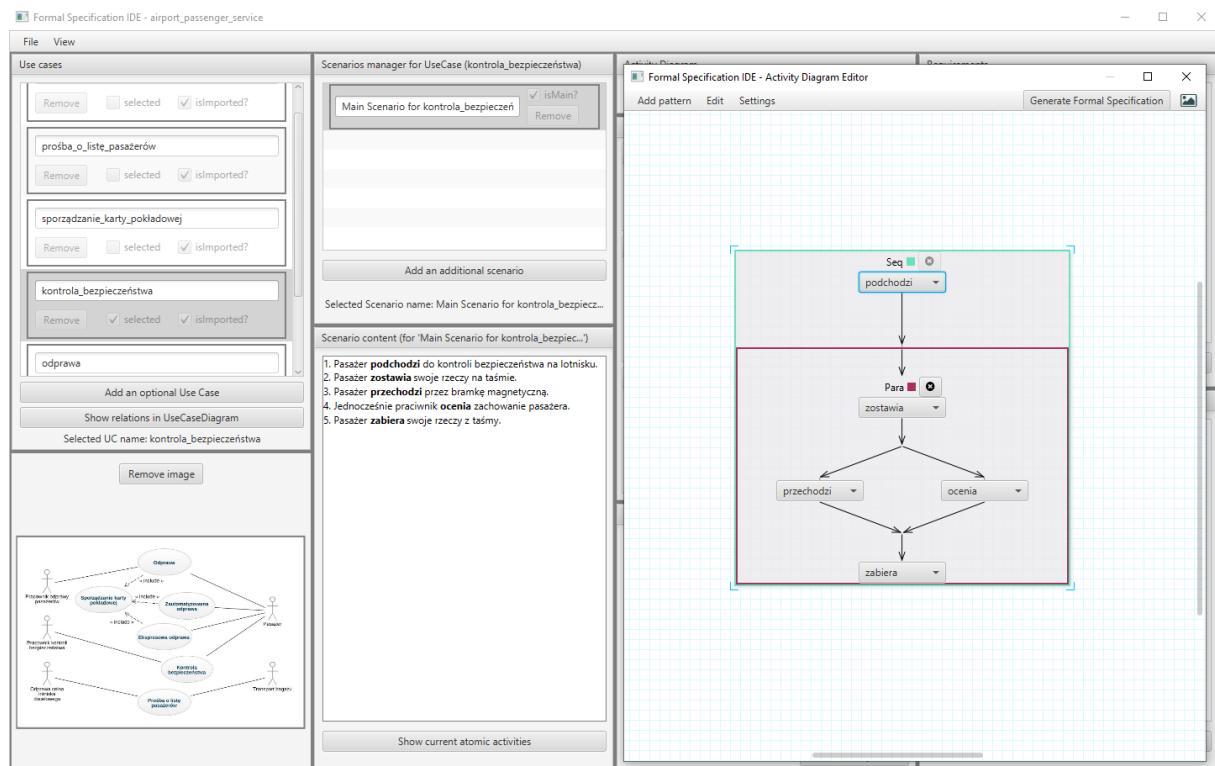
Rys. 4.52. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



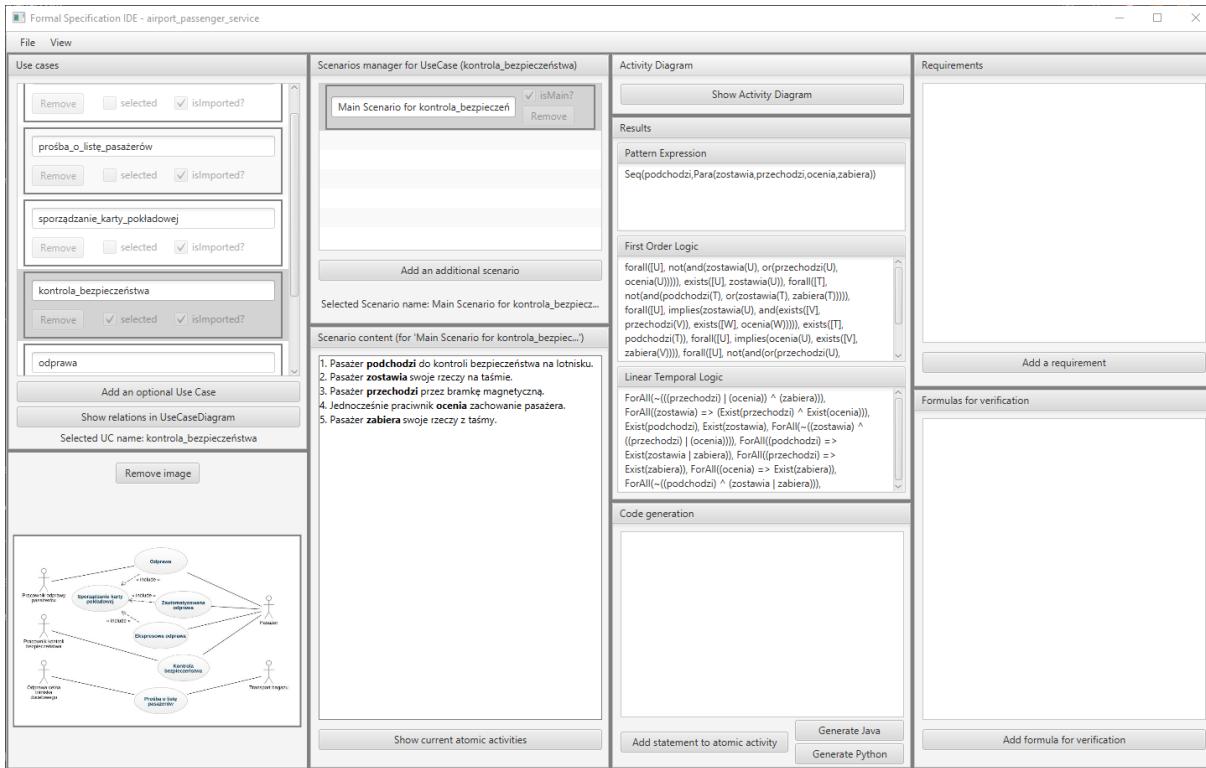
Rys. 4.53. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Zautomatyzowana odpawa”.



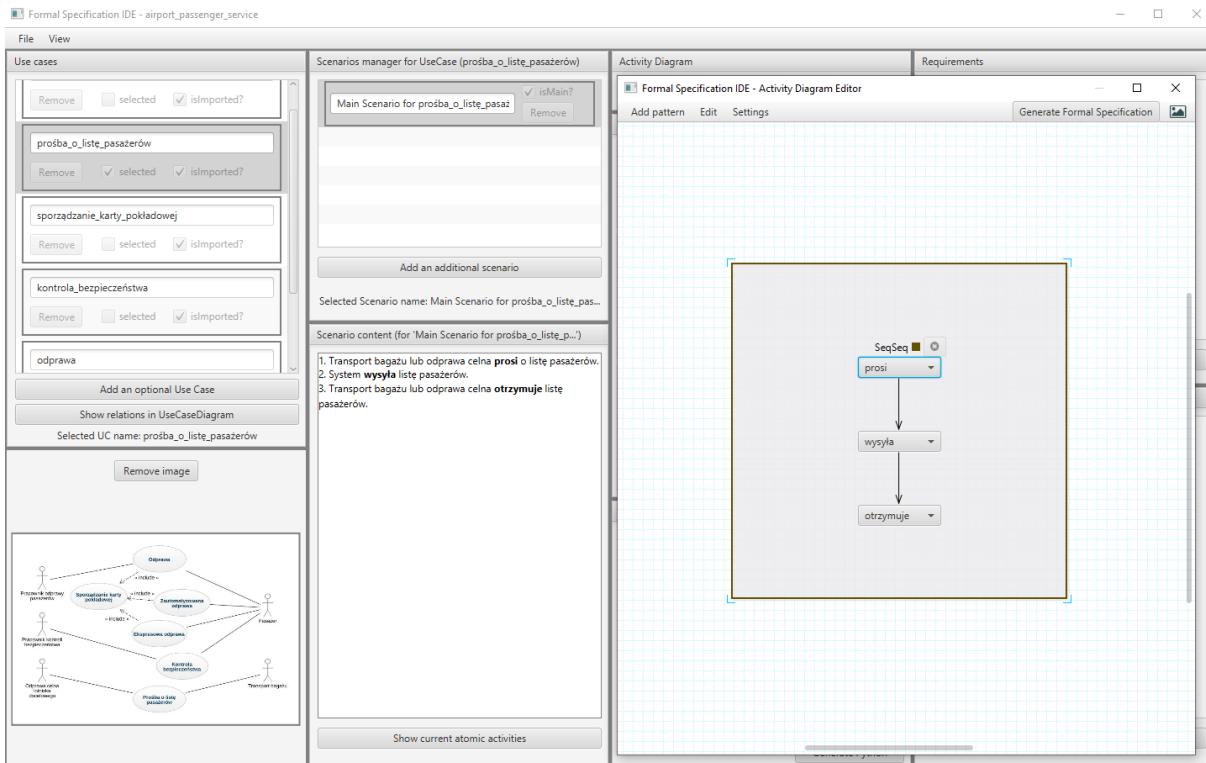
Rys. 4.54. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



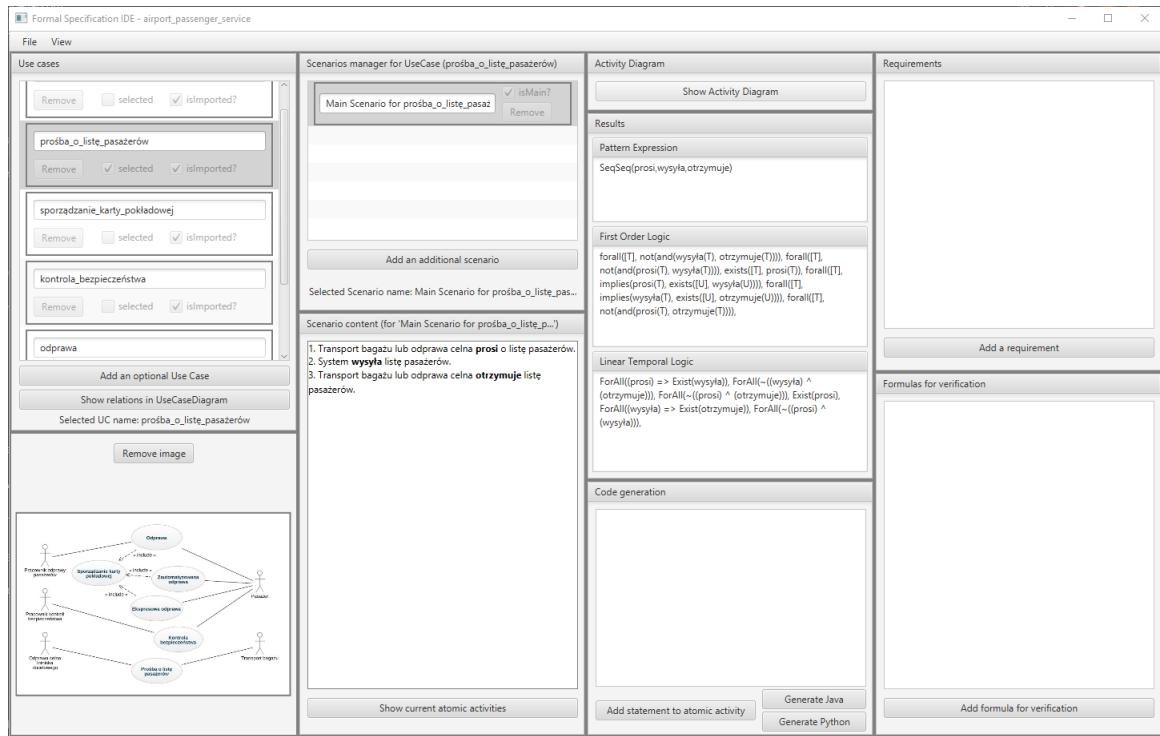
Rys. 4.55. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Kontrola bezpieczeństwa”.



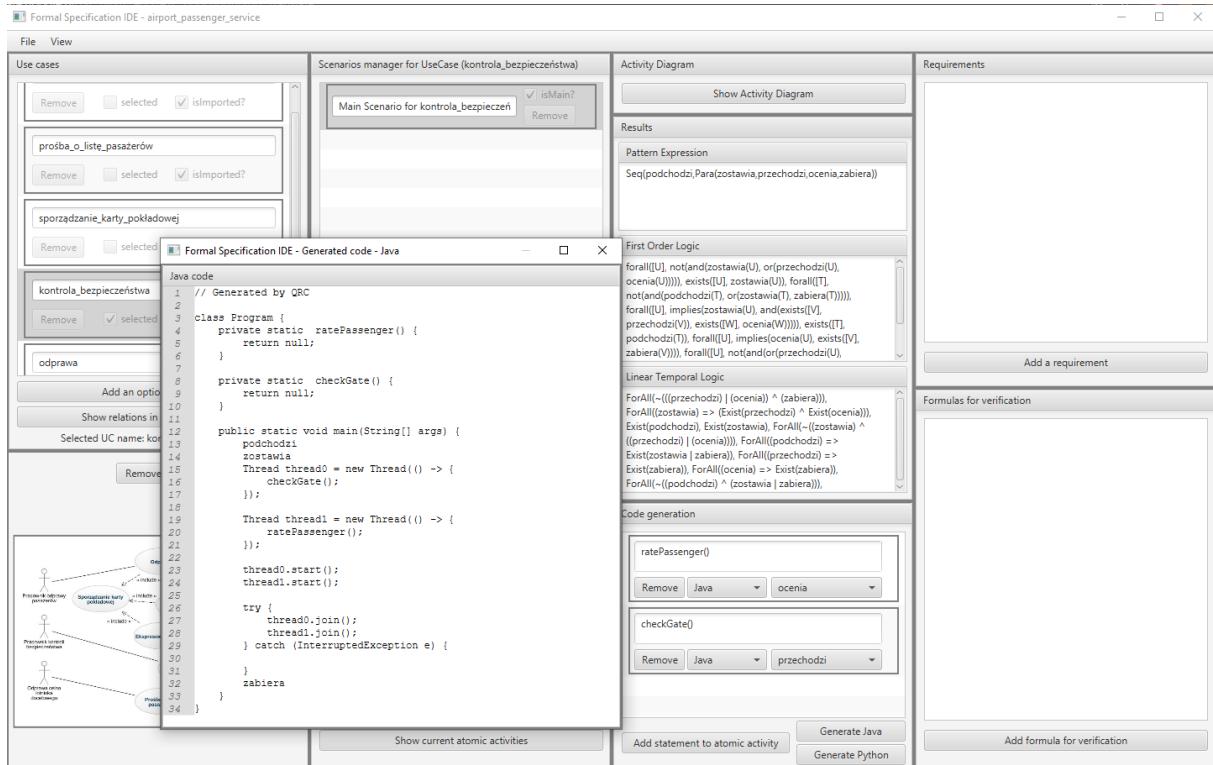
Rys. 4.56. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



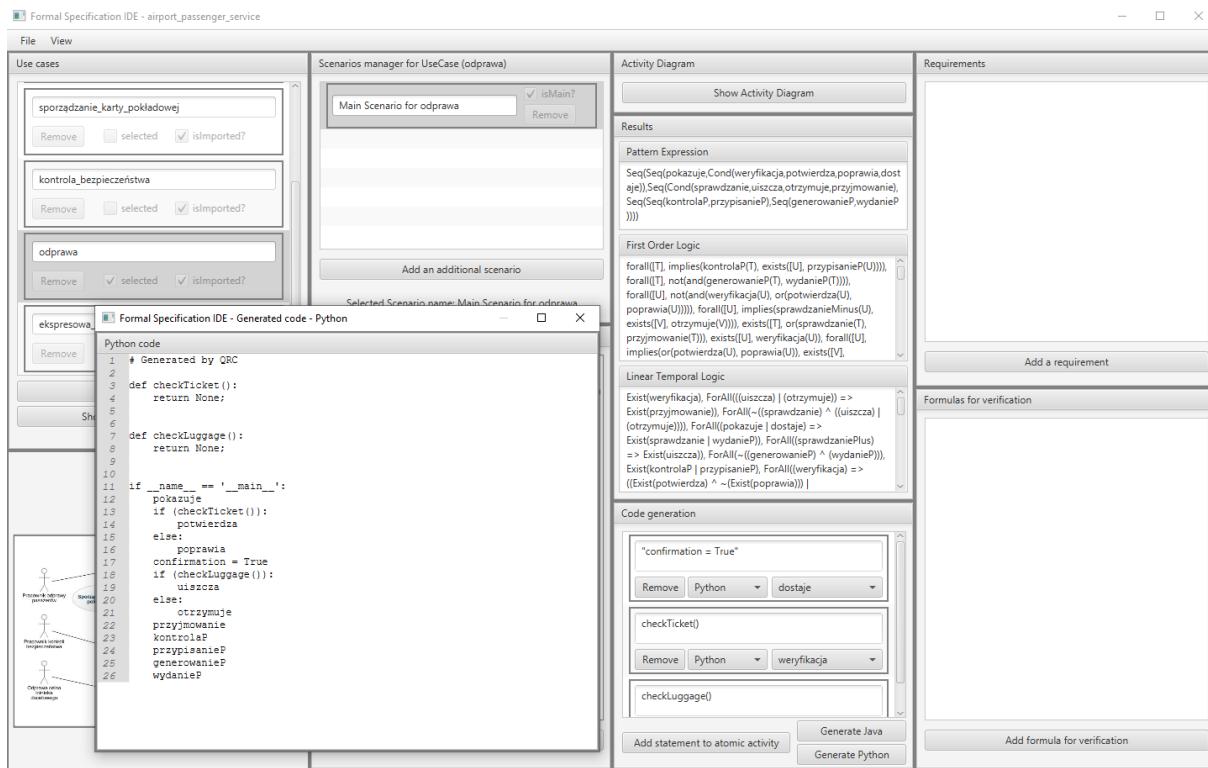
Rys. 4.57. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Prośba o listę pasażerów”.



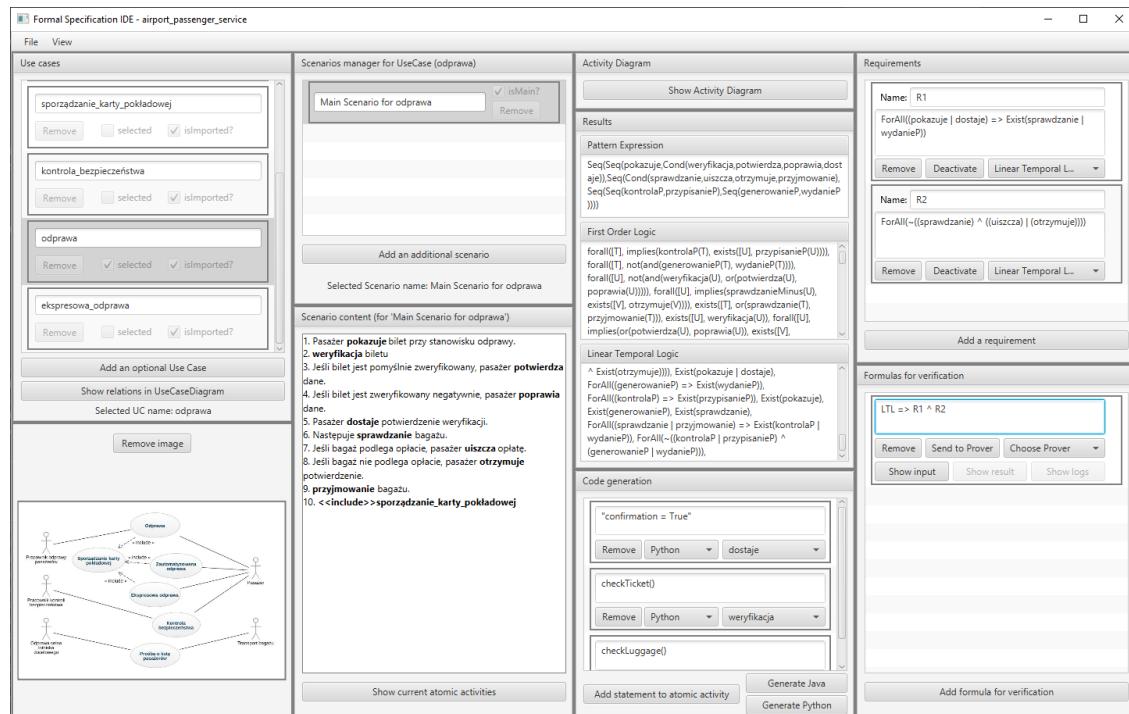
Rys. 4.58. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



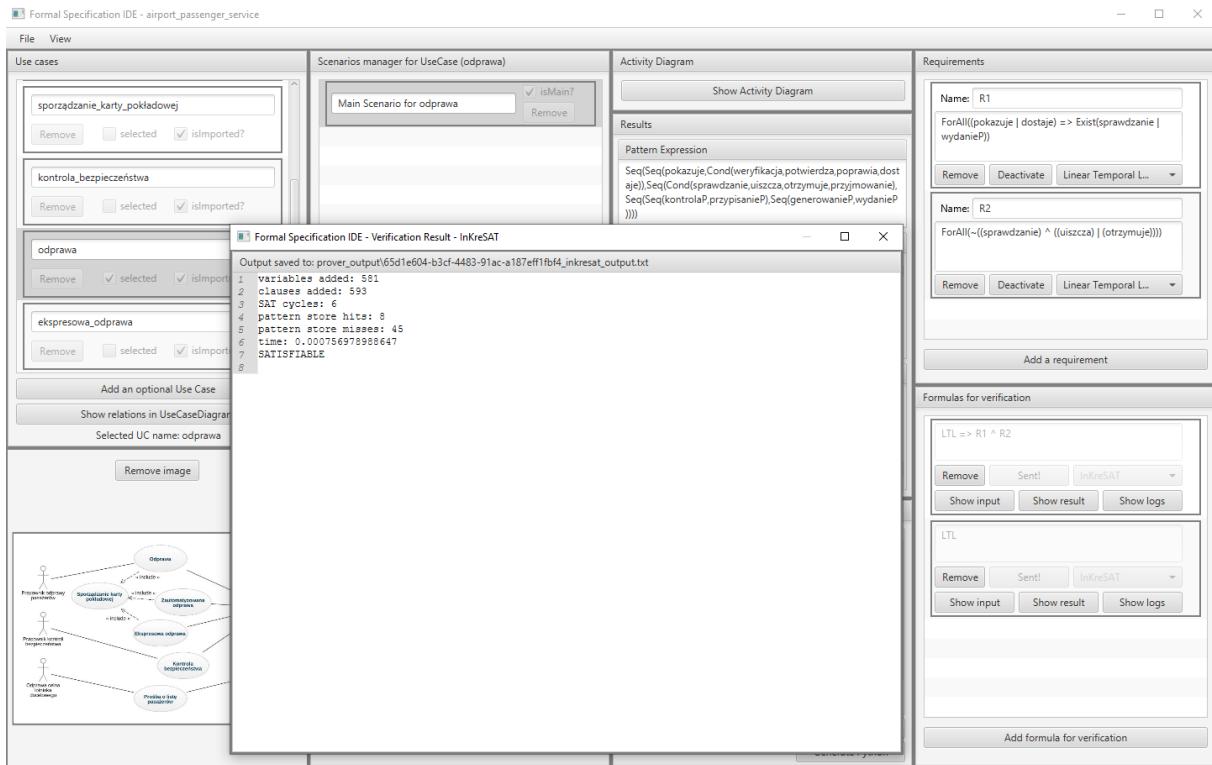
Rys. 4.59. Przykład wygenerowanego kodu źródłowego w języku Java. Zastosowano mapowanie aktywności na instrukcje języka programowania.



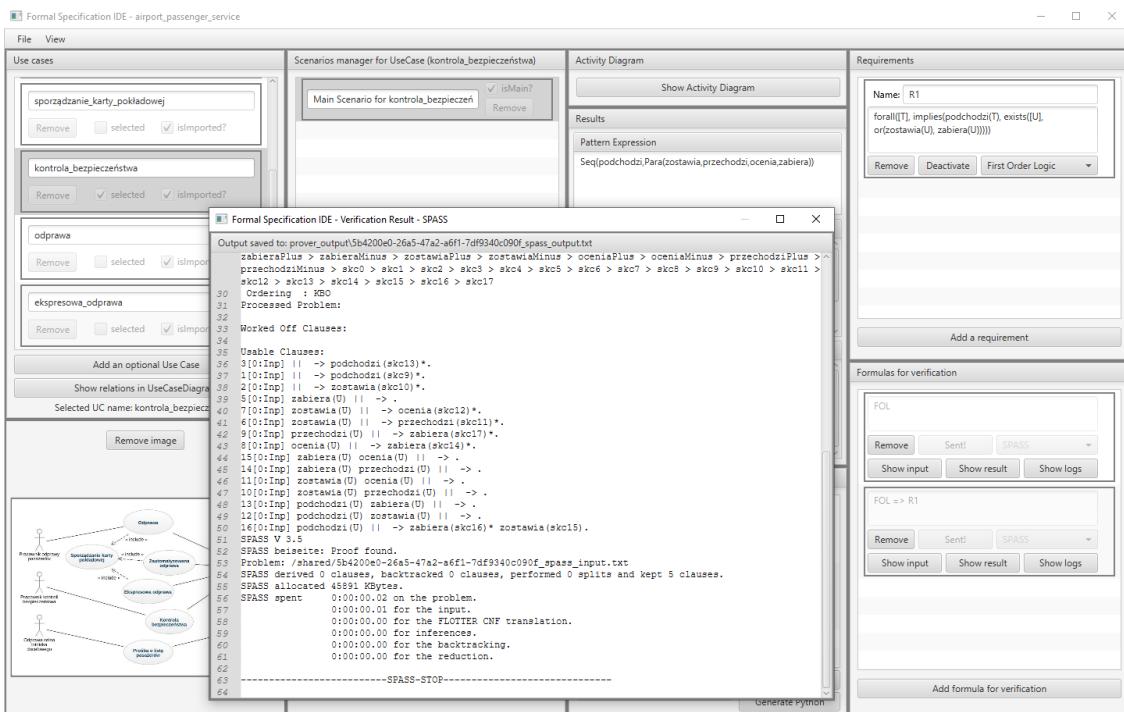
Rys. 4.60. Przykład wygenerowanego kodu źródłowego w języku Python. Zastosowano mapowanie aktywności na instrukcje języka programowania.



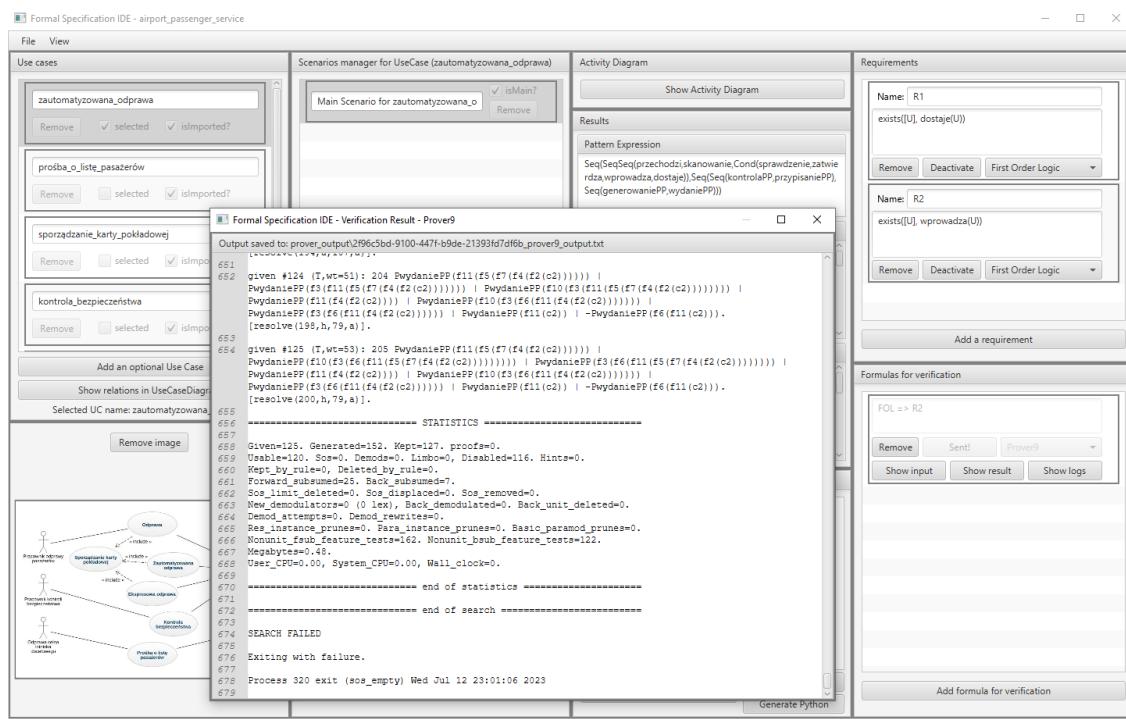
Rys. 4.61. Posiadając wygenerowaną specyfikację, można przeprowadzić formalną weryfikację. Należy wprowadzić treść, wybrać prover i kliknąć „Send to Prover”.



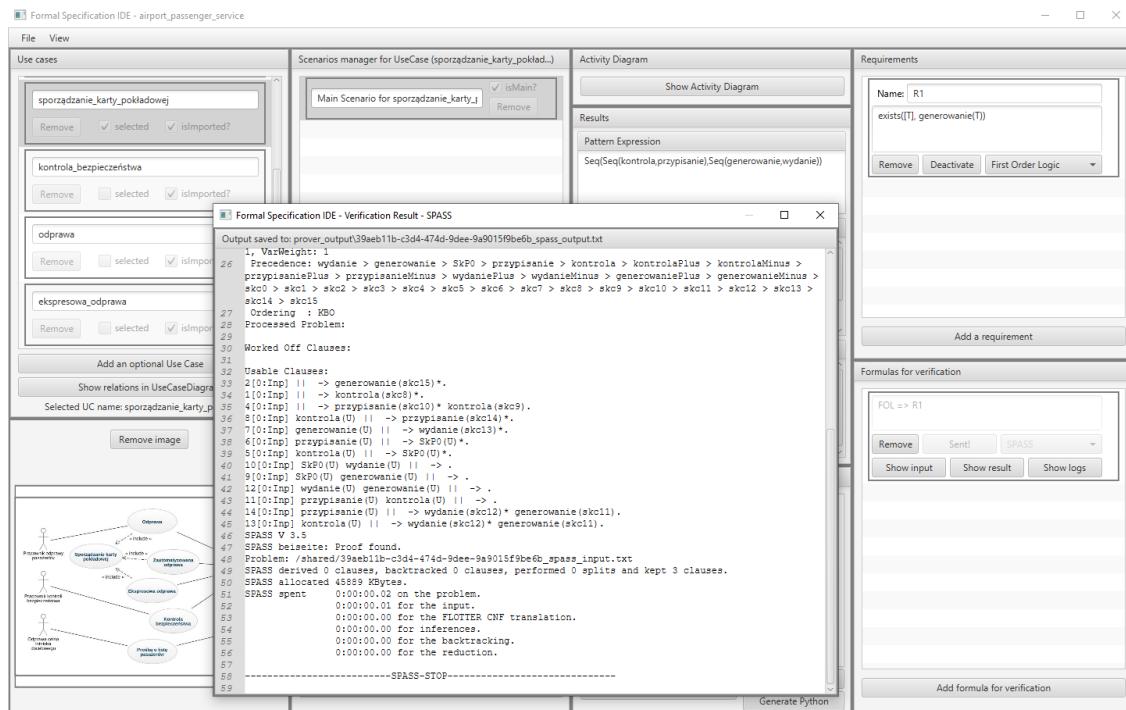
Rys. 4.62. Przykład pozytywnego wyniku weryfikacji formuły  $LTL \Rightarrow R1 \wedge R2$ , wykonanej z użyciem InKreSAT, dla przypadku użycia „Odprawa”.



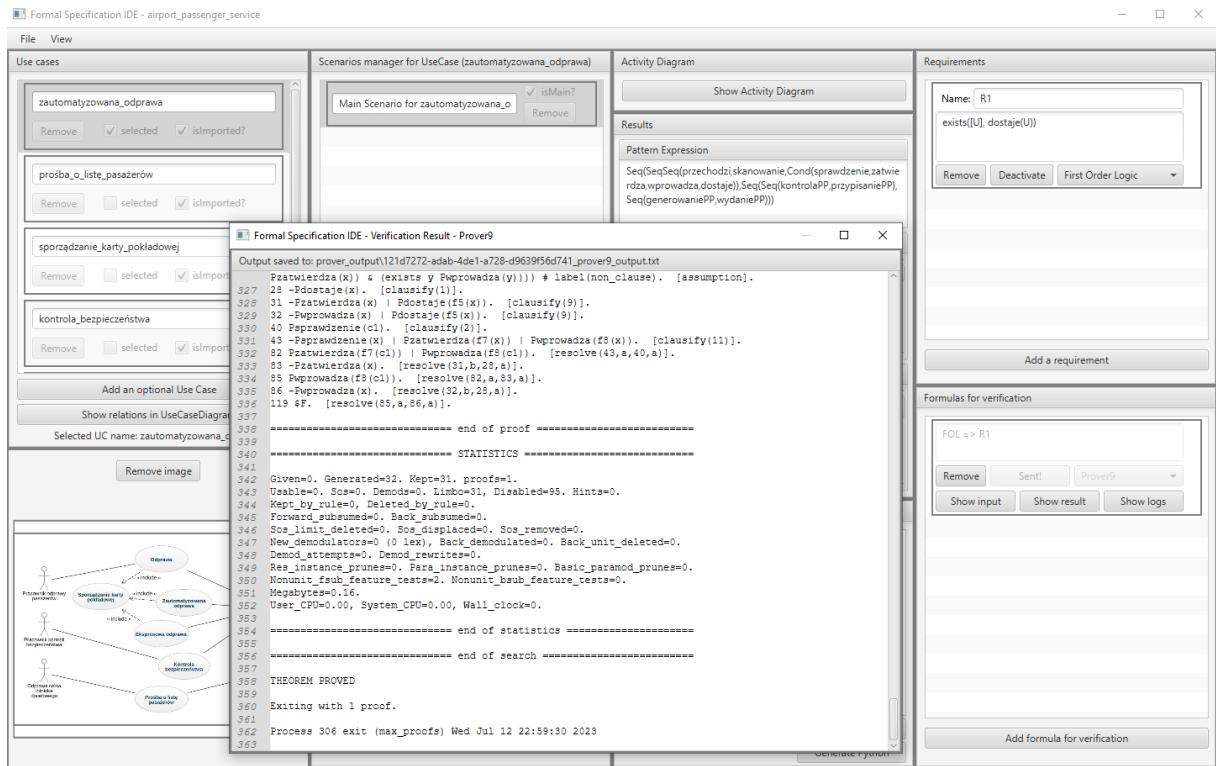
Rys. 4.63. Przykład pozytywnego wyniku weryfikacji formuły  $FOL \Rightarrow R1$ , z użyciem SPASS Prover, dla przypadku użycia „Kontrola bezpieczeństwa”.



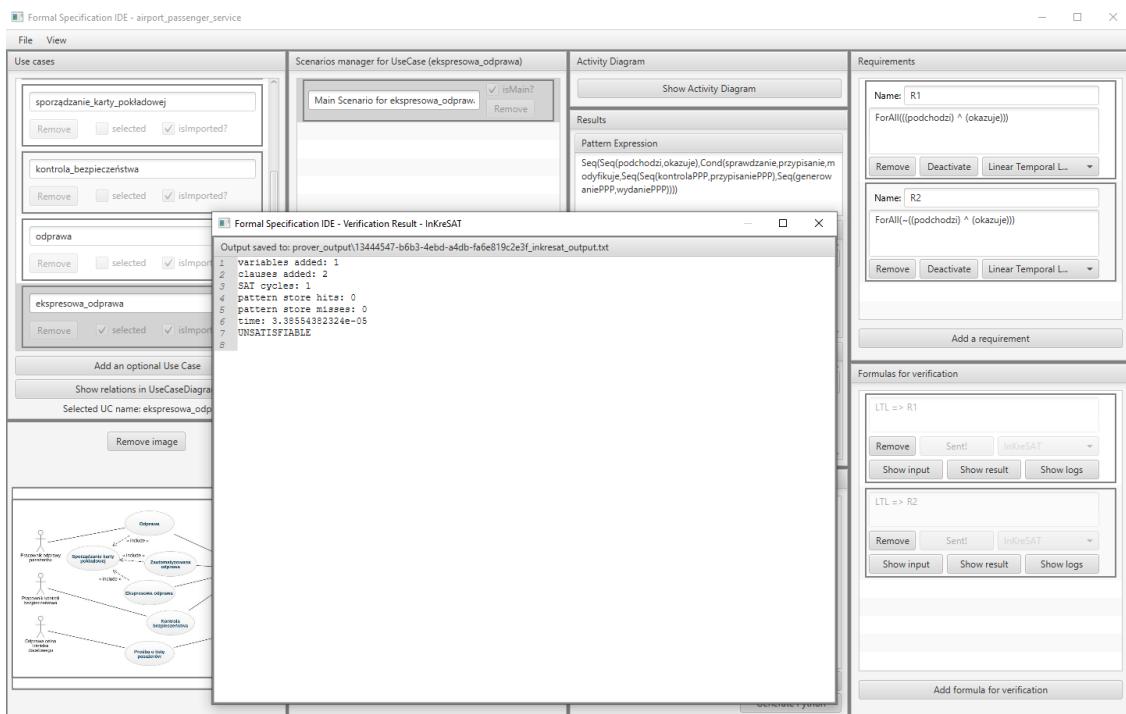
Rys. 4.64. Przykład negatywnego wyniku weryfikacji Prover9, z użyciem dodatkowego wymagania R2, w logice pierwszego rzędu, dla przypadku użycia „Zautomatyzowana odprawa”.



Rys. 4.65. Przykład pozytywnego wyniku weryfikacji formuły  $FOL \Rightarrow R1$ , z użyciem SPASS Prover, w logice pierwszego rzędu, dla przypadku użycia „Sporządzanie karty pokładowej”.



**Rys. 4.66.** Przykład pozytywnego wyniku weryfikacji Prover9, z użyciem dodatkowego wymagania R1, w logice pierwszego rzędu, dla przypadku użycia „Zautomatyzowana odprawa”.



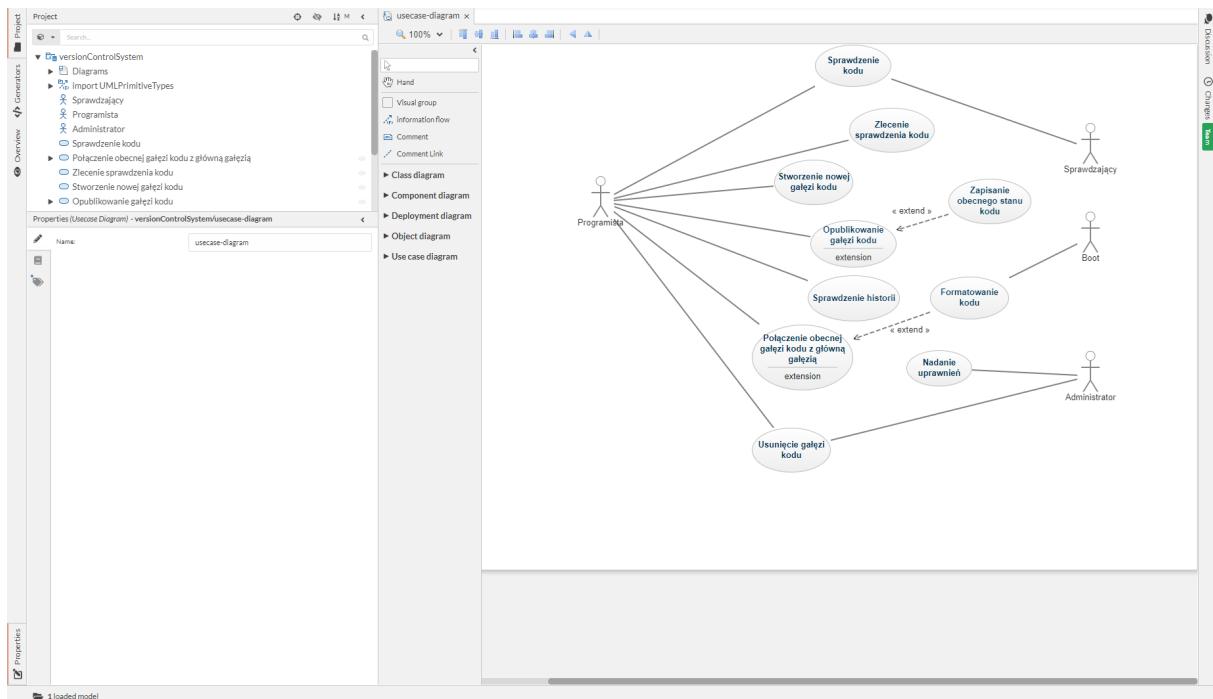
**Rys. 4.67.** Przykład negatywnego wyniku weryfikacji formuły  $LTL \Rightarrow R2$ , z użyciem InKreSAT, dla przypadku użycia „Ekspresowa odprawa”.

Przypadek użycia	Badana formula	Prover	Rezultat	Wnioski
Odprawa	FOL	SPASS Prover	Completion found	Formuła jest poprawna.
Odprawa	LTL => R1	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Zautomatyzowana odprawa	LTL	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Zautomatyzowana odprawa	FOL => R2	Prover9	SEARCH FAILED	Formuła nie jest spełnialna. Należy poprawić wymaganie.
Zautomatyzowana odprawa	FOL => R3	Prover9	THEOREM PROVED	Formuła jest spełnialna. Formuła została udowodniona.
Ekspresowa odprawa	LTL	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Ekspresowa odprawa	LTL => R4	InKreSAT	UNSATISFIABLE	Formuła nie jest spełnialna. Należy poprawić formułę.
Ekspresowa odprawa	LTL => R5	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Sporządzanie karty pokładowej	FOL => R6	SPASS Prover	Proof found	Formuła została udowodniona.
Prośba o listę pasażerów	FOL	SPASS Prover	Completion found	Formuła jest poprawna.
Prośba o listę pasażerów	LTL	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Kontrola bezpieczeństwa	FOL => R7	SPASS Prover	Proof found	Formuła została udowodniona.

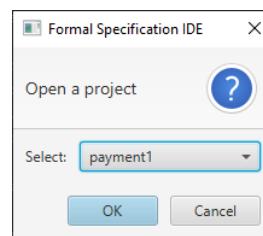
**Tabela 4.1.** Zestawienie uzyskanych wyników weryfikacji przeprowadzonych dla projektowanego systemu obsług pasażerów na lotnisku. R1 =  $\text{ForAll}(\sim(\text{sprawdzanie} \mid \text{przyjmowanie}))$ , R2 =  $\text{exists}([U], \text{wprowadza}(U))$ , R3 =  $\text{exists}([U], \text{dostaje}(U))$ , R4 =  $\text{ForAll}(\sim(\text{podchodzi}) \wedge (\text{okazuje}))$ , R5 =  $\text{ForAll}((\text{podchodzi}) \wedge (\text{okazuje}))$ , R6 =  $\text{exists}([T], \text{generowanie}(T))$ , R7 =  $\text{forall}([T], \text{implies}(\text{podchodzi}(T), \text{exists}([U], \text{or}(\text{zostawia}(U), \text{zabiera}(U))))$

## 4.11.2. System kontroli wersji kodu

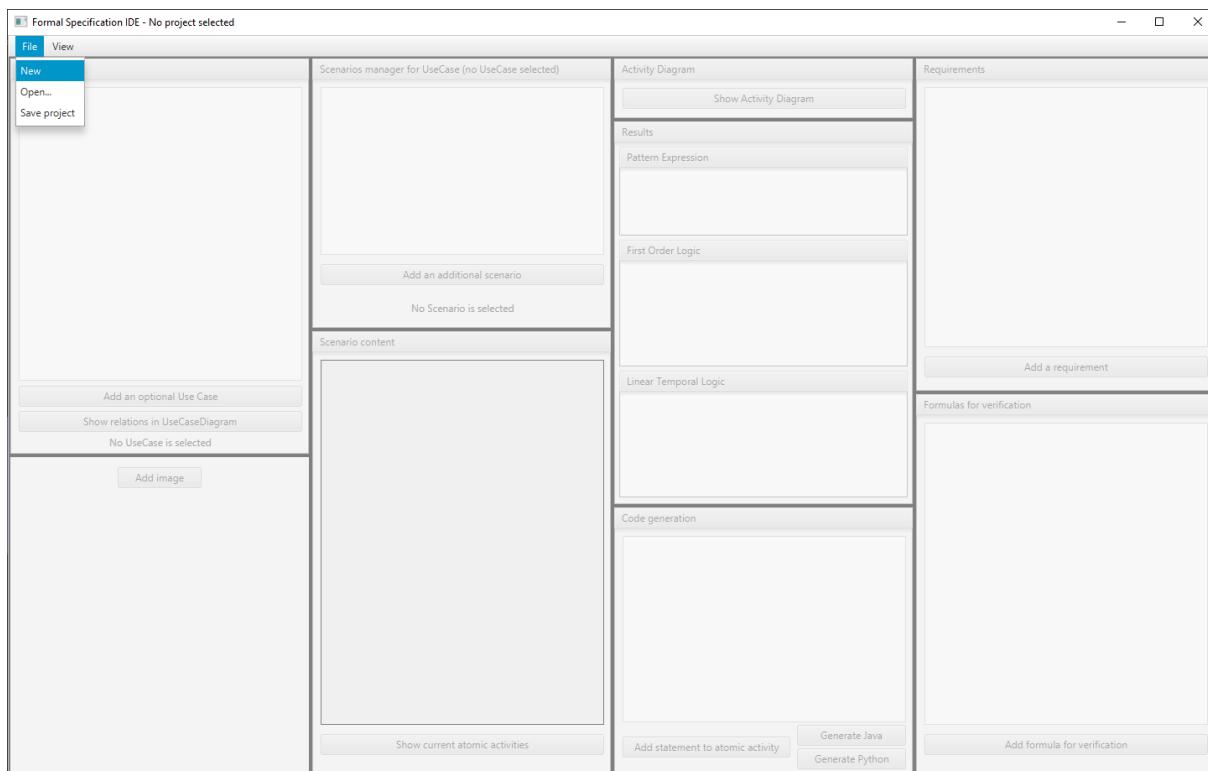
Poniższy przykład przedstawia system kontroli wersji kodu, oparty na systemie Git. System ten umożliwia programistom sprawdzanie poprzednich wersji kodu, zlecanie sprawdzenia kodu przez innych użytkowników oraz dołączanie swojego kodu do głównej gałęzi. Obsługuje także zewnętrzne usługi do formatowania kodu.



Rys. 4.68. Diagram przypadków użycia dla systemu kontroli wersji kodu. Projekt został wykonany z użyciem aplikacji GenMyModel[15].



Rys. 4.69. Bezpośrednio po uruchomieniu aplikacji *FS-IDE* pojawia się okno z możliwością wyboru jednego z istniejących projektów. W przypadku wyboru „Cancel”, zostanie wyświetcone główne okno, z możliwością stworzenia nowego projektu.

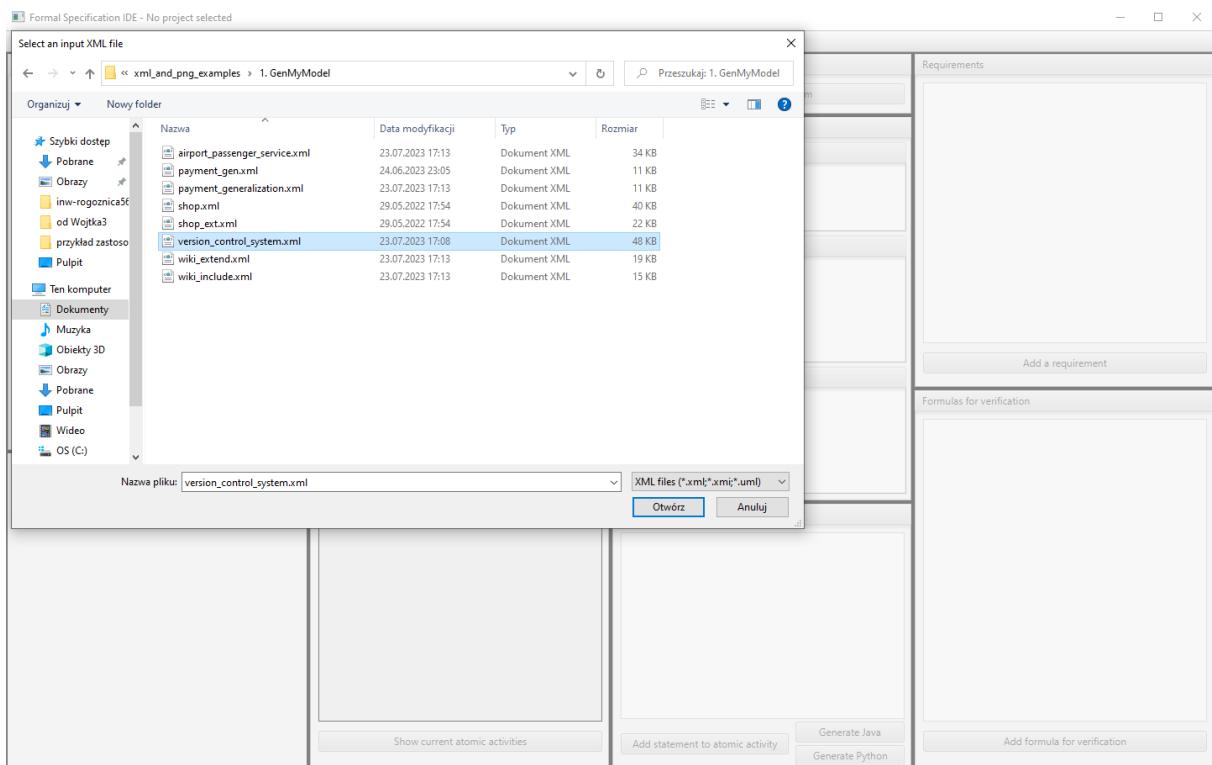


**Rys. 4.70.** W menu, pod opcją „File” znajduje się opcja stworzenia nowego projektu („New”).

System kontroli wersji kodu umożliwia przechowywanie, śledzenie i zarządzanie zmianami w kodzie źródłowym projektu. Dzięki temu programiści mogą łatwo powracać do wcześniejszych wersji kodu, porównywać zmiany oraz skutecznie współzleżnić pracę w zespole. Wszystkie zmiany w kodzie są przechowywane w repozytorium, które stanowi centralne miejsce dla wszystkich członków zespołu.

Na rysunku 4.68 przedstawiono zamodelowany diagram przypadków użycia dla omawianego przykładu systemu kontroli wersji kodu, wykonany przy użyciu zewnętrznego narzędzia GenMyModel [15]. Z aplikacji GenMyModel został wyeksportowany plik projektu .xml oraz grafika .jpeg. Następnie te elementy mogą zostać zimportowane do systemu *Formal Specification IDE*. Diagram przypadków użycia przedstawia aktorów: programistę, sprawdzającego, administratora oraz Boota. Wykazano również przypadki użycia związane z kontrolą wersji kodu, takie jak stworzenie nowej gałęzi kodu, opublikowanie gałęzi kodu, zapisanie obecnego stanu kodu.

Aby rozpocząć korzystanie z systemu, należy otworzyć aplikację *Formal Specification IDE*. Po uruchomieniu pojawi się okno umożliwiające wybór istniejącego projektu (rysunek 4.69). Wybór opcji „Cancel” spowoduje wyświetlenie głównego okna, w którym można stworzyć nowy projekt (rysunek 4.70).



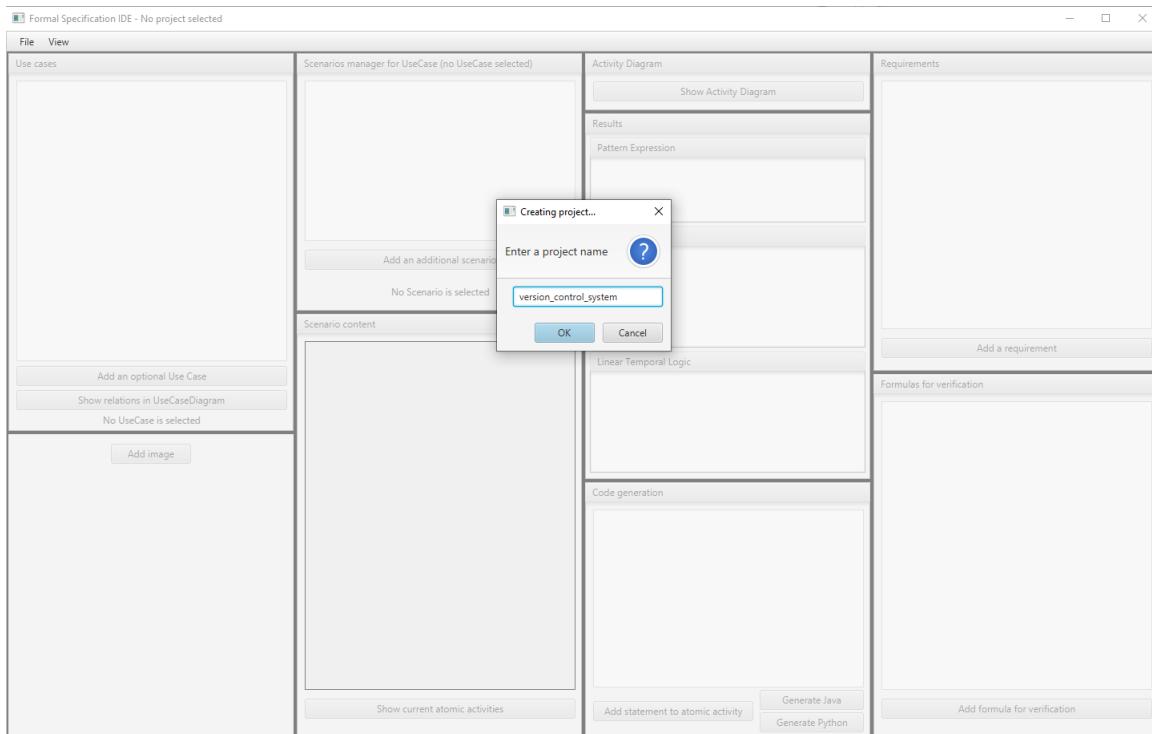
**Rys. 4.71.** Po wyborze opcji stworzenia nowego projektu, pojawia się okno z prośbą o wskazanie pliku .xml z diagramem przypadków użycia.

Następnie, w menu „File” można wybrać opcję „New” w celu utworzenia nowego projektu. Po wybraniu tej opcji pojawi się okno proszące o wskazanie pliku .xml z diagramem przypadków użycia (rysunek 4.71). Po wskazaniu pliku .xml, pojawi się okno z prośbą o nadanie nazwy projektowi (rysunek 4.72). Po wpisaniu nazwy, przypadki użycia zostaną zainportowane i wyświetlane na liście w panelu „Use Cases” (rysunek 4.73). Istnieje również możliwość dodania grafiki diagramu przypadków użycia (rysunek 4.74) w celu lepszego wizualnego zrozumienia modelu (rysunek 4.75). W przypadku chęci powiększenia grafiki w systemie, po kliknięciu na nią, zostaje wyświetlone okno podglądu, z przybliżania oraz przesuwania grafiki (rysunek 4.76).

Kolejnym krokiem jest wypełnienie scenariuszy przypadków użycia w panelu „Scenario content” (rysunek 4.77), co odbywa się poprzez wybór konkretnego przypadku użycia z listy dostępnych i dodanie odpowiednich kroków scenariusza, w formie tekstu naturalnego.

Przykładowe treści scenariuszy, dla wszystkich rozważanych przypadków użycia, zostały przedstawione na rysunkach: 4.78, 4.79, 4.80, 4.81, 4.82, 4.83, 4.84, 4.85, 4.86, 4.87.

Podczas tworzenia scenariuszy, w każdym momencie dostępny jest podgląd aktualnych atomicznych aktywności, pod przyciskiem „Show current atomic activities”, co zostało pokazane na rysunku 4.88.



Rys. 4.72. Po wskazaniu pliku .xml pojawia się okno z prośbą o nadanie nazwy projektowi.

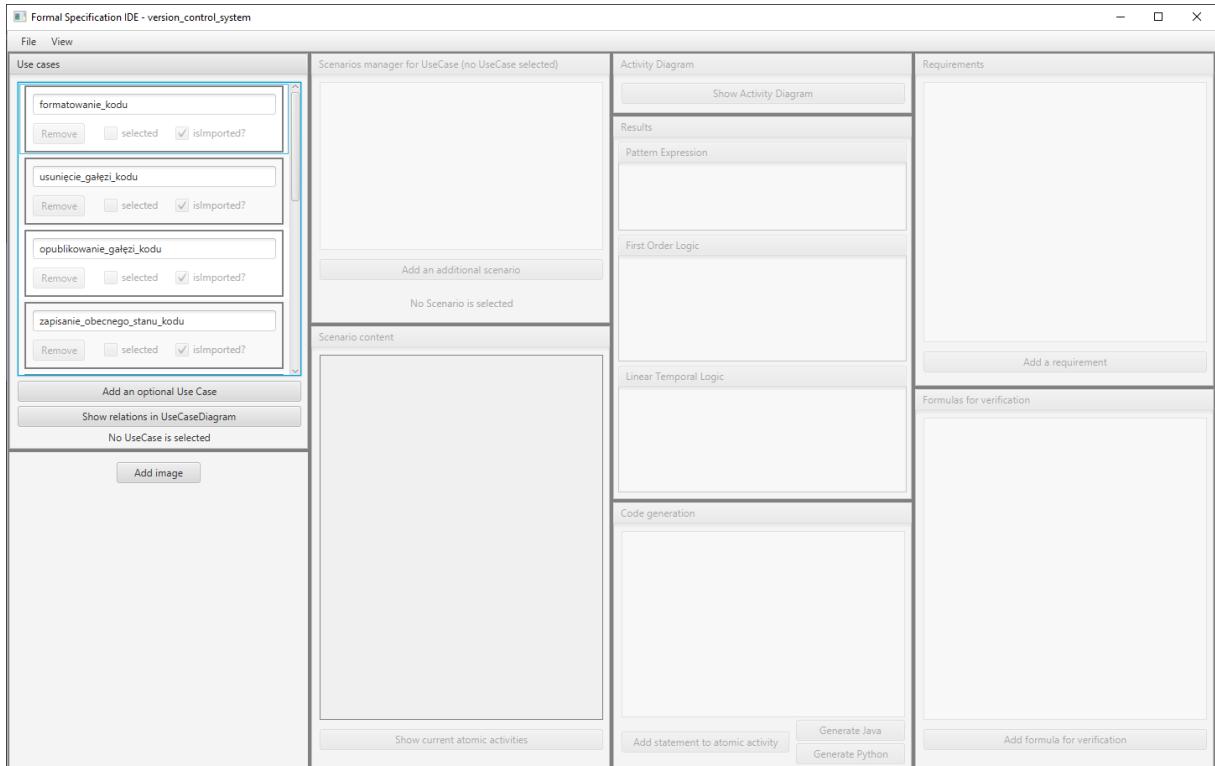
Po wypełnieniu scenariuszy, można przejść do modelowania diagramu aktywności, który ilustruje przebieg akcji i zachowań w systemie. Na diagramie można zobaczyć, jakie aktywności są wykonywane i w jakiej kolejności. Edytor diagramów aktywności można uruchomić klikając przycisk „Show Activity Diagram” (rysunek 4.89).

Przykładowe diagramy aktywności, dla wszystkich rozważanych przypadków użycia, zostały przedstawione na rysunkach: 4.90, 4.92, 4.94, 4.96, 4.98, 4.100, 4.102, 4.104, 4.106, 4.108.

Następnie, po zamodelowaniu diagramu aktywności, można przystąpić do generowania specyfikacji. System generuje wyrażenia wzorcowe oraz specyfikacje logiczne w logice temporalnej LTL oraz logice pierwszego rzędu FOL. Wygenerowane specyfikacje można wykorzystać do formalnej weryfikacji, która może być przeprowadzona za pomocą odpowiednich narzędzi.

Przykładowe stworzone wyrażenia wzorcowe oraz wygenerowane specyfikacje logiczne, dla wszystkich rozważanych przypadków użycia, zostały przedstawione na rysunkach: 4.91, 4.93, 4.95, 4.97, 4.99, 4.101, 4.103, 4.105, 4.107, 4.109.

System *Formal Specification IDE* dostarcza również możliwość przeprowadzenia formalnej weryfikacji specyfikacji dotyczących systemu kontroli wersji kodu. Proces ten pozwala na



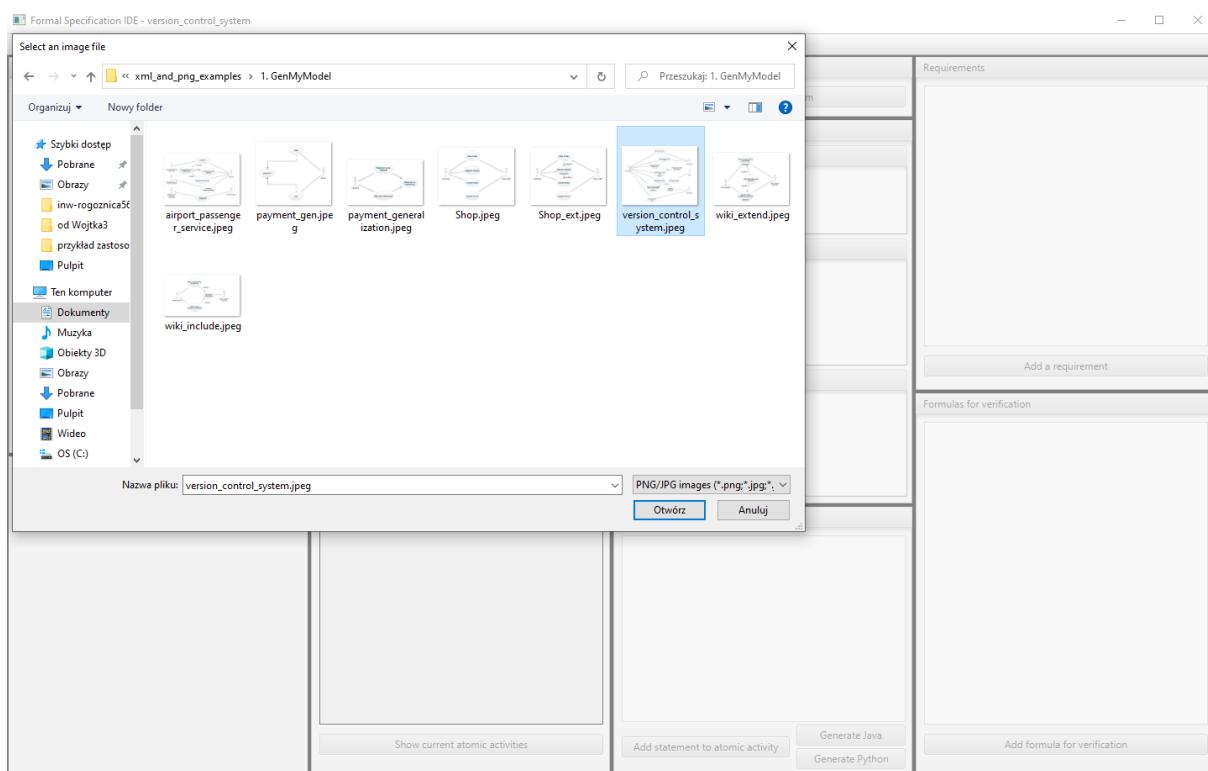
**Rys. 4.73.** Po wpisaniu nazwy projektu, przypadki użycia zostają zainportowane i wyświetlane na liście w panelu „Use cases”.

dokładne sprawdzenie i potwierdzenie zgodności wygenerowanych modeli z założeniami projektowymi oraz określonymi wymaganiem dotyczącymi zarządzania kodem źródłowym. Specyfikacje stworzone w aplikacji mogą być wysłane do specjalistycznych narzędzi weryfikacyjnych, takich jak InKreSAT, SPASS Prover oraz Prover9 (rysunek 4.112). Rezultaty weryfikacji dostarczają użytkownikowi szczegółowych informacji na temat spełnienia lub niespełnienia określonych warunków lub własności systemu kontroli wersji.

Przykładowo, dla przypadku użycia „Opublikowanie gałęzi kodu”, można przeprowadzić weryfikację w logice temporalnej LTL. Rezultaty takiej weryfikacji mogą być zarówno pozytywne, jak i negatywne, zależnie od spełnienia lub niespełnienia określonych warunków logicznych. Podobnie, dla innych przypadków użycia, takich jak „Sprawdzenie historii” czy „Stworzenie nowej gałęzi”, można przeprowadzić weryfikację w logice pierwszego rzędu FOL.

Przykłady wyników weryfikacji można zobaczyć na diagramach 4.113, 4.114, 4.115 i 4.116.

Aby pokazać cały proces weryfikacji w kontekście systemu kontroli wersji kodu, w tabeli 4.2 przedstawiono zestawienie wyników weryfikacji przeprowadzonych dla projektowanego systemu kontroli wersji. Tabela zawiera przykładowe formuły, które mogą być poddane weryfikacji, oraz prezentuje wyniki zarówno pozytywne, jak i negatywne, uzyskane dla każdego zintegrowanego narzędzia weryfikacyjnego. W sytuacjach, w których otrzymano negatywne wyniki, formuły zostały dostosowane, co pozwoliło na uzyskanie pozytywnych rezultatów.

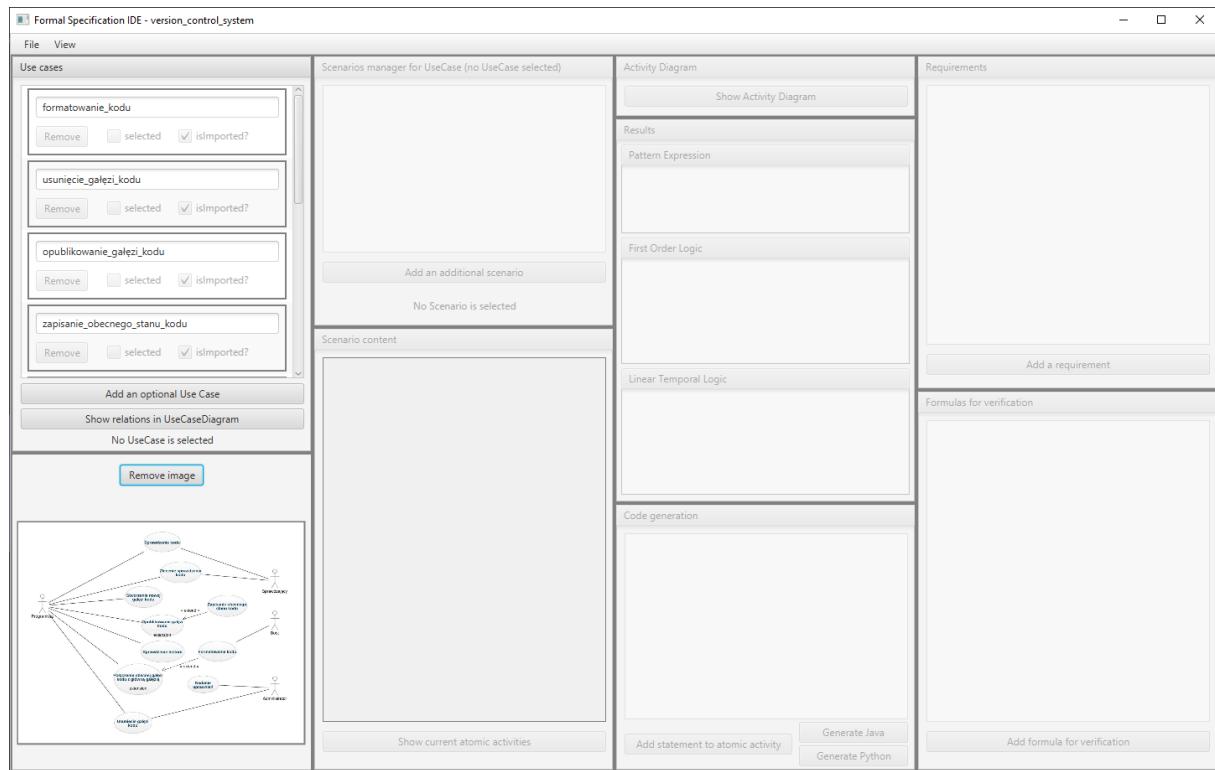


Rys. 4.74. Istnieje możliwość wskazania pliku („Add image”) z grafiką diagramu przypadków użycia, w celu wizualizacji.

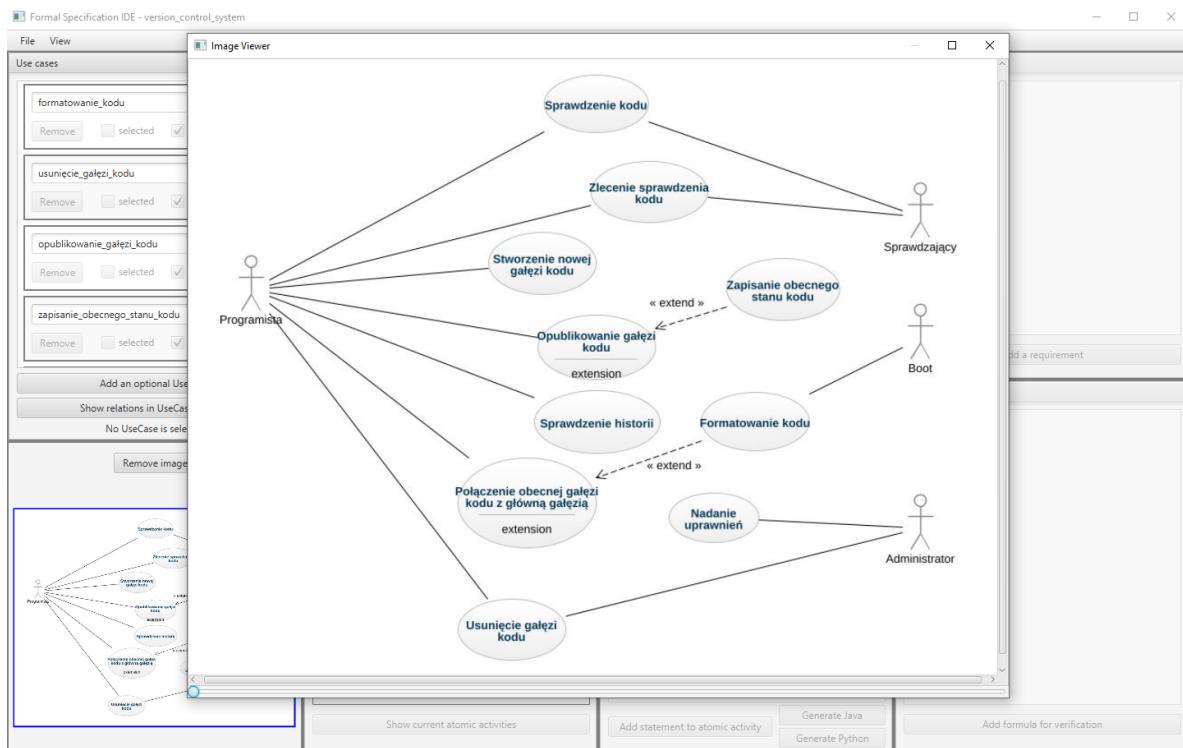
System *Formal Specification IDE* umożliwia również generowanie kodu źródłowego na podstawie zamodelowanego systemu. Wygenerowany kod może być w dwóch różnych językach programowania, takich jak Java, Python. Wykorzystuje się mapowanie aktywności na instrukcje języka programowania.

Przykłady wygenerowanego kodu źródłowego w języku Java i Python można zobaczyć na diagramach 4.110 i 4.111.

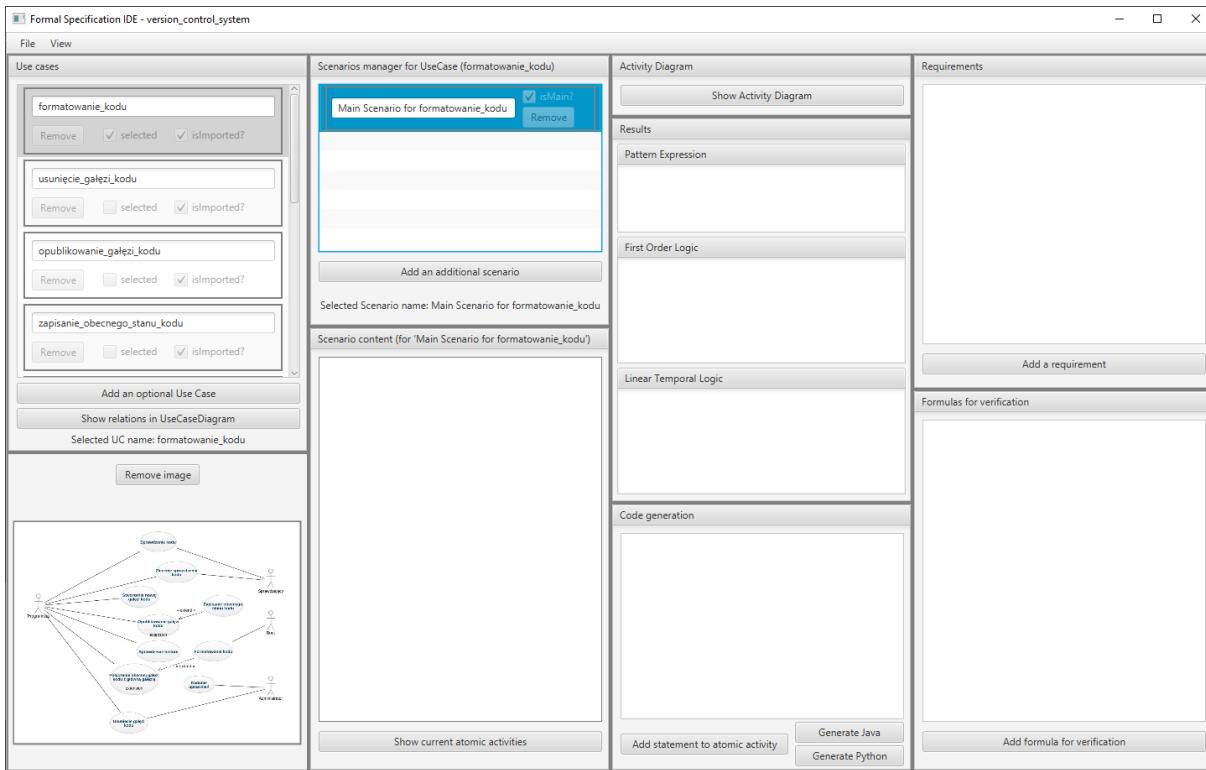
W ten sposób system *Formal Specification IDE* umożliwia modelowanie, generowanie specyfikacji, formalną weryfikację oraz generowanie kodu źródłowego na podstawie zamodelowanego systemu kontroli wersji kodu.



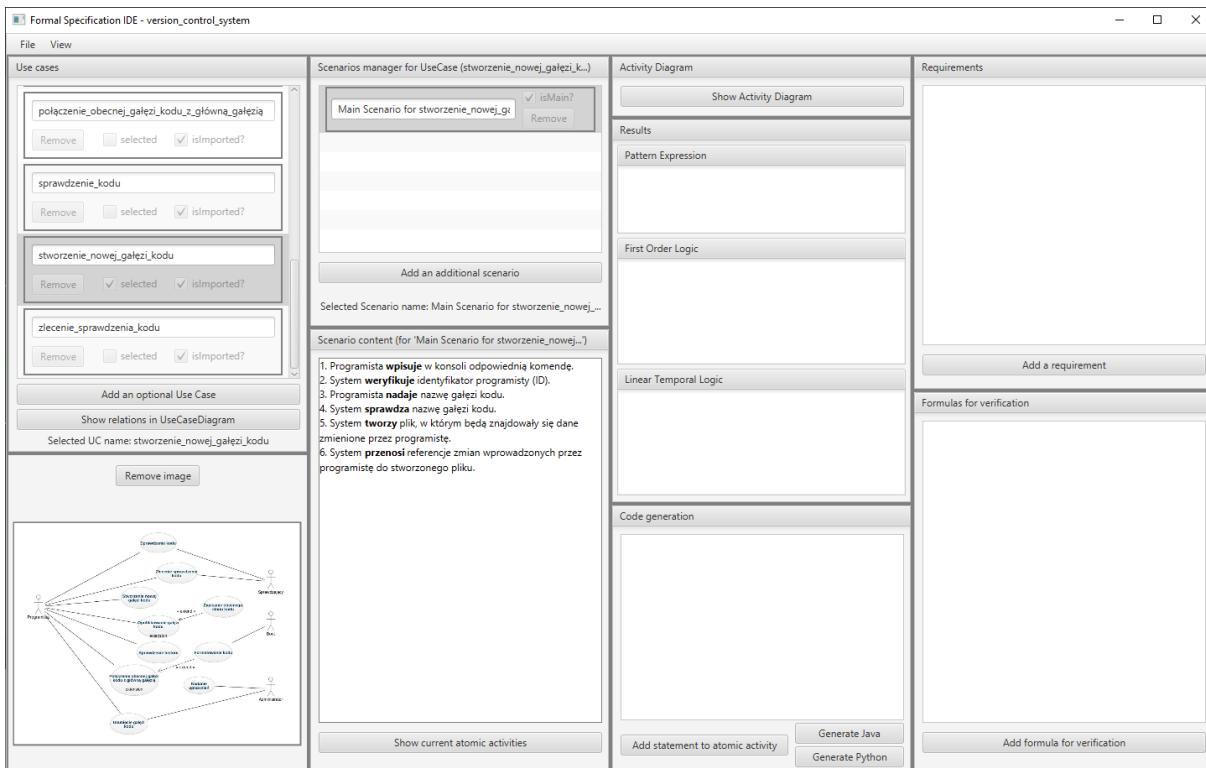
Rys. 4.75. Po wyborze pliku z grafiką, obraz zostaje wyświetlony w lewym dolnym rogu okna aplikacji.



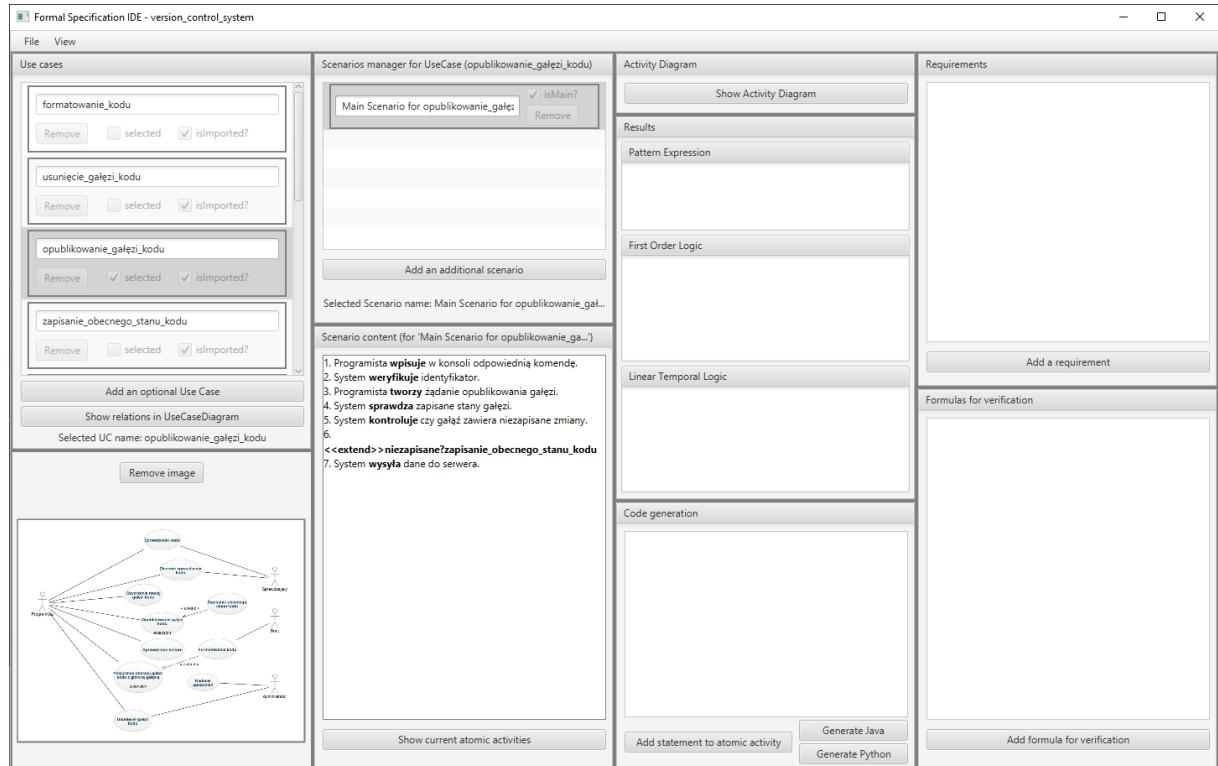
Rys. 4.76. W celu lepszego podglądu grafiki, po kliknięciu na obraz, zostaje wyświetlone nowe okno, z możliwością przesuwania i przybliżania.



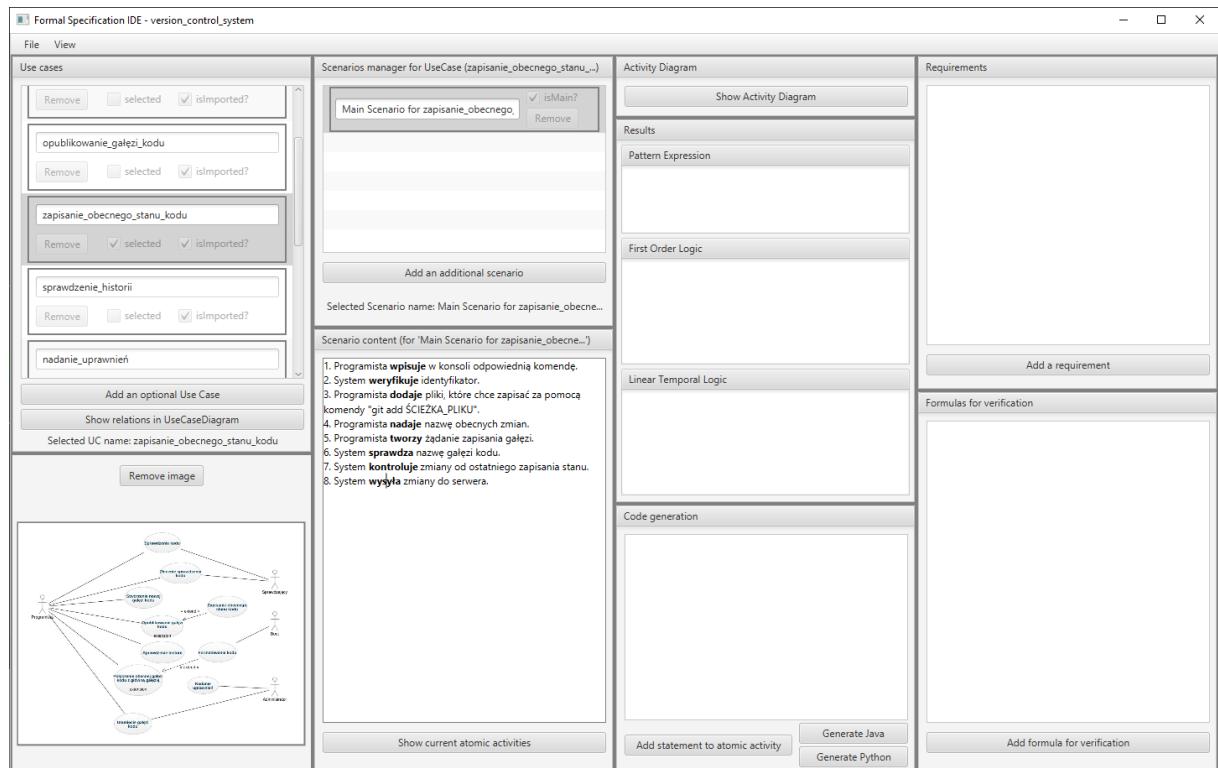
Rys. 4.77. Po kliknięciu na konkretny przypadek użycia, w panelu „Scenarios manager” wyświetlany jest główny scenariusz, który jest domyślnie dodany.



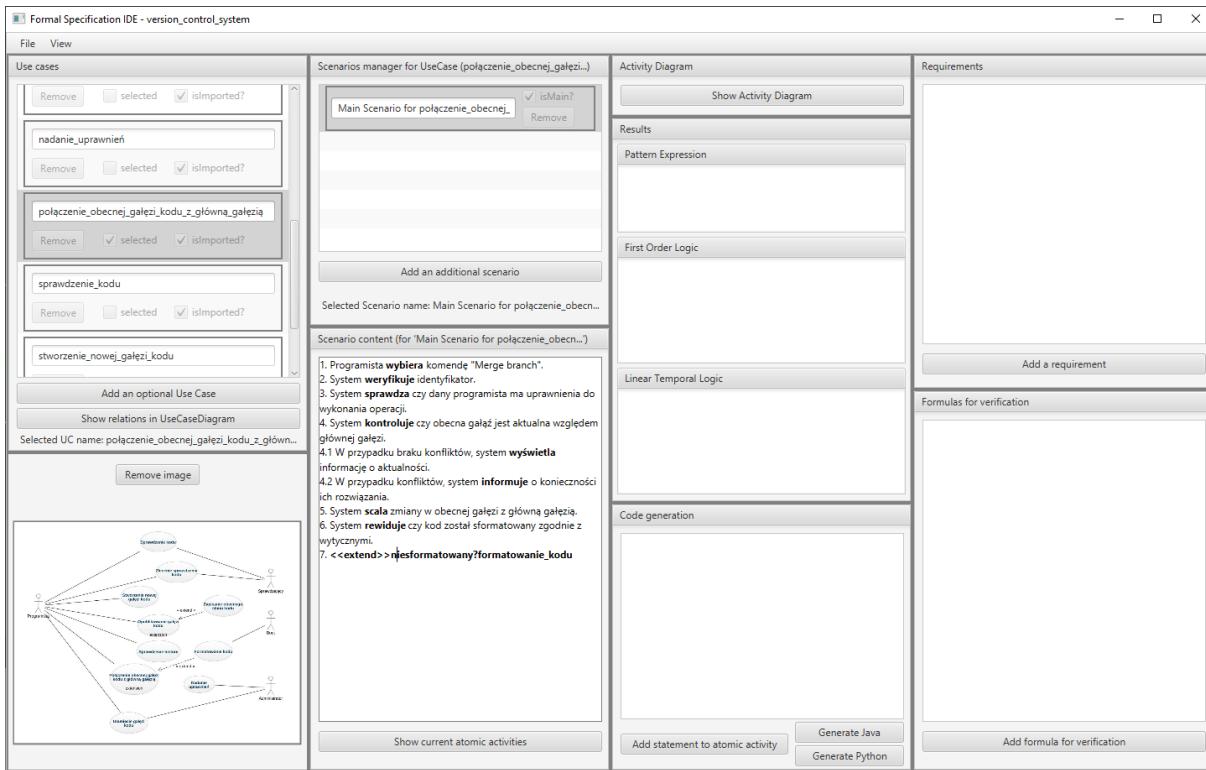
Rys. 4.78. Przykład wypełnienia scenariusza przypadku użycia „Stworzenie nowej gałęzi kodu”.



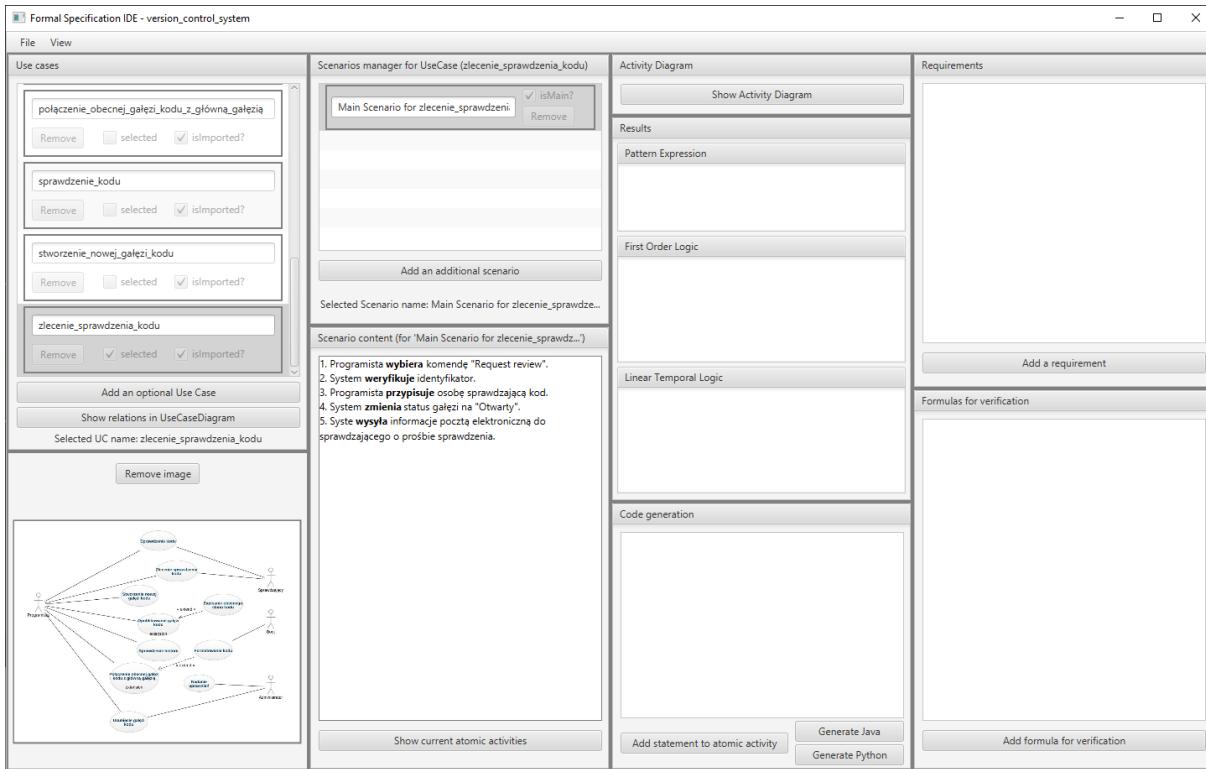
Rys. 4.79. Przykład wypełnienia scenariusza przypadku użycia „Opublikowanie gałęzi kodu”.



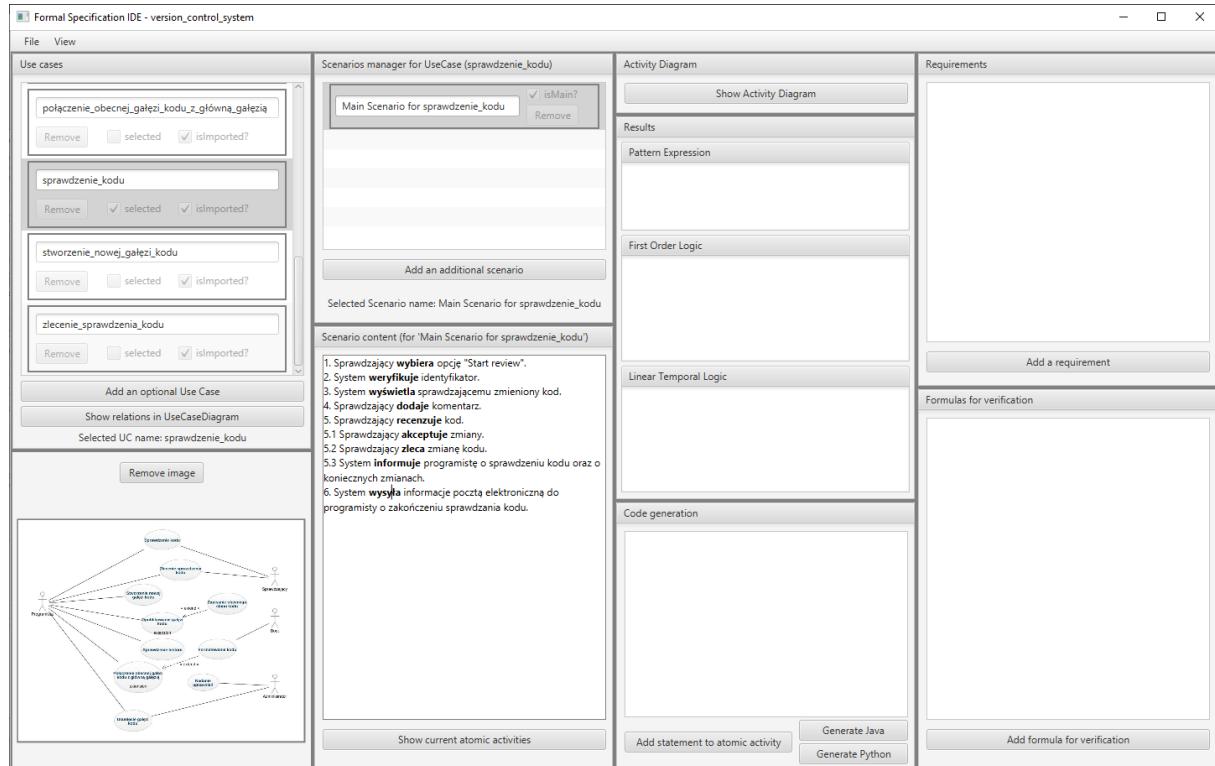
Rys. 4.80. Przykład wypełnienia scenariusza przypadku użycia „Zapisanie obecnego stanu kodu”.



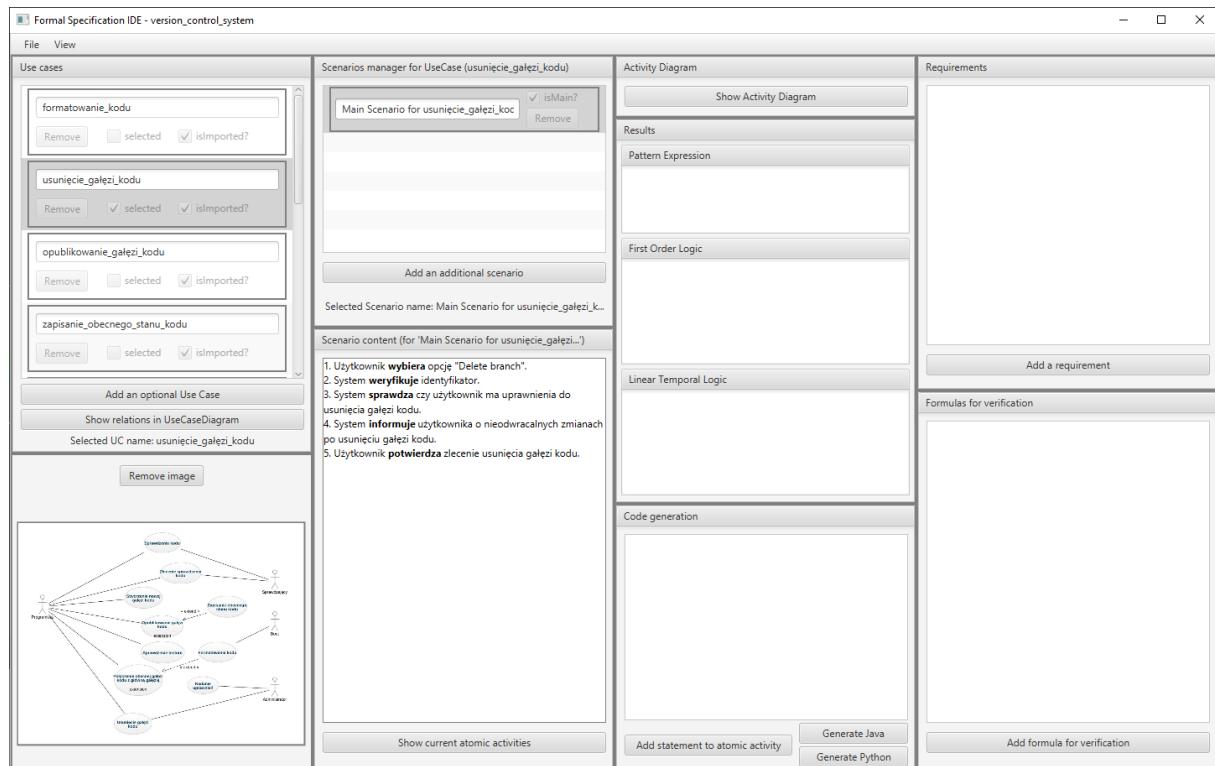
Rys. 4.81. Przykład wypełnienia scenariusza przypadku użycia „Połączenie obecnej gałęzi kodu z główną gałęzią”.



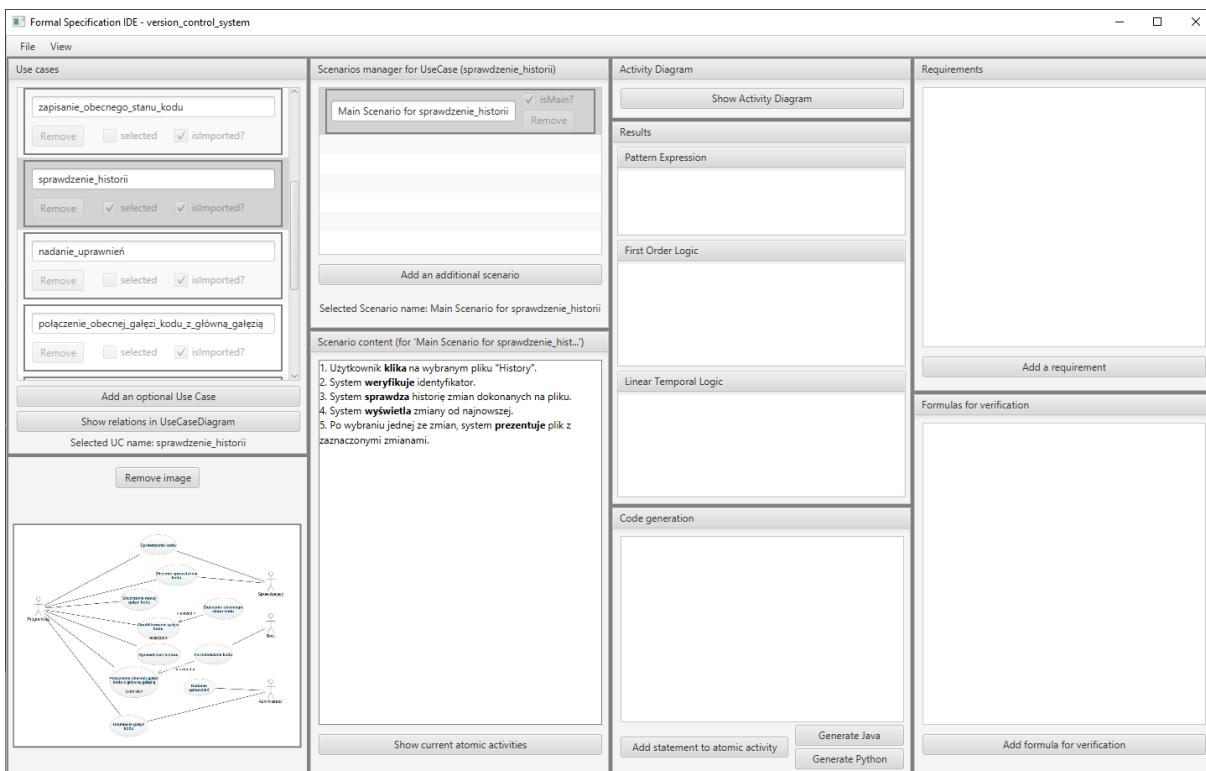
Rys. 4.82. Przykład wypełnienia scenariusza przypadku użycia „Zlecenie sprawdzenie kodu”.



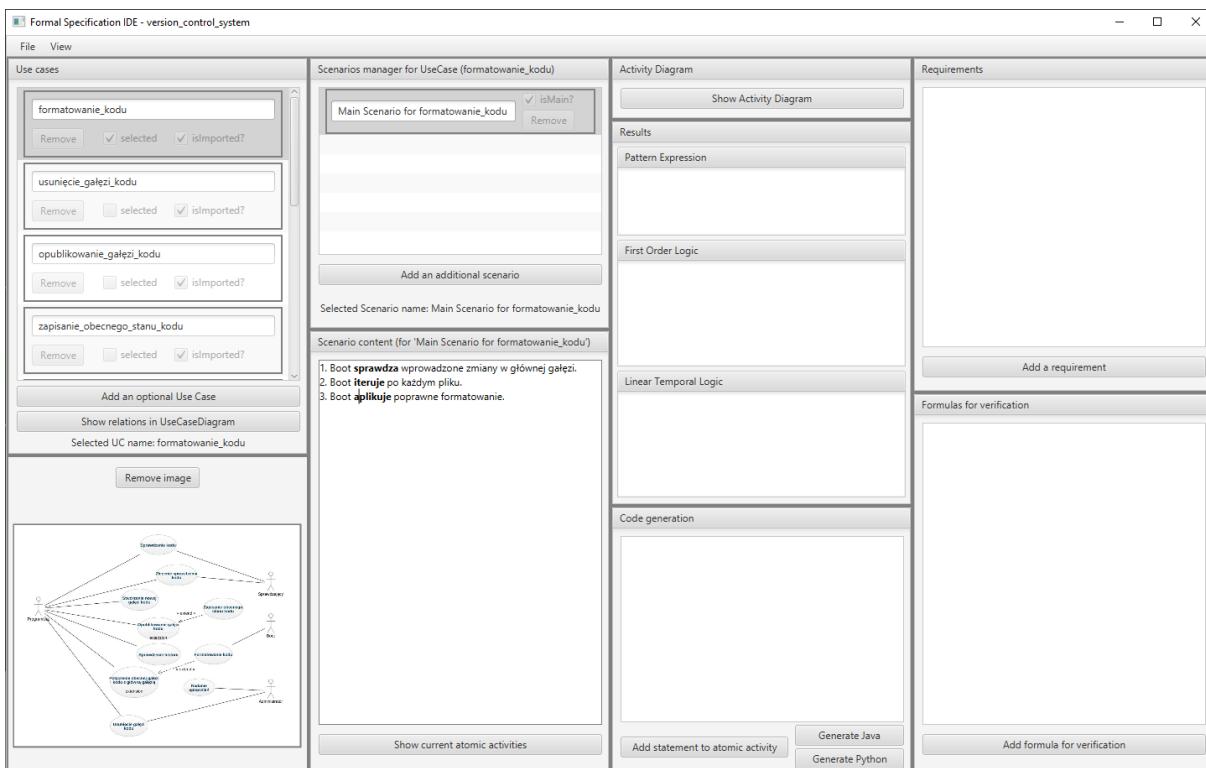
Rys. 4.83. Przykład wypełnienia scenariusza przypadku użycia „Sprawdzenie kodu”.



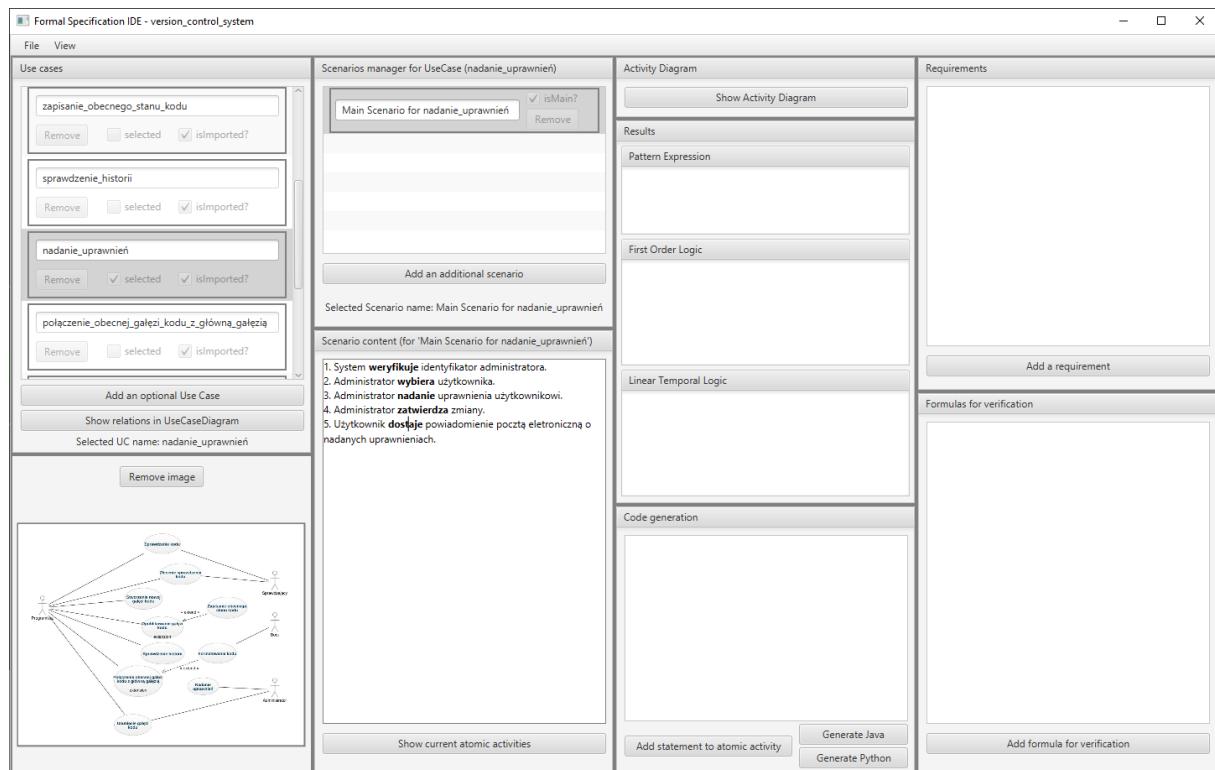
Rys. 4.84. Przykład wypełnienia scenariusza przypadku użycia „Usunięcie gałęzi kodu”.



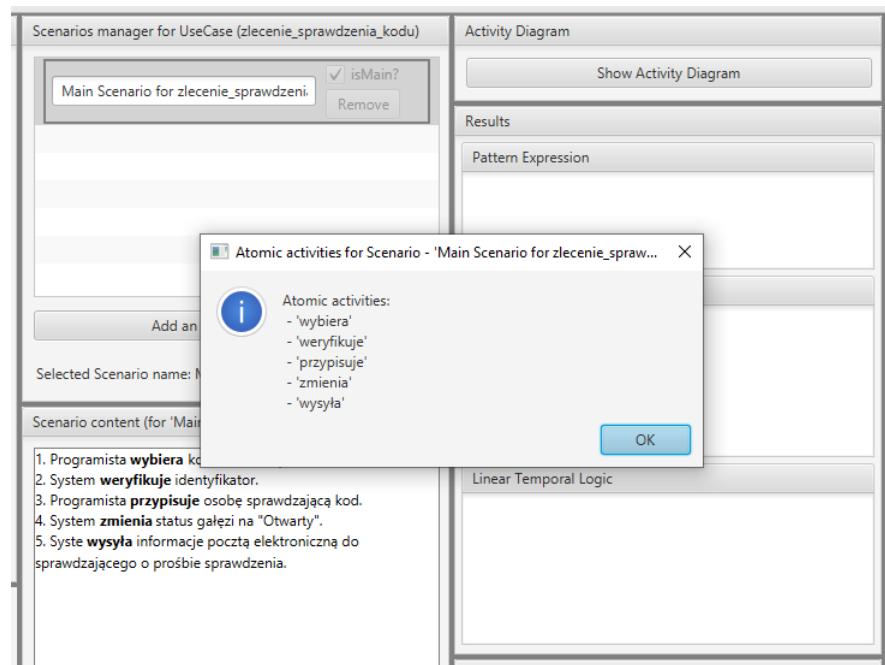
Rys. 4.85. Przykład wypełnienia scenariusza przypadku użycia „Sprawdzenie historii”.



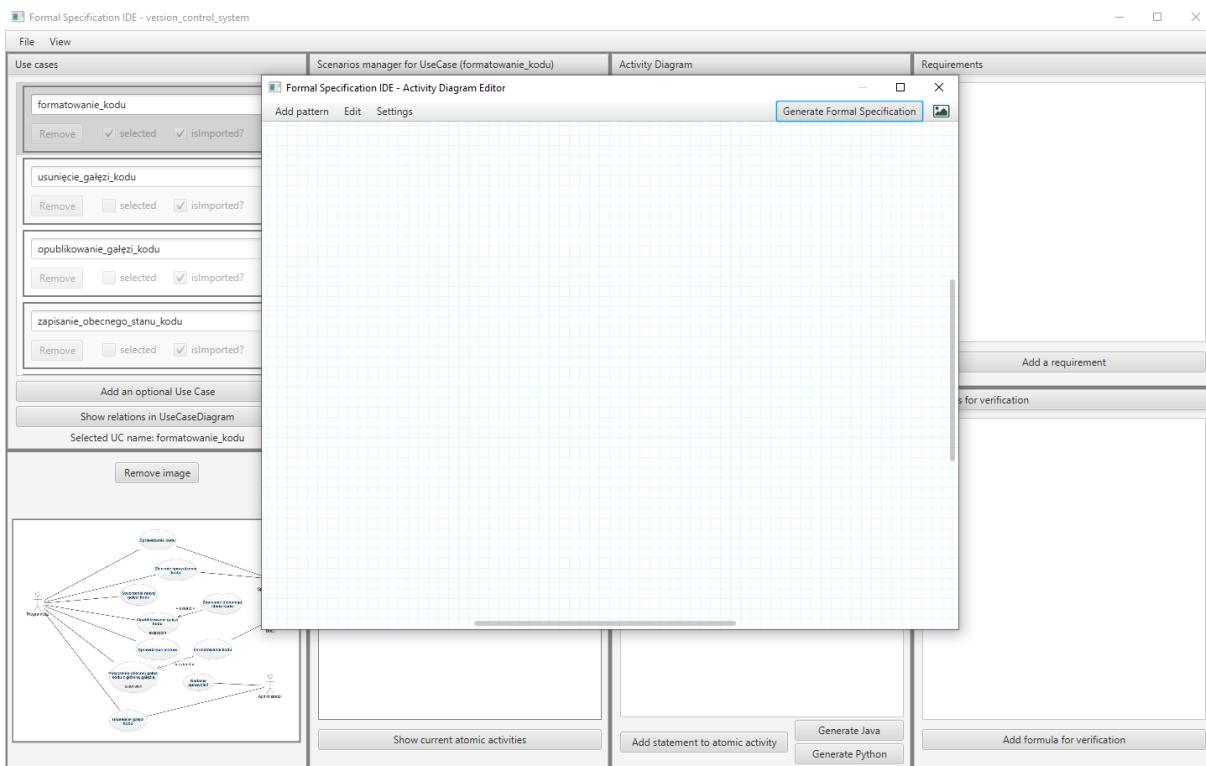
Rys. 4.86. Przykład wypełnienia scenariusza przypadku użycia „Formatowanie kodu”.



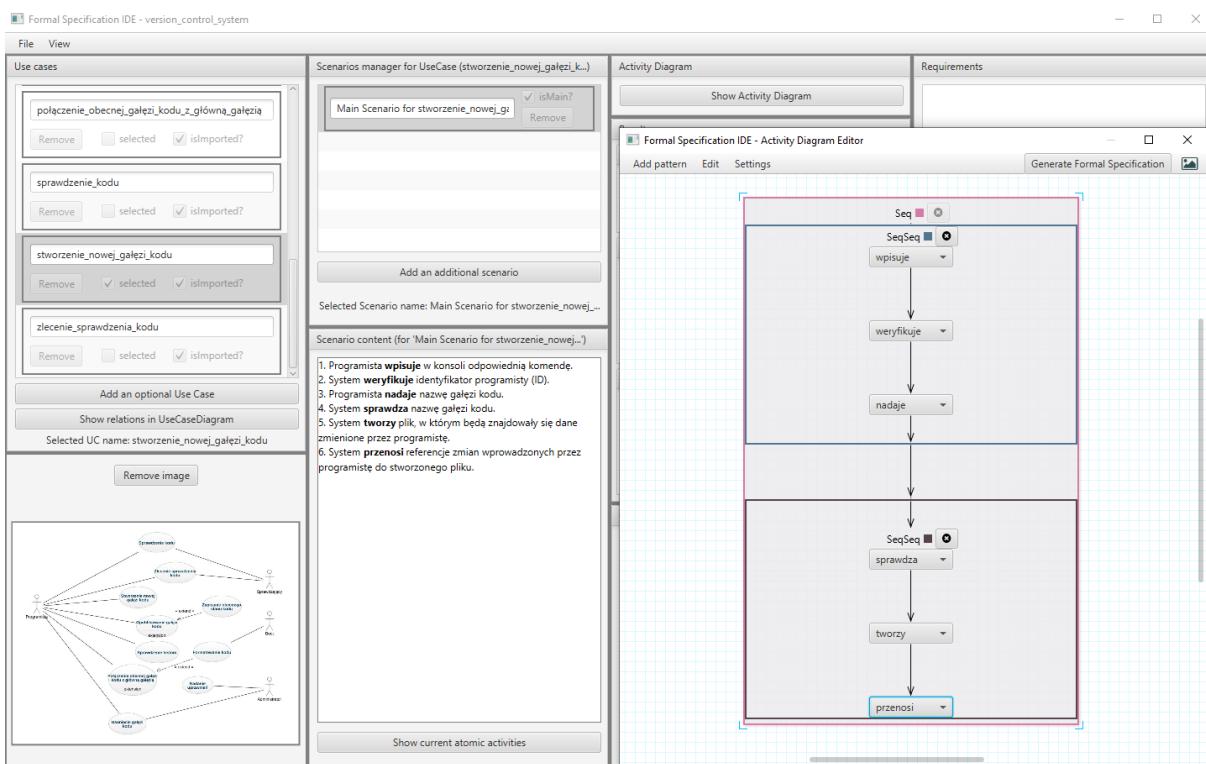
Rys. 4.87. Przykład wypełnienia scenariusza przypadku użycia „Nadanie uprawnień”.



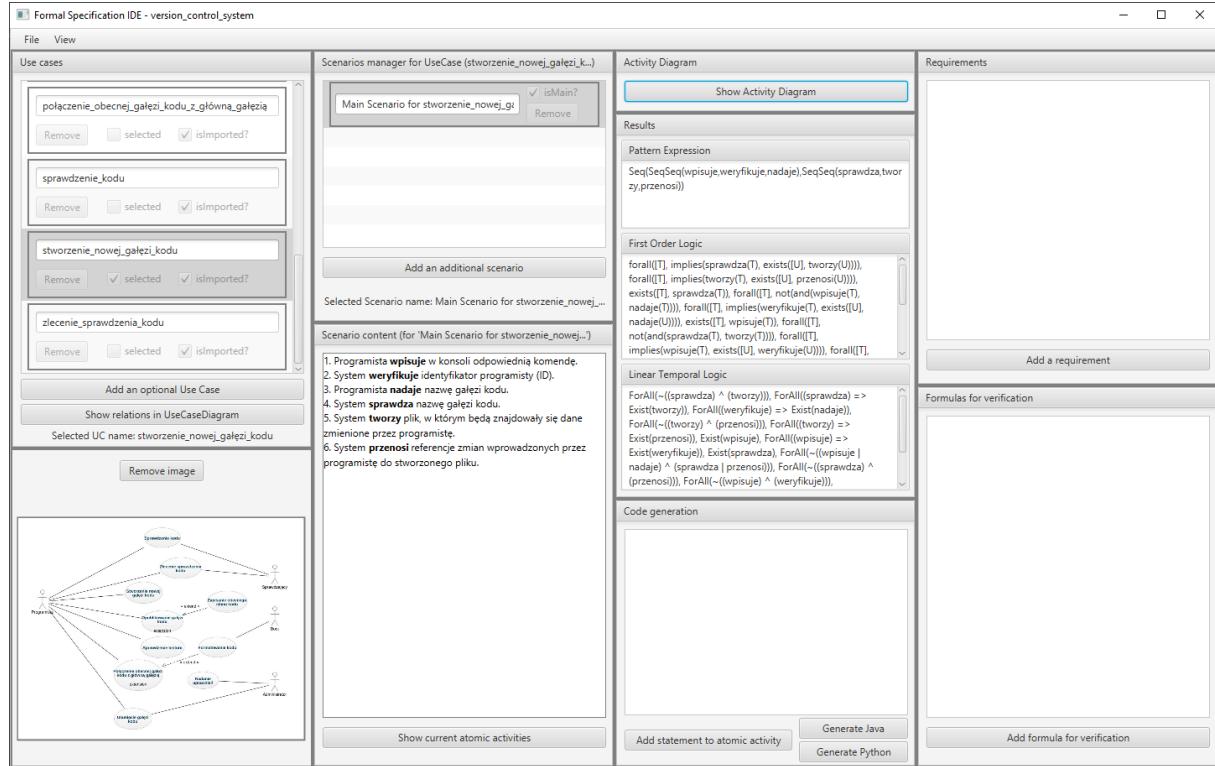
Rys. 4.88. Podczas tworzenia scenariuszy, w każdym momencie dostępny jest podgląd aktualnych atomicznych aktywności, pod przyciskiem „Show current atomic activities”.



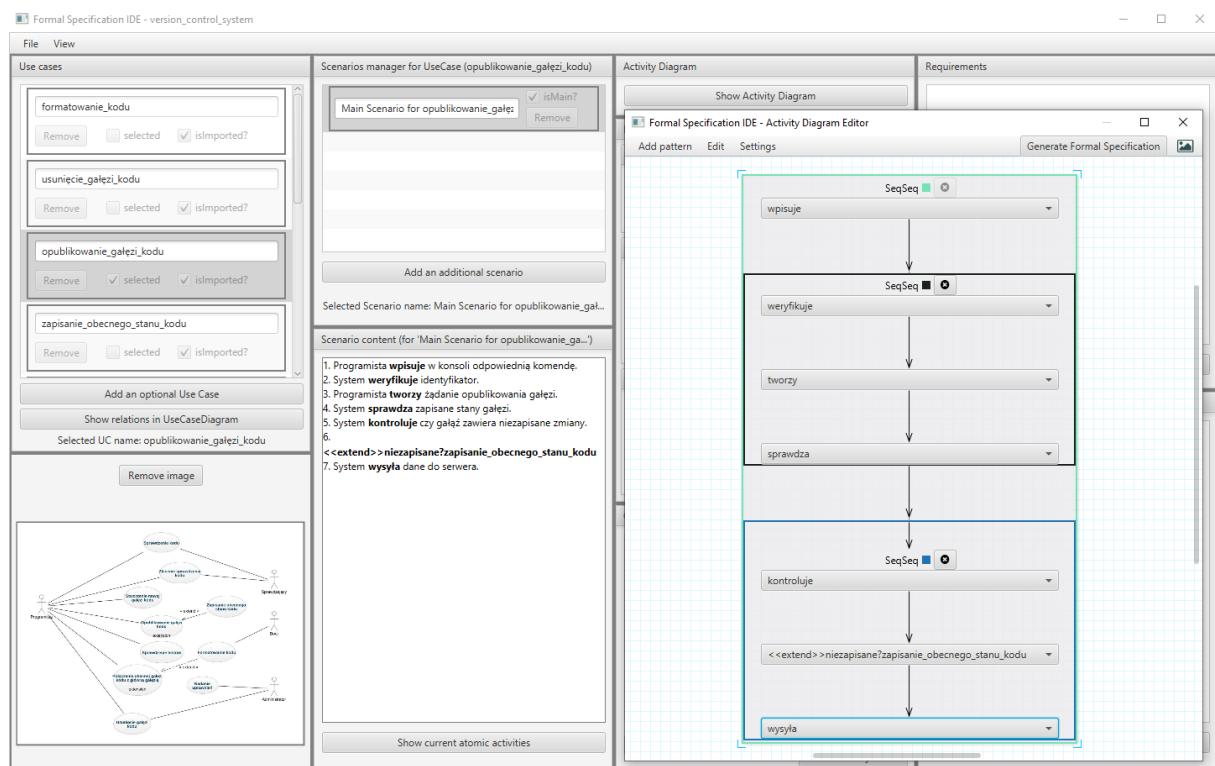
Rys. 4.89. Po uzupełnieniu treści scenariusza, następnym etapem jest modelowanie diagramu aktywności, poprzez kliknięcie „Show Activity Diagram”.



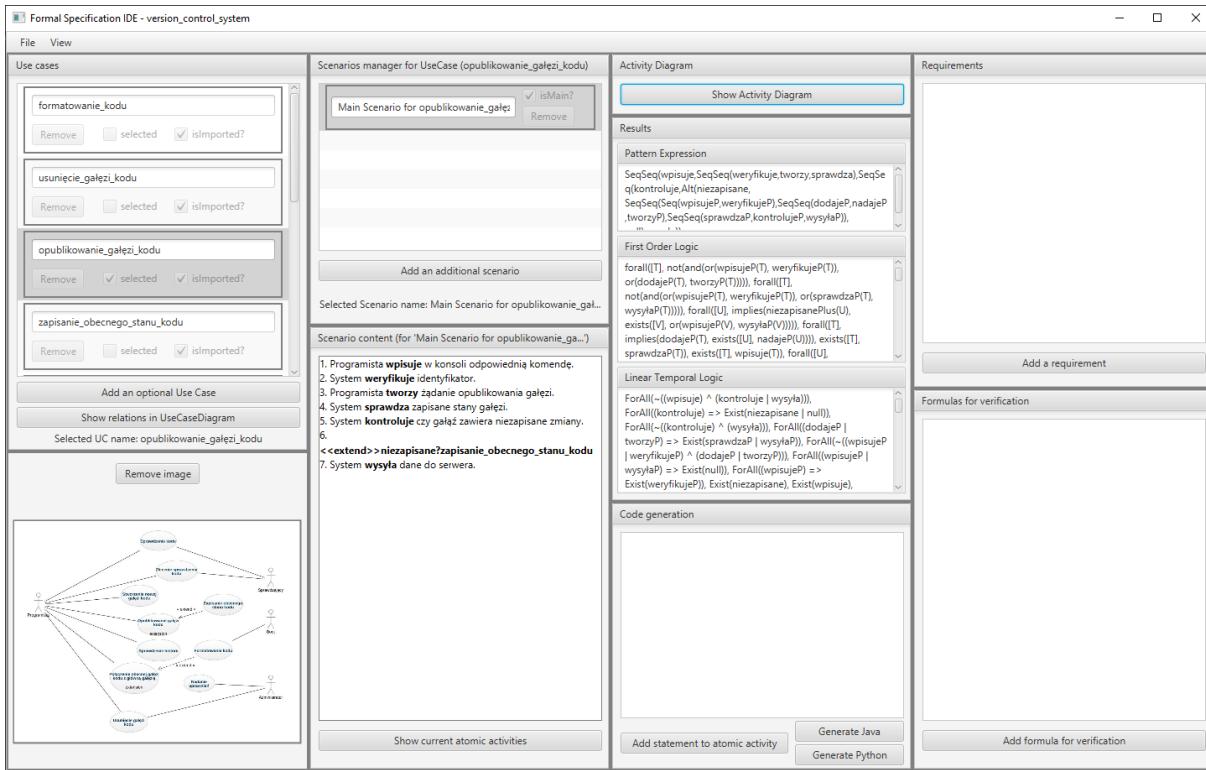
Rys. 4.90. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Stworzenie nowej gałęzi kodu”.



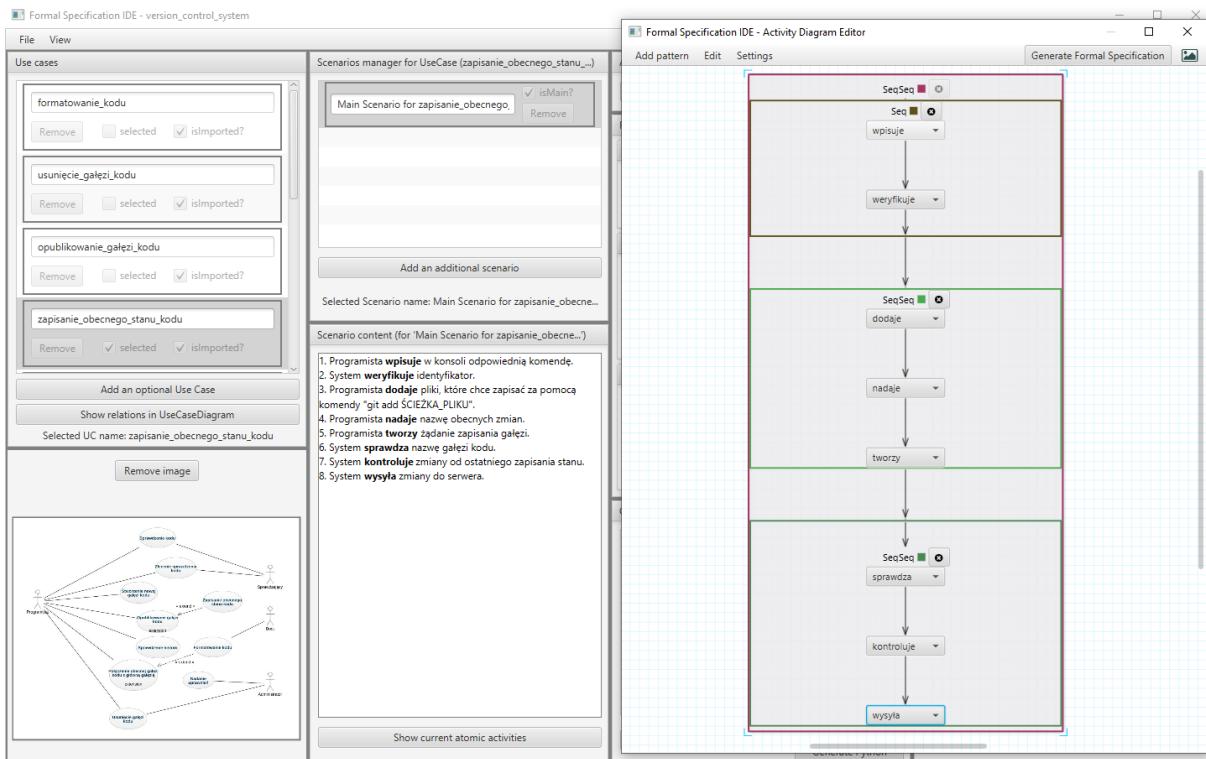
**Rys. 4.91.** Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



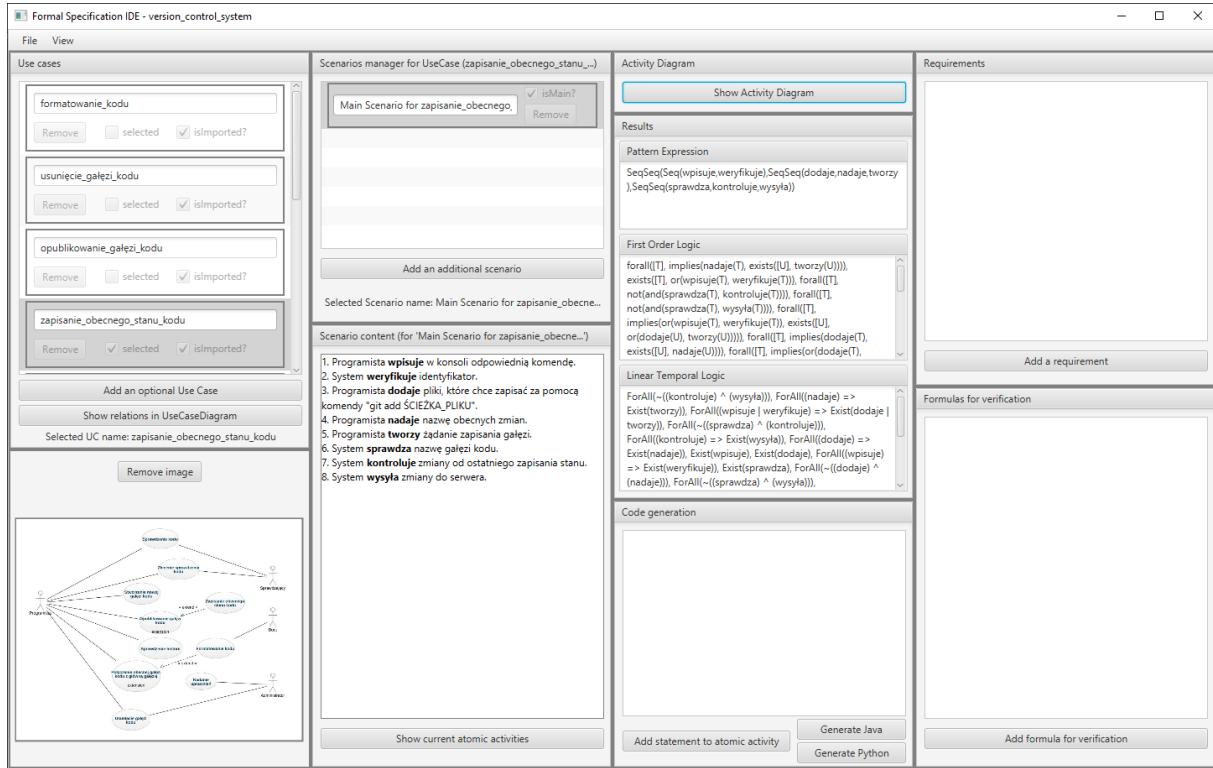
**Rys. 4.92.** Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Opublikowanie gałęzi kodu”.



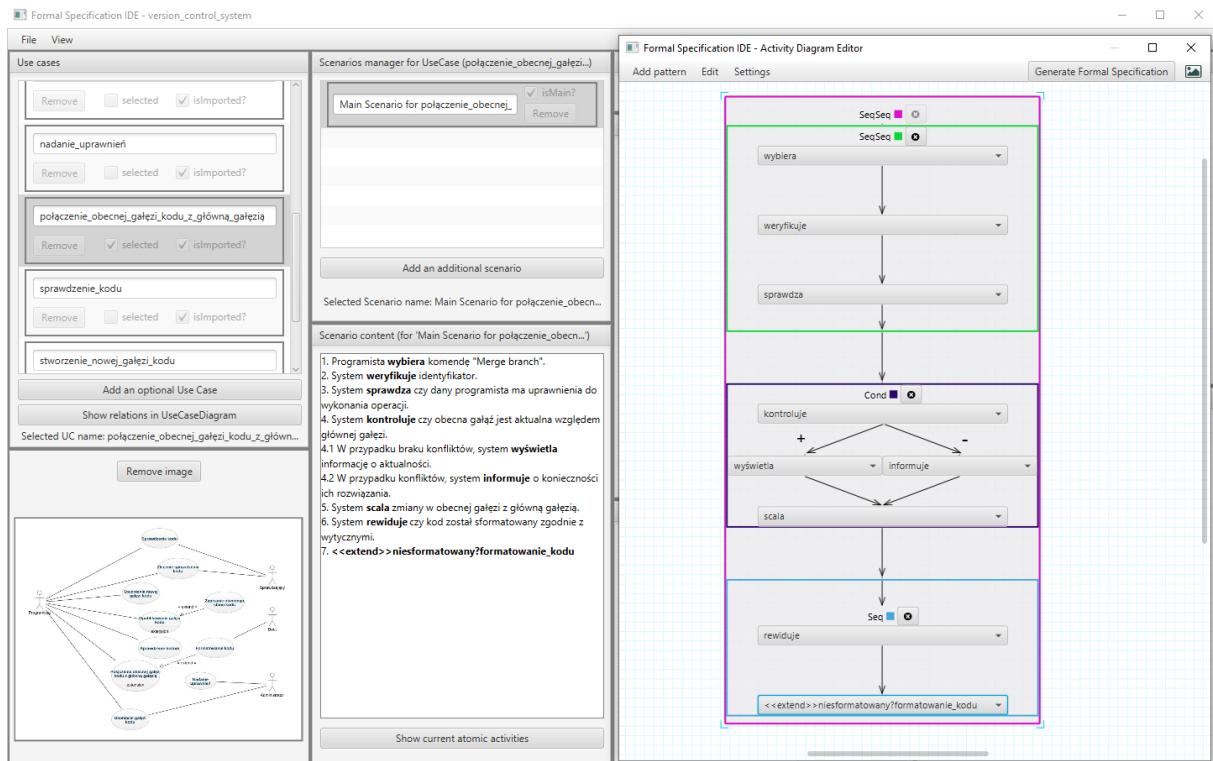
Rys. 4.93. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



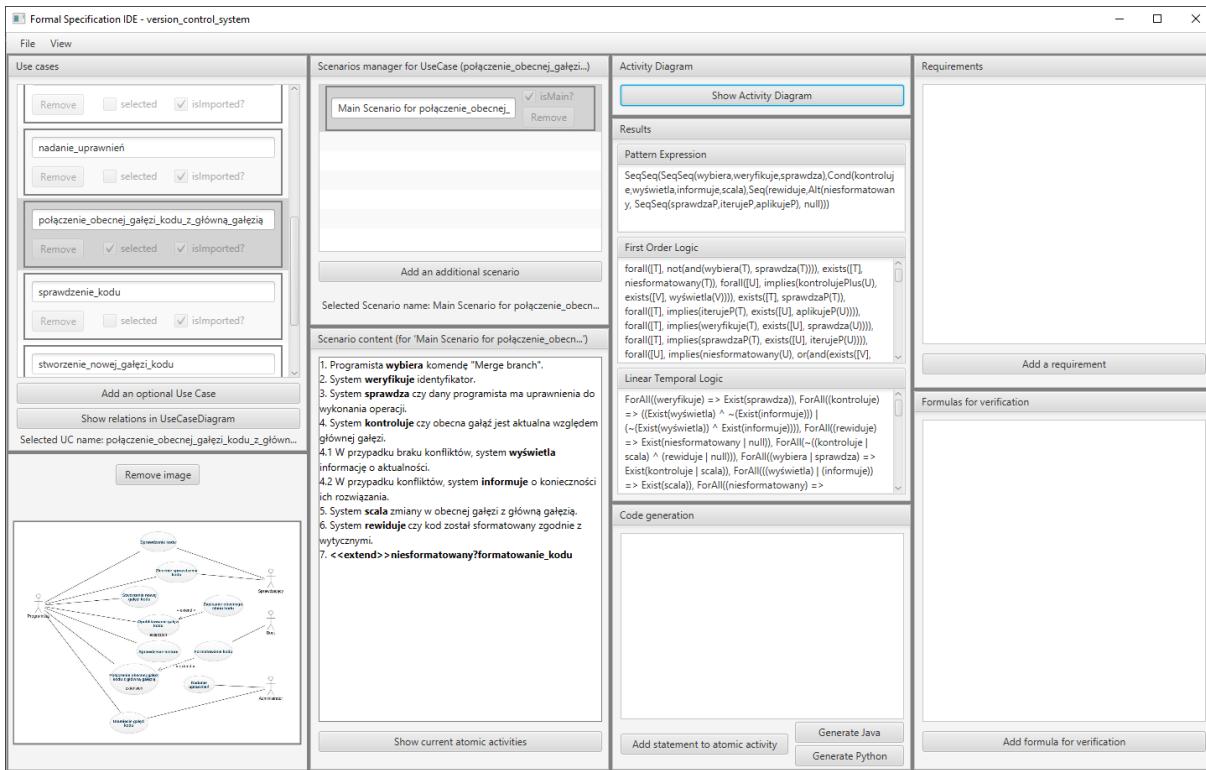
Rys. 4.94. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Zapisanie obecnego stanu kodu”.



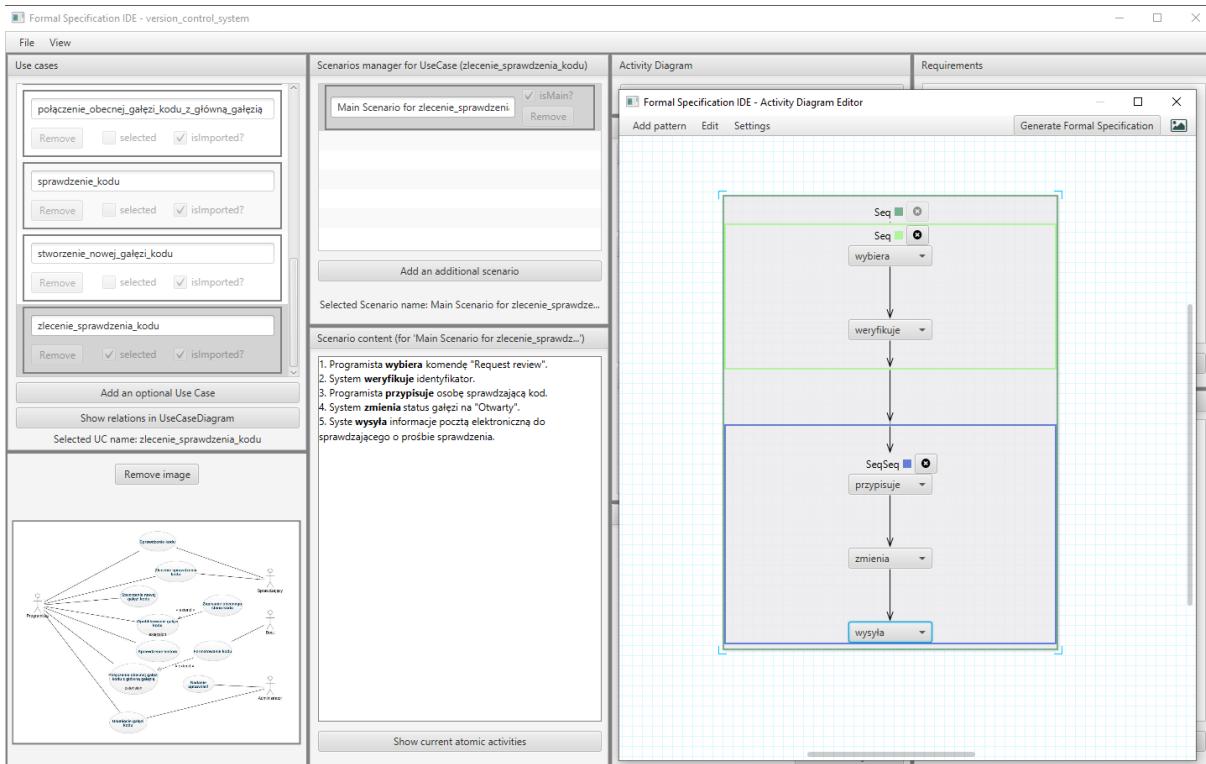
Rys. 4.95. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



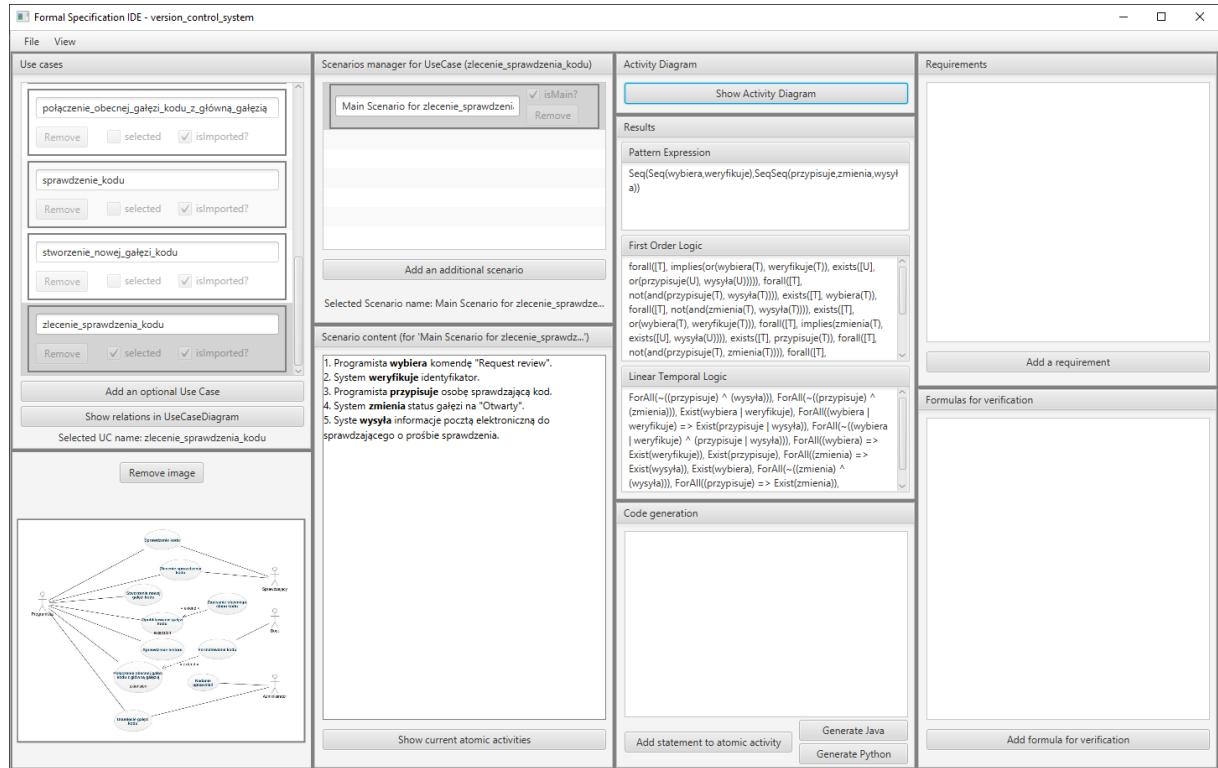
Rys. 4.96. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Połączenie obecnej gałęzi kodu z główną gałęzią”.



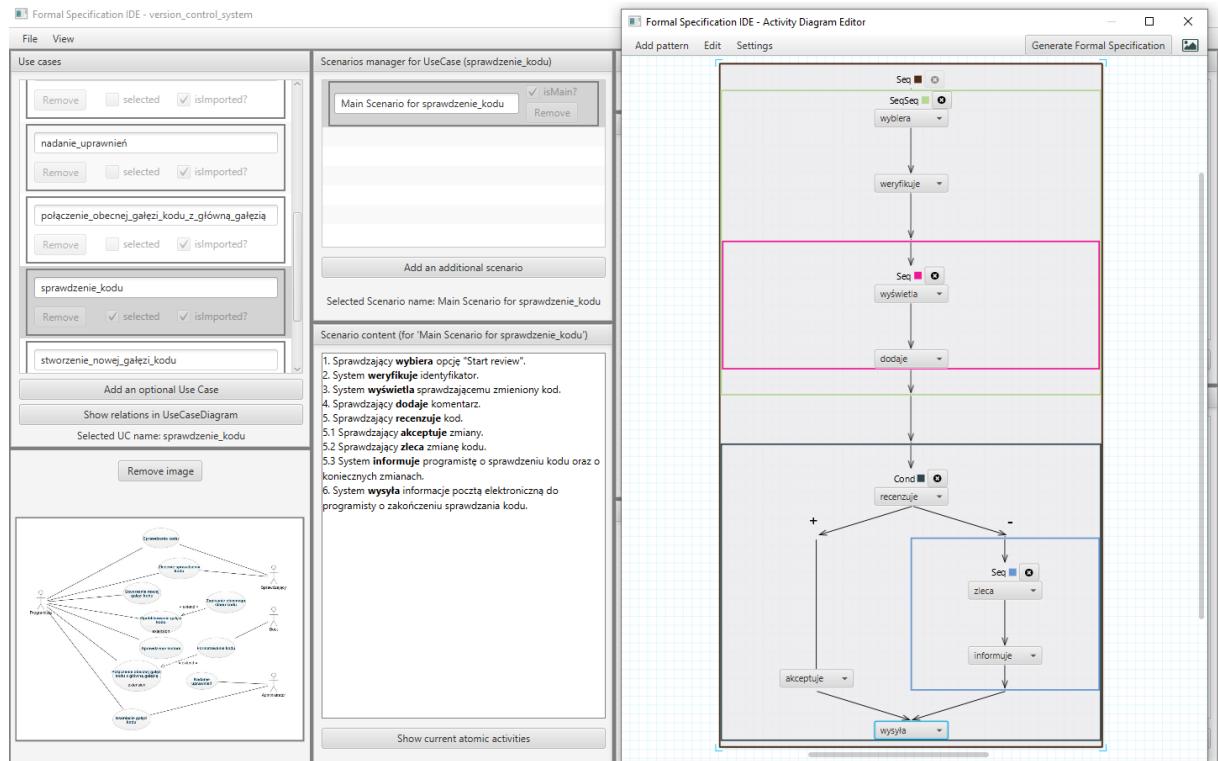
Rys. 4.97. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



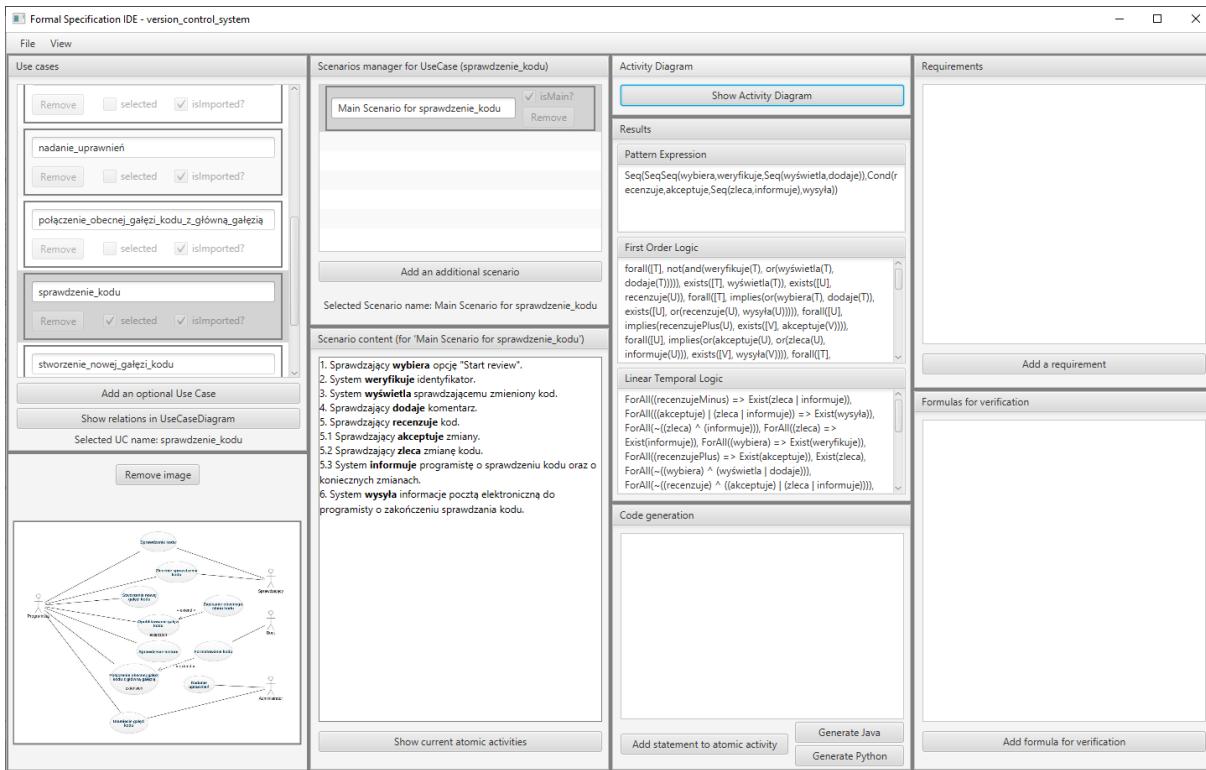
Rys. 4.98. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Zlecenie sprawdzenie kodu”.



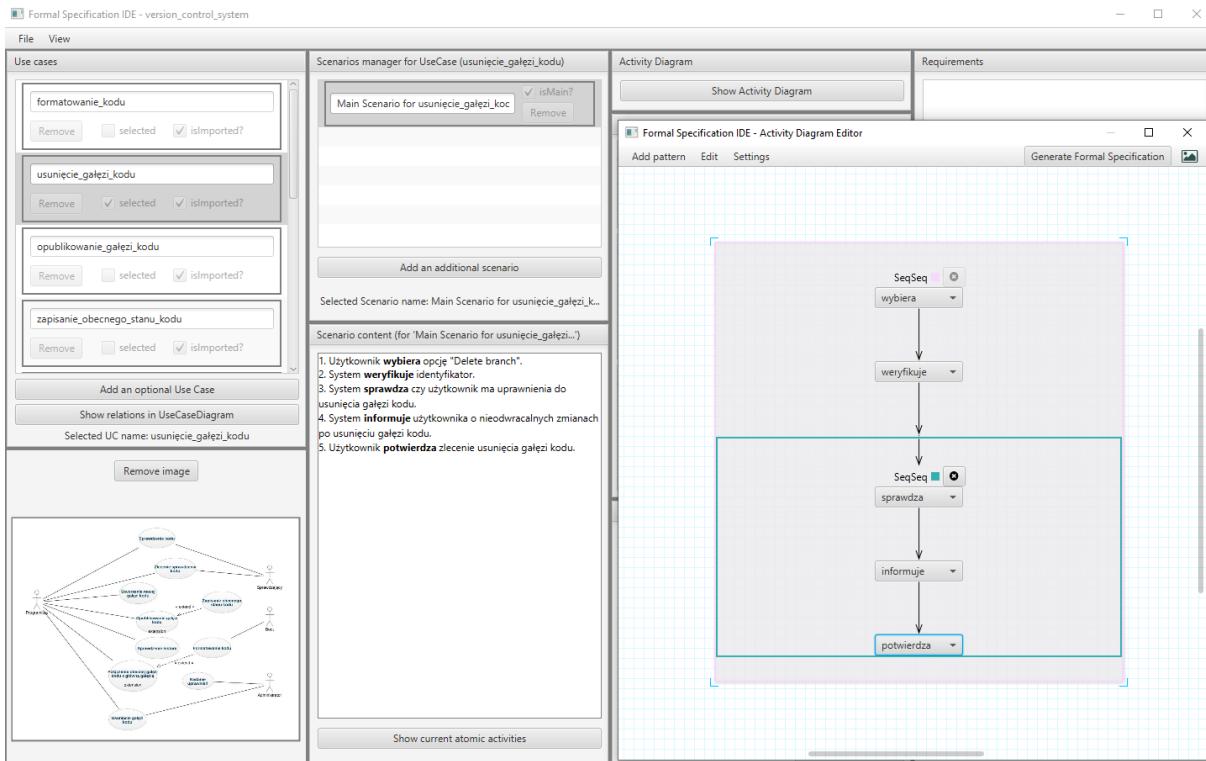
Rys. 4.99. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



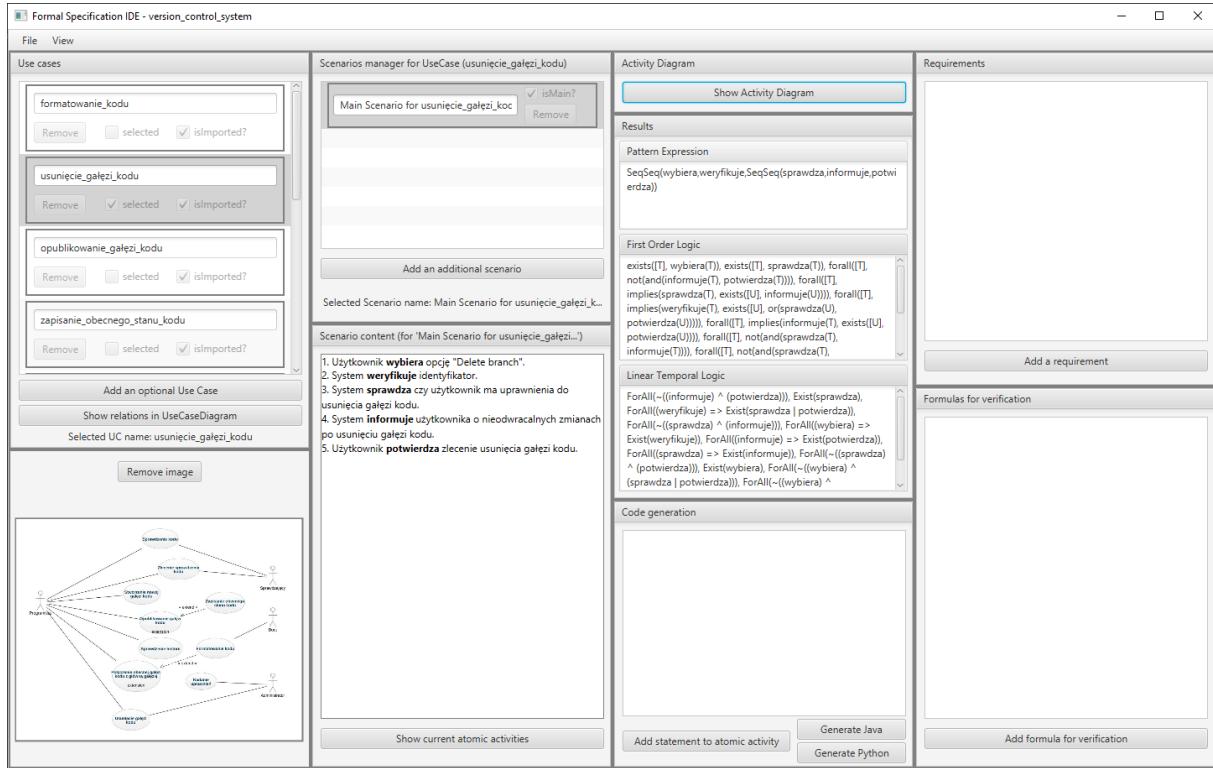
Rys. 4.100. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Sprawdzenie kodu”.



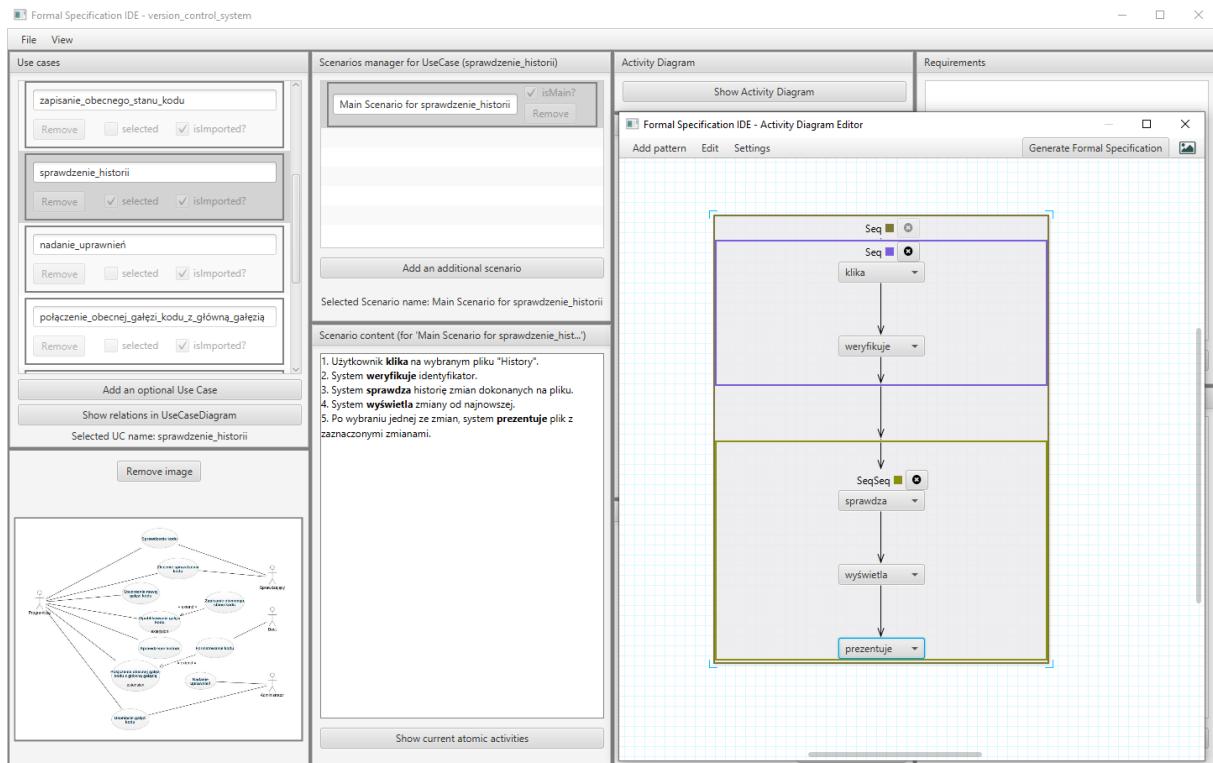
Rys. 4.101. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



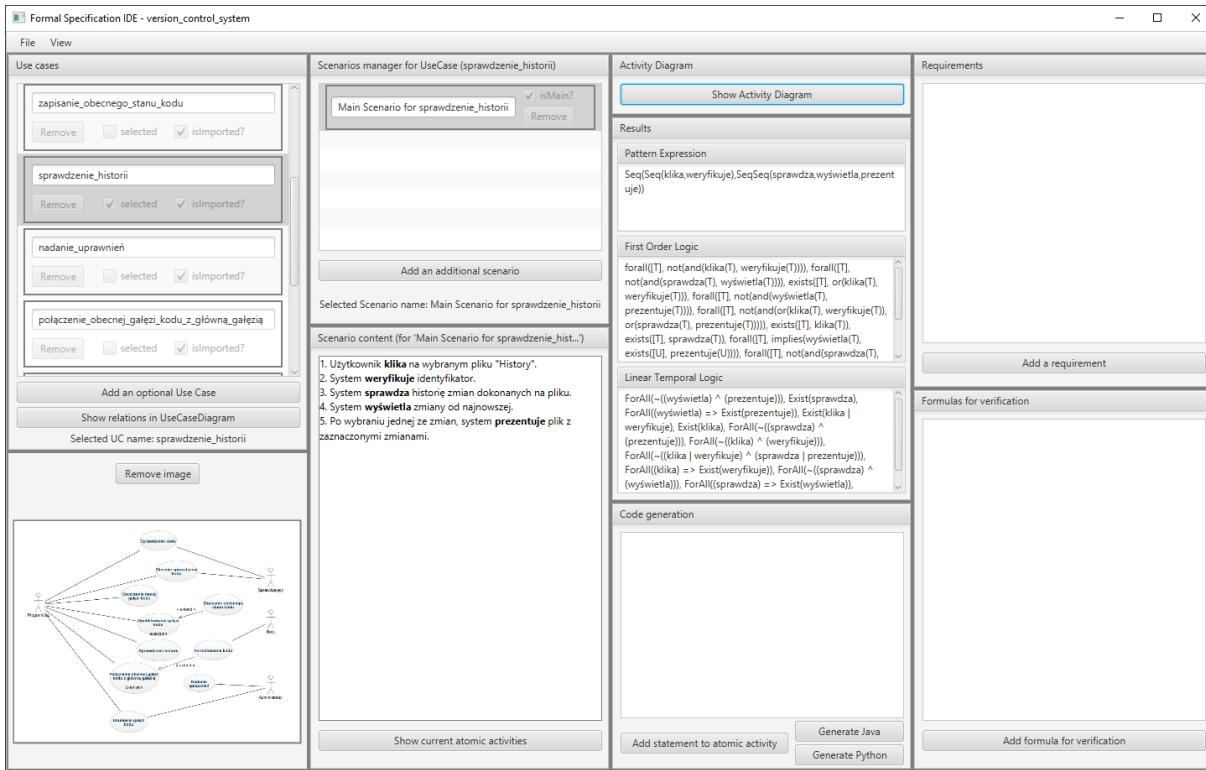
Rys. 4.102. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Usunięcie gałęzi kodu”.



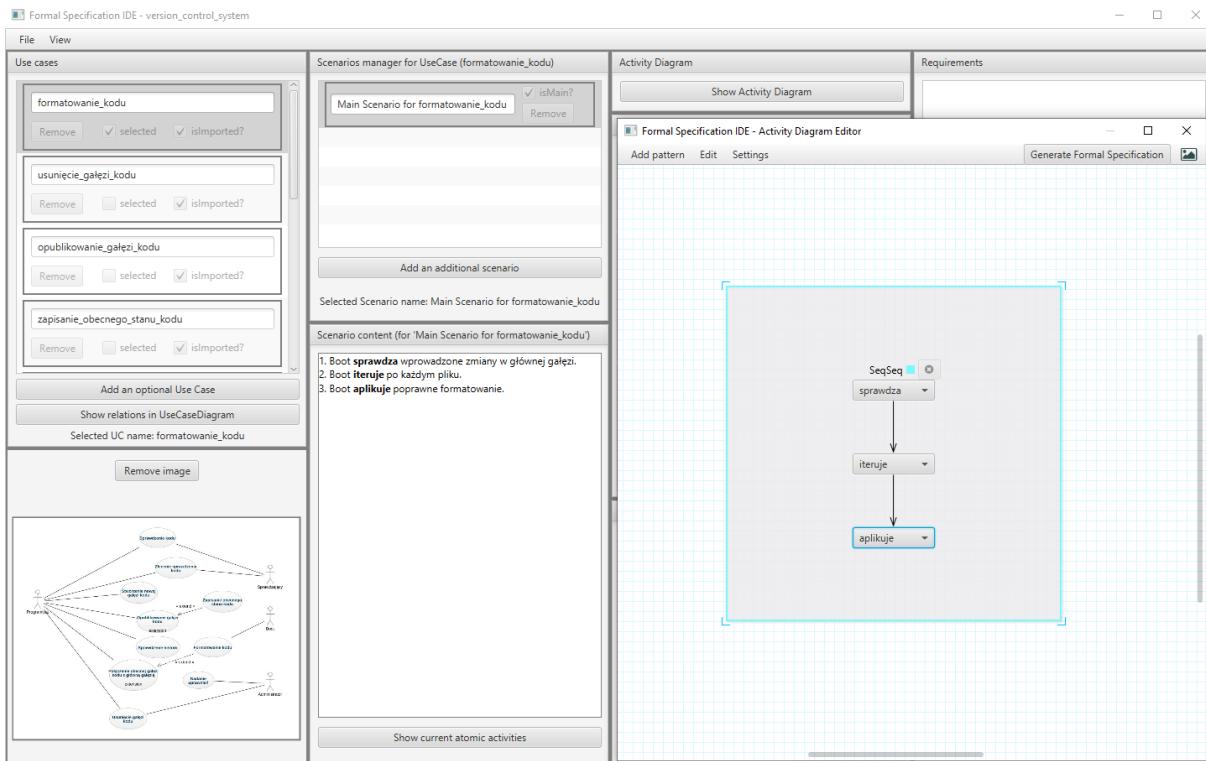
Rys. 4.103. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



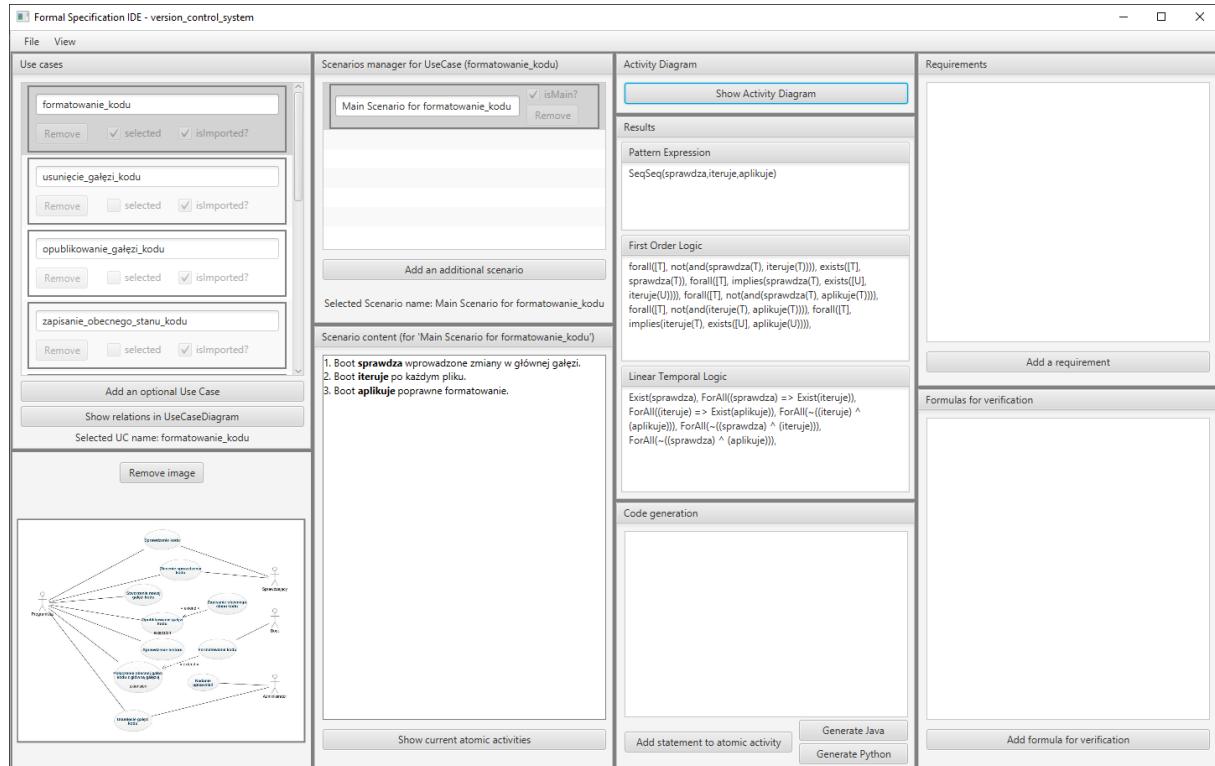
Rys. 4.104. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Sprawdzenie historii”.



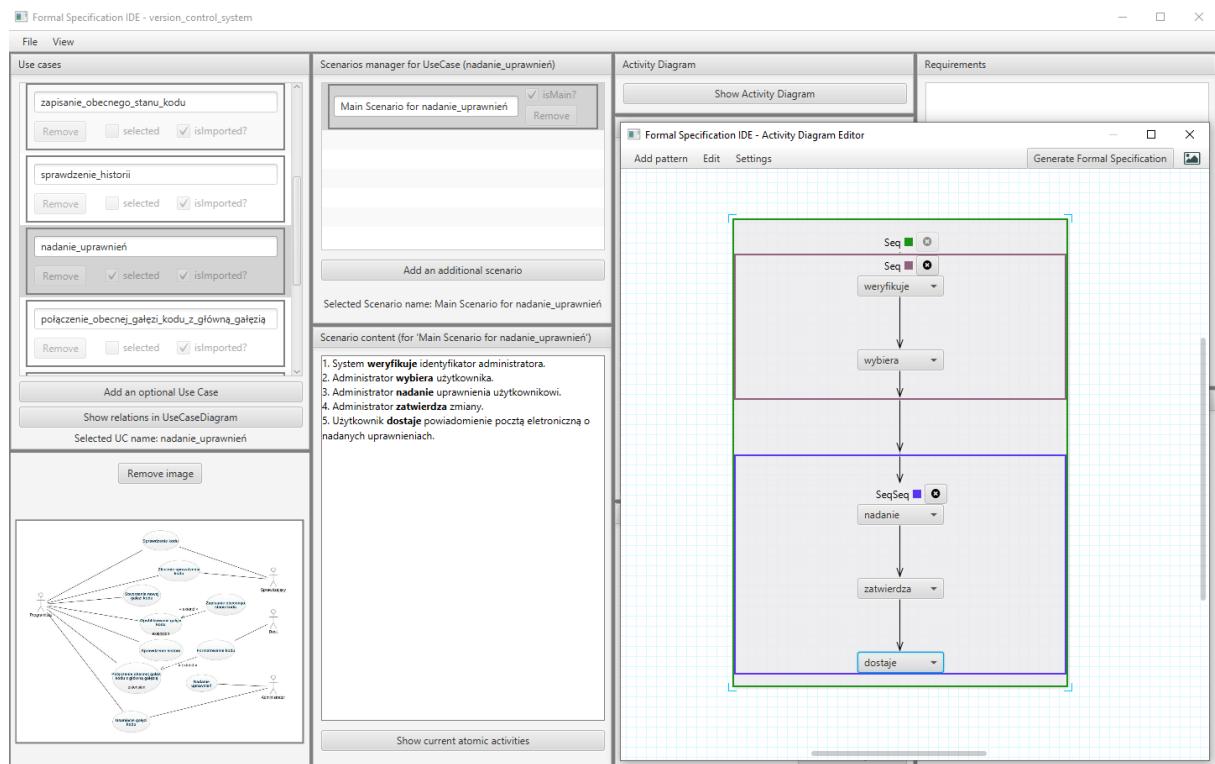
Rys. 4.105. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



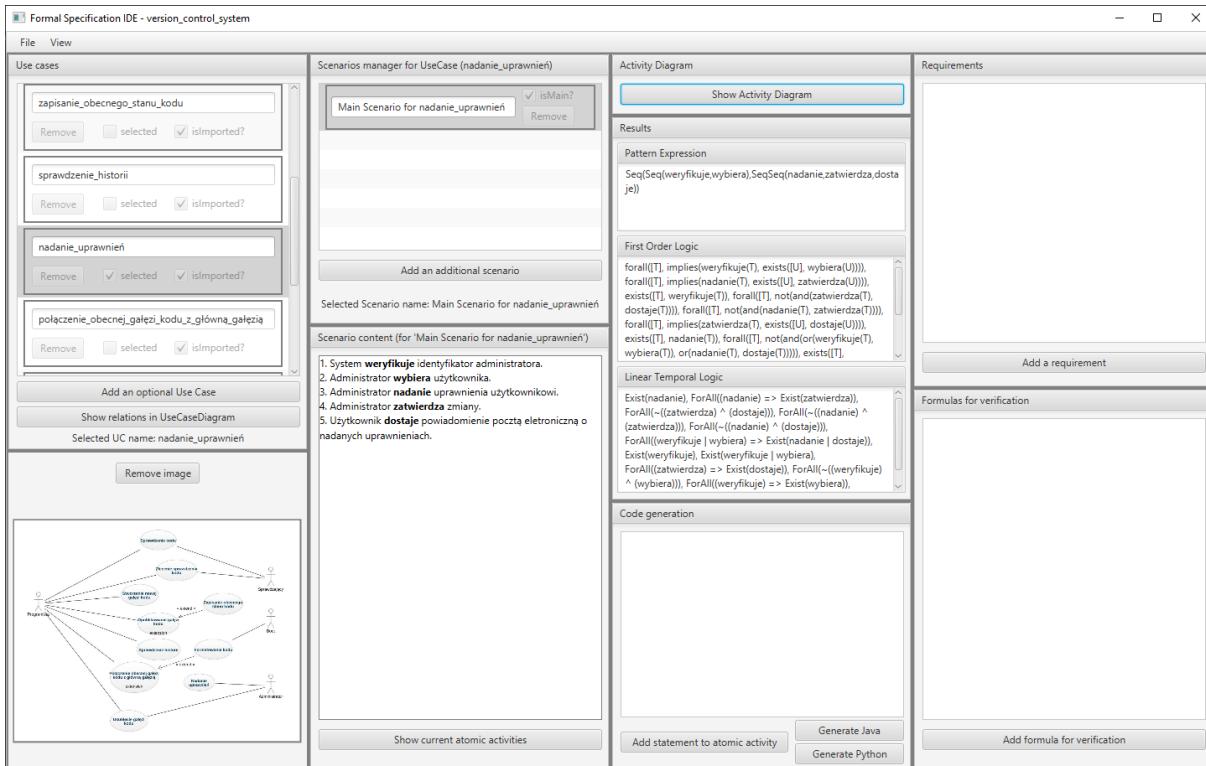
Rys. 4.106. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Formatowanie kodu”.



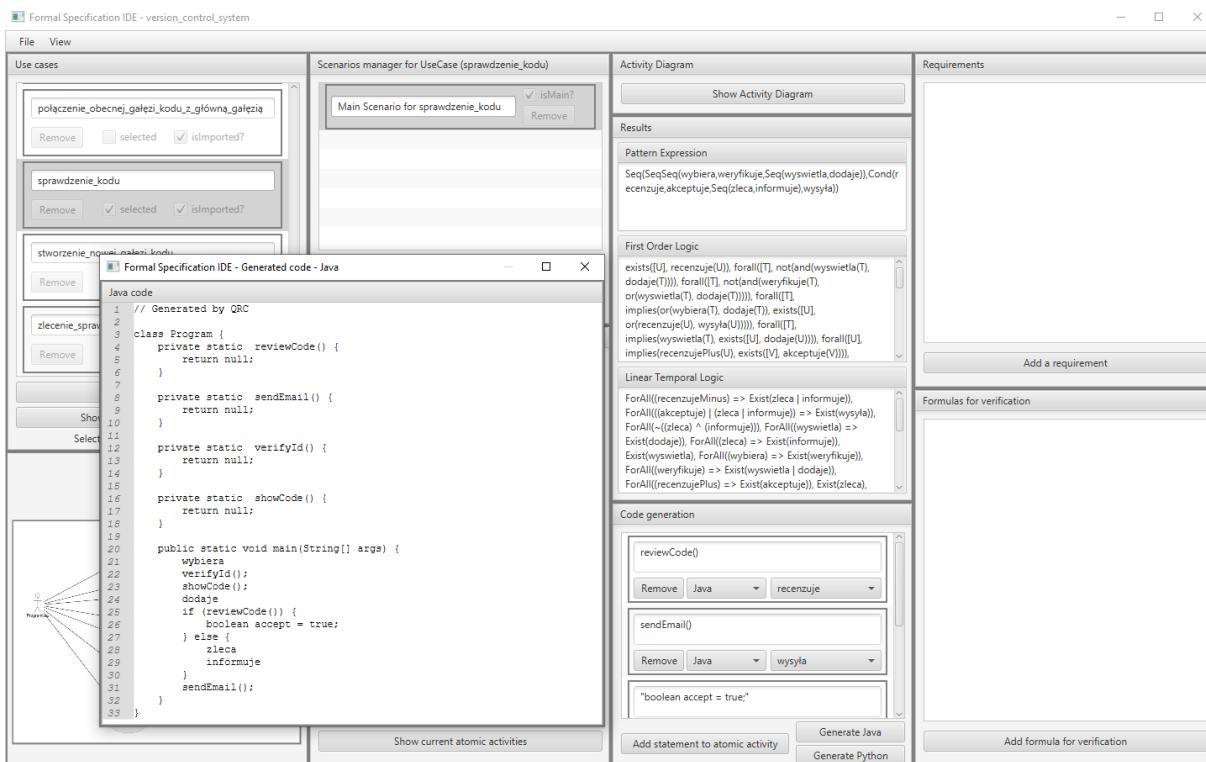
Rys. 4.107. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



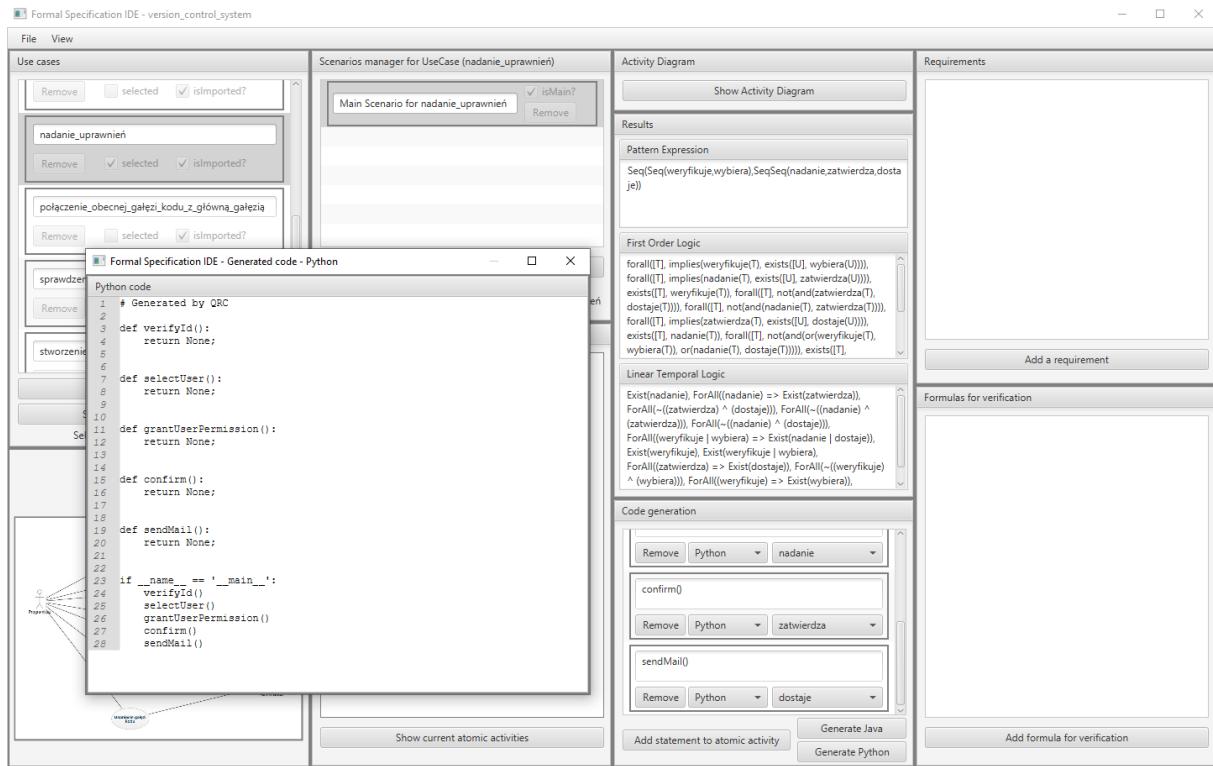
Rys. 4.108. Przykład zamodelowanego diagramu aktywności na podstawie scenariusza przypadku użycia „Nadanie uprawnień”.



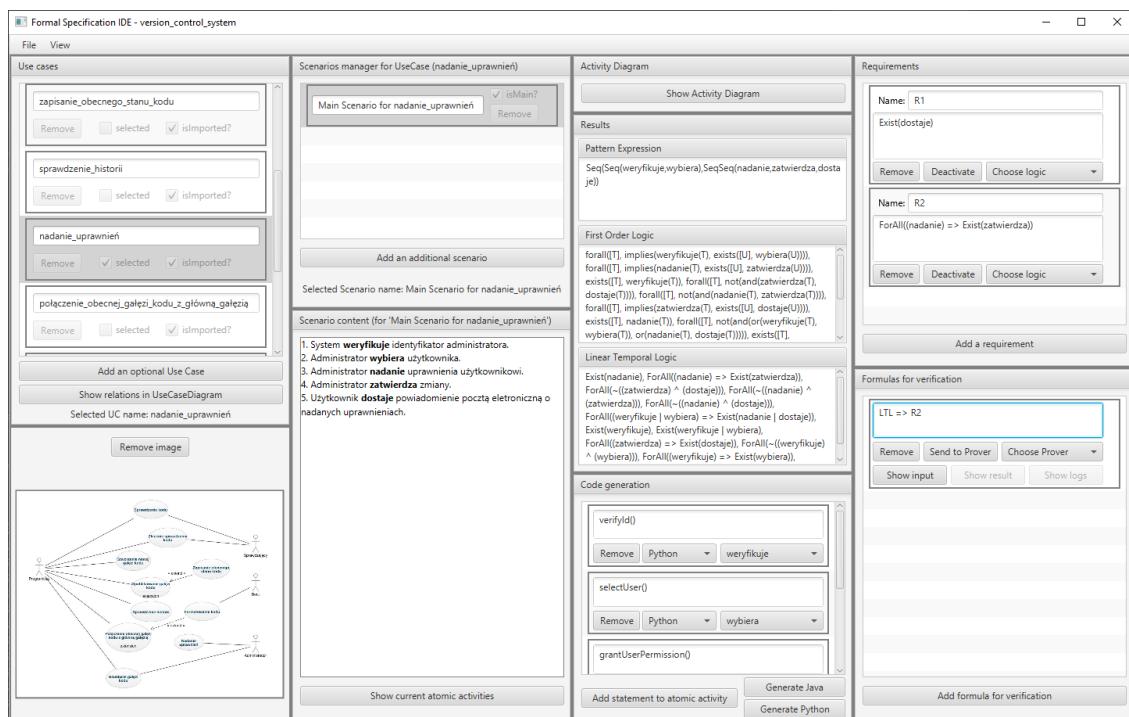
Rys. 4.109. Po stworzeniu diagramu aktywności oraz kliknięciu „Generate Specification”, w panelu „Results” pojawiają się wyniki generacji.



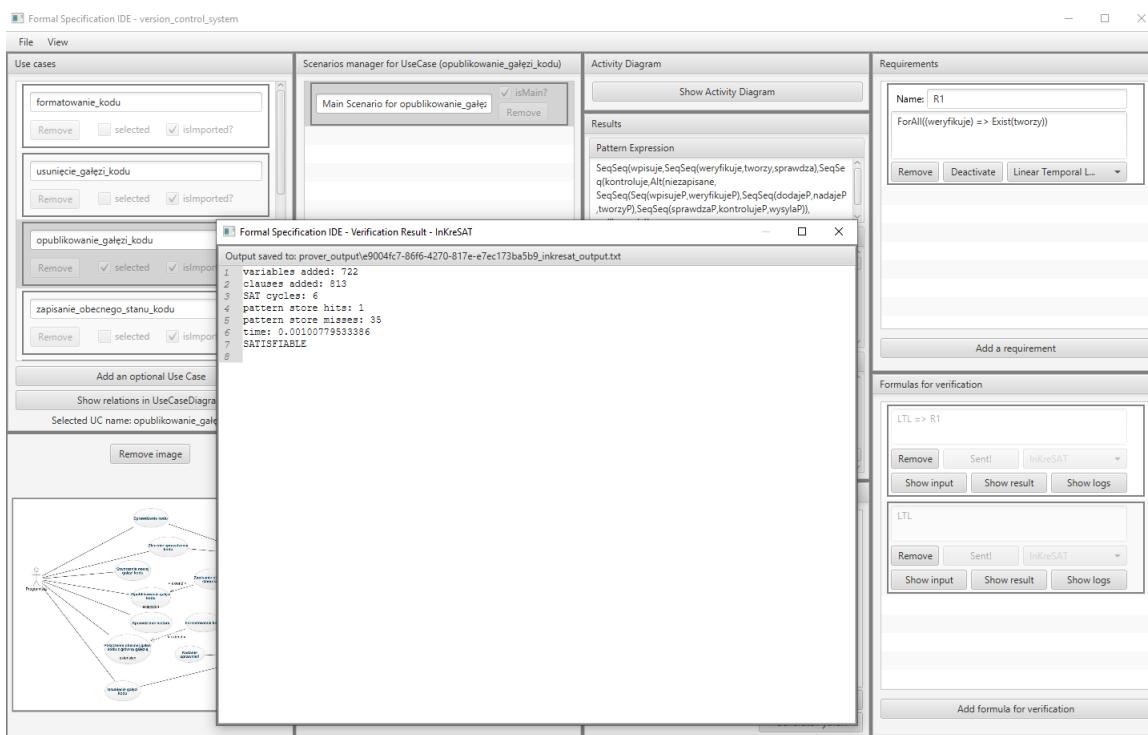
Rys. 4.110. Przykład wygenerowanego kodu źródłowego w języku Java. Zastosowano mapowanie aktywności na instrukcje języka programowania.



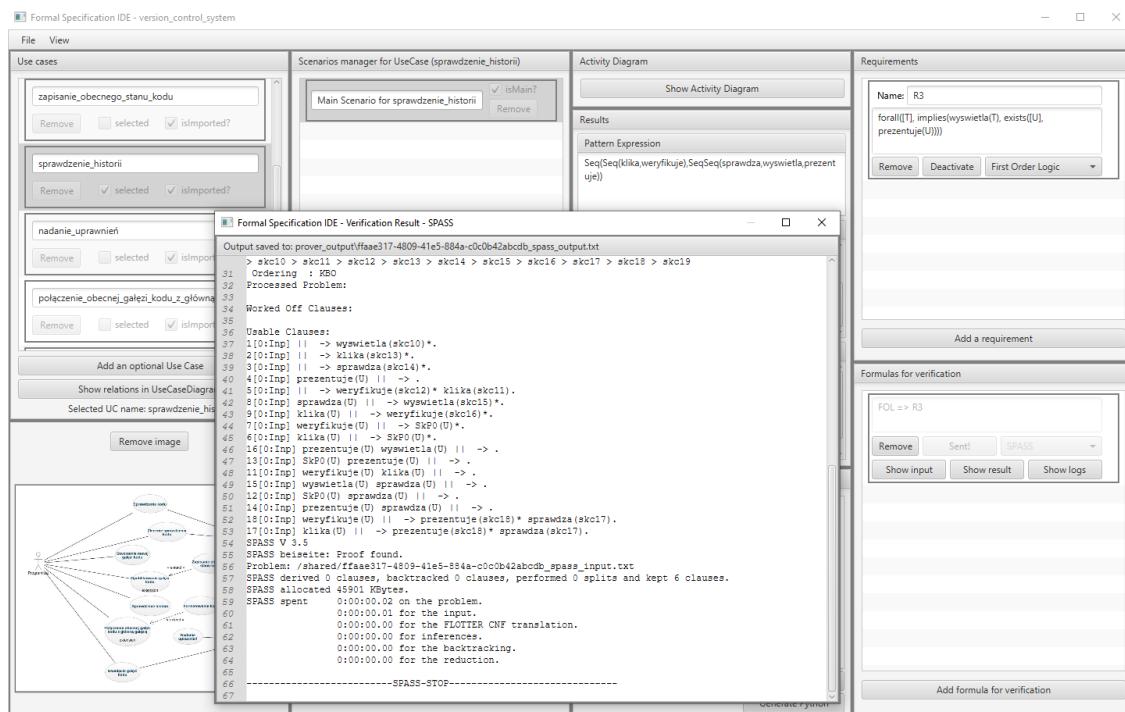
Rys. 4.111. Przykład wygenerowanego kodu źródłowego w języku Python.  
Zastosowano mapowanie aktywności na instrukcje języka programowania.



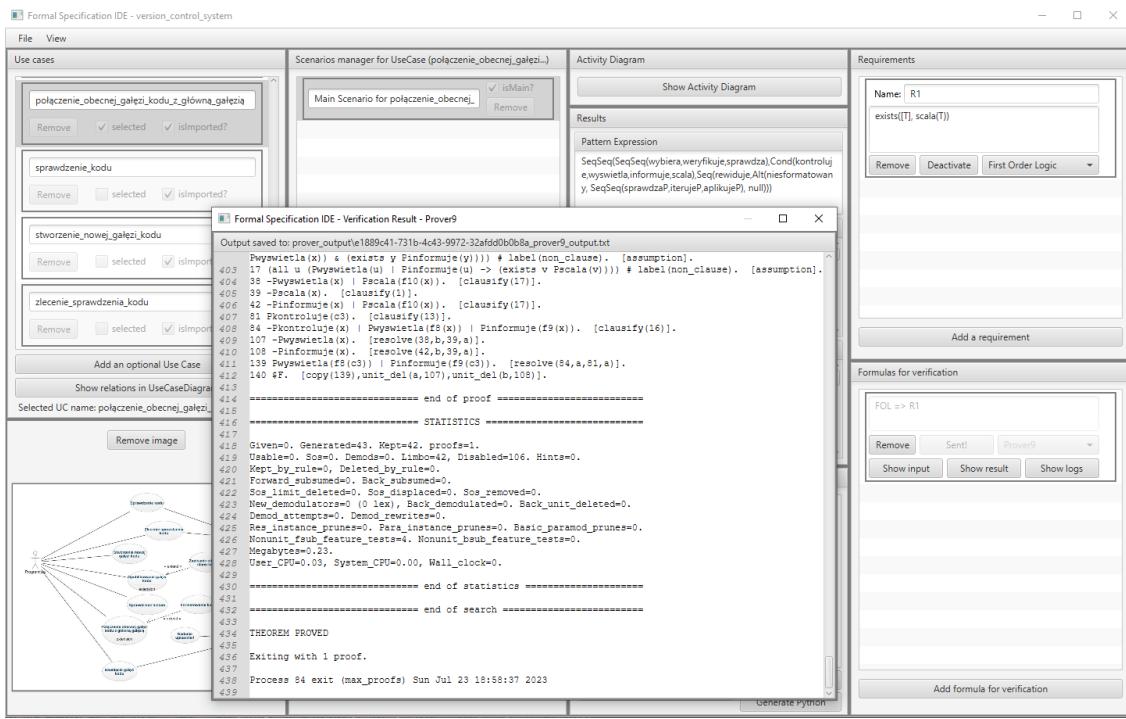
Rys. 4.112. Posiadając wygenerowaną specyfikację, można przeprowadzić formalną weryfikację. Należy wprowadzić treść, wybrać prover i kliknąć „Send to Prover”.



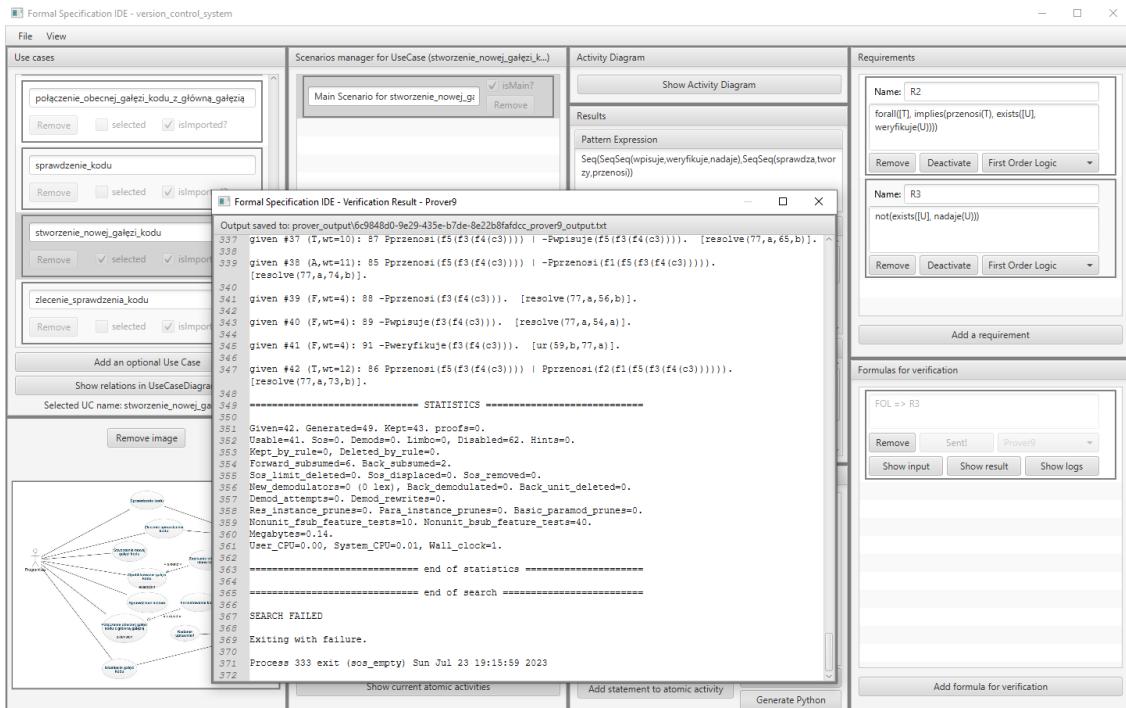
Rys. 4.113. Przykład pozytywnego wyniku weryfikacji formuły  $LTL \Rightarrow R1$ , z użyciem InKreSAT, dla przypadku użycia „Opublikowanie gałęzi kodu”.



Rys. 4.114. Przykład wyniku weryfikacji SPASS Prover, w logice pierwszego rzędu, dla przypadku użycia „Sprawdzenie historii”.



Rys. 4.115. Przykład pozytywnego wyniku weryfikacji Prover9, z użyciem dodatkowego wymagania R1, w logice pierwszego rzędu, dla przypadku używania „Połączenie obecnej gałęzi kodu z główną gałęzią”.



Rys. 4.116. Przykład negatywnego wyniku weryfikacji Prover9, z użyciem dodatkowego wymagania R2, w logice pierwszego rzędu, dla przypadku używania „Stworzenie nowej gałęzi kodu”.

Przypadek użycia	Badana formula	Prover	Rezultat	Wnioski
Formatowanie kodu	FOL	SPASS Prover	Completion found	Formuła jest poprawna.
Formatowanie kodu	FOL => R1	SPASS Prover	Proof found	Formuła została udowodniona.
Usunięcie gałęzi kodu	LTL	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Usunięcie gałęzi kodu	LTL => R2	InKreSAT	UNSATISFIABLE	Formuła nie jest spełnialna. Należy poprawić formułę.
Usunięcie gałęzi kodu	LTL => R3	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Opublikowanie gałęzi kodu	FOL	SPASS Prover	Completion found	Formuła jest poprawna.
Opublikowanie gałęzi kodu	LTL	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Opublikowanie gałęzi kodu	LTL => R4	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Zapisanie obecnego stanu kodu	FOL => R5	Prover9	THEOREM PROVED	Formuła została udowodniona.
Sprawdzenie historii	LTL => R6	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Nadanie uprawnień	LTL	InKreSAT	SATISFIABLE	Formuła jest spełnialna.
Połączenie obecnej gałęzi kodu z główną gałęzią	FOL => R7	Prover9	THEOREM PROVED	Formuła jest spełnialna.
Sprawdzenie kodu	FOL	SPASS Prover	Completion found	Formuła jest poprawna.
Stworzenie nowej gałęzi kodu	FOL => R8	Prover9	THEOREM PROVED	Formuła została udowodniona
Zlecenie sprawdzenia kodu	LTL	InKreSAT	SATISFIABLE	Formuła jest spełnialna.

**Tabela 4.2.** Zestawienie uzyskanych wyników weryfikacji przeprowadzonych dla projektowanego systemu kontroli wersji kodu. R1 =  $\text{forall}([T], \text{implies}(\text{iteruje}(T), \text{exists}([U], \text{aplikuje}(U))))$ , R2 =  $\text{ForAll}(\sim(\text{weryfikuje}) \wedge (\text{sprawdza} \mid \text{potwierdza}))$ , R3 =  $\text{ForAll}((\text{weryfikuje}) \wedge (\text{sprawdza} \mid \text{potwierdza}))$ , R4 =  $\text{ForAll}(\text{weryfikuje}) \Rightarrow \text{Exist}(\text{tworzy})$ , R5 =  $\text{forall}([T], \text{not}(\text{and}(\text{wpisuje}(T), \text{weryfikuje}(T))))$ , R6 =  $\text{ForAll}((\text{klik} \mid \text{weryfikuje}) \Rightarrow \text{Exist}(\text{sprawdza} \mid \text{prezentuje}))$ , R7 =  $\text{exists}([T], \text{scala}(T))$ , R8 =  $\text{not}(\text{exists}([U], \text{nadaje}(U)))$

### 4.11.3. Wnioski

W sekcjach 4.11.1 oraz 4.11.2 przedstawiono możliwości oraz zastosowania narzędzia *Formal Specification IDE* w kontekście weryfikacji dwóch różnych systemów: systemu obsługi pasażerów na lotnisku oraz systemu kontroli wersji kodu. Celem było ukazanie, jak to narzędzie może wpływać na proces projektowania i weryfikacji systemów informatycznych oraz jakie korzyści i wyzwania niesie ze sobą tego typu podejście.

W przeprowadzonych przykładach wykorzystano różnorodność narzędzi weryfikacyjnych (Prover9, SPASS Prover, InKreSAT). Analizę wyników weryfikacji przedstawiono w tabelach 4.1 i 4.2. Każde z tych narzędzi posiada lekko odmienną charakterystykę działania, co umożliwia dopasowanie danego narzędzia do konkretnego przypadku.

Warto podkreślić, że podejście oparte na formalnej weryfikacji może być bardziej kosztowne i czasochłonne niż tradycyjne metody programistyczne. W praktyce, wybór metodyki projektowania i weryfikacji zależy od rodzaju projektowanego systemu. Istnieją systemy, gdzie skrupulatne dowodzenie własności jest niezbędne ze względu na bezpieczeństwo lub niezawodność, takie jak systemy wbudowane czy sterowniki. Z drugiej strony, w systemach, gdzie priorytetem jest szybki rozwój i dostarczanie nowych funkcji, formalna weryfikacja może być mniej istotna, szczególnie jeśli koszt ewentualnych poprawek ma mniejsze znaczenie. W takich przypadkach, skupienie na specyfikacji może być niewielkie.

Niezależnie od konkretnego zastosowania, narzędzie *Formal Specification IDE* wydaje się być wartościowym dodatkiem do procesu projektowania oprogramowania. Może ono wspomagać projektantów i programistów w tworzeniu precyzyjnych specyfikacji, pomagać w wykrywaniu potencjalnych błędów już na etapie projektowania, a także dostarczać narzędzi do formalnej weryfikacji, co znacząco podnosi jakość i niezawodność finalnego produktu.

Proces projektowania i weryfikacji oprogramowania to złożone zagadnienie. Dobór konkretnych narzędzi i metod zależy od konkretnego kontekstu i wymagań projektu. Narzędzie *Formal Specification IDE* stanowi cenne wsparcie w procesie projektowania i weryfikacji systemów informatycznych, a jego wybór i zastosowanie powinno być rozważane w kontekście konkretnej domeny projektu oraz priorytetów związanych z niezawodnością i bezpieczeństwem systemu.



## 5. Implementacja systemu i stos technologiczny

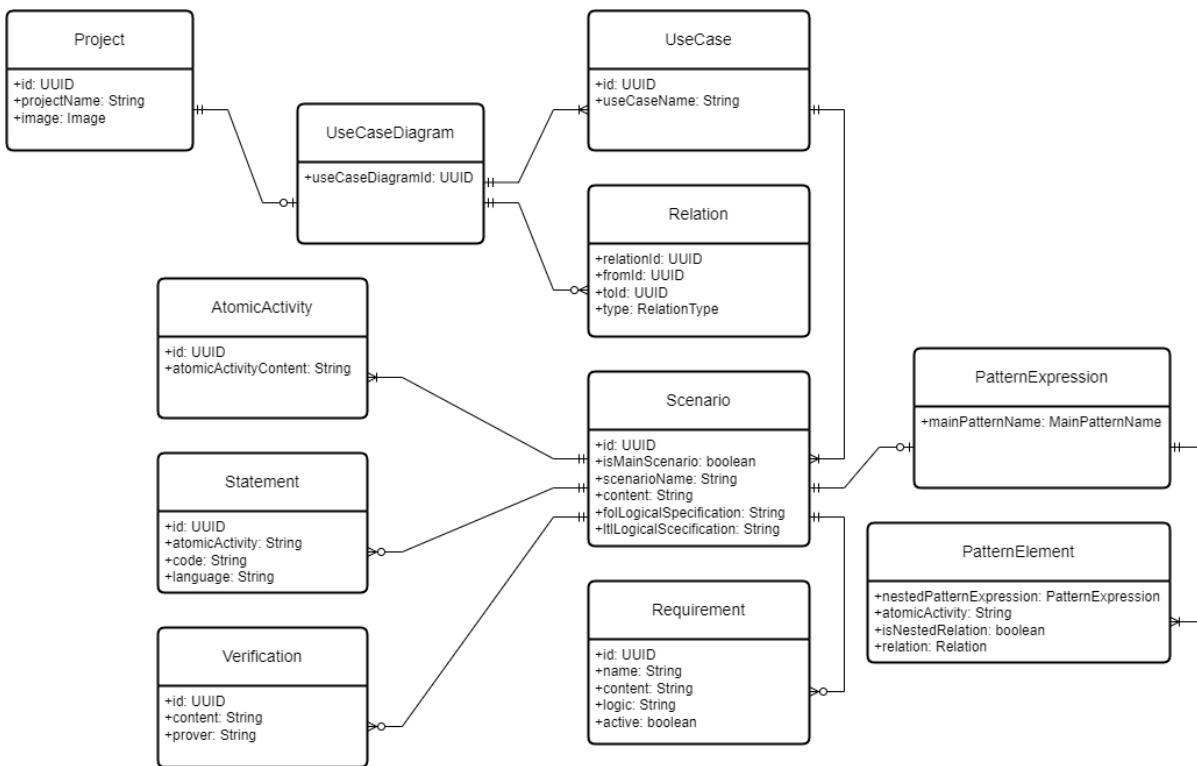
W niniejszym rozdziale przedstawiono realizację przedmiotowego systemu. Określono zastosowaną architekturę systemu, opisano konfigurowanie parametrów podczas korzystania z aplikacji. Zaprezentowano sposób uzyskiwania danych o drogach i dzielnicach wybranego miasta, na podstawie których możliwe jest urealnienie poruszania się patroli. Wymieniono również wykorzystane technologie podczas implementowania programu. Przedstawiono kilka zrzutów ekranu pokazujących wygląd aplikacji.

### 5.1. Encje w systemie

W stworzonym systemie można wyróżnić różne encje, które odzwierciedlają różne aspekty modelowania i weryfikacji inżynierii wymagań. Na rysunku 5.1 przedstawiono diagram związków encji, ukazujący relacje między tymi encjami.

Poniżej opisano główne encje występujące w systemie:

1. **Project** – Reprezentuje projekt tworzony w systemie. Jest jednostką nadzczną dla innych encji. Każdy projekt posiada unikalny identyfikator, nazwę i opcjonalnie może zawierać obraz z diagramem.
2. **UseCaseDiagram** – Encja reprezentująca diagram przypadków użycia. Po zimportowaniu pliku w formacie .xml, system tworzy odpowiedni diagram przypadków użycia, który zawiera powiązane przypadki użycia oraz relacje między nimi.
3. **UseCase** – Przypadek użycia wyodrębniony z diagramu przypadków użycia podczas importu pliku w formacie .xml. Każdy przypadek użycia posiada swoją nazwę oraz jest powiązany z konkretnym diagramem przypadków użycia.
4. **Relation** – Encja reprezentująca relacje między przypadkami użycia, takie jak *include*, *extend* i *inherit*. Relacje te są analizowane podczas importu pliku .xml i są związane z odpowiednimi przypadkami użycia. Każda relacja posiada swój typ oraz źródłowy i docelowy przypadek użycia.



Rys. 5.1. Diagram związków encji (ang. *Entity–relationship model*) – przedstawia związki między encjami. Demonstruje model danych używanych w systemie.

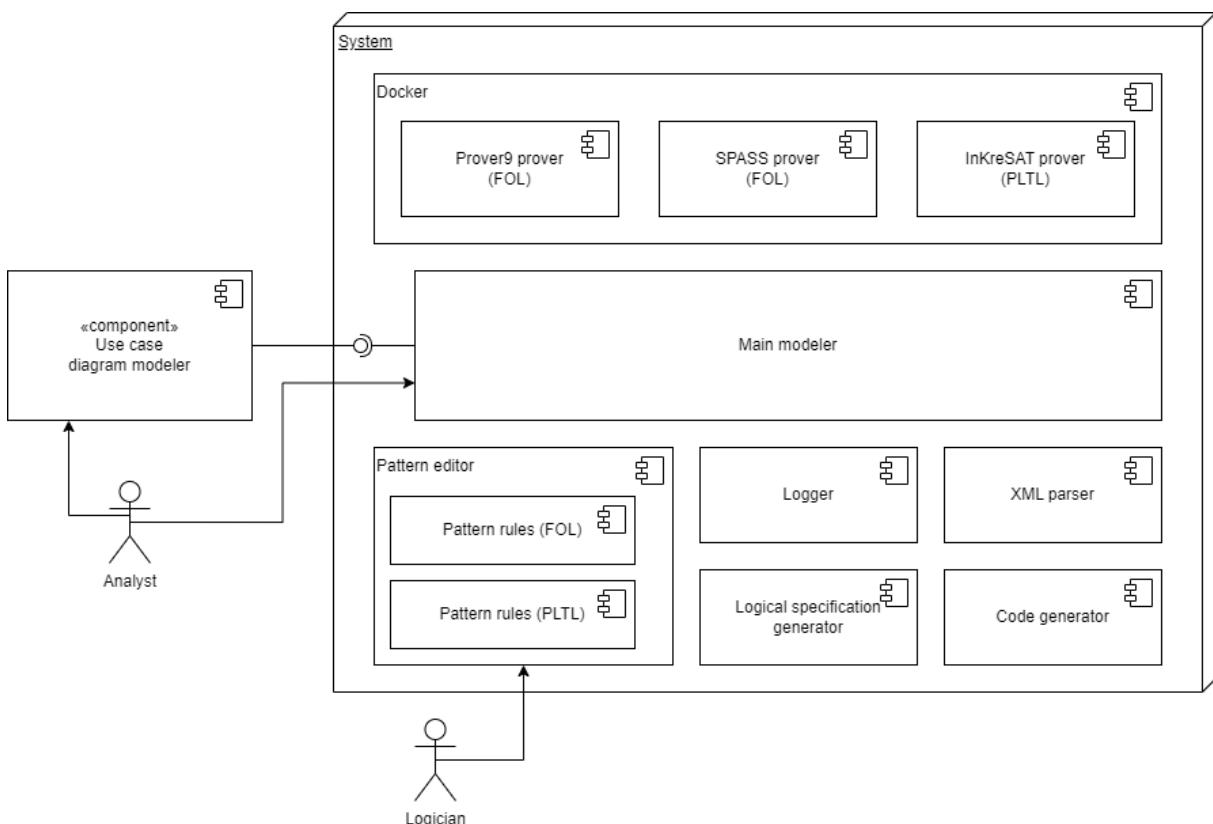
5. **Scenario** – Scenariusz przypadku użycia, który jest tworzony przez użytkownika. Scenariusz składa się z kroków i jest powiązany z konkretnym przypadkiem użycia. Scenariusz może być główny lub alternatywny.
6. **AtomicActivity** – Encja reprezentująca atomiczną aktywność w scenariuszu. Użytkownik może zaznaczać atomiczne aktywności w scenariuszu, które są istotne dla modelu inżynierii wymagań.
7. **PatternExpression** – Wyrażenie wzorcowe tworzone na podstawie diagramu aktywności. Wyrażenie wzorcowe jest przekazywane do generatora formuł logicznych i służy do generowania specyfikacji logicznej.
8. **PatternElement** – Element wzorca używany do budowy diagramu aktywności. Użytkownik może korzystać z predefiniowanych wzorców oraz zaznaczonych atomicznych aktywności.
9. **Statement** – Wyrażenie, kod źródłowy, napisany w jednym z dwóch języków programowania – Java lub Python. Na celu zastąpienie danej aktywności w procesie generacji kodu.

10. **Requirement** – Dodatkowe wymaganie zdefiniowane przez użytkownika. Wymaganie to może zostać połączone z wygenerowanymi formułami logicznymi i poddane formalnej weryfikacji.
11. **Verification** – Encja reprezentująca proces formalnej weryfikacji. Wygenerowane formuły logiczne oraz dodatkowe wymagania mogą być przekazane do zintegrowanych proverów w celu sprawdzenia ich poprawności logicznej.

Dzięki zdefiniowanym encjom system *Formal Specification IDE* umożliwia kompleksową reprezentację i manipulację modelami inżynierii wymagań oraz prowadzenie formalnej weryfikacji.

## 5.2. Komponenty systemu

Komponenty zawarte w stworzonym systemie zostały wizualnie przedstawione na rysunku 5.2.



Rys. 5.2. Diagram komponentów systemu. Przedstawia wizualnie zależności między komponentami systemu oprogramowania.

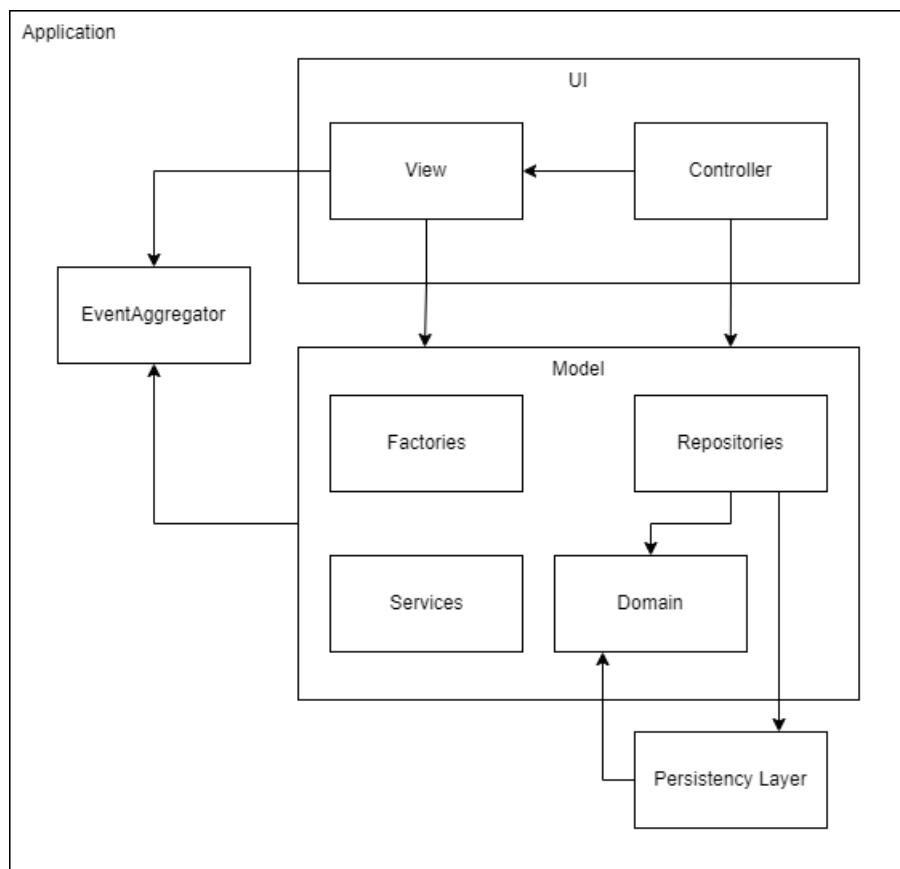
Poniżej opisano główne komponenty systemu:

1. **Use case diagram modeler** – Zewnętrzny program do modelowania diagramów przypadków użycia, który jest importowany do głównego systemu. Ten komponent umożliwia użytkownikom tworzenie i modyfikowanie diagramów przypadków użycia, które są podstawą dla procesu inżynierii wymagań.
2. **Main modeler** – Komponent odpowiedzialny za zarządzanie przypadkami użycia w systemie. Użytkownicy mogą tworzyć, uzupełniać i zarządzać scenariuszami dla poszczególnych przypadków użycia. Ponadto, *main modeler* umożliwia modelowanie diagramów aktywności, które są wykorzystywane do dalszej analizy i generacji specyfikacji logicznej.
3. **XML parser** – Komponent odpowiedzialny za parsowanie plików w formacie .xml, które zawierają diagramy przypadków użycia. Parser analizuje strukturę plików XML i ekstrahuje informacje dotyczące przypadków użycia oraz ich relacji. Te dane są następnie przekazywane do systemu w celu dalszej obróbki i wykorzystania.
4. **Pattern editor** – Komponent umożliwiający edycję plików w formacie .json, które zawierają reguły dla formuł logiki pierwszego rzędu (FOL) oraz linearnej logiki temporalnej (LTL). Użytkownicy mogą dostosować reguły zgodnie z ich potrzebami i preferencjami, co wpływa na generację specyfikacji logicznej.
5. **Logical specification generator** – Komponent odpowiedzialny za generowanie specyfikacji logicznej na podstawie wyrażeń wzorcowych zdefiniowanych w diagramach aktywności. Na podstawie reguł logicznych, ten komponent generuje formalne specyfikacje logiczne, które mogą być dalej poddane procesowi weryfikacji.
6. **Code generator** – Komponent odpowiedzialny za generację kodu na podstawie wyrażenie wzorcowego. Ten komponent generuje kod źródłowy w jednym z dwóch języków programowania: Java lub Python. Wygenerowany kod stanowi implementację opartą na zdefiniowanych wymaganiach.
7. **Docker** – Platforma Docker jest wykorzystywana do komunikacji z proverami. Proverys są narzędziami służącymi do przeprowadzania formalnej weryfikacji specyfikacji logicznych. Docker umożliwia uruchomienie proverów w izolowanym środowisku kontenerowym, co zapewnia bezpieczeństwo i niezależność działania systemu.
8. **Logger** – Komponent odpowiedzialny za logowanie zdarzeń w systemie. Logger rejestruje różnego rodzaju operacje, działania użytkowników oraz błędy lub ostrzeżenia, które mogą wystąpić podczas korzystania z systemu. Rejestrowane informacje są przydatne do analizy, monitorowania i rozwiązywania problemów w systemie.

Dzięki zdefiniowanym komponentom system *Formal Specification IDE* umożliwia kompleksowe zarządzanie, modelowanie i weryfikację inżynierii wymagań w procesie tworzenia oprogramowania.

### 5.3. Architektura systemu

Architektura stworzonego systemu została przedstawiona wizualnie na rysunku 5.3.



Rys. 5.3. Diagram architektury systemu. Wizualna reprezentacja implementacji systemu oprogramowania.

1. **Model** – Warstwa modelu zawiera wszystkie byty aplikacji, takie jak Project, UseCase-Diagram, UseCase, Scenario, Relation, PatternExpression, PatternElement, AtomicActivity, Statement, Requirement, Verification. Model oparty jest na wiedzy dziedzinowej i posiada niezbędną informację na temat obiektów potrzebnych do tworzenia funkcjonalności systemu.
2. **Services** – Warstwa usług zawiera różne serwisy, które wykonują określone zadania w systemie. Te serwisy to:

- **LoggerService** – Serwis do logowania zdarzeń i akcji w celu debugowania i odnajdywania błędów w systemie.
- **XMLParserService** – Serwis odpowiedzialny za parsowanie plików XML, z których wyodrębniane są przypadku użycia, potrzebne do konstrukcji scenariuszy w języku naturalnym.
- **ResourceService** – Serwis odpowiedzialny za odczyt plików źródłowych, które są wykorzystywane w interfejsie użytkownika do wyświetlania wiadomości. Dzięki temu mechanizmowi możliwa jest łatwa zmiana tych wiadomości na inną lokalizację lub język.
- **DockerService** – Serwis odpowiedzialny za komunikację z platformą Docker. Wykorzystywany do zarządzania kontenerami i uruchamiania środowiska izolowanego dla procesu weryfikacji.
- **LogicalSpecificationGenerationService** – Serwis generujący specyfikację logiczną na podstawie wyrażeń wzorcowych. Przetwarza informacje z modelu i tworzy logiczne specyfikacje wymagań, które są używane do weryfikacji.
- **CodeGenerationService** – Serwis generujący kod na podstawie wyrażeń wzorcowych. Przetwarza wyrażenia i tworzy kod źródłowy w wybranym języku programowania, który implementuje określone funkcje systemu.

3. **UI** – Warstwa interfejsu użytkownika oparta na wzorcu MVP (ang. *Model-View-Presenter*). W tej warstwie wykorzystywana jest biblioteka JavaFX do tworzenia graficznego interfejsu użytkownika oraz modelowania diagramów.
4. **View** – Warstwa widoku odpowiedzialna za prezentację danych użytkownikowi. Wzorzec MVP zapewnia oddzielenie warstwy prezentacji od logiki biznesowej.
5. **Controller** – Warstwa kontrolera obsługująca interakcje użytkownika i przekazującą żądania do odpowiednich serwisów w celu wykonania odpowiednich operacji.
6. **Persistency Layer** – Warstwa odpowiadająca za serializację i deserializację danych. Relacje rodzic-dziecko są zapisywane i odtwarzane dzięki mechanizmom serializacji. Pozwala to na zapisywanie stanu pracy oraz umożliwia rozbudowę aplikacji bez utraty dotychczasowej funkcjonalności.
7. **Repositories** – Warstwa repozytoriów służy do oddzielenia logiki biznesowej od warstwy danych. Repozytoria zarządzają operacjami związanymi z dostępem do danych.
8. **Factories** – Warstwa fabryk służy do tworzenia obiektów związanych z konkretnymi interfejsami. Zapewnia to jednolitość w tworzeniu obiektów i ich podstawowej struktury.

9. **EventAggregator** – Komponent centralizujący obsługę wszystkich akcji i zdarzeń w systemie. Jest odpowiedzialny za zarządzanie przepływem informacji o występujących zdarzeniach w różnych częściach systemu.

Zastosowana architektura umożliwia kompleksowe zarządzanie, modelowanie i weryfikację inżynierii wymagań, zapewniając elastyczność, rozszerzalność i separację poszczególnych warstw aplikacji.

## 5.4. Algorytm generowania specyfikacji logicznej

Algorytm generowania specyfikacji logicznej został przedstawiony w artykule [1]. W ramach opracowanego systemu, omawiany algorytm został zaimplementowany.

Generowanie specyfikacji logicznej polega na przekształceniu wyrażenia wzorcowego na docelową specyfikację logiczną. Specyfikacja logiczna (ang. *software model logical specification*) składa się z formuł, które są wynikiem przetwarzania wyrażenia wzorcowego przy użyciu tego algorytmu.

Algorytm generowania specyfikacji logicznej przyjmuje dwa wejścia. Pierwszym z nich jest wyrażenie wzorcowe, które jest zmienną dla modelu przepływu pracy. Drugim wejściem jest predefiniowany zestaw reguł, który jest stały dla pewnej klasy modeli oprogramowania. Wyjściem algorytmu jest wygenerowana specyfikacja logiczna, czyli zbiór formuł logicznych.

### Algorytm 5.1. Implementacja algorytmu generacji specyfikacji logicznej

```
1 public static String generateLogicalSpecifications(String patternExpression
, List<WorkflowPatternTemplate> patternPropertySet) throws Exception {
2     List<String> logicalSpecification = new ArrayList<>();
3     String labelledExpression = LabellingPatternExpressions.
4         labelExpressions(patternExpression);
5     int highestLabelNumber = getHighestLabel(labelledExpression);
6     for (int l = highestLabelNumber; l > 0; l--) {
7         int c = 1;
8         WorkflowPattern pat = getPat(labelledExpression, l, c,
9             patternPropertySet);
10        while (pat != null) {
11            List<String> L2 = pat.getWorkflowPatternFilledRules();
12            L2.remove(0);
13            L2.remove(0);
14            for (String arg : pat.getPatternArguments()) {
15                if (WorkflowPattern.isNotAtomic(arg)) {
```

```

14         String cons = CalculatingConsolidatedExpression.
15             generateConsolidatedExpression(arg, "ini",
16                 patternPropertySet) + " | " +
17             CalculatingConsolidatedExpression.
18                 generateConsolidatedExpression(arg, "fin",
19                     patternPropertySet);
20
21     }
22 }
23 c++;
24 logicalSpecification.addAll(L2);
25 pat = getPat(labelledExpression, l, c, patternPropertySet);
26 }
27 }
28
29 Set<String> set = new HashSet<>(logicalSpecification);
30 logicalSpecification.clear();
31 logicalSpecification.addAll(set);
32 String connectedString = "";
33 System.out.println("\nWynik: ");
34 for (String lValue : logicalSpecification) {
35     connectedString = connectedString + lValue + ", ";
36     System.out.println(lValue);
37 }
38 return connectedString;
39 }
```

Opis algorytmu:

1. Algorytm przyjmuje jako wejście wyrażenie wzorcowe *patternExpression* oraz listę reguł wzorca *patternPropertySet*.
2. Tworzy pustą listę *logicalSpecification*, która będzie przechowywać specyfikację logiczną.
3. Wykonuje etykietowanie wyrażenia wzorcowego przy użyciu funkcji *LabellingPatternExpressions.labelExpressions*, co daje wynik *labelledExpression*.
4. Znajduje najwyższy numer etykiety w *labelledExpression* przy użyciu funkcji *getHighestLabel*.

5. Rozpoczyna iterację od najwyższej etykiety w dół, zmienna  $l$  przechowuje numer etykiety.
6. Wewnątrz pętli, ustawia zmienną  $c$  na 1. Odpowiada ona obecnemu wzorcowi dla etykiety  $l$ , w celu otrzymania pierwszego (od lewej) wzorca dla etykiety  $l$ .
7. Pobiera wzorzec  $pat$  dla etykiety  $l$  i numeru  $c$  przy użyciu funkcji  $getPat()$ .
8. Dopóki wzorzec  $pat$  nie jest pusty, wykonuje poniższe kroki:
  - (a) Pobiera listę reguł wzorca  $L2$  dla  $pat$ .
  - (b) Usuwa dwa pierwsze elementy z listy  $L2$ .
  - (c) Iteruje przez argumenty wzorca  $pat$ .
  - (d) Jeśli argument wzorca nie jest atomowy, generuje skonsolidowane wyrażenie za pomocą funkcji *CalculatingConsolidatedExpression.generateConsolidatedExpression*, podając argument *ini* oraz *patternPropertySet*, i łączy je z wyrażeniem dla *fin*.
  - (e) Tworzy nową listę  $L2\_cons$  i dla każdego wyniku w  $L2$ , zamienia argument na skonsolidowane wyrażenie i dodaje go do  $L2\_cons$ .
  - (f) Przypisuje  $L2\_cons$  do  $L2$ .
  - (g) Zwiększa zmienną  $c$  o 1.
  - (h) Dodaje wszystkie elementy z listy  $L2$  do *logicalSpecification*.
  - (i) Pobiera kolejny wzorzec  $pat$  dla etykiety  $l$  i numeru  $c$  przy użyciu funkcji  $getPat()$ .
9. Po zakończeniu pętli, usuwa duplikaty z listy *logicalSpecification*, pozostawiając tylko unikalne formuły logiczne.
10. Czyści listę *logicalSpecification* i dodaje elementy ze zbioru *set* z powrotem do niej.
11. Inicjalizuje pusty łańcuch znaków *connectedString*.
12. Iteruje przez elementy w *logicalSpecification*, dodając je do *connectedString* oraz wyświetlaając je na standardowym wyjściu.
13. Zwraca *connectedString*, który zawiera specyfikację logiczną.

Funkcja *getPat()* zwraca następny wzorzec z wyrażenia wzorcowego, z etykietą  $l$ , na  $c$ -tej pozycji od lewej strony. Przykładowo, dla wyrażenia wzorcowego 2.5 funkcja *getPat(w,3,2)* zwróci *ConcurRe(e,f,g)*.  $L2$  jest zmienną przechowującą specyfikację dla konkretnego wzorca.

Jeśli wszystkie argumenty wzorca są atomiczne, wtedy  $L2$  jest dołączana do docelowej specyfikacji bez żadnych modyfikacji. W przeciwnym razie potrzebna jest przynajmniej jedna modyfikacja w wierszu 14 algorytmu 5.1. Zawarta jest tam alternatywa skonsolidowanych formuł *ini* i *fin*. Formuły *ini* i *fin* reprezentują widok wzorca z zewnątrz, stąd alternatywna reprezentacja jest najbardziej ogólną reprezentacją wzorca. Logiczna alternatywna formuł *ini* i *fin* pozwala przedstawić wzorzec w generalny sposób, przy użyciu zewnętrznej perspektywy. Bez rozważania wewnętrznego zachowania wzorca. Tylko formuły *ini* i *fin* są widoczne z zewnątrz.

Docelowa specyfikacja logiczna może być uznana za sumę specyfikacji uzyskanych dla poszczególnych wzorców, tj.  $L = L_1 \cup L_2 \cup \dots$

Algorytm generowania specyfikacji logicznej używa dwóch pomocniczych algorytmów, pochodzących z artykułu [1], zaimplementowanych w przedmiotowym systemie.

#### 5.4.1. Etykietowanie wyrażenia wzorcowego

W artykule [1] przedstawiono algorytm etykietowania wyrażeń wzorcowych, który został zaimplementowany i wykorzystany w opracowanym systemie do generacji specyfikacji logicznej.

Algorytm ma na celu wprowadzenie etykiet numerycznych do wyrażeń wzorcowych.

Wejściem do algorytmu jest wyrażenie wzorcowe, które jest zmienną dla modelu przepływu pracy. Wyjściem algorytmu jest etykietowane wyrażenie wzorcowe.

**Algorytm 5.2.** Implementacja algorytmu etykietowania wyrażenia wzorcowego

```

1 public static String labelExpressions(String expression) {
2     StringBuilder labelledExpression = new StringBuilder();
3     int labelNumber = 0;
4     for (char c : expression.toCharArray()) {
5         if (c == '(') {
6             labelNumber++;
7             labelledExpression.append(" (").append(labelNumber).append(" ] "));
8         } else if (c == ')') {
9             labelledExpression.append(" [ ").append(labelNumber).append(" ) ");
10            labelNumber--;
11        } else {
12            labelledExpression.append(c);
13        }
14    }
15    return labelledExpression.toString();
16 }
```

Opis algorytmu:

1. Zmienna *labelledExpression* jest inicjalizowana jako pusty łańcuch znaków.
  2. Zmienna *labelNumber* jest inicjalizowana jako 0.
  3. Dla każdego znaku *c* w wyrażeniu wejściowym:
    - (a) Jeśli *c* jest lewym nawiasem okrągłym '(', zostają wykonane kroki 3b-c.
    - (b) Wartość *labelNumber* jest zwiększana o 1.
    - (c) Do *labelledExpression* jest dołączany łańcuch znaków "(" + *labelNumber* + ")".
    - (d) W przeciwnym przypadku, jeśli *c* jest prawym nawiasem okrągłym ')', wykonaj kroki 3e-f.
    - (e) Do *labelledExpression* dołączany jest łańcuch znaków "[" + *labelNumber* + "]".
    - (f) Wartość *labelNumber* jest zmniejszana o 1.
    - (g) W przeciwnym przypadku, czyli gdy *c* nie jest nawiasem, zostaje wykonany krok 3h.
    - (h) *c* jest dołączane do *labelledExpression*.

Zaimplementowany algorytm jest używany podczas generacji specyfikacji logicznej.

#### **5.4.2. Obliczanie skonsolidowanego wyrażenia**

W artykule [1] przedstawiono algorytm obliczania skonsolidowanego wyrażenia, który został zaimplementowany i wykorzystany w opracowanym systemie do generacji specyfikacji logicznej.

Algorytm polega obliczeniu zbioru argumentów należących do wyrażenia *ini*- lub *fn*-.

Wejściem do algorytmu jest wyrażenie wzorcowe, które jest zmienne dla modelu przepływu pracy, typ wyrażania: *ini* lub *fin* oraz predefiniowany zestaw reguł, który jest stały dla pewnej klasy modeli oprogramowania. Wyjściem algorytmu jest skonsolidowane wyrażenie.

**Algorytm 5.3.** Implementacja algorytmu obliczania skonsolidowanego wyrażenia

```
1 public static String generateConsolidatedExpression(String
2     patternExpression, String type, List<WorkflowPatternTemplate>
3     patternPropertySet) throws Exception {
4
5     if (!type.equals("ini") && !type.equals("fin")) throw new Exception(
6         "type must equal 'ini' or 'fin'!");
7
8     String ex = "";
9 }
```

```

6   WorkflowPattern workflowPattern = WorkflowPattern.
7       getWorkflowPatternFromExpression(patternExpression,
8           patternPropertySet);
9
10  List<String> rulesWithAtomicActivities = workflowPattern.
11      getWorkflowPatternFilledRules();
12
13  String ini = rulesWithAtomicActivities.get(0);
14  String fin = rulesWithAtomicActivities.get(1);
15  rulesWithAtomicActivities.remove(0);
16  rulesWithAtomicActivities.remove(0);
17
18
19  List<String> expressionArguments = WorkflowPattern.
20      extractArgumentsFromLabelledExpression(patternExpression,
21          patternPropertySet);
22
23  for (String argument : expressionArguments) {
24      if (WorkflowPattern.isNotAtomic(argument)) {
25          String innerConsolidatedExpression =
26              generateConsolidatedExpression(argument, type,
27                  patternPropertySet);
28          ex = ex.replace(argument, innerConsolidatedExpression);
29      }
30  }
31
32  return ex;
33}

```

Opis algorytmu:

1. Sprawdzana jest wartość zmiennej *type*. Może równać się *ini* lub *fin*.
2. Zmienna *ex* jest inicjalizowana jako pusty ciąg znaków.
3. Tworzony jest obiekt *workflowPattern* na podstawie wyrażenia wzorcowego *patternExpression* i zestawu właściwości wzorców *patternPropertySet*.
4. Pobierana jest lista reguł wzorca z atomicznymi aktywnościami *rulesWithAtomicActivities* z obiektu *workflowPattern*.
5. Przypisywana jest pierwsza reguła do zmiennej *ini*, a druga reguła do zmiennej *fin*.
6. Usuwane są pierwsze dwie reguły z listy *rulesWithAtomicActivities*.

7. Jeśli *type* jest równy „ini”, wartość *ini* przypisywana jest do zmiennej *ex*. W przeciwnym przypadku, wartość *fin* przypisywana jest do *ex*.
8. Pobierana jest lista argumentów wyrażenia wzorcowego za pomocą funkcji *extractArgumentsFromLabelledExpression(patternExpression, patternPropertySet)*.
9. Dla każdego argumentu *argument* z listy *expressionArguments*:
  - (a) Jeśli argument nie jest atomowy, zostają wykonane kroki 9b-c.
  - (b) Skonsolidowane wyrażenie wzorcowe *innerConsolidatedExpression* jest generowane dla argumentu rekurencyjnie, wywołując funkcję *generateConsolidatedExpression(argument, type, patternPropertySet)*.
  - (c) Zamianie jest wystąpienie argumentu w *ex* na skonsolidowane wyrażenie wzorcowe *innerConsolidatedExpression*.
10. Zwracana jest zmienna *ex* jako wynik.

Algorytm przetwarza wyrażenie wzorcowe, uwzględniając typ „ini” lub „fin”, generując skonsolidowane wyrażenie wzorcowe. Rekurencyjnie przetwarza argumenty wyrażenia wzorcowego, zamieniając nieatomowe argumenty na ich skonsolidowane odpowiedniki. Zaimplementowany algorytm jest używany podczas generacji specyfikacji logicznej.

## 5.5. Reguły logiczne dla wzorców

W celu umożliwienia generowania specyfikacji logicznej w systemie *Formal Specification IDE*, zaimplementowano reguły logiczne dla wzorców. Reguły te określają właściwości i zachowanie poszczególnych wzorców przepływu pracy. Przykładowy zbiór reguł logicznych dla wzorców został zaprezentowany przez formuły 2.1 i 2.2, na podstawie artykułu [1] i ilustrowany na rysunkach 2.9 i 2.10. W celu umożliwienia łatwego edytowania i wykorzystania tych reguł w systemie, zostały one przepisane do bardziej czytelnego formatu i dostosowane do logiki pierwszego rzędu, gdyż zaprezentowane reguły w artykule [1] zostały określone w logice temporalnej.

Reguły logiczne dla wzorców są przechowywane w plikach *pattern\_rules\_FOL.json* i *pattern\_rules\_LTL.json*, znajdujących się w folderze *pattern\_rules* w głównym katalogu projektu. Pliki te zawierają zbiór reguł dla poszczególnych wzorców, które mogą być dowolnie edytowane, zachowując strukturę pliku.

Przykładowe reguły dla wybranych wzorców w kontekście liniowej logiki temporalnej (*pattern\_rules\_LTL.json*) przedstawiają się następująco:

```
{
  "Seq": {
    "number of args": 2,
    "rules": [
      "arg0",
      "arg1",
      "Exist(arg0)",
      "ForAll(arg0 => Exist(arg1))",
      "ForAll(~(arg0 ^ arg1))"
    ]
  },
  "Concur": {
    "number of args": 3,
    "rules": [
      "arg0",
      "arg1 | arg2",
      "Exist(arg0)",
      "ForAll((arg0) => (Exist(arg1) ^ Exist(arg2)))",
      "ForAll(~((arg0) ^ (arg1)))",
      "ForAll(~((arg0) ^ (arg2)))"
    ]
  },
  // Pozostałe wzorce...
}
```

Przykładowe reguły dla wybranych wzorców w kontekście logiki pierwszego rzędu (*pattern\_rules\_FOL.json*) przedstawiają się następująco:

```
{
  "Seq": {
    "number of args": 2,
    "rules": [
      "arg0",
      "arg1",
      "exists([T], arg0(T))",
      "forall([T], implies(arg0(T), exists([U], arg1(U))))",
      "forall([T], not(and(arg0(T), arg1(T))))"
    ]
  },
  "Concur": {
    "number of args": 3,
    "rules": [
      "arg0",
      "arg1 | arg2",
      "exists([U], arg0(U))",
      "forall([U], implies(arg0(U), exists([V], arg1(V))))",
      "forall([U], not(and(arg0(U), arg1(U))))"
    ]
  }
}
```

```
"forall([U], implies(arg0(U), and(exists([V], arg1(V)), exists([W],
    arg2(W))))",
"forall([U], not(and(arg0(U), arg1(U))))",
"forall([U], not(and(arg0(U), arg2(U))))"
]
},
// Pozostałe wzorce...
}
```

Reguły te definiują właściwości i ograniczenia dla poszczególnych wzorców. Na przykład dla wzorca Seq (sekwencja) w logice pierwszego rzędu istnieje reguła `exists([T], arg0(T))`, która mówi, że istnieje pewien element T, dla którego warunek `arg0(T)` jest prawdziwy. Innym przykładem jest reguła `forall([T], implies(arg0(T), exists([U], arg1(U))))`, która mówi, że dla każdego elementu T, jeśli warunek `arg0(T)` jest prawdziwy, to istnieje element U, dla którego warunek `arg1(U)` jest prawdziwy. Reguły te umożliwiają modelowanie i weryfikację zachowań wzorców w sposób logiczny i formalny.

Dostęp do reguł dla wzorców pozwala na elastyczne dostosowanie ich zachowania i właściwości do konkretnych wymagań i specyfikacji projektu. Użytkownik może edytować te reguły, uwzględniając dodatkowe warunki i ograniczenia, co przyczynia się do dokładniejszej analizy i weryfikacji stworzonych modeli. Dzięki temu system *Formal Specification IDE* zapewnia większą precyzję i dokładność w procesie modelowania i weryfikacji systemów opartych na wzorcach przepływu pracy.

Zastosowany format reguł w kontekście logiki LTL umożliwia łatwą translację do formatu wejściowego akceptowanego przez program InKreSAT, który jest systemem dowodzenia twierdzeń dla logiki LTL. Szczegóły dotyczące systemu InKreSAT zostały opisane w sekcji 2.4.4.

W pliku z regułami w kontekście logiki FOL zastosowano format bardzo zbliżony do formatu SPASS Prover. Jest on jednym z dwóch zastosowanych proverów dla logiki pierwszego rzędu. Drugim jest Prover9, którego wejście tworzone jest za pomocą wbudowanej w SPASS Prover funkcjonalności konwersji do formatu akceptowanego przez Prover9. Szczegółowe charakterystyki obu systemów zostały opisane w sekcji 2.4.2 oraz 2.4.3.

## 5.6. Integracja z systemami dowodzenia twierdzeń

W ramach aplikacji *Formal Specification IDE* przeprowadzono integrację z istniejącymi i zaimplementowanymi systemami dowodzenia twierdzeń. Wybranymi systemami są Prover9, SPASS Prover i InKreSAT, które służą do dowodzenia twierdzeń w różnych logikach. Integracja ta umożliwia użytkownikom aplikacji przeprowadzanie formalnej weryfikacji wygenerowanych

formuł logicznych. W tej sekcji opisano charakterystyki tych systemów, a także przedstawiono metodologię integracji. Charakterystyki wybranych systemów opisano w sekcji 2.4.

Prover9 jest systemem automatycznego dowodzenia twierdzeń opartym na logice pierwszego rzędu (ang. *First-Order Logic* – FOL). System ten został zaimplementowany w języku C i oferuje zaawansowane narzędzia do dowodzenia automatycznego. Prover9 obsługuje dowodzenie w logice pierwszego rzędu z równościami, a także operuje na formułach logicznych z kwantyfikatorami, relacjami i funkcjami. System ten jest wykorzystywany do dowodzenia twierdzeń zapisanych w postaci logiki FOL, co umożliwia sprawdzenie poprawności logicznej stworzonych modeli i wymagań.

SPASS Prover jest innym popularnym systemem automatycznego dowodzenia twierdzeń w logice pierwszego rzędu (FOL). Podobnie jak Prover9, SPASS Prover umożliwia dowodzenie automatyczne na podstawie zdefiniowanych formuł logicznych. System ten jest również zaimplementowany w języku C i oferuje wszechstronne funkcje dowodzenia w logice FOL. SPASS Prover wspiera pełne rachunki rezolucji oraz inne strategie dowodzenia, takie jak superpozycja termów. Dzięki temu użytkownicy mogą przeprowadzać formalne weryfikacje swoich modeli i wymagań, wykorzystując ten system dowodzenia twierdzeń.

InKreSAT to system automatycznego dowodzenia twierdzeń, który skupia się na logice temporalnej (ang. *Linear Temporal Logic* – LTL). Ten system został zaprojektowany specjalnie do analizy i weryfikacji formuł logicznych związanych z czasem. InKreSAT operuje na formułach LTL i oferuje efektywne metody dowodzenia automatycznego. Dzięki temu użytkownicy mogą weryfikować poprawność temporalnych wymagań i właściwości systemów.

### 5.6.1. Konteneryzacja

W celu integracji wybranych systemów dowodzenia twierdzeń z aplikacją *Formal Specification IDE*, zastosowano konteneryzację za pomocą narzędzia Docker. Docker jest popularnym oprogramowaniem służącym do wirtualizacji na poziomie systemu operacyjnego, które umożliwia tworzenie, wdrażanie i uruchamianie aplikacji w izolowanych kontenerach [25]. Kontenery Docker są przenośne i mogą być uruchamiane na różnych systemach, co pozwala na skuteczną integrację systemów działających w różnych środowiskach.

W przypadku wybranych systemów dowodzenia twierdzeń, takich jak Prover9, SPASS Prover i InKreSAT, konteneryzacja pozwoliła na uruchomienie tych systemów w izolowanych kontenerach Docker. Dzięki temu aplikacja *Formal Specification IDE* może komunikować się z tymi systemami i przeprowadzać formalną weryfikację formuł logicznych generowanych przez użytkowników.

W przypadku systemów dowodzenia twierdzeń, które posiadają ograniczenia platformowe – Prover9 i InKreSAT, konieczne było zastosowanie wirtualizacji na poziomie systemu operacyjnego. Oznacza to, że kontenery Docker zostały uruchomione na systemach operacyjnych, które są zgodne z tymi narzędziami. Dla przykładu, Prover9 oryginalnie działa tylko na systemach Unix, dlatego konieczne było uruchomienie kontenera Docker na systemie Unix, aby umożliwić integrację z *Formal Specification IDE*. Podobnie, SPASS Prover działa na systemach Unix, ale istnieją również wersje dostosowane do systemu Windows. Niemniej jednak, dla większej pewności i zgodności, zdecydowano się użyć wersji systemu Unix w kontenerze Docker.

W ramach aplikacji został stworzony serwis o nazwie *DockerService*, który jest odpowiedzialny za zarządzanie kontenerami Docker. Serwis ten umożliwia budowanie obrazów Docker, tworzenie i zarządzanie kontenerami, oraz wykonywanie poleceń wewnętrz kontenerów. W celu osiągnięcia tego celu, wykorzystano bibliotekę docker-java [26].

Podczas inicjalizacji aplikacji, klient Docker jest inicjalizowany poprzez nawiązanie połączenia z serwerem Docker przy użyciu adresu *tcp://localhost:2375*. Następnie, na podstawie pliku Dockerfile, budowany jest obraz Docker. Ten plik definiuje kroki niezbędne do skonfigurowania kontenera, w tym instalację i konfigurację odpowiednich narzędzi i zależności. Po zbudowaniu obrazu, tworzony jest nowy kontener oparty na tym obrazie. Kontener ten jest uruchamiany i udostępnia swoje zasoby (takie jak systemy dowodzenia twierdzeń) dla aplikacji *Formal Specification IDE*.

Systemy dowodzenia twierdzeń są instalowane w kontenerze Docker przy użyciu pliku Dockerfile, który definiuje szczegóły budowy obrazu Docker. Szczegółowe zawartości pliku Dockerfile:

- Używany jest obraz bazowy najnowszej wersji systemu Ubuntu.
- Wykonuje się aktualizacja i instalacja zależności w systemie Ubuntu, w tym narzędzi takich jak *GCC* i *make*, które są potrzebne do budowy programów.
- **Prover9:** Skopiowany zostaje plik tar.gz z kodem źródłowym Prover9 do obrazu Docker w lokalizacji */tmp*. Następnie plik ten jest rozpakowywany do katalogu */opt*. Kopiowany jest również plik Makefile do katalogu */opt/LADR-2009-11A/provers.src/*. W celu skompilowania programu Prover9 konieczne było wprowadzenie pewnych poprawek, które zostały dokładnie opisane w sekcji 5.6.2. Wykonywana jest komplikacja Prover9 oraz powiązanych programów (Mace4).
- **SPASS Prover:** Skopiowany zostaje plik tar.gz z kodem źródłowym SPASS do obrazu Docker w lokalizacji */tmp*. Plik ten jest rozpakowywany do katalogu */opt*. Następnie przeprowadzana jest konfiguracja, komplikacja i instalacja SPASS.

- **InKreSAT:** Skopiowany zostaje plik tar.bz2 z kodem źródłowym InKreSAT do obrazu Docker w lokalizacji `/tmp`. Plik ten jest rozpakowywany do katalogu `/opt`. Następnie instalowane są wymagane zależności, a InKreSAT jest budowany. W celu skompilowania InKreSAT konieczne było wprowadzenie kilku poprawek, które zostały dokładnie opisane w sekcji 5.6.2.
- Definiowany jest współdzielony katalog `/shared`, który można używać do wymiany plików między kontenerem a hostem.

Zastosowanie konteneryzacji za pomocą Docker umożliwiło skutecną integrację systemów dowodzenia twierdzeń z aplikacją *Formal Specification IDE*. Dzięki temu użytkownicy mogą wykorzystywać narzędzia do dowodzenia twierdzeń, takie jak Prover9, SPASS Prover i InKreSAT, bez konieczności instalowania i konfigurowania ich na swoich własnych maszynach.

### 5.6.2. Rozwiążane problemy z instalacją

W trakcie integracji istniejących systemów dowodzenia twierdzeń napotkano kilka problemów związanych z instalacją tych systemów. W tej sekcji opisano rozwiązania tych problemów, które zostały podjęte w ramach stworzonego systemu *Formal Specification IDE*.

Podczas komplikacji Prover9 wystąpił błąd wskazujący na niezdefiniowane odwołanie do funkcji *round* w pliku *search.c*. Problematyczne było brakujące połaczenie biblioteki matematycznej podczas procesu komplikacji, co skutkowało niepowodzeniem komplikacji. Aby rozwiązać ten problem, zmodyfikowano plik Makefile w celu jawnego komplikacji z połączoną biblioteką matematyczną (*-lm*). Dodanie opcji *-lm* do poleceń *gcc* umożliwiło skorzystanie z domyślnych ustawień kompilatora, co skutkowało poprawnym połaczeniem biblioteki i rozwiązaniem problemu. Podczas budowy obrazu Docker, po rozpakowaniu kodu źródłowego Prover9, domyślny plik Makefile jest zastępowany zaktualizowanym, który zawiera wymienione poprawki. Poprawiony plik Makefile znajduje się w folderze `/docker` w głównym katalogu projektu *Formal Specification IDE*.

Proces konfiguracji, komplikacji i instalacji SPASS Prover odbył się bez żadnych problemów. Wystarczyło postępować zgodnie z dołączonymi instrukcjami i nie napotkano nieoczekiwanych trudności.

Największe trudności pojawiły się podczas budowy systemu InKreSAT, który opiera się na MiniSAT - minimalistycznym i wysoce wydajnym SAT Solverze. Ze względu na fakt, że projekt MiniSAT nie był rozwijany od roku 2013, wystąpiły problemy z komplikacją i integracją z InKreSAT. Pierwszy problem pojawił się podczas komplikacji i dotyczył pliku *SolverTypes.h*. Zdefiniowane zaprzyjaźnione funkcje *mkLit* nie miały przypisanych domyślnych wartości argumentów. Aby rozwiązać ten problem, zakomentowano zaprzyjaźnione deklaracje i dodano domyślną wartość dla parametru *sign*:

```

50      -     friend Lit mkLit(Var var, bool sign = false);
50      + // friend Lit mkLit(Var var, bool sign = false);

58      -     inline Lit mkLit      (Var var, bool sign)
58      +     inline Lit mkLit      (Var var, bool sign = false)

```

Następnym problemem, który wystąpił podczas procesu komplikacji projektu InKreSAT, wskazywał na problem z komplikacją i łączeniem pliku *minisat\_if.o*. Aby rozwiązać ten problem, w pliku Makefile dodano opcję *-fpermissive*, która pozwala na obniżenie poziomu narzędzi diagnostycznych dotyczących niezgodnego kodu z poziomu błędów do ostrzeżeń.

```

67      -     g++ -c -I$(MINISATSRC) -I`ocamlc -where`
              -D __STDC_LIMIT_MACROS -D __STDC_FORMAT_MACROS minisat_if.cc
              -o minisat_if.o -fPIC
67      +     g++ -c -I$(MINISATSRC) -I`ocamlc -where`
              -D __STDC_LIMIT_MACROS -D __STDC_FORMAT_MACROS minisat_if.cc
              -o minisat_if.o -fPIC -fpermissive

```

Kolejny problem, który pojawił się podczas komplikacji systemu InKreSAT, dotyczył pliku *template.mk*. Komunikat o błędzie wskazywał na problem z łączeniem pliku *minisat.cmxa* podczas próby utworzenia współdzielonego obiektu. Sugerowało to konieczność ponownej komplikacji MiniSAT z użyciem flagi *-fPIC* (Position Independent Code). Po dokonaniu tej zmiany MiniSAT został skompilowany z flagą *-fPIC*, co rozwiązało problem z łączeniem.

```

44      -     CFLAGS += $(COPTIMIZE) -g -D DEBUG
45      -     CFLAGS += $(COPTIMIZE) -pg -g -D NDEBUG
46      -     CFLAGS += -O0 -g -D DEBUG
47      -     CFLAGS += $(COPTIMIZE) -g -D NDEBUG
44      +     CFLAGS += $(COPTIMIZE) -g -D DEBUG -fpermissive -fPIC
45      +     CFLAGS += $(COPTIMIZE) -pg -g -D NDEBUG -fpermissive -fPIC
46      +     CFLAGS += -O0 -g -D DEBUG -fpermissive -fPIC
47      +     CFLAGS += $(COPTIMIZE) -g -D NDEBUG -fpermissive -fPIC

```

Podsumowując, w ramach pracy magisterskiej udało się skutecznie rozwiązać problemy z instalacją systemów dowodzenia twierdzeń. Poprawki w plikach źródłowych oraz dostosowanie opcji komplikacji pozwoliły na poprawną komplikację i instalację narzędzi Prover9, SPASS Prover oraz InKreSAT. Dzięki temu obraz Docker, który zawiera te narzędzia, może być wykorzystywany w aplikacji *Formal Specification IDE* do udostępnienia zaawansowanych funkcji dowodzenia twierdzeń.

## 5.7. Generacja kodu

Jednym z etapów w systemie *Formal Specification IDE* jest modelowanie diagramów aktywności przy użyciu predefiniowanych wzorców oraz atomicznych aktywności wyodrębnionych z przypadków użycia. W kolejnym etapie, zamodelowany diagram aktywności jest przekształcany na wyrażenie wzorcowe, co zostało opisane w sekcji 2.2.2. Przykładowy diagram aktywności modelowany przez język wzorców został przedstawiony na rysunku 5.4. Stworzone wyrażenie można następnie poddać procesowi generacji kodu źródłowego przy użyciu dedykowanego narzędzia. Proces ten umożliwia programistom tworzenie struktur programowych i generowanie kodu wynikowego, co przyspiesza proces tworzenia oprogramowania. Niniejsza sekcja przedstawia szczegółowy opis stworzonego komponentu systemu, który pozwala na komplikowanie języka wzorców na kod źródłowy programu komputerowego.

W języku wzorców, wzorzec ma następującą postać:

$$W(A_1, \dots, A_n), \quad (5.1)$$

gdzie  $W$  oznacza nazwę wzorca, a  $A_1, \dots, A_n$  jest sekwencją  $n$  wzorców lub atomów. Atom, zwany również atomiczną aktywnością, stanowi najmniejszą jednostkę w modelu i umożliwia opisanie instrukcji lub wyrażeń w języku docelowym. Szczegółowy opis formy wzorców został przedstawiony w artykule [1] (sekcja 2.2).

Stworzone wyrażenie wzorcowe może zostać przekształcone na szkielet programu komputerowego przy użyciu dedykowanego generatora kodu. W generowanym języku programowania, wzorce mogą reprezentować literały, funkcje lub puste wyrażenia. Poniżej przedstawiona jest lista obsługiwanych wzorców wraz z krótkim objaśnieniem ich znaczenia oraz przykładowym kodem w języku Java:

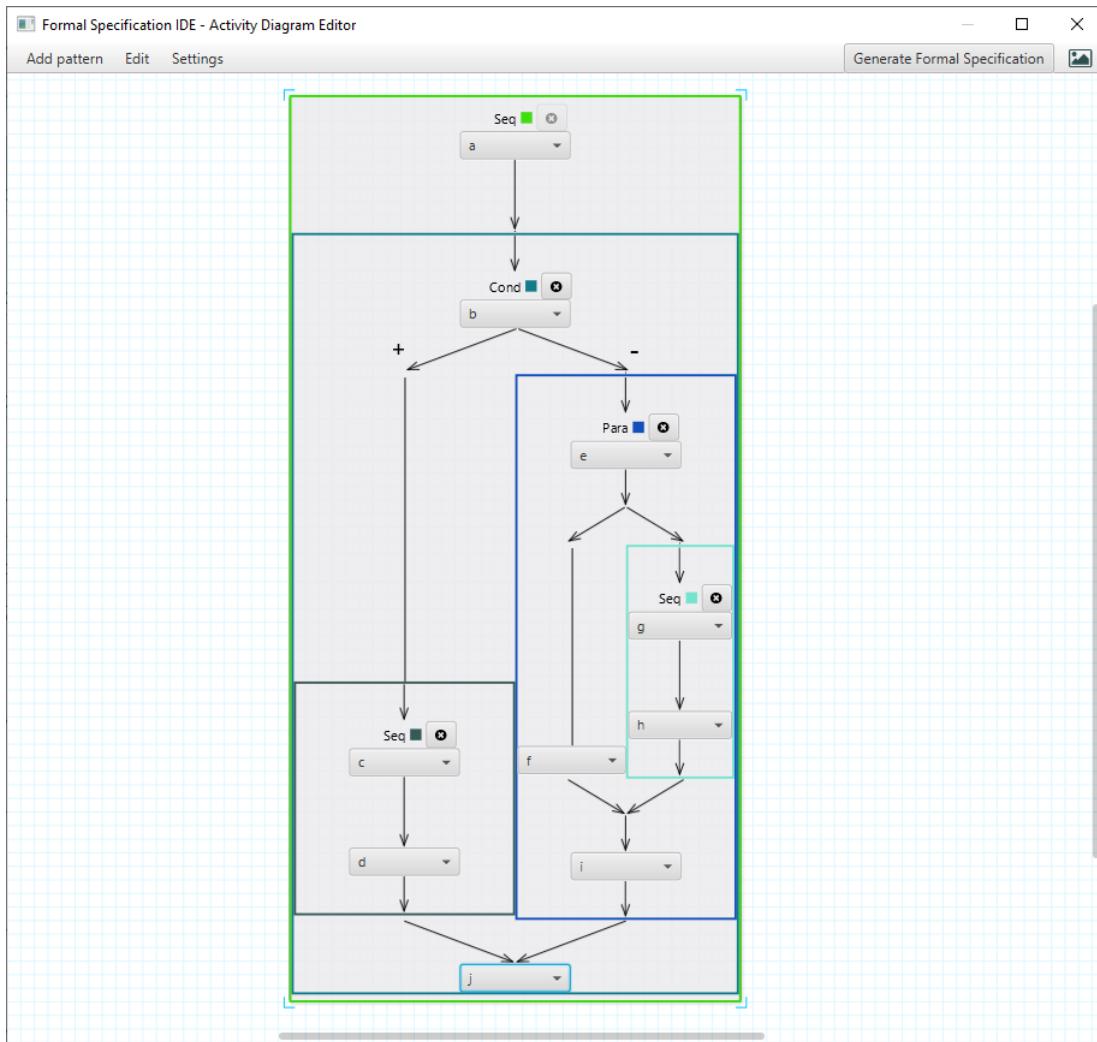
- $\text{Seq}(w_1, w_2)$  – wzorzec modelujący sekwencyjne wykonanie instrukcji wzorca  $w_2$  po  $w_1$ .

```
w1;
w2;
```

- $\text{SeqSeq}(w_1, w_2, w_3)$  – wzorzec modelujący sekwencję trzech instrukcji, równoważny wzorcowi  $\text{Seq}(w_1, \text{Seq}(w_2, w_3))$ .

```
w1;
w2;
w3;
```

- $\text{Alt}(w_1, w_2, w_3)$  wzorzec modelujący decyzję podejmowaną na podstawie wyrażenia opisanego przez  $w_1$ , jeżeli zwraca prawdę, następuje wykonanie  $w_2$ , następnie zawsze wykonywana jest instrukcja opisana przez  $w_3$ .



Rys. 5.4. Przykładowy diagram aktywności modelowany przez język wzorców. Na jego podstawie automatycznie tworzone jest wyrażenie wzorcowe, które służy jako wejście dla generatora kodu.

```
if (w1) {
    w2;
}
w3;
```

- $Branch(w_1, w_2, w_3)$  – wzorzec modelujący konstrukcję instrukcji warunkowej o warunku  $w_1$ , gdzie  $w_2$  jest wykonywany jeśli  $w_1$  zwróci prawdę,  $w_3$  w przeciwnym wypadku.

```
if (w1) {
    w2;
} else {
    w3;
```

```
}
```

- $Concur(w_1, w_2, w_3)$  – wzorzec modelujący wykonanie  $w_1$ , a następnie równolegle wykonanie instrukcji  $w_2$  i  $w_3$ .

```
w1;
Thread thread0 = new Thread(() -> {
    w2;
})
Thread thread1 = new Thread(() -> {
    w3;
})
thread0.start();
thread1.start();
```

- $Cond(w_1, w_2, w_3, w_4)$  – wzorzec modelujący konstrukcję instrukcji warunkowej języka, równoważny  $Seq(Branch(w_1, w_2, w_3), w_4)$ .

```
if(w1) {
    w2;
} else {
    w3;
}
w4;
```

- $Para(w_1, w_2, w_3, w_4)$  – wzorzec modelujący wykonanie  $w_1$ , a następnie równolegle wykonanie instrukcji  $w_2$  i  $w_3$ , a po zakończeniu wykonanie  $w_4$ .

```
w1;
Thread thread0 = new Thread(() -> {
    w2;
})
Thread thread1 = new Thread(() -> {
    w3;
})
thread0.start();
thread1.start();
try {
    thread0.join();
    thread1.join();
}
w4;
```

- $\text{Loop}(w_1, w_2, w_3, w_4)$  – wzorzec modelujący pętlę while. Instrukcja opisana przez  $w_1$  jest wykonywana przed rozpoczęciem iteracji, następnie instrukcja opisana przez  $w_3$  jest wykonywana, dopóki warunek  $w_2$  jest prawdziwy, po zakończeniu iteracji wykonywana jest instrukcja opisywana przez  $w_4$ ,

```
w1;  
while(w2) {  
    w3;  
}  
w4;
```

Na rysunku 5.5 przedstawiono przykład wygenerowanego kodu w języku Java. Atomiczne aktywności, umieszczone na diagramie aktywności, mogą zostać zmapowane w procesie generacji kodu na dowolne funkcje lub instrukcje w języku docelowym.

```
// Generated by QRC  
class Program {  
    public static void main(String[] args) {  
        a  
        if (b) {  
            c  
            d  
        } else {  
            e  
            Thread thread0 = new Thread(() -> {  
                f  
            });  
            Thread thread1 = new Thread(() -> {  
                g  
                h  
            });  
            thread0.start();  
            thread1.start();  
            try {  
                thread0.join();  
                thread1.join();  
            } catch (InterruptedException e) {  
                i  
            }  
            j  
        }  
    }  
}
```

Rys. 5.5. Przykład wygenerowanego kodu w języku Java. Atomiczne aktywności mogą być zmapowane na funkcje lub instrukcje języka docelowego.

Wyrażenia, których nazwy zaczynają się od wielkich liter, oznaczają nazwy wzorców, natomiast te, których pierwsze litery są małe, oznaczają atomy. Jeśli atom jest zakończony nawiasami (), traktowany jest jako funkcja, a odpowiadający mu szkielet również zostaje wygenerowany. Jeśli natomiast potrzebne jest bezpośrednie wpisanie instrukcji do kodu, należy umieścić je w cudzysłowie, na przykład `Seq("int a = 1; ", "a++;")`.

Implementacja generatora kodu została oparta na projekcie autorstwa Stanisława Borowego, Piotra Kubali i Łukasza Łabuza, wykonanym w ramach przedmiotu Teoria Kompilacji i Komplikatory na kierunku Informatyka i Systemy Inteligentne na AGH. Projekt ten, stworzony pod opieką profesora Radosława Klimka, stanowił podstawę dla komponentu w systemie *Formal Specification IDE*.

Generator kodu został zaimplementowany w języku Java, przy wykorzystaniu narzędzia ANTLR [27]. Główną klasą odpowiedzialną za proces komplikacji języka wzorców do szkieletu programu jest klasa `Compiler` z metodą `String compile(String pattern, Language language)`. Obecnie obsługiwane są języki Java i Python.

W celu analizy składniowej wzorców, zdefiniowano formalną gramatykę bezkontekstową języka wzorców. Gramatyka ta została podzielona na dwie części: analizę leksykalną i analizę składniową. Program pomija wszelkie białe znaki, co ułatwia formatowanie kodu dla programistów.

Produkcje dla leksera definiujące tokeny wyglądają następująco:

```
PATTERN_NAME_LEX ::= [A-Z] [A-Za-z]* ;
DELIMITER ::= ',' ;
LEFT_BRACKET ::= '(' ;
RIGHT_BRACKET ::= ')' ;
COLON ::= ':' ;
BOOLEAN ::= 'true' | 'false' ;
EMPTY ::= 'empty' ;
ATOM ::= [a-z] [A-Za-z0-9_]* ;
INTEGER ::= ('+' | '-') ? ('0' | [1-9][0-9]*) ;
FLOATING ::= ('+' | '-') ? ('0' | [1-9][0-9]*) '.' [0-9]+ ;
STRING ::= '\"' '\"' | .)*? '\"' ;
```

Natomiast produkcje dla parsera, które określają składnię wzorców, wyglądają następująco:

```
pattern :=
    patternName LEFT_BRACKET (DELIMITER pattern)* RIGHT_BRACKET
  | method
  | STRING
  | EMPTY
;
patternName :=
    PATTERN_NAME_LEX
;
```

```
method ::=  
    ATOM LEFT_BRACKET (DELIMITER) * RIGHT_BRACKET (COLON STRING)?  
    ;  
parameter ::=  
    INTEGER  
    | FLOATING  
    | BOOLEAN  
    | STRING  
    | method  
    | typedExpr  
    ;  
typedExpr ::=  
    STRING COLON STRING  
    ;
```

Zintegrowany generator kodu w systemie *Formal Specification IDE* umożliwia komplikację wzorców, wprowadzanie fragmentów kodu i otrzymywanie kodu wynikowego. Jest to istotny komponent, który przyczynia się do efektywnego procesu tworzenia oprogramowania.

## 5.8. Wykorzystywane technologie

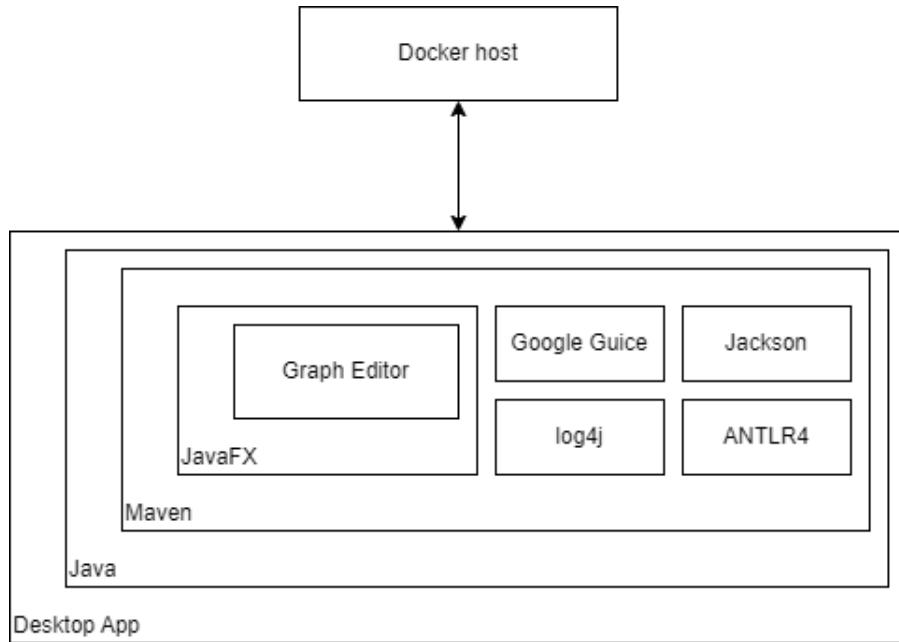
W niniejszej sekcji opisane są technologie, które zostały wykorzystane podczas tworzenia aplikacji *Formal Specification IDE*. Przedstawione technologie zapewniają funkcjonalności związane z konteneryzacją, komplikacją języka wzorców, edytorem diagramów aktywności, interfejsem użytkownika, zapisem pracy w systemie, językiem programowania oraz logami.

### 5.8.1. Konteneryzacja

Jedną z wykorzystanych technologii jest Docker[25], popularne narzędzie do konteneryzacji aplikacji. Docker pozwala na tworzenie izolowanych, niezależnych od siebie kontenerów, w których uruchamiana jest aplikacja wraz z jej zależnościami. Konteneryzacja przy użyciu Dockera zapewnia jednolite i niezawodne środowisko dla aplikacji, co ułatwia przenoszenie aplikacji między różnymi środowiskami.

Dzięki dostosowaniu konteneryzacji możliwe było zintegrowanie systemu *Formal Specification IDE* z istniejącymi systemami dowodzenia twierdzeń, które są potrzebne do przeprowadzenia procesu formalnej weryfikacji.

Dodatkowo, w projekcie wykorzystano bibliotekę docker-java[26], która umożliwia programowe zarządzanie kontenerami Docker z poziomu języka Java. Biblioteka ta dostarcza interfejs do tworzenia, uruchamiania, zarządzania i monitorowania kontenerów Docker, co pozwala na automatyzację procesów związanych z konteneryzacją w aplikacji.



Rys. 5.6. Wykorzystane technologie w aplikacji.

### 5.8.2. Kompilacja języka wzorców do kodu źródłowego

Do komplikacji języka wzorców na kod źródłowy zostało wykorzystane narzędzie ANTLR[27]. ANTLR (ANother Tool for Language Recognition) jest narzędziem do generowania analizatorów składniowych i leksykalnych dla różnych języków programowania. Pozwala ono na zdefiniowanie gramatyki języka oraz generowanie parsera, który potrafi analizować tekst wejściowy zgodnie z tą gramatyką.

ANTLR zostało wykorzystane w projekcie do zdefiniowania formalnej gramatyki języka wzorców. Dzięki temu możliwe jest analizowanie składni wzorców i generowanie odpowiadającego im kodu źródłowego.

### 5.8.3. Edytor diagramów aktywności

W aplikacji *Formal Specification IDE*, edytor diagramów aktywności został stworzony w oparciu o bibliotekę Graph Editor[28]. Graph Editor umożliwia tworzenie, edycję i wizualizację diagramów grafowych w JavaFX. Biblioteka jest wysoce konfigurowalna, co pozwoliło na stworzenie narzędzia do modelowania diagramu aktywności. Dzięki temu narzędziu użytkownicy mogą w prosty sposób tworzyć diagramy aktywności, co ułatwia proces tworzenia i analizy specyfikacji formalnej.

### 5.8.4. Interfejs użytkownika

Aplikacja *Formal Specification IDE* korzysta z framework JavaFX[29] do implementacji interfejsu użytkownika. JavaFX to biblioteka graficzna, która umożliwia tworzenie nowoczesnych i interaktywnych aplikacji desktopowych w języku Java. JavaFX oferuje wiele gotowych komponentów graficznych, takich jak przyciski, pola tekstowe, tabele, które mogą być łatwo dostosowane i stylizowane.

### 5.8.5. Zapis pracy w systemie

Aby umożliwić zapisywanie pracy użytkowników w systemie, w aplikacji *Formal Specification IDE* wykorzystano bibliotekę Jackson. Jackson[30] to biblioteka do przetwarzania danych w formacie JSON w języku Java. Biblioteka ta umożliwia serializację obiektów Java do formatu JSON oraz deserializację danych JSON do obiektów Java.

Dzięki Jacksonowi, aplikacja może zapisywać dane dotyczące specyfikacji formalnej, diagramów aktywności i innych informacji w formacie JSON. To pozwala na łatwe przechowywanie i udostępnianie danych w systemie, a także integrację z innymi narzędziami i systemami, które obsługują format JSON.

### 5.8.6. Język programowania

Aplikacja *Formal Specification IDE* została w całości napisana w języku programowania Java[31]. Java to obiektowy język programowania, który jest powszechnie używany w branży informatycznej. Java oferuje wiele zaawansowanych funkcji, takich jak zarządzanie pamięcią, wielowątkowość i obsługę wyjątków, co sprawia, że jest popularnym wyborem do tworzenia aplikacji desktopowych i systemów o dużej skali.

Podczas tworzenia aplikacji *Formal Specification IDE* wykorzystano JDK (Java Development Kit) w wersji 19. W procesie tworzenia użyto również zintegrowanego środowiska programistycznego IntelliJ IDEA[32] firmy JetBrains, które jest jednym z najpopularniejszych narzędzi do programowania w języku Java.

Do automatyzacji procesu budowania projektu wykorzystano narzędzie Maven[33]. Maven jest narzędziem do zarządzania projektem i budowania oprogramowania. Umożliwia ono definiowanie struktury projektu, zarządzanie zależnościami, budowanie aplikacji oraz wykonywanie innych zadań związanych z cyklem życia projektu.

Dodatkowo, w projekcie wykorzystano bibliotekę Google Guice[34], która jest frameworkm do wstrzykiwania zależności w języku Java. Guice ułatwia tworzenie skalowalnych i modułowych aplikacji poprzez automatyczne rozwiązywanie zależności między komponentami.

### 5.8.7. Tworzenie logów

Do obsługi logowania w aplikacji *Formal Specification IDE* została wykorzystana biblioteka log4j[35]. Log4j jest biblioteką do logowania zdarzeń w języku Java. Pozwala ona na elastyczne konfigurowanie logowania, zarządzanie poziomami logowania, a także zapisywanie logów w różnych formatach (np. plikach tekstowych).

Log4j umożliwia śledzenie działań i zdarzeń w aplikacji, co ułatwia debugowanie i monitorowanie działania programu. Dzięki temu narzędziu, aplikacja może generować logi dotyczące różnych aspektów działania, takich jak błędy, ostrzeżenia czy informacje diagnostyczne, co ułatwia analizę i rozwiązywanie problemów w systemie.

## 6. Podsumowanie

W ramach tej pracy został przedstawiony projekt i implementacja systemu wspierającego budowę i formalną weryfikację modeli inżynierii wymagań. Celem pracy było stworzenie kompleksowego narzędzia opartego na znanych metodach i algorytmach, które umożliwiałyby skuteczną analizę i weryfikację modeli behawioralnych, składających się z diagramów przypadków użycia, scenariuszy i diagramów aktywności.

W rozdziale 1 przedstawiono strukturę pracy, określając kolejne rozdziały i zawartość. Wprowadzenie zawierało motywację do stworzenia systemu *Formal Specification IDE* oraz cele i założenia pracy.

Rozdział 2 skupiał się na przeglądzie istniejących rozwiązań związanych z inżynierią wymagań. Omówiono wzorce przepływu pracy, modelowanie diagramu przypadków użycia oraz systemy dowodzenia twierdzeń. Przedstawiono istniejące metody i narzędzia, które mogą być wykorzystane w tworzonym systemie.

Rozdział 3 zawiera opis zastosowanego rozwiązania oraz przyjętej metodologii, która składa się z kilku etapów: modelowanie diagramów przypadków użycia, tworzenie scenariuszy przypadków użycia, modelowanie diagramów aktywności, generowanie specyfikacji logicznej, generowanie kodu oraz formalna weryfikacja formuł logicznych. Każdy z tych etapów ma na celu zapewnienie precyzyjnego i jednoznacznego opisu wymagań oraz możliwość ich weryfikacji pod kątem poprawności logicznej.

Rozdział 4 przedstawia wszystkie elementy stworzonego systemu. Zaprezentowano również pełen przykład zastosowania systemu. Opisano kroki budowy modelu inżynierii wymagań oraz przeprowadzono weryfikację za pomocą dedukcji logicznej.

Rozdział 5 poświęcono na implementację stworzonego systemu. Przedstawiono architekturę systemu, opisano poszczególne komponenty oraz szczegóły implementacyjne, takie jak algorytmy i integracje z systemami dowodzenia twierdzeń. Zawarto również informacje na temat użytych technologii.

Praca ta stanowi istotny wkład w dziedzinę inżynierii wymagań, dostarczając narzędzia i metody umożliwiające efektywną budowę i formalną weryfikację modeli behawioralnych. Stworzony system może znaleźć zastosowanie w praktyce, wspierając proces projektowania oprogramowania i analizy wymagań.

## 6.1. Metryka projektu

W celu oceny skali i złożoności stworzonego projektu, przeprowadzono analizę metryczną. Poniżej przedstawione są wybrane metryki, które dostarczają informacji o liczbie plików, linii kodu i innych aspektach projektu.

Projekt składa się z następującej liczby plików:

- Pliki źródłowe Java: 138 plików
- Pliki .fxml (JavaFX): 18 plików
- Pliki konfiguracyjne: 3 pliki

Projekt składa się z następującej liczby linii kodu:

- Pliki źródłowe Java: 16,499 linii kodu
- Pliki .fxml (JavaFX): 370 linii kodu

Projekt składa się z 8 pakietów, w tym m.in. modelu, serwisów i interfejsu użytkownika (UI).

Analiza metryczna projektu dostarcza informacji o rozmiarze, złożoności i strukturze stworzonego systemu. Otrzymane wyniki pozwalają na ocenę skali pracy oraz mogą stanowić punkt wyjścia do dalszych analiz i optymalizacji projektu.

## 6.2. Możliwości rozbudowy aplikacji

Dalszy rozwój *Formal Specification IDE* może obejmować kilka obszarów, które rozszerzą możliwości stworzonego systemu. Poniżej przedstawione są niektóre z tych możliwości.

W celu usprawnienia procesu definiowania reguł logicznych dla wzorców, istnieje możliwość stworzenia dedykowanego edytora. Taki edytor pozwoliłby na intuicyjne tworzenie i modyfikowanie reguł w bardziej zaawansowany sposób. Narzędzie to mogłoby również zapewnić validację wprowadzanych zmian, co przyczyniłoby się do zwiększenia użyteczności systemu.

W celu zapewnienia kompletnego środowiska pracy, możliwe jest stworzenie dedykowanego edytora diagramów przypadków użycia. Taki edytor umożliwiłby tworzenie, modyfikowanie i wizualizację diagramów przypadków użycia bez konieczności korzystania z zewnętrznych narzędzi.

Obecna wersja edytora diagramów aktywności umożliwia tworzenie i modyfikowanie diagramów z wykorzystaniem zdefiniowanych wzorców. Jednak możliwe jest rozwinięcie tego

edytora, aby umożliwić użytkownikom definiowanie własnych struktur wzorców i ich integrację w procesie tworzenia modeli.

W celu usprawnienia integracji z systemami dowodzenia twierdzeń, możliwe jest stworzenie dedykowanego panelu komunikacyjnego. Taki panel pozwoliłby użytkownikom na bezpośrednie wprowadzanie danych wejściowych dla systemów dowodzenia twierdzeń oraz otrzymywanie wyników dowodzenia w interfejsie systemu.

W celu zapewnienia jeszcze większej elastyczności i użytkowej konfigurowalności interfejsu użytkownika, warto rozważyć możliwość rozbudowy systemu. Aktualnie, ułożenie paneli w oknie aplikacji jest ograniczone, nie umożliwiając przesuwania czy ukrywania kolumn i paneli według indywidualnych preferencji użytkowników. Byłoby zasadne wprowadzenie opcji dostosowywania interfejsu w celu zapewnienia optymalnej przestrzeni roboczej. Sposobem realizacji byłoby wprowadzenie dodatkowej kolumny, która została przeznaczona na wyświetlanie diagramów aktywności. W tym układzie, użytkownicy mogliby swobodnie pracować nad diagramami w głównym oknie aplikacji, a dodatkowe informacje z pozostałych paneli byłyby dostępne w bocznych kolumnach. Taka konfiguracja umożliwiłyby intuicyjne zarządzanie zawartością i umiejscowieniem paneli, zwiększając przy tym komfort pracy. Dodatkową opcją jest umożliwienie dostosowywania szerokości kolumn i układu paneli w taki sposób, aby wszystkie kolumny były widoczne jednocześnie na ekranie. To podejście pozwoliłoby na jednoczesne korzystanie z różnych paneli, co wpłynęłoby na skrócenie czasu przechodzenia między różnymi funkcjonalnościami systemu. Użytkownicy mieliby możliwość ukrywania mniej istotnych dla nich informacji, dzięki czemu pozostałe kolumny mogłyby zajmować większą przestrzeń roboczą. Wprowadzenie powyższej propozycji znacząco wpłynęłyby na zwiększenie użyteczności i elastyczności interfejsu *Formal Specification IDE*. Dostosowywalny interfejs użytkownika daje możliwość konfiguracji pracy zgodnie z indywidualnymi preferencjami, co przekłada się na efektywność pracy i zadowolenie użytkowników.

W celu jeszcze większej wszechstronności i użyteczności systemu, możliwe jest rozszerzenie jego funkcjonalności. Przykładowe obszary rozwoju obejmują obsługę innych rodzajów modeli behawioralnych, integrację z innymi narzędziami inżynierii wymagań oraz dalszy rozwój algorytmów weryfikacji i optymalizacji. Ponadto, adaptacja systemu do innych dziedzin, gdzie analiza i weryfikacja modeli behawioralnych odgrywają istotną rolę, stanowi kolejną możliwość rozwoju.

Podsumowując, stworzony system stanowi solidny fundament do dalszego rozwoju. Suggerowane rozbudowy umożliwią dostosowanie systemu do różnorodnych potrzeb i kontekstów. Dalsze badania, eksperymenty oraz praktyczne zastosowanie systemu w procesach projektowania oprogramowania stanowią perspektywy rozwoju tej pracy magisterskiej.



# Bibliografia

- [1] R. Klimek. „*Pattern-based and composition-driven automatic generation of logical specifications for workflow-oriented software models*”. W: *Journal of Logical and Algebraic Methods in Programming* 104 (2019), s. 201–226.
- [2] Manfred Broy. „*On the Role of Logic and Algebra in Software Engineering*”. W: *Mathematics, Computer Science and Logic - A Never Ending Story: The Bruno Buchberger Festschrift*. Springer International Publishing, 2013, s. 51–68. ISBN: 978-3-319-00966-7. DOI: [10.1007/978-3-319-00966-7\\_2](https://doi.org/10.1007/978-3-319-00966-7_2).
- [3] J. N. Reed, J. E. Sinclair i F. Guigand. „*Deductive Reasoning versus Model Checking: Two Formal Approaches for System Development*”. W: *IFM’99*. Springer London, 1999, s. 375–394. ISBN: 978-1-4471-0851-1.
- [4] Liesbeth De Mol i Giuseppe Primiero. „*When Logic Meets Engineering: Introduction to Logical Issues in the History and Philosophy of Computer Science*”. W: *History and Philosophy of Logic* 36.3 (2015), s. 195–204. DOI: [10.1080/01445340.2015.1084183](https://doi.org/10.1080/01445340.2015.1084183).
- [5] Manfred Broy. „*Yesterday, Today, and Tomorrow: 50 Years of Software Engineering*”. W: *IEEE Software* 35.5 (2018), s. 38–43. DOI: [10.1109/MS.2018.290111138](https://doi.org/10.1109/MS.2018.290111138).
- [6] Dimitri Belli i Franco Mazzanti. „*A Case Study In Formal Analysis Of System Requirements*”. W: *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops: AI4EA, F-IDE, CoSim-CPS, CIFMA, Berlin, Germany, September 26–30, 2022, Revised Selected Papers*. Springer-Verlag, 2023, s. 164–173. ISBN: 978-3-031-26235-7. DOI: [10.1007/978-3-031-26236-4\\_14](https://doi.org/10.1007/978-3-031-26236-4_14).
- [7] José Proença i Andrei Paskevich. „*Proceedings of the 6th Workshop on Formal Integrated Development Environment*”. 2021. DOI: [10.4204/eptcs.338](https://doi.org/10.4204/eptcs.338).
- [8] Catherine Dubois, Paolo Masci i Dominique Méry. „*Proceedings of the Third Workshop on Formal Integrated Development Environment*”. 2017. DOI: [10.4204/eptcs.240](https://doi.org/10.4204/eptcs.240).
- [9] Vincent Aravantinos i in. „*AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems*.” W: *ACES-MB&WUCOR@ MoDELS* 1508 (2015), s. 19–26.

- [10] Frank Wolter i Michael Wooldridge. „*Temporal and Dynamic Logic*”. W: (paź. 2010).
- [11] Oprogramowanie Visual Paradigm.  
<https://www.visual-paradigm.com/download/community.jsp>.
- [12] Oprogramowanie Sparx Enterprise Architect.  
<https://sparxsystems.com/products/ea/trial/request.html>.
- [13] Oprogramowanie Sparx Eclipse Papyrus.  
<https://www.eclipse.org/papyrus/download.html>.
- [14] Oprogramowanie Sinvas UML.  
<https://sourceforge.net/projects/sinvas-uml/>.
- [15] Oprogramowanie GenMyModel.  
<https://app.genmymodel.com/>.
- [16] M. Huth i M. Ryan. „*Logic in Computer Science: Modelling and Reasoning about Systems*”. Cambridge University Press, 2004. ISBN: 9781139453059.
- [17] W. McCune. „*Prover9 and Mace4*”. <http://www.cs.unm.edu/~mccune/prover9/>. 2005–2010.
- [18] Christoph Weidenbach i in. „*SPASS Version 3.5*”. W: Springer Berlin Heidelberg, 2009, s. 140–145.
- [19] Wolfgang Bibel. „*DFG Schwerpunktprogramm "Deduktion"*”. W: *KI - Künstliche Intelligenz* 12 (paź. 1998), s. 38–40.
- [20] Dokumentacja Classic SPASS Theorem Prover.  
<https://webspass.spass-prover.org/help/syntax/index.html>.
- [21] Mark Kaminski i Tobias Tebbi. „*InKreSAT: Modal Reasoning via Incremental Reduction to SAT*”. W: *CADE-24*. Red. Maria Paola Bonacina. T. 7898. LNCS. Springer, czer. 2013, s. 436–442.
- [22] Niklas Eén i Niklas Sörensson. „*An Extensible SAT-solver*”. W: *Theory and Applications of Satisfiability Testing*. Springer Berlin Heidelberg, 2004, s. 502–518.
- [23] Russ Miles i Kim Hamilton. „*UML 2.0. Wprowadzenie*”. O'Reilly, 2007.
- [24] H. Baumann, P. Grässle i P. Baumann. „*UML 2.0 in Action: A Project-Based Tutorial*”. From technologies to solutions. PACKT, 2005. ISBN: 9781904811558.
- [25] Docker.com: What is Docker.  
<https://www.docker.com/what-is-docker/>.
- [26] Biblioteka docker-java.  
<https://github.com/docker-java/>.

- [27] Terence J. Parr i Russell W. Quong. „ANTLR: A predicated-LL ( $k$ ) parser generator”. W: 25 (7 1995), s. 789–810.
- [28] Biblioteka Graph Editor.  
<https://github.com/eckig/graph-editor/>.
- [29] Dokumentacja JavaFX.  
<https://docs.oracle.com/javafx/2/>.
- [30] Biblioteka Jackson.  
<https://github.com/FasterXML/jackson-databind/>.
- [31] Oprogramowanie JDK 19.  
<https://openjdk.java.net/projects/jdk/19/>.
- [32] Oprogramowanie IntelliJ IDEA.  
<https://www.jetbrains.com/idea/>.
- [33] Narzędzie Apache Maven 3.9.3.  
<https://maven.apache.org/>.
- [34] Dokumentacja Google Guice.  
<https://github.com/google/guice/wiki/Guice510>.
- [35] Oprogramowanie Log4j.  
<https://logging.apache.org/log4j/2.x/javadoc.html>.



# A. Dokumentacja techniczna systemu dla użytkownika

## A.1. Wykorzystywane biblioteki

W ramach systemu *Formal Specification IDE* wykorzystywane są różne biblioteki, które zapewniają niezbędne funkcjonalności i ułatwiają rozwój aplikacji. Poniżej przedstawiamy listę bibliotek używanych w systemie:

1. **JavaFX**: JavaFX jest biblioteką graficzną, która umożliwia tworzenie interfejsu użytkownika w aplikacjach Java. Wykorzystano JavaFX do budowy intuicyjnego i responsywnego interfejsu w *Formal Specification IDE*.
2. **Google Guice**: Guice to lekka biblioteka wstrzykiwania zależności (dependency injection) dla języka Java. Użyto Guice do zarządzania zależnościami między komponentami systemu, co ułatwia rozwój i testowanie aplikacji.
3. **log4j**: log4j to biblioteka do logowania zdarzeń w aplikacjach Java. Wykorzystano log4j do rejestrowania i śledzenia zdarzeń w systemie *Formal Specification IDE*, co umożliwia debugowanie i monitorowanie działania aplikacji.
4. **jackson-databind**: jackson-databind to biblioteka do przekształcania danych JSON na obiekty Java i odwrotnie. Użyto jackson-databind do serializacji i deserializacji danych w formacie JSON, co jest przydatne w różnych częściach systemu, w tym przy importowaniu i eksportowaniu projektów.
5. **jaxp-api**: jaxp-api to zestaw interfejsów programistycznych aplikacji (API) do obsługi przetwarzania XML w języku Java. Wykorzystano jaxp-api do parsowania i manipulowania plikami XML w systemie *Formal Specification IDE*.
6. **grapheditor (eckig)**: grapheditor to biblioteka do edycji i wizualizacji grafów w języku Java. Wykorzystano grapheditor do tworzenia i manipulowania diagramami aktywności w

systemie *Formal Specification IDE*, co umożliwia wygodne modelowanie struktur danych i zależności.

7. **antlr4**: antlr4 to narzędzie do generowania analizatorów leksykalnych i parserów dla języków programowania. Wykorzystano antlr4 do tworzenia generatorów kodu źródłowego dla dwóch różnych języków programowania w systemie *Formal Specification IDE*.
8. **docker-java**: docker-java to biblioteka Java do programowego zarządzania kontenerami Docker. Użyto docker-java do interakcji z istniejącymi systemami dowodzenia twierdzeń, co umożliwia tworzenie i zarządzanie kontenerami wewnątrz systemu *Formal Specification IDE*.

## A.2. Systemy dowodzenia twierdzeń

W ramach systemu *Formal Specification IDE* zostały zintegrowane trzy systemy do dowodzenia twierdzeń:

- Prover9
- SPASS Prover
- InKreSAT

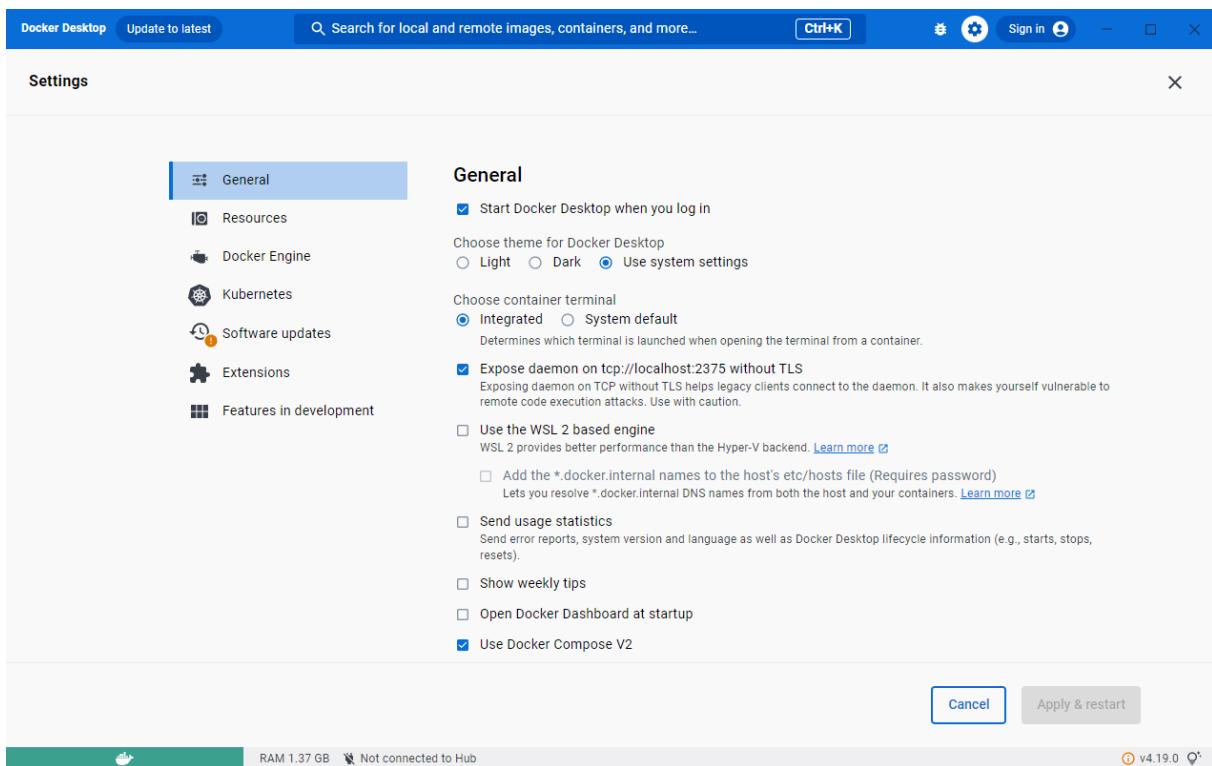
Szczegóły integracji zostały szczegółowo opisane w sekcji 5.6. W przypadku systemów Prover9 i InKreSAT konieczne było wprowadzenie pewnych modyfikacji w celu rozwiązania problemów związanych z procesem instalacji. Opis przyjętych rozwiązań znajduje się w sekcji 5.6.2. Aktualizowane źródła wszystkich systemów są dostępne w folderze /docker. Niezmieniony kod programu SPASS Prover znajduje się w pliku spass35pclinux64.tgz. Oryginalny kod programu Prover9 zawarty jest w pliku LADR-2009-11A.tar.gz. Dodatkowo, w tym samym folderze, znajduje się plik Makefile, który należy podmienić w przypadku próby instalacji. Zaktualizowany kod programu InKreSAT dostępny jest w pliku inkresat-1.0.tar. Plik ten zawiera wszystkie wprowadzone poprawki.

## A.3. Uruchomienie aplikacji

### A.3.1. Warunki wstępne

Przed uruchomieniem aplikacji *Formal Specification IDE* należy upewnić się, że spełnione są następujące warunki:

- **JDK 19:** Aplikacja wymaga zainstalowanego JDK 19 (Java Development Kit) lub nowszej wersji. JDK zapewnia środowisko wykonawcze dla języka Java, niezbędne do uruchomienia aplikacji.
- **Docker desktop:** Aplikacja wykorzystuje kontenery Docker do izolacji i zarządzania różnymi składnikami systemu. W związku z tym, przed uruchomieniem *Formal Specification IDE*, należy mieć zainstalowany Docker desktop. Ustawienia Docker desktop powinny umożliwiać aplikacji *Formal Specification IDE* tworzenie i zarządzanie kontenerami. Przykład konfiguracji ustawień Docker desktop przedstawiono na rysunku A.1.



Rys. A.1. Przykład konfiguracji ustawień Docker desktop, umożliwiających aplikacji *Formal Specification IDE* tworzenie i zarządzanie kontenerami.

### A.3.2. Instrukcje uruchomienia

Aplikacja *Formal Specification IDE* została napisana w języku programowania Java 19. Istnieją dwa sposoby uruchomienia aplikacji: poprzez plik wykonywalny JAR lub za pomocą środowiska programistycznego.

#### 1. Uruchomienie poprzez plik wykonywalny JAR:

- Należy pobrać plik wykonywalny Formal-Specification-IDE-1.0-SNAPSHOT.jar z odpowiedniego źródła.
- JAR jest to rozszerzenie plików Java. Aplikacja została stworzona do działania w środowisku JAVA, wymaga zainstalowania oprogramowania w systemie. Sam system Windows nie może go uruchomić.
- Należy upewnić się, że w tym samym folderze znajduje się plik config.json, który wskazuje aplikacji dostępne położenia informacji w plikach .xml, co jest potrzebne podczas wydobywania informacji o przypadkach użycia.
- Możliwe jest również uruchomienie aplikacji z poziomu wiersza poleceń: Należy otworzyć terminal lub wiersz polecenia i przejść do folderu, w którym znajduje się plik JAR. Następnie aplikację można uruchomić za pomocą polecenia: `java -jar Formal-Specification-IDE-1.0-SNAPSHOT.jar`

## 2. Uruchomienie za pomocą środowiska programistycznego IntelliJ IDEA:

- Należy uruchomić Uruchom IntelliJ IDEA. Użycie tej opcji pozwoli użytkownikowi na wprowadzanie zmian w kodzie źródłowym systemu.
- Należy Wybrać opcję File -> New -> Project from Existing Sources...
- Należy Wskazać plik pom.xml i postępować zgodnie z instrukcjami, aby skonfigurować projekt. W tym momencie IntelliJ powinien ustalić zależności i skonfigurować projekt.
- Po pomyślnym skonfigurowaniu projektu, można uruchomić aplikację z poziomu IntelliJ IDEA.
- Ta opcja umożliwia modyfikację kodu źródłowego aplikacji, dostosowywanie jej działania lub tworzenie nowych funkcjonalności.

Po wykonaniu powyższych instrukcji, aplikacja *Formal Specification IDE* zostanie uruchomiona i będzie gotowa do użytku. Należy pamiętać, że przed rozpoczęciem pracy z aplikacją, warto zapoznać się z dostępna dokumentacją i instrukcjami obsługi, aby jak najlepiej wykorzystać wszystkie funkcjonalności oferowane przez system.

## A.4. Obsługa zewnętrznych programów do modelowania diagramów przypadków użycia

Zewnętrzne programy służące do modelowania diagramów przypadków użycia zostały wymienione w sekcji 2.3. Oto lista tych programów:

- Visual Paradigm
- Enterprise Architect
- Eclipse Papyrus
- Sinvias UML
- GenMyModel

Aplikacja *Formal Specification IDE* umożliwia integrację z dowolnym programem obsługującym diagramy przypadków użycia. Użytkownik może wybrać preferowany program do modelowania i skonfigurować go w pliku config.json. W pliku tym należy wskazać odpowiednią ścieżkę, wskazującą położenie danych w pliku .xml dotyczących przypadków użycia.

Dzięki tej funkcjonalności, użytkownik ma elastyczność w wyborze programu do modelowania diagramów przypadków użycia, zgodnie z własnymi preferencjami i potrzebami. Aplikacja *Formal Specification IDE* jest w stanie współpracować z różnymi narzędziami, co ułatwia proces tworzenia i zarządzania przypadkami użycia w systemie.

## A.5. Generacja kodu

W programie *Formal Specification IDE* wygenerowany kod jest automatycznie zapisywany do folderu /generated\_code. Dzięki temu użytkownik ma łatwy dostęp do wygenerowanych plików źródłowych.

Dodatkową funkcjonalnością aplikacji jest możliwość zmiany typu generatora kodu. W trakcie tworzenia oprogramowania powstały dwie wersje generatora kodu źródłowego. Jeśli użytkownik chce dokonać zmiany wersji generatora, może to zrobić w pliku config.json, poprzez zmianę wartości pola code\_generator\_type na jedną z dwóch akceptowanych wartości: „v1” lub „v2”.

Dostęp do różnych wersji generatora kodu daje użytkownikowi elastyczność w wyborze preferowanego narzędzia do generowania kodu źródłowego. Może to być przydatne, gdy użytkownik chce przetestować różne podejścia lub dostosować generowany kod do konkretnych wymagań projektowych. Poprzez prostą zmianę wartości w pliku konfiguracyjnym, użytkownik może eksperymentować z różnymi wersjami generatora kodu w celu uzyskania optymalnych wyników.

## A.6. Wyniki formalnej weryfikacji

Wyniki przeprowadzonej weryfikacji w systemie *Formal Specification IDE* są starannie zapisywane do plików w celu zapewnienia trwałości i łatwego dostępu do nich.

Wejścia dla proverów, takie jak dane wejściowe i specyfikacje, są zapisywane w folderze `/prover_input`. W przypadku provera Prover9, jego wejście jest generowane przez skrypt dostarczony wraz z oprogramowaniem SPASS Prover. Skrypt `dfg2otter` konwertuje dane wejściowe na format zrozumiały dla Prover9, a następnie plik wynikowy jest zapisywany w folderze `/prover_converting_logs`.

Wyniki działania proverów, takie jak dowody, informacje o spełnieniu czy nie-spełnieniu warunków, są zapisywane w folderze `/prover_output`. Wszelkie logi generowane przez provera w trakcie ich działania są również zapisywane dla celów diagnostycznych i analizy w folderze `/prover_logs`.

Dzięki tej strukturze przechowywania danych użytkownik ma łatwy dostęp do wyników weryfikacji i może przeglądać zarówno wejścia, jak i wyjścia proverów. Ponadto, przechowywanie logów pozwala na analizę procesu weryfikacji i identyfikację ewentualnych problemów lub błędów, co jest istotne w przypadku bardziej zaawansowanych zastosowań formalnej weryfikacji.

## A.7. Reguły logiczne wzorców przepływu pracy

Możliwa jest edycja reguł logicznych dla wzorców przepływu pracy poprzez modyfikację plików `.json` znajdujących się w folderze `/pattern_rules`. W tym folderze dostępne są dwa pliki:

- `pattern_rules_FOL.json` – zawierający reguły dla logiki pierwszego rzędu,
- `pattern_rules_LTL.json` – zawierający reguły dla liniowej logiki temporalnej.

Pliki te zawierają zbiór reguł logicznych dla poszczególnych wzorców przepływu pracy. Wzorce te zostały omówione w sekcji 2.2.

Przykład reguł wzorców przedstawiono poniżej:

```
{
  "Seq": {
    "number of args": 2,
    "rules": [
      "arg0",
      "arg1",
      "Exist(arg0)",
      "ForAll(arg0 => Exist(arg1))",
      "ForAll(~(arg0 ^ arg1))"
    ]
  },
  "Concur": {
    ...
  }
}
```

```
"number of args": 3,  
"rules": [  
    "arg0",  
    "arg1 | arg2",  
    "Exist(arg0)",  
    "ForAll((arg0) => (Exist(arg1) ^ Exist(arg2)))",  
    "ForAll(~((arg0) ^ (arg1)))",  
    "ForAll(~((arg0) ^ (arg2)))"  
]  
,  
// Pozostałe wzorce...  
}
```

W plikach .json główny obiekt zawiera podobiekty, których nazwy odpowiadają nazwom wzorców. Każdy podobiekt wzorca zawiera dwa parametry:

- `number of args` – określa liczbę argumentów wzorca,
- `rules` – zawiera listę reguł logicznych.

Pierwsze dwa elementy listy reguł to argumenty wejściowe i wyjściowe wzorca, a pozostałe elementy to reguły logiczne.

Pliki .json można edytować według własnych potrzeb, pamiętając o zachowaniu struktury pliku i poprawnej składni formatu .json. Modyfikacja reguł logicznych umożliwia dostosowanie systemu do konkretnych wymagań i preferencji użytkownika.

## A.8. Przykłady diagramów przypadków użycia

System *Formal Specification IDE* dostarcza przykładowe projekty diagramów przypadków użycia wraz z odpowiadającymi im grafikami. Pliki przykładów znajdują się w folderze /xml\_and\_png\_examples. Przykłady te zostały stworzone przy użyciu popularnych programów do modelowania, takich jak Visual Paradigm, Enterprise Architect, Eclipse Papyrus, Sinvas UML oraz GenMyModel.

Dostępność tych przykładów ma na celu zaprezentowanie możliwości systemu oraz umożliwienie użytkownikom zapoznania się z różnymi typami diagramów przypadków użycia. Mogą one służyć jako punkt wyjścia do eksploracji i testowania funkcjonalności *Formal Specification IDE*, a także jako inspiracja przy tworzeniu własnych projektów. Przykłady są w formacie plików XML i są dostępne do odczytu i edycji w systemie. Odpowiadające im grafiki w formacie PNG zostały dołączone, aby wizualnie przedstawić strukturę i relacje między aktorami oraz przypadkami użycia.

Użytkownicy mogą eksperymentować z przykładami, analizować ich konstrukcję i wykorzystywać je jako wzorce do tworzenia własnych diagramów przypadków użycia. To interaktywne podejście pozwala na lepsze zrozumienie i opanowanie tworzenia i analizy diagramów przypadków użycia w *Formal Specification IDE*.

## A.9. Podział klas w projekcie

W projekcie *Formal Specification IDE* klasy zostały podzielone na 8 pakietów, które mają różne zadania i funkcje. Poniżej przedstawiono krótki opis każdego pakietu:

1. **main:** Ten pakiet zawiera główną klasę `FormalSpecificationIDE`, która pełni rolę punktu startowego programu. Znajdują się tutaj również klasy odpowiedzialne za konfigurację i inicjalizację systemu.
2. **exceptions:** Pakiet ten zawiera klasy odpowiedzialne za definicję specyficznych wyjątków w systemie.
3. **factories:** Ten pakiet zawiera klasy fabryk, które są odpowiedzialne za tworzenie instancji obiektów w systemie. Klasy fabryk dostarczają interfejsy i metody umożliwiające elastyczne tworzenie obiektów zgodnie z określonymi parametrami i wymaganiami.
4. **model:** Pakiet `model` zawiera klasy reprezentujące model danych systemu. Te klasy definiują struktury danych i ich relacje, które są wykorzystywane przez inne komponenty systemu do przechowywania i manipulacji danymi.
5. **persistence:** Ten pakiet zawiera klasy odpowiedzialne za obsługę trwałości danych, takie jak zapisywanie i odczytywanie danych z plików. Komponenty w tym pakiecie umożliwiają przechowywanie danych systemowych w formacie plików, co umożliwia ich zachowanie między kolejnymi uruchomieniami systemu.
6. **services:** Pakiet `services` zawiera klasy serwisów, które implementują główne funkcje i logikę biznesową systemu. Te klasy są odpowiedzialne za przetwarzanie danych, wykonywanie operacji i udostępnianie funkcjonalności systemowej dla innych komponentów.
7. **ui:** W pakiecie `ui` znajdują się klasy związane z interfejsem użytkownika. Komponenty w tym pakiecie odpowiadają za prezentację danych i interakcję z użytkownikiem. Zawiera również logikę interfejsu użytkownika, taką jak obsługa zdarzeń i reakcje na akcje użytkownika.
8. **utilities:** Ten pakiet zawiera narzędzia pomocnicze, takie jak klasy do obsługi plików, formatowania danych, generowania identyfikatorów itp. Klasy w tym pakiecie dostarczają funkcje pomocnicze, które mogą być użyteczne w różnych częściach systemu.

Podział klas na odpowiednie pakiety ułatwia organizację kodu, zapewnia czytelność i separację odpowiedzialności. Każdy pakiet ma określony zakres funkcjonalności, co ułatwia zarządzanie i rozwijanie projektu *Formal Specification IDE*.