Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

ROZPRAWA DOKTORSKA

METODY FORMALNEJ ANALIZY SYSTEMÓW WBUDOWANYCH CZASU RZECZYWISTEGO

AUTOR:

mgr inż. Jarosław Baniewicz

PROMOTOR:

prof. dr hab. Marcin Szpyrka

Kraków 2018

University of Science and Technology in Kraków

Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering

PhD Dissertation

Methods of formal analysis of embedded real-time systems

Author:

mgr inż. Jarosław Baniewicz

Supervisor:

Prof. Marcin Szpyrka, PhD, DSc

Kraków 2018

Mojej żonie Eli
oraz dzieciom Weronice, Jasiowi i Jackowi.

W sposób szczególny dziękuję
Panu profesorowi Marcinowi Szpyrce za jego nieocenione zaangażowanie w powstanie niniejszej rozprawy.

Abstract

This doctoral dissertation deals with the possibility of formal modeling of real-time embedded systems using the Alvis environment. Formal analysis is gaining importance today, taking into account the huge demand for IT systems as well as the costs caused by their erroneous operation. Analysis of information systems using formal methods is a very popular topic in scientific literature. Approaches based mostly on formalisms such as timed automata or Petri nets are rarely used in engineering projects. The doctoral dissertation is part of the trend of searching for new solutions adapted to engineering practice.

The adopted hardware basis for any systems considered in the dissertation is any single-processor platform. The tasks or processes running on it that are real-time system artefacts are managed by means of a special scheduling algorithm that takes into account their priorities and position in the two-dimensional FIFO queue.

This dissertation uses the innovative Alvis formal modeling language. This language has been developed and is still being developed at the Department of Applied Computer Science at the Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering of the AGH University of Science and Technology in Krakow. The main features of the Alvis language are primarily the extremely clear syntax and the possibility of formal verification of the model using external environments implementing model checking techniques. Modeling in the Alvis environment takes place in two stages. In the first phase, a graphic model of the system structure and communication between its elements is created. In the next step, the dynamics of the system being created is described using the high-level language. The final stage of the modeling process is generating *labeled transition system*, which forms the basis for formal verification of the system.

The main purpose of this dissertation was to develop the Alvis language extension for modeling embedded real-time systems, with the assumption that they operate on a single-processor platform. To this end, a new system layer has been developed α^1_{FPPS}, which primarily takes into account the aspects related to time and defines the algorithm for scheduling real time system's tasks. For the use of a system layer α^1_{FPPS} a new algorithm for determining labeled transition systems was developed and implemented. Implementation is an extension of IHR (Intermediate Haskell Representation) used in the Alvis environment, therefore, it constitutes a natural stage in the development of Alvis language and supporting software.

This dissertation also includes a case study that shows the practical use of Alvis to create formal models for two real-time systems. The final element of the dissertation is an attempt to compare the Alvis modeling language, extended by the α^1_{FPPS} system layer, with other popular formalisms for modeling such systems.

Streszczenie

Przedstawiona rozprawa doktorska dotyczy możliwości formalnego modelowania systemów wbudowanych czasu rzeczywistego za pomocą środowiska Alvis. Analiza formalna zyskuje dzisiaj na znaczeniu, biorąc pod uwagę ogromne zapotrzebowanie na systemy informatyczne, jak również koszty spowodowane ich błędnym działaniem. Analiza systemów informatycznych za pomocą metod formalnych to temat bardzo popularny w literaturze naukowej. Podejścia bazujące najczęściej na formalizmach takich jak automaty czasowe, czy sieci Petriego rzadko stosowane są w projektach inżynierskich. Rozprawa doktorska wpisuje się w nurt poszukiwania nowych rozwiązań przystosowanych do praktyki inżynierskiej.

Przyjętą podstawą sprzętową, dla rozważanych w rozprawie systemów jest dowolna platforma jednoprocesorowa. Uruchomione na niej zadania lub procesy, będące artefaktami systemu czasu rzeczywistego, zarządzane są za pomocą specjalnego algorytmu szeregującego, który uwzględnia ich priorytety oraz pozycję w dwuwymiarowej kolejce FIFO.

W niniejszej rozprawie zastosowano nowatorski język modelowania formalnego Alvis. Język ten został opracowany i jest nadal rozwijany w Katedrze Informatyki Stosowanej na Wydziale Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej Akademii Górniczo-Hutniczej w Krakowie. Główne cechy języka Alvis to przede wszystkim wyjątkowo czytelna składnia i możliwość formalnej weryfikacji modelu z użyciem zewnętrznych środowisk implementujących techniki weryfikacji modelowej. Modelowanie w środowisku Alvis przebiega dwuetapowo. W pierwszej fazie tworzy się graficzny model struktury systemu oraz komunikacji pomiędzy jego elementami. W następnym kroku za pomocą języka wysokiego poziomu opisuje się dynamikę tworzonego systemu. Końcowym etapem procesu modelowania jest wygenerowanie *etykietowanego systemu przejść*, który stanowi podstawę formalnej weryfikacji systemu.

Głównym celem niniejszej rozprawy było opracowanie rozszerzenia języka Alvis na potrzeby modelowania wbudowanych systemów czasu rzeczywistego, z założeniem, że działają one na platformie jednoprocesorowej. W tym celu opracowano nową warstwę systemową (α^1_{FPPS}) języka Alvis, która przede wszystkim uwzględnia aspekty związane z czasem oraz definiuje algorytm szeregowania zadań systemu czasu rzeczywistego. Na potrzeby stosowania warswy systemowej α^1_{FPPS} opracowano i zaimplementowano nowy algorytm wyznaczania etykietowanych systemów przejść. Implementacja jest rozszerzeniem stosowanej w środowisku Alvis reprezentacji IHR (Intermediate Haskell Representation), przez co stanowi naturalny etap rozwoju języka Alvis i wspierającego go oprogramowania.

Niniejsza rozprawa zawiera także studium przypadków, w którym pokazano praktyczne wykorzystanie Alvisa do tworzenia modeli formalnych, dla dwóch systemów czasu rzeczywistego. Końcowym elementem rozprawy jest próba porównania języka modelowania Alvis, rozszerzonego o warstwę systemową α^1_{FPPS}, z innymi popularnymi formalizmami do modelowania tego typu systemów.

Spis treści

Spis rysunków

1. Wprowadzenie

Systemy informatyczne nigdy nie były domeną związaną jedynie ze światem nauki, elektroniki, telekomunikacji lub informatyki. Od początku ich istnienia zyskały na znaczeniu praktycznie w każdej gałęzi przemysłu i rozwoju myśli technicznej. W obecnych czasach niemal każda idea będąca odpowiedzią na rynek potrzeb, od razu zyskuje wsparcie w postaci implementującego ją systemu, aplikacji lub oprogramowania. Powstające algorytmy i rozwiązania programistyczne bardzo często wpływają na koszty związane ze wspierającymi je platformami sprzętowymi. Ilość rdzeni mikroprocesora w przypadku potrzeby bardzo szybkich, równoległych operacji, pojemność przestrzeni dyskowej magazynującej duże obszary danych lub chociażby zależności sprzętowe związane z samouczeniem się maszyn (ang. *machine learning* [14]), to przykładowe problemy a zarazem czynniki wpływające na koszty projektu, a co za tym idzie ich wdrożenia w gotowych rozwiązaniach komercyjnych.

Kolejną bardzo ważną kwestią w przypadku projektowania systemów informatycznych jest to, że muszą one wspierać dedykowany im poziom krytyczności, czyli nigdy nie powinny wykazać się błędnym działaniem w przypadku zdefiniowanych dla nich funkcjonalności. Biorąc pod uwagę ogromne zapotrzebowanie na systemy informatyczne, koszty związane z ich projektowaniem, a także żądaną bezawaryjność, należy ze szczególną atencją odnieść się do spraw związanych z ich analizą i weryfikacją. Czynności te nie mogą być podejmowane jedynie na początku projektu lecz również powinny być wykonywane poprzez cały okres implementacji danego systemu [41]. Ze względu na niezawodność projektowanego systemu wydaje się zasadne zastosowanie modelu formalnego do weryfikacji ich poprawności i braku rozbieżności ze specyfikacją wymagań [1].

Niniejsza rozprawa podejmuje próbę zaprezentowania praktycznego aparatu formalnej analizy systemu wbudowanego czasu rzeczywistego, opartej na weryfikacji modelu utworzonego dla danego systemu.

Systemy współbieżne czasu rzeczywistego

Proces projektowania systemu czasu rzeczywistego, uwzględniający jego analizę wzbogaconą o modelowanie systemu, a później implementację i walidację, nie należy do zadań banalnych w szeroko pojętej inżynierii oprogramowania [41]. Systemy te, z uwagi na współbieżność wykonywanych w nich akcji, nie są

do końca deterministyczne. Mogą być one w pewnym stopniu przewidywalne na podstawie stawianym im wymagań, ich architektury oraz wyników przeprowadzonych symulacji. Dość powszechnym zjawiskiem w tego typu systemach jest ciągła rywalizacja pomiędzy procesami (ang. *processes*) lub zadaniami (ang. *tasks*) o dostępność i przejęcie kontroli nad zasobami systemu, takimi jak obszar pamięci lub procesor. Mamy tutaj do czynienia z szeregiem różnych zachowań planowanych, jak wzajemne wywłaszczanie, priorytetyzacja, kolejkowanie oraz tymi z zachowań, które przewidzieć jest niezwykle trudno, jak zagłodzenie zadań (ang. *starvation*), odwrócenie priorytetów (ang. *priority inversion*) lub zakleszczenie (ang. *deadlock*). Dodatkowo systemy takie mogą mieć nieskończoną wariację rożnych stanów, które ze swojej natury mogą być niezwykle trudne do przewidzenia przez projektanta na etapie definiowania architektury.

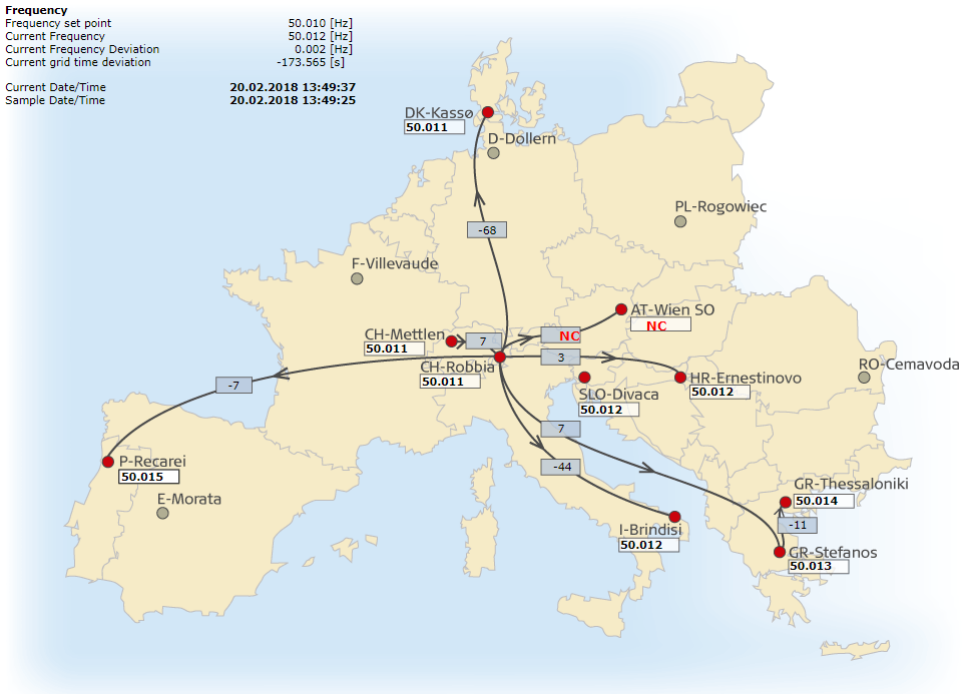
Systemy czasu rzeczywistego [17], [42] mogą być rozpatrywane w różnej skali, biorąc pod uwagę chociażby ilość zarządzanych przez nie procesów, funkcjonalność jaką spełniają oraz ich zasięg. Przez zasięg systemu rozumie się tutaj, czy system działa jedynie lokalnie, czy też komunikuje się z innymi systemami lub jest systemem rozproszonym analizującym i zarządzającym przesyłaniem informacji z różnych lokalizacji oraz jej przechowywaniem.

Przykładem systemów czasu rzeczywistego małej skali mogą być systemy zarządzające pracą komponentów samochodowych [7], [6]. Istnieje tutaj cały zestaw funkcjonalności, które mogą być wspierane przez takie systemy. Najbardziej znane systemy zarządzające pracą układu hamulcowego to ASB (ang. *Anti-Lock Braking System*), który zapobiega blokowaniu się kół pojazdu podczas hamowania lub ESP (ang. *Electronic Stability Program*), którego głównym zadaniem jest stabilizacja toru jazdy pojazdu podczas pokonywania zakrętów. Kolejnymi przykładami systemów czasu rzeczywistego tej skali mogą być systemy kontrolujące i wspierające pracą silnika, alarmu lub emisji dźwięków w pojazdach hybrydowych. Ostatnio systemy takie przejmują nawet niektóre czynności kierującego pojazdem, jak bezpieczne parkowanie, czy chociażby utrzymywanie stałej prędkości na drodze w odniesieniu do innych przemieszczających się aut.

Poza przemysłem samochodowym istnieje spora grupa systemów czasu rzeczywistego związana z przemysłem medycznym, jak respiratory, aparatura pomiarowa, systemy asystujące lekarzom przy zabiegach i operacjach. Obie przedstawione grupy systemów należą do grupy systemów krytycznych ze względu na możliwość spowodowania uszczerbku na zdrowiu człowieka, a w przypadkach ekstremalnych również utraty życia.

Przykładem systemu nieco bardziej rozbudowanego jest system zarządzający pracą podstacji energetycznej, której główną rolą jest zarządzanie dystrybucją energii elektrycznej do odpowiednich konsumentów, widzianych w systemie jako punkty lokalne. Za przykład może tu posłużyć system SDM600 firmy ABB, który monitoruje i zarządza pracą wszystkich układów IED wpiętych do tego systemu (ang. *Intelligent Electonic Device*), poprzez kontrolę ich oprogramowania, zbieranie i przechowywanie logów, dostarczanie raportów, konfigurację całej sieci, a także wspieranie jej od strony cyberbezpieczeństwa.

Grupa systemów czasu rzeczywistego dużej skali zawiera rozwiązania polegające na przesyłaniu informacji na skalę dużych obszarów, nierzadko między poszczególnymi krajami, czy też kontynentami. W energetyce systemy takie dzieli się dodatkowo na WAMS (ang. *Wide Area Monitoring Systems*) i WAMPAC (ang. *Wide Area Monitoring Protection And Control*). Przykładem pierwszej podgrupy może być system PSGuard, stworzony przez firmę ABB do monitorowania transferu energii elektrycznej. Skalowalność tego systemu zależy od potrzeb konsumenta, może monitorować transfer energii elektrycznej na terytorium danego obszaru, kraju lub też pomiędzy państwami rozmieszczonymi po globie ziemskim. System PSGuard przechowuje też synchrofazorowe dane i rezultaty działających na nim aplikacji. Te dane historyczne służą do odtworzenia sytuacji, które zaistniały podczas przesyłania energii. Bieżące monitorowanie stanu sieci może ustrzec przed najgorszym z przypadków, czyli zerwaniem transferu energii (ang. *blackout*). Przykład wykorzystania systemu PSGuard przez Swissgrid (szwajcarskie konsorcjum państwowe zajmujące się energetyką) można zobaczyć na rysunku 1.1.



Rysunek 1.1: Przykład wykorzystania systemu PSGuard przez Swissgrid

Można powiedzieć, że systemy WAMPAC stanowią nową generację w świecie informatycznych systemów energetycznych, gdyż zawierają zautomatyzowane reakcje na zdarzenia mogące wystąpić w systemie, dzięki czemu ingerencja ludzka, nierzadko spóźniona, nie będzie już potrzebna. Poza energetyką, spotykane systemy średniej i dużej skali to systemy kontroli lotów, systemy sterujące pracą fabryki, elektrowni jądrowej, lotami kosmicznymi itp.

Systemy wbudowane czasu rzeczywistego

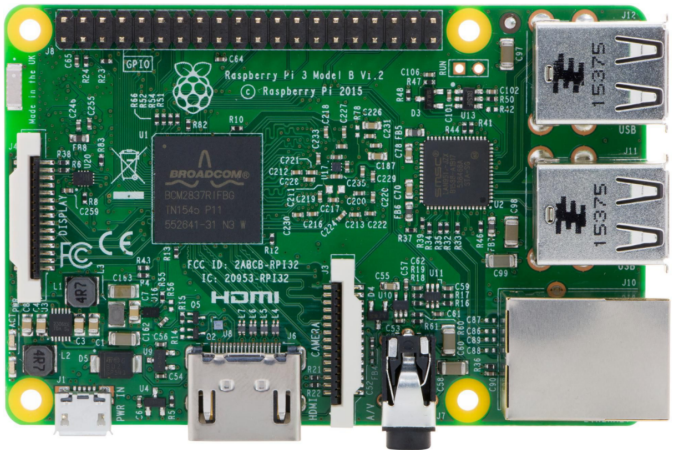
Pewnego rodzaju podgrupą systemów czasu rzeczywistego są systemy wbudowane (ang. *embedded systems*). Nazwa ich wprost określa jak jest widziane ich oprogramowanie. Zostaje wbudowane, czyli innymi słowy zaprogramowane, na danym urządzeniu, sprzęcie komputerowym (ang. *hardware*), czy też platformie n-procesorowej (lub n-rdzeniowej). Systemy wbudowane posiadają wszystkie charakterystyczne cechy systemów czasowych, do których możemy zaliczyć:

- Duży stopień współbieżności procesów.
- Wymagania dotyczące terminowych odpowiedzi przed upływem zadeklarowanego czasu. – Z uwagi na tą zależność od czasu odpowiedzi systemy wbudowane można podzielić na:
  - *twarde* – w których zdefiniowano najdłuższy czas odpowiedzi i przyjęto, że nie zostanie on przekroczony; każda odpowiedź w czasie dłuższym, niż zakładany jest sytuacją błędną działania takiego systemu;
  - *miękkie* – w których system odpowiada w możliwie skończonym przedziale czasowym, lecz jego maksymalna wartość nie jest znana; systemy takie nie są krytyczne pod względem czasu odpowiedzi.
- Zapewnienie nieprzerwalnego, ciągłego w czasie działania danego systemu.
- Wsparcie dla cyberbezpieczeństwa. – Ta cecha zyskała ostatnio na wartości i zakłada dwa etapy kontroli użytkownika/klienta. Pierwszym jest uwierzytelnienie (ang. *authentication*), czyli sprawdzenie, czy dana osoba próbująca zdobyć dostęp do danego systemu, jest tą za którą się podaje. Kolejny etap to autoryzacja (ang. *authorisation*), czyli sprawdzenie jakie prawa dostępu do danych i zasobów posiada dany użytkownik systemu.

W obecnych czasach systemy wbudowane czasu rzeczywistego przeżywają renesans za sprawą rozwoju wielu gałęzi przemysłu bazujących na idei znanej pod hasłem Internet rzeczy (ang. *Internet of Things* – IoT). Rozwiązanie to polega na zbieraniu, gromadzeniu oraz przetwarzaniu danych przez komponenty elektroniczne i współdzielenie tych danych za pomocą sieci komputerowej do innych zainteresowanych odbiorców danego systemu. Dzięki temu możliwe jest projektowanie systemów do zarządzania inteligentnymi budynkami, przedsiębiorstwami, a nawet całymi miastami. Dodatkowo dzięki takim systemom można monitorować środowisko, systemy pomiarowe, systemy energetyczne i obserwować potencjalne zagrożenia.

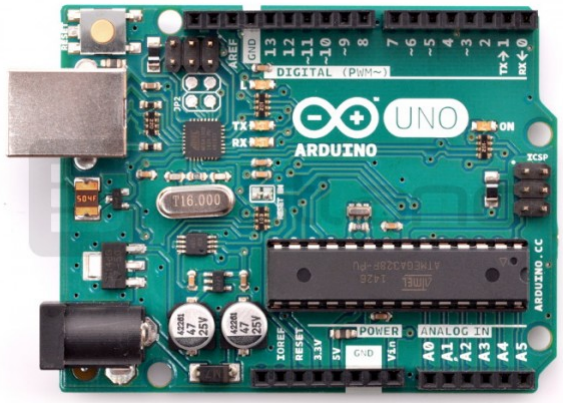
Przykładem urządzenia będącym flagowym komponentem systemów IoT jest komputer Raspberry Pi 3 pokazany na rysunku 1.2. Głównym elementem tego zminiaturyzowanego zestawu komputerowego jest 64-bitowy procesor Broadcom Quad Core 1.2GHz BCM2837. Dodatkowo wyposażony został on w 1 GB pamięci RAM oraz port Micro SD, służący do wgrywania systemu operacyjnego i przechowywania danych. Komunikacja z otoczeniem możliwa jest za pomocą modułów WiFi BCM43143 i Bluetooth Low Energy

(BLE), które są umieszczone na płytce. Oczywiście standardowo dostępne są piny GPIO (ang. *General Purpose Input Output*) oraz cztery porty USB 2. Istnieje także możliwość podłączenia kamery poprzez port CSI i wyświetlacza dotykowego poprzez port DSI. Układ współpracuje również z urządzeniami wspierającymi pełnowymiarowe HDMI.



Rysunek 1.2: Raspberry Pi 3

Kolejnym bardzo popularnym urządzeniem wykorzystywanym w rozwiązaniach IoT jest układ Arduino Uno Rev3, który zaprezentowano na rysunku 1.3. Zestaw ten posiada mikrokontroler ATmega326 z 8-bitowym rdzeniem AVR, który taktowany jest zegarem o częstotliwości 16 MHz. Dodatkowo wyposażony jest w 2 kB pamięci RAM, 1 kB pamięci EEPROM i 32 kB pamięci Flash. Posiada ponadto wejścia analogowe, piny cyfrowe, kanały PWM, przetwornik A/C i gniazdo USB.



Rysunek 1.3: Arduino Uno Rev3

Systemy wbudowane czasu rzeczywistego stanowią główny element analizy dla niniejszej rozprawy doktorskiej. Przykłady prezentowane w rozprawie były implementowane i sprawdzane na wyżej wymienionych platformach sprzętowych z użyciem systemu operacyjnego FreeRTOS.

Model systemu

Jednym ze sposobów zapanowania nad złożonością wbudowanego systemu czasu rzeczywistego jest stworzenie jego modelu. Model taki może służyć nie tylko podczas analizy wymagań, architektury, czy też kwestiom związanym z poszczególnymi funkcjonalnościami systemu. Można go również wykorzystać dla celów walidacji, czyli sprawdzenia czy bieżąca implementacja odpowiada przyjętym wymaganiom, normom i ustaleniom.

Modelowanie jest uniwersalną techniką, która może być używana przez różne kompetencje w projekcie, a więc przez architektów do zbudowania ogólnej, systemowej wizji dla danego rozwiązania, dalej przez programistów zajmujących się implementacją tego rozwiązania i finalnie przez osoby testujące bieżącą implementację. Przykładem powszechnie używanego przez inżynierów języka modelowania pół-formalnego jest UML (ang. *Unified Modeling Language*) stworzony przez Grady Boocha, Jamesa Rumbaugha oraz Ivara Jacobsona, a obecnie rozwijany przez Object Management Group.

Model wcale nie musi być używany jedynie na początku definiowania systemu, jak to jest przyjęte w metodykach opartych na modelu kaskadowym (ang. *waterfall*) wytwarzania oprogramowania. W popularnych obecnie metodykach zwinnych (ang. *agile*) model systemu może być także artefaktem, który akceptuje zmiany wymagań, uwzględnia je i staje się wzorcem tychże zmian dla kolejnej iteracji prac nad danym systemem.

Model systemu może przybierać dowolną formę w zależności od potrzeb. I tak model może być przedstawiony w postaci graficznej, pseudo-kodu, symulacji, czy nawet wykonywalnego (ang. *executable*) fragmentu oprogramowania. Bardzo ważną cechą charakterystyczną dla modelu jest to, że wprowadza on pewnego rodzaju abstrakcję, która może być różnie definiowana w zależności od etapów projektowania, implementacji bądź wdrażania danego systemu [30]. Przykładowo w fazie projektowania, ukierunkowanej na aspekty związane z funkcjonalnością systemu widzianą przez użytkownika, niekoniecznie istotne są kwestie związane z budową rdzenia procesora odpowiedzialnego za operacje arytmetyczno-logiczne na jego rejestrach. Zatem abstrakcja modelu może przykryć niepotrzebne na chwilę obecną detale, jednak możliwość ta wcale nie wyklucza tego, że osoba projektująca dany system wbudowany czasu rzeczywistego, może stworzyć dodatkowy model uwzględniający te właśnie poziomy szczegółowości. Bardzo przydatną i praktyczną cechą modeli jest to, że można panować nad ich abstrakcją, w zależności od ich bieżących potrzeb oraz od etapów prac nad danym systemem. Konstruując model, częstym pytaniem jest pytanie o zasadność uwzględnienia konkretnego komponentu, jego funkcjonowanie w danym systemie i cele, dla których został w ten model wkomponowany. Tego rodzaju pytania stwarzają możliwość podjęcia decyzji, które elementy systemu powinny być zawarte w modelu jako elementy istotne a nawet krytyczne z punktu widzenia chociażby analizy i działania całości. Elementy modelu mogą też uwzględniać te komponenty systemu, które są

istotne z punku widzenia integracji z innymi systemami, komunikacji, cyberbezpieczeństwa, a także, ważnych dla tej rozprawy zależności czasowych. Odpowiednio dobrana abstrakcja stanowi *state of art* całego procesu definiowania i tworzenia modelu.

Tworząc model dostrzega się kolejną ważna cechę, jaką jest przewaga modelu nad tradycyjną dokumentacją projektową. Model jest prostszy i niekiedy bardziej kompletny. Ponadto nie musi być płaski, jak to często bywa w przypadku zbioru wymagań, gdzie trudno dostrzec powiązania hierarchiczne pomiędzy funkcjonalnościami jakie opisują. Cechy systemu (ang. *features*) mogą być uwypuklone w sposób bardziej przejrzysty niż ich spisanie w postaci płaskiej listy wymagań. Użytkownik czytając wymagania, nawet pogrupowane domenowo, może odnieść wrażenie, że w dość prosty sposób traci się połączenie (ang. *linkage*) pomiędzy nimi. Na modelu takie zależności są widoczne i dodatkowo, mogą zostać poddane symulacji. Strukturowość systemu, hierarchia jego komponentów jest z reguły cechą, którą widać w modelu natychmiast. Projektant tworząc model, najczęściej uwzględnia te zależności pomiędzy komponentami jako pierwsze. Dodatkowo osoby zajmujące się modelem systemu, mogą w dość prosty sposób wychwycić błędy samych wymagań, które w spłaszczone wersji tradycyjnego dokumentu mogą być trudne do zaobserwowania. Czas spędzony na tworzeniu modelu przyczynia się ponadto do lepszego zrozumienia całego systemu przez projektanta. Podobną korzyść osiągną też osoby analizujące dany model pod kątem, np. architektury, implementacji i testów. Osoby pracujące w projekcie lepiej rozumieją ogólną ideę systemu i jego projekt (ang. *design*), poprzez obserwację modelu, analizę wyników jego symulacji i robione na nim zmiany. Model o odpowiednim poziomie abstrakcji stwarza również użytkownikom możliwość zobaczenia tego, co jest planowane do zaimplementowania i późniejszego wdrożenia. Mogą upewnić się, że ich wymagania co do funkcjonalności danego systemu są właściwie zinterpretowane i rozumiane, przez osoby zajmujące się realizacją ich wizji i potrzeb.

Samo modelowanie pozwala lepiej zrozumieć wymagania. Projektant dyskutuje z innymi projektantami a przede wszystkim użytkownikami przyszłego systemu, widzi zależności pomiędzy elementami modelu, przez co w bardzo prosty sposób może zauważyć nieobecność danego modułu, która może prowadzić do błędów w całym systemie. Symulacja modelu może wskazać stany niemożliwe do osiągnięcia, wbrew wcześniejszemu projektowi i wymaganiom lub takie, których istnienia się nie zakładało na wcześniejszych etapach prac nad modelem. Dzięki temu możliwe staje się wychwycenie usterek (ang. *faults*) oraz błędnych scenariuszy już na etapie tworzenia modelu. W rezultacie zmniejszają się wydatki projektu o koszta związane z testowaniem oraz powtórną implementacją. Dodatkowo spadają koszta utrzymania całego systemu, ponieważ ewentualne zmiany mogą być dokonywane na jego modelu, symulowane i analizowane zanim dojdzie do ich adaptacji w kodzie. Biorąc powyższe, można powiedzieć, że na podstawie analizy samego modelu i jego symulacji, można wysnuć wnioski prowadzące do usprawnienia całego systemu. Dodatkowo użycie modelu jako formy dokumentacji wpisuje się w manifest powszechnie stosowanych metodyk mięk-

kich (ang. *agile*), który to zakłada stosowanie mniej dokumentacji w projekcie na rzecz rozmowy pomiędzy kompetencjami. Doskonałym punktem wyjścia oraz styku wymiany poglądów na tematy związane z funkcjonowaniem systemu jest właśnie jego model.

Metody formalne

Za metody formalne w inżynierii oprogramowania [43] uważa się oparte na matematyce procesy, służące do:

- modelowania i definiowania wymagań oraz specyfikacji dla danego systemu informatycznego;
- wsparcia implementacji, testowania i weryfikacji poszczególnych elementów oraz systemu uważanego za całość.

Podstawowym założeniem stosowania metod formalnych jest to, że model formalny dokładnie i precyzyjnie określa funkcjonalność oraz cechy systemu. Dzięki temu można zweryfikować poprawność implementacji danego systemu w oparciu właśnie o jego model formalny.

Metody formalne, jako idea, pojawiły się w inżynierii oprogramowania w latach 60-tych XX wieku. Okres ten przypadał również na czas pojawiania się pierwszych mikrokomputerów i co za tym idzie, w naturalny sposób metody formalne zostały zaadresowane jako aparat matematyczny badający poprawność zaimplementowanych na nich systemów informatycznych. Początkowo sądzono, że stosunkowo niskim kosztem oraz z wielką precyzją będzie można przeprowadzać dowody na poprawność działania wszystkich systemów informatycznych. W zderzeniu z rzeczywistością i praktyką okazało się, że procesy formalne zajmują sporą ilość czasu, a brak automatyzacji tych procesów dodatkowo ten czas zwielokrotnia. Dodatkowym niepowodzeniem okazała się również trudność w zdefiniowaniu kompletnego modelu formalnego w większości systemów. Nie istniały też żadne standardy, normy i wytyczne, mogące uwspólnić niektóre cechy formalizmów. Nawet popularne sieci Petriego posiadały kilka możliwych rozwiązań, chociażby dla modelowania systemów z czasem. Nie istniało też żadne szerzej znane oprogramowanie, ani narzędzia, które mogłyby wesprzeć informatyczne rozwiązania komercyjne. W efekcie tych niepowodzeń i rozczarowań metody formalne dość szybko zostały uznane za niezbyt praktyczne i do dzisiaj toczą się dyskusje, czy są one warte dalszej pracy nad nimi [39], [44]. Nie mniej jednak dobrze zaprojektowany model formalny może przyczynić się do zmniejszenia kosztów prac nad rozwijanym systemem. Metody formalne mogą oczywiście być uznane za trudne do opanowania, a co za tym idzie drogie do wdrożenie lecz niekiedy koszty błędów systemu mogą je przewyższać kilkakrotnie, przez co potrafią wymusić użycie tychże metod. Dlatego też, mimo głosów krytycznych, metody formalne są ciągle rozwijane i przyczyniają się do rozwoju inżynierii oprogramowania systemów.

Wraz z rozwojem systemów informatycznych zaczęto dostrzegać i zdawać sobie sprawę z tego, jak bardzo systemy te są ważne ze względu na życie człowieka, ochronę środków i zasobów naturalnych, inte-

lektualnych oraz finansowych. Dostrzeżono, że ewentualne koszty pojawienia się błędów w tych systemach mogą urosnąć do gigantycznych sum. Przykładowo straty ekonomiczne spowodowane awarią sieci energetycznych (ang. *blackout*) mogą doprowadzić dany kraj na skraj bankructwa lub do stanu klęski. W efekcie tego Komisja Europejska wprowadziła normę ITSEC (ang. *Information Technology Security Evaluation Criteria*), w której określono konieczność stosowania metod formalnych w procesie wytwarzania oprogramowania dla czterech z siedmiu poziomów bezpieczeństwa. Dodatkowo w normie ISO od 1999 r. zarekomendowano zastosowanie metod formalnych powyżej piątego poziomu bezpieczeństwa.

W obecnym okresie trudno powiedzieć, że metody formalne zostały na stałe zaadoptowane i są powszechnie używane w komercyjnych projektach informatycznych. Rozwijający się rynek zapotrzebowań na systemy informatyczne, w tym również systemy wbudowane, spowodował, że systemy te powinny powstawać szybko oraz przy minimalnych kosztach projektowania i produkcji. Dodatkowo szybkość reakcji na zmiany, czyli zaadoptowanie zmian od klienta na etapie implementacji, jest również kluczowym aspektem wytwarzania systemu. Kolejnym niezwykle ważnych elementem w dyskusji nad procesem wytwarzania oprogramowania jest ich wykorzystanie w metodyka zwinnych (ang. *agile*). Biorąc pod uwagę, że metodyki te są stosowane w projektach dosyć powszechnie, z punktu widzenia potrzeb dalszego rozwoju metod formalnych, dobrze by było znaleźć niepusty punkt przecięcia w relacjach pomiędzy nimi.

Na pewno kluczowym aspektem w rozwoju metod formalnych są i nadal będą narzędzie wspierające ich zastosowanie. Nawet najlepiej wymyślony i opracowany formalizm będzie komercyjnie bezużyteczny bez odpowiedniego wsparcia od strony dostępnego oprogramowania. Sposób w jaki tworzy się model formalny danego systemu informatycznego ma kluczowe znaczenie dla czasu spędzonego nad projektem oraz poniesionych kosztów. Robiąc go na kartce papieru, na pewno nie przyspieszy się całego procesu, a co za tym idzie opóźni się jego wdrożenie po stronie klienta. Rozwiązaniem są tutaj narzędzia oparte na najnowszych zdobyczach informatycznych, rozumianych jako najnowsze technologie, platformy programistyczne (ang. *frameworks*), algorytmy i zestawy bibliotek programistycznych.

Kolejną kwestią jest oddzielenie matematycznych definicji i pojęć danego formalizmu, od tego co będzie używane bezpośrednio przez końcowego użytkownika, jakim w większości będzie inżynier. Wszelkie aspekty związane z tzw. *doświadczeniem użytkownika* (ang. *User Experience* – UX) powinny być wzięte pod uwagę, przeanalizowane i wdrożone.

Biorąc pod uwagę powyższe rozważania, można stwierdzić, że metody formalne nadają się do tego, by być powszechnie wdrażanymi w procesy produkcji oprogramowania. Należy jednak zastrzec, że nie wszystkie projekty mogą się do tego nadawać a tam, gdzie będzie to możliwe a nawet konieczne, powinno istnieć wsparcie ze strony narzędziowej danego formalizmu.

1.1. Przedstawienie problemu

Język modelowania Alvis [51], [47], [50] powstał z założeniem stworzenia takiego formalizmu, który z jednej strony byłby prosty do opanowania przez inżyniera informatyka, a z drugiej strony stwarzałby możliwość formalnej analizy projektowanego przez niego systemu. Język Alvis został opracowany i jest nadal rozwijany w Katedrze Informatyki Stosowanej na Wydziale Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej Akademii Górniczo-Hutniczej w Krakowie. Główne cechy języka Alvis to:

- możliwość weryfikacji modelu formalnego stworzonego dla danego systemu;
- graficzny język umożliwiający modelowanie struktury tworzonego systemu z uwzględnieniem przepływu informacji i sterowania.
- możliwość graficznej hierarchizacji modelu;
- język wysokiego poziomu definiujący dynamikę elementów składowych (*agentów*) modelu;

Język Alvis powstał jako następca języka XCCS [5], [49], [33], który z kolei rozszerza algebry procesów CCS [35], [22] o możliwość graficznego modelowania połączeń między agentami.

Alvis zapożycza termin *agent* z algebr procesów i przypisuje go każdemu elementowi systemu, który stanowi wybraną, trwającą w czasie funkcjonalność danego systemu. Dodatkowo agent jest niepodzielną jednostką systemu (ang. *entity*), której można przypisać stan. Stan modelu składającego się ze skończonego zbioru agentów jest ciągiem stanów wszystkich agentów.

W celu zdefiniowania dynamiki agentów, zamiast równań algebraicznych, w Alvisie używa się imperatywnego języka wysokiego poziomu oraz języka Haskell [38], [28], [31]. Kod agenta składa się z pojedynczych instrukcji, które dla wersji czasowych Alvisa posiadają zdefiniowany czas ich wykonywania. Cecha ta jest istotna dla niniejszej rozprawy, gdyż umożliwia uwzględnienie aspektów czasowych w modelu. Użycie składni języka Haskell daje projektantowi możliwość zdefiniowania typów danych oraz operatorów i funkcji działających na tych typach.

Warstwa graficzna języka Alvis umożliwia prezentację interfejsów poszczególnych agentów, którymi są odpowiednio porty wejściowe, wyjściowe i dwukierunkowe. Dzięki tym interfejsom agenty mogą się ze sobą komunikować, wymieniać dane lub po prostu synchronizować się na wzajem. Mechanizm komunikacji został oparty na idei spotkań, zaczerpniętej z języka Ada [21], [15]. Agent inicjalizujący spotkanie czeka, aż inny agent dołączy i zacznie z nim wymieniać dane lub zsynchronizuje się. Z punktu widzenia tej rozprawy ważną kwestią jest dostępność dla agentów zasobów procesora. Agent, który je przejął i czeka na reakcję ze strony innego agenta, przechodzi w stan oczekiwania. W przypadku portów wejściowych agent oczekuje wartości o konkretnym typie lub sygnału, w przypadku komunikacji polegającej na synchronizowaniu się agentów. Jeżeli komunikacja odbywa się na porcie wyjściowym agenta, oznacza to, że agent próbuje wysłać wartość danego typu, lub sygnał. Podobnie jak dla portu wejściowego agent również oczekuje na agenta,

który pobierze tą wartość albo sygnał z portu wyjściowego agenta, który tą komunikację rozpoczął. Mamy tutaj do czynienia ze współbieżnym wykonywaniem akcji przez agenty, ponieważ agent rozpoczynający spotkanie zwalnia zasoby procesora, w celu umożliwienia przejęcia tychże przez innego agenta i dokończenia spotkania. Dodatkowo agenty posiadają swoje priorytety, co sprawia, że agent, który jest oczekiwany, wcale nie musi być tym, który w danym momencie czasowym zostanie wypromowany do stanu aktywnego, umożliwiającego mu przejęcie zasobów procesora i co za tym idzie wykonywanie swoich akcji.

Opisany powyżej rodzaj komunikacji jest określany jako *komunikacja blokująca*, ponieważ agent, który tą komunikację zaczyna, jest blokowany do momentu czasu, w którym inny agent ją podejmie. W Alvisie istnieje także rodzaj *komunikacji nieblokującej* niezwykle istotnej z punktu widzenia tej rozprawy, ponieważ uwzględnia ona zjawisko upływu czasu [34]. Komunikacja ta polega na tym, że agent inicjalizujący komunikację, odczekuje przez czas równy skończonej liczbie jednostek czasu, na odpowiedź ze strony innego agenta. Jeżeli takowa nie nastąpi, komunikacja jest przerywana i agent, o ile nie został wywłaszczony przez innego agenta, zaczyna procesować kolejną instrukcję.

Dzięki warstwie graficznej projektant może również zadecydować o hierarchicznych zależnościach pomiędzy elementami systemu. Cecha ta wprowadza do modelu modułowość systemu, czyli pozwala uwzględnić potencjalne podsystemy danego systemu.

Kolejną ważną składową języka Alvis, z uwagi na weryfikację systemu, jest warstwa systemowa modelu. Pozwala ona zdefiniować kwestie sprzętowe projektowanego systemu, co ma istotny wpływ na budowany docelowo etykietowany system przejść. Poszczególne wersje systemowe różnią się od siebie poprzez to, że definiują określoną liczbę dostępnych procesorów oraz uwzględniają, bądź nie kwestie związane z domeną czasu. Wersje warstwy systemowej przyjęto oznaczać symbolem α_s^n, gdzie:

- $n \in \mathbb{N}$ oznacza liczbę procesorów dostępnych dla danej platformy sprzętowej;
- s jest nazwą danej wersji, która odróżnia wersje, w przypadku gdy mamy do czynienia z wersjami, które mają tą samą ilość dostępnych procesorów. Jest to uzasadnione tym, że nie ma takiego wymagania, aby istniała tylko jedna wersja dla np. platform jednoprocesorowych. Przykładowo projektant może stworzyć dwie wersje $\alpha_{s_1}^3$ i $\alpha_{s_2}^3$, które różnią się sposobem szeregowania agentów, ale obie będą przystosowane do platform sprzętowych opartych na trzech procesorach.

Warstwa podstawowa w Alvisie jest oznaczana jako α^0. Warstwa ta każdemu agentowi w modelu przydziela swobodny dostęp do procesora. Oznacza to, że agenty ani przez chwilę nie rywalizują pomiędzy sobą o dostęp do zasobów procesora. W warstwie tej główny nacisk kładziony jest na współbieżność wykonywanych akcji.

Warstwa systemowa języka modelowania Alvis pozwala wyznaczyć *etykietowany system przejść*, który stanowi reprezentację przestrzeni stanów danego modelu.

1.2. Cel badań i teza pracy

Jako podstawowy cel podjętych badań przyjęto próbę opracowania formalnego opisu modeli w języku Alvis, stosujących warstwę systemową α^1_{FPPS}. Za nazwę tej wersji wzięto nazwę algorytmu szeregującego (ang. *scheduler*) – *Fixed Priority Preemptive Scheduling*. Poszczególne człony tej nazwy określają następujące założenia algorytmu szeregowania agentów w warstwie α^1_{FPPS}:

- *Fixed Priority* – stały, ustalony priorytet agenta, niezmienny w czasie;
- *Preemptive Scheduling* – zarządzanie dostępem do zasobów procesora poprzez wywłaszczanie jednych agentów na rzecz drugich, posiadających większy priorytet.

Pod względem sprzętowym warstwa ta ma wspierać modelowanie systemów wbudowanych czasu rzeczywistego uruchomionych na platformach jednoprocesorowych, a więc takich, w których istnieje ciągła rywalizacja pomiędzy agentami o przejęcie kontroli nad zasobami procesora.

W celach weryfikacji modelu powinien zostać opisany algorytm tworzenia *etykietowanego systemu przejść* dla modeli z warstwą systemową α^1_{FPPS}. Dodatkowo zaproponowane rozwiązanie ma być zgodne z podstawowymi założeniami języka Alvis i wpisywać się w obowiązujące rozwiązania i definicje. Oznacza to, że powstała warstwa systemowa α^1_{FPPS} byłaby rozszerzeniem dla już istniejących warstw – α^0 i jej wersji czasowej.

Formalnie tezę rozprawy zdefiniowano następująco:
Język Alvis, wsparty odpowiednimi narzędziami komputerowymi, może być efektywnie użyty do modelowania systemów wbudowanych, umożliwiając jednocześnie formalną analizę modelu z zastosowaniem metod i narzędzi typowych dla technik weryfikacji modelowej.

Dla uzasadnienia tezy w rozprawie pokazano, że warstwa systemowa α^1_{FPPS} rozszerza język modelowania Alvis w następujących aspektach:

- możliwe jest modelowanie systemów wbudowanych czasu rzeczywistego działających na platformie jednoprocesorowej;
- algorytm szeregowania zadań uwzględnia priorytety występujących w modelu agentów i na tej podstawie dokonuje wywłaszczania agentów o niższych priorytetach na rzecz tych agentów, których priorytety są wyższe;
- możliwe jest wygenerowanie *etykietowanego systemu przejść*, który stanowi reprezentację przestrzeni stanów dla modelu systemu wbudowanego czasu rzeczywistego.

1.3. Zawartość pracy

Niniejsza rozprawa została zredagowana w ten sposób, aby wpierw przedstawić pokrótce ogólne założenia języka modelowania Alvis, czyli na czym polega tworzenie modelu projektowanego systemu, jak definiowany i rozumiany jest stan modelu i jak przebiegają tranzycje pomiędzy poszczególnymi stanami. W trakcie prezentowania ogólnych twierdzeń i definicji wpleciono te związane z nową wersją czasową warstwy systemowej – α^1_{FPPS}. Zaprezentowano propozycję algorytmu szeregowania dla tej warstwy i przedstawiono jego definicję. Dużą rolę przywiązano do tych aspektów modelu formalnego, które są związane ze zjawiskiem upływu czasu. Elementem wieńczącym prezentację idei nowej warstwy systemowej α^1_{FPPS} jest przedstawienie algorytmu generowania *etykietowanego systemu przejść* dla tej warstwy. Rozprawa zawiera także studium dwóch przypadków praktycznego tworzenia modeli dla wybranych systemów wbudowanych czasu rzeczywistego. Pod koniec rozprawy przedstawiono porównanie języka Alvis, poszerzonego o nową wersję warstwy systemowej, z innymi popularnymi formalizmami do modelowania systemów z czasem.

Zawartość poszczególnych rozdziałów i dodatków (z pominięciem wstępu) kształtuje się następująco:

- **Rozdział 2** zawiera krótkie wprowadzenie do języka modelowania Alvis. Przedstawiono w nim na czym polega koncepcja tworzenia modelu i jak wygląda jego trójwarstwowa struktura – *warstwa diagramów komunikacji*, *warstwa kodu* i *warstwa systemowa*. Opisano znaczenie każdej warstwy. Wyjaśniono jak modelowany jest agent w warstwie graficznej i jak definiuje się jego dynamikę w warstwie kodu. Przestawiono dwa rodzaje agentów: *aktywne* i *pasywne* oraz wyjaśniono różnice pomiędzy nimi. Następnie pokazano czym są, jakie są rodzaje i jak są używane porty agentów służące do wymiany danych oraz synchronizacji pomiędzy agentami. Rozdział zawiera również formalną definicję niehierarchicznego diagramu komunikacji oraz opis instrukcji dostępnych w języku Alvis. Na koniec zaprezentowano proces transformacji od modelu w języku Alvis do reprezentacji przestrzeni stanów w postaci *etykietowanego systemu przejść*.
- **Rozdział 3** jest pierwszym rozdziałem, w którym prezentowane są oryginalne idee zawarte w rozprawie, dotyczące nowej warstwy systemowej α^1_{FPPS}. Na jego wstępie przedstawiono ogólną definicję modelu w języku Alvis. Rozwijając to zagadnienie, zaprezentowano definicję stanu agenta, omawiając kolejno, jakie wyróżnia się tryby pracy agenta, jak obliczany jest licznik rozkazów, czym jest lista kontekstowa agenta i jakie informacje zawiera oraz jak zarządza się parametrami zdefiniowanymi w kodzie danego agenta. Idąc w kierunku nowej warstwy systemowej α^1_{FPPS}, przedstawiono jej ideę oraz główne założenia. W następstwie tego zaprezentowano pomysł oraz formalną definicję algorytmu szeregowania agentów. Część ta zawiera główne elementy algorytmu jak dwuwymiarowa kolejka FIFO oraz proces kolejkowania w niej agentów. Wyjaśniono ważną kwestię związaną z zarządzaniem pracą agentów – ideę *przeplotu*, czyli wywłaszczenie agenta w trakcie wykonywania dłu-

gotrwałej instrukcji. Na koniec zaprezentowano ogólną definicję stanu modelu w Alvisie, jako zbioru stanów poszczególnych agentów oraz zdefiniowano stan początkowy dla nowej warstwy systemowej α^1_{FPPS}.

- **Rozdział 4** prezentuje listę możliwych tranzycji pomiędzy stanami agenta z uwzględnieniem tranzycji ogólnych oraz systemowych, w których to znajdują się specyficzny tranzycje związane z warstwą systemową α^1_{FPPS}. Do tej grupy należy tranzycja związana z wystąpieniem przerwania systemowego, określanego jako $SysTick$ oraz tranzycja związana z upływem czasu. W przypadku przerwania systemowego przedstawiono jego wpływ na uruchamianie algorytmu szeregującego pracę agentów. Opisano również zależność tego algorytmu od sekcji krytycznej, modelowanej za pomocą instrukcji `critical`. Prezentując tranzycje, przedstawiono zależności pomiędzy agentami pasywnymi i aktywnymi. Pokazano tu, jak agenty pasywne uruchamiane są w kontekście akcji wykonywanych przez agenty aktywne.
- **Rozdział 5** zawiera formalną definicję *etykietowanego systemu przejść* (ang. *Labeled Transition System*, krótko: graf LTS), będącego reprezentacją przestrzeni stanów osiągalnych dla danego modelu. Główną częścią tego rozdziału i zarazem bardzo kluczową dla całej rozprawy, jest opis algorytmu generowania grafu LTS dla systemów modelowanych za pomocą warstwy systemowej α^1_{FPPS}.
- **Rozdział 6** przedstawia studium przypadków praktycznego wykorzystania języka Alvis z warstwą systemową α^1_{FPPS} do modelowania systemów wbudowanych czasu rzeczywistego.
- **Rozdział 7** jest próbą porównania języka modelowania Alvis, rozszerzonego o warstwę systemową α^1_{FPPS}, z innymi popularnymi formalizmami do modelowania tego typu systemów. Wzięto tu pod uwagę automaty czasowe oraz czasowe kolorowane sieci Petriego.
- **Rozdział 8** zawiera podsumowanie pracy. Poza wnioskami końcowymi zawarto w nim również perspektywy dalszych badań i rozwoju dla języka modelowania Alvis.

2. Wprowadzenie do języka Alvis

Alvis [47], [51], [50] jest formalnym językiem modelowania, który został opracowany w Katedrze Informatyki Stosowanej na Wydziale Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej Akademii Górniczo-Hutniczej w Krakowie. Głównym celem projektu o nazwie *Alvis* jest dostarczenie języka modelowania, który byłby łatwy do stosowania przez inżynierów oprogramowania do formalnej weryfikacji systemów współbieżnych. Jednocześnie założono, że oprogramowanie wspierające stosowanie języka Alvis powinno współpracować z popularnymi systemami do weryfikacji modelowej, jak nuXmv [16] lub CADP toolbox [24].

2.1. Model

Język Alvis został opracowany przede wszystkim do modelowania systemów współbieżnych. Model w tym języku jest systemem składającym się z komponentów nazywanych *agentami*. *Agenty* mogą operować równoległe z innymi *agentami*, komunikować się ze sobą, rywalizować o kontrolę nad obiektami współdzielonymi itp. W modelu takim można wyróżnić trzy warstwy, przy czym tylko dwie pierwsze są projektowane przez użytkownika.

Warstwa graficzna – nazywana *diagramem komunikacji* [50], jest używana do opisu struktury modelowanego systemu z punktu widzenia kontroli i przepływu danych pomiędzy agentami.

Warstwa kodu – jest używana do opisu zachowania poszczególnych agentów. Zachowanie agenta jest definiowane poprzez zbiór instrukcji, które w swej składni podobne są do tych stosowanych w typowych językach programowania wysokiego poziomu (zob. sekcja 2.3).

Warstwa systemowa – jest warstwą predefiniowaną i dostarcza informacji na temat środowiska uruchomianego (systemy jedno- lub wieloprocesorowe).

Podstawową warstwą systemową w języku Alvis jest warstwa o nazwie α^0. W definicji tej warstwy założono, że liczba dostępnych procesorów jest nielimitowana i z tego względu nadaje się do modelowania zachowania tzw. platform wieloprocesorowych. W praktyce oznacza to, że każdy aktywny agent zdefiniowany w tej warstwie ma nieograniczony dostęp do własnego procesora. Procesor, w ujęciu definicji tej

warstwy systemowej, nie jest obiektem współdzielonym. Taki rodzaj zdefiniowania warstwy systemowej daje możliwość użycia języka Alvis w modelowaniu systemów współbieżnych zamiast innych popularnych formalizmów, np. sieci Petriego [46], [30], automatów czasowych [58], [8], [3] lub algebr procesów [35], [11], [49].

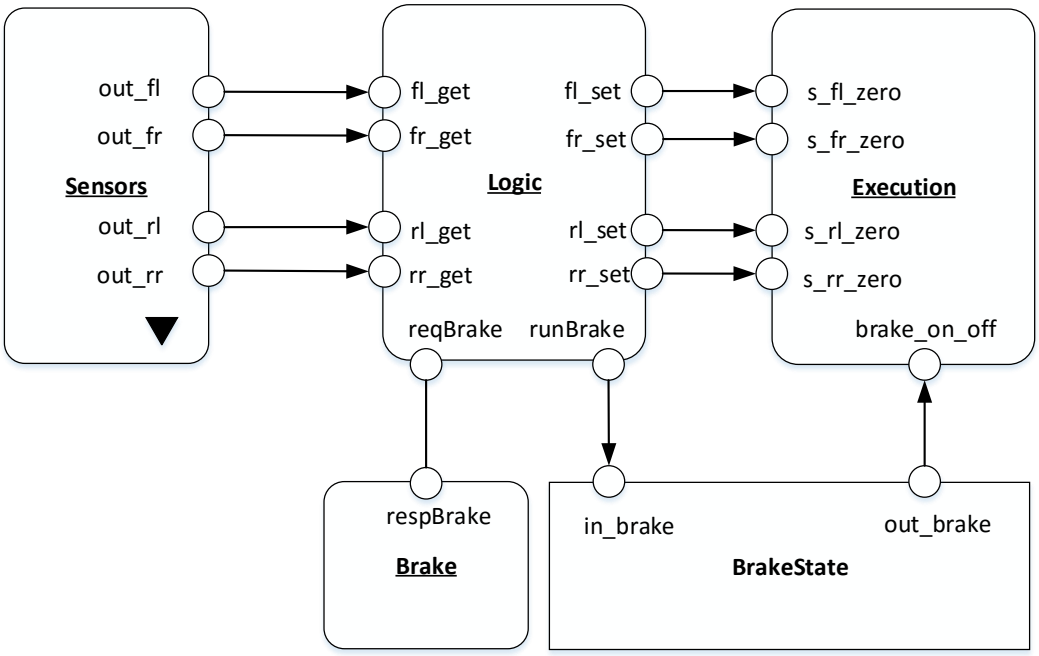
Jednym z podstawowych celów niniejszej rozprawy doktorskiej jest zdefiniowanie i opisanie warstwy systemowej α^1_{FPPS}, która w odróżnieniu od warstwy α^0 służyłaby do modelowania systemów współbieżnych w środowisku jednoprocesorowym. W takim modelu procesor jest obiektem współdzielonym przez wszystkie zdefiniowane agenty. W praktyce oznacza to, że agenty danego systemu muszą rywalizować pomiędzy sobą o dostęp do procesora, ponieważ w danym momencie czasu tylko jeden agent może wykonywać swoje instrukcje, podczas gdy inne agenty oczekują, aż procesor zostanie zwolniony. Rolą algorytmu szeregującego (ang. *scheduler*) jest arbitraż pomiędzy agentami, rywalizującymi o dostęp do zasobów procesora takiego systemu współbieżnego.

2.2. Diagramy komunikacji

Diagram komunikacji jest graficzną reprezentacją modelu w języku Alvis. Warstwa przyjmuje formę grafu skierowanego z węzłami, które reprezentują *agenty* i krawędziami, które służą do graficznego modelowania kanałów komunikacyjnych pomiędzy nimi. *Agent* stanowi kluczowy element w koncepcji języka Alvis. Służy do opisu dowolnego rozróżnialnego elementu modelowanego systemu z uwzględnieniem stanu tego elementu. Z punktu widzenia semantyki języka Alvis agenty podzielone są na dwie grupy: *agenty aktywne* oraz *agenty pasywne*. Główną różnicą pomiędzy tymi grupami jest to, że agenty aktywne, w przeciwieństwie do agentów pasywnych, mogą przejmować kontrolę nad zasobami procesora. Agenty pasywne mogą jedynie być wywoływane na rzecz agentów aktywnych, jako procedury realizujące pewne założone funkcjonalności. Dodatkowo agenty pasywne mogą być wywoływane kaskadowo przez inne agenty pasywne.

Różnica pomiędzy dwoma typami agentów występuje również w ich graficznej reprezentacji. Agenty aktywne przedstawiane są jako prostokąty z zaokrąglonymi rogami, a agenty pasywne mają reprezentację prostokątów z rogami ostrymi. Dla obu typów agentów istnieje ta sama zasada co do ich nazw. Muszą być to nazwy unikatowe w zakresie całego modelu i muszą się zaczynać od wielkiej litery. Przykład graficznej reprezentacji agentów przedstawiono na rysunku 2.1. Na umieszczonym tam diagramie pokazano m.in. agenty aktywne: *Sensors*, *Logic* i *Execution* oraz agenta pasywnego *BrakeState*.

Komunikacja pomiędzy agentami jest możliwa jedynie poprzez ich *porty*. Połączenie takie nazywane jest *kanałem komunikacyjnym* i realizowane jest pomiędzy dwoma różnymi agentami. Dla obu typów agentów porty reprezentowane są w ten sam sposób – jako koła o tej samej średnicy. Nazwa portu musi być unikatowa, ale w odróżnieniu od nazwy agenta jej unikatowość odnosi się jedynie do zakresu agenta, w



Rysunek 2.1: Przykład modelu zawierającego trzy agenty aktywne, jednego agenta pasywnego i jednego agenta hierarchicznego.

którym dany port występuje. W praktyce oznacza to, że mogą istnieć dwa agenty o takich samych nazwach zdefiniowanych dla ich portów. Nazwa portu musi zaczynać się od małej litery.

W zależności od zdefiniowanego zachowania agenta i zastosowanych połączeń port może być sklasyfikowany jako *wejściowy*, *wyjściowy* lub *dwukierunkowy*. Port wejściowy służy do pobierania danych lub sygnałów, a port wyjściowy do ich wysyłania do innych agentów. W przypadku agentów pasywnych porty mogę mieć zdefiniowane procedury (reprezentują usługi). Takie *porty proceduralne* muszą być określone jako wejściowe, albo wyjściowe.

Model w Alvisie dopuszcza dwa rodzaje kanałów komunikacyjnych: jednokierunkowe i dwukierunkowe. Jednokierunkowe kanały komunikacyjne są reprezentowane za pomocą linii zakończonej grotem – grot wskazuje kierunek przepływu danych pomiędzy agentami. Przykładem takiego połączenia może być komunikacja pomiędzy agentami *Sensors*, *Logic* i *Execution*, zaprezentowana na rysunku 2.1. Dwukierunkowe kanały komunikacyjne są formalnie parą kanałów jednokierunkowych o przeciwnych kierunkach przepływu informacji. Połączenia takie są przedstawiane jako linia bez grotu. Przykładem połączenia dwukierunkowego jest kanał komunikacyjny między agentami *Brake* i *Logic* na rysunku 2.1.

Komunikacja z portem proceduralnym agenta pasywnego musi być zdefiniowana jako jednokierunkowa, a kierunek tej komunikacji zależy od rodzaju danej procedury. Przykładem takiego połączenia jest wywołanie *procedury wejściowej in-brake* agenta pasywnego *BrakeState* przez agenta *Logic* (rysunek 2.1) oraz

wywołanie *procedury wyjściowej out-brake* przez agenta *Execution*.

Niech $\mathcal{P}(X)$ oznacza zbiór portów agenta X. Można rozróżnić następujące podzbiory zbioru $\mathcal{P}(X)$ [50]:

- $\mathcal{P}_{in}(X)$ oznacza zbiór *portów wejściowych* agenta X.
- $\mathcal{P}_{out}(X)$ oznacza zbiór *portów wyjściowych* agenta X.
- $\mathcal{P}_{proc}(X)$ oznacza zbiór *portów portów proceduralnych* agenta pasywnego X – porty zdefiniowane ze słowem kluczowym *proc* (zob. sekcja 2.3).

Powyższe oznaczenia można uogólnić na zbiory agentów. Dla zbioru agentów W przyjmujemy $\mathcal{P}(W) = \bigcup_{X \in W} \mathcal{P}(X)$, $\mathcal{P}_{in}(W) = \bigcup_{X \in W} \mathcal{P}_{in}(X)$ itd.

Ponadto symbolem $\mathcal{P}$ oznaczamy zbiór wszystkich portów zdefiniowanych w modelu, $\mathcal{P}_{in}$ oznaczamy zbiór wszystkich portów wejściowych zdefiniowanych w modelu itd.

Niech $X.p$ oznacza port p agenta X. Jeśli nazwa agenta nie będzie wymagana w rozważaniach teoretycznych, zostanie pominięta. Niehierarchiczny diagram komunikacji definiujemy następująco [50]:

Definicja 2.1. *Niehierarchiczny diagram komunikacji* jest krotką $D = (\mathcal{A}, \mathcal{C}, \sigma)$, gdzie: $\mathcal{A} = \{X_1, \ldots, X_n\}$ jest zbiorem *agentów* składającym się z dwóch rozłącznych podzbiorów, $\mathcal{A}_A$, $\mathcal{A}_P$ takich, że $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$, zawierających odpowiednio agenty *aktywne* oraz *pasywne*; $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$ jest *relacją komunikacji* taką, że:

$$\forall_{X \in \mathcal{A}} (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C} = \emptyset, \tag{2.1}$$

$$\mathcal{P}_{proc} \cap \mathcal{P}_{in} \cap \mathcal{P}_{out} = \emptyset, \tag{2.2}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_A) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow q \in \mathcal{P}_{proc}, \tag{2.3}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_A)) \cap \mathcal{C} \Rightarrow p \in \mathcal{P}_{proc}, \tag{2.4}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \ \Rightarrow (p \in \mathcal{P}_{proc} \wedge q \notin \mathcal{P}_{proc}) \vee (q \in \mathcal{P}_{proc} \wedge p \notin \mathcal{P}_{proc}). \tag{2.5}$$

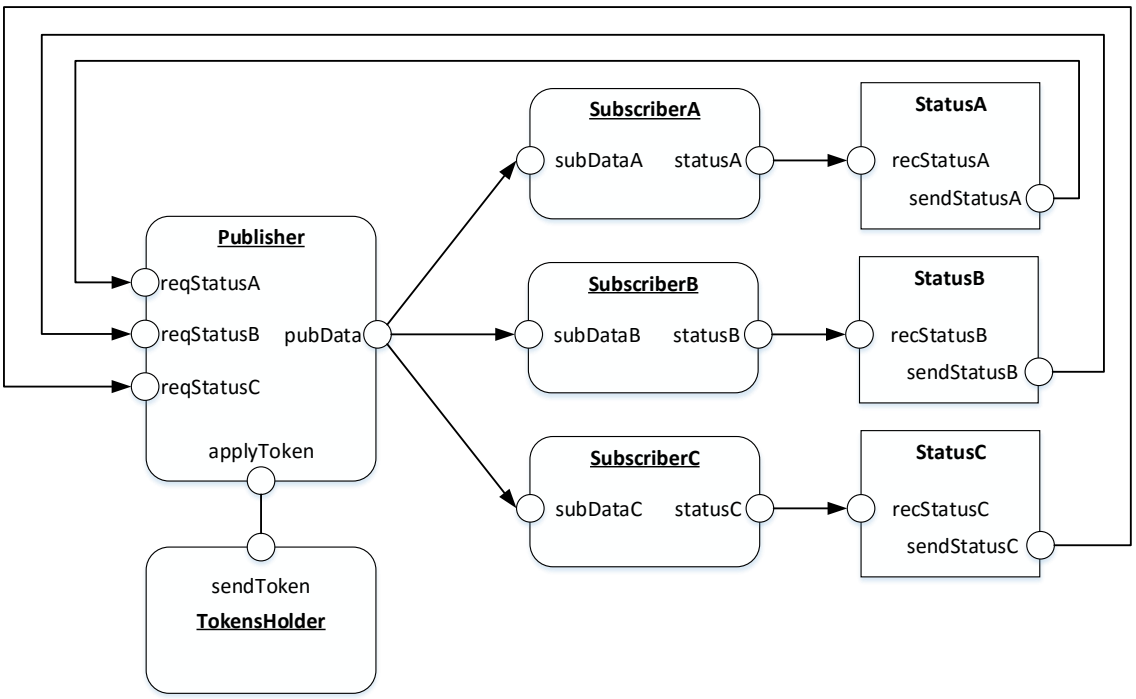
oraz $\sigma \colon \mathcal{A}_A \rightarrow \{False, True\}$ jest *funkcją aktywności*, która wskazuje, które agenty aktywne są uruchamiane przy starcie.

Powyższa definicja niehierarchicznego diagramu komunikacji ustanawia następujące zasady:

- Kanały komunikacyjne nie mogą być definiowane pomiędzy portami należącymi do tego samego agenta (2.1).
- Porty proceduralne agenta pasywnego mogą być definiowane jedynie jako wejściowe albo wyjściowe (2.2).
- Kanał komunikacyjny pomiędzy agentem aktywnym i agentem pasywnym może być ustalony jedynie z portem proceduralnym (2.3), (2.4).
- Kanał komunikacyjny pomiędzy agentami pasywnymi może być ustalony jedynie poprzez wywołanie procedury z portu nieproceduralnego (2.5).

- Kanał komunikacyjny który zawiera agenta pasywnego może być zrealizowany jedynie jako jednokierunkowy (2.3), (2.4), (2.5).

Przykład niehierarchicznego diagramu komunikacji zaprezentowano na rysunku 2.2. W modelu tym agent aktywny *Publisher* wysyła dane na swój wyjściowy port *pubData*, a subskrybenci w postaci agentów aktywnych SubscriberA, SubsciberB i SubscriberC konsumują te dane na swoich portach wejściowych, kolejno subDataA, subDataB i subData poprzez jednokierunkowy kanał komunikacyjny. Dodatkowo każdy subskrybent oblicza swój status na podstawie otrzymywanych danych wejściowych i zapisuje go, poprzez wywołanie procedury na porcie proceduralnym powiązanego agenta pasywnego.

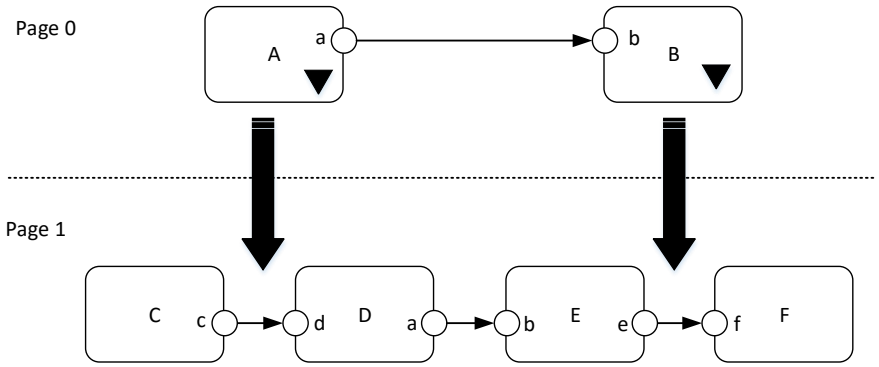


Rysunek 2.2: Przykład niehierarchicznego diagramu komunikacji

Zanim aktywny agent Publisher wyśle dane do swoich subskrybentów weryfikuje najpierw status każdego z nich, poprzez wywołanie procedur agentów pasywnych kolejno: StatusA, StatusB i StatusC. Cała komunikacja z agentami pasywnymi realizowana jest jako jednokierunkowe kanały komunikacyjne. W kolejnym kroku aktywny agent *Publisher* decyduje, jaki rodzaj danych może być wysłany do poszczególnych subskrybentów. Decyzję tą ustala na podstawie obliczeń, których argumentem jest informacja o statusie poszczególnego subskrybenta, otrzymana poprzez wywołanie procedury na powiązanym z danym subskrybentem agencie pasywnym. Struktura danych, które są przesyłane od aktywnego agenta Publisher do subskrybentów, zawiera *token*, który z kolei określa dozwolonych odbiorców tych danych. Na podstawie wartości *tokena* dany subskrybent zostaje poinformowany, czy przesyłane dane są dozwolone dla niego, czy

nie. Może zaistnieć przypadek, w którym dla przykładu aktywny agent SubscriberB może otrzymać dane na swój port wejściowy lecz wartość *tokena* będzie jednoznacznie określała, że nie są to dane przeznaczone dla niego i co za tym idzie, nie powinien brać ich pod uwagę przy obliczaniu swojego statusu. W prezentowanym przykładzie dopuszcza się kilka możliwych schematów w jakich definiowany jest dany *token*. W celu zapoznania się z obecnie obowiązującym schematem *tokena*, aktywny agent *Publisher* odpytuje aktywnego agenta *TokensHolder* poprzez dwukierunkowy kanał komunikacyjny. Warto zwrócić uwagę, że w prezentowanym modelu wszystkie nazwy agentów są unikatowe oraz nie ma takiej sytuacji, w której kanał komunikacyjny ustanowiony jest pomiędzy portami należącymi do tego samego agenta.

Diagram komunikacyjny może być również przedstawiony w formie hierarchicznej, gdzie poszczególne części diagramu są rozmieszczone pomiędzy diagramami hierarchiczne od siebie zależnymi. Diagramy te nazywane są *stronami* (ang. *page*). Strona jest reprezentowana na wyższym poziomie przez agenta hierarchicznego (agent z ikoną w kształcie czarnego trójkąta).



Rysunek 2.3: Przykład hierarchicznego diagramu komunikacyjnego

Detaliczność informacji zawartych w modelu zwiększa się w kierunku od stron umieszczonych najwyżej w hierarchii do tych położonych najniżej. Szczegółowość wzrasta, ponieważ agent hierarchiczny prezentowany na wyższej stronie jest zastępowany przez odpowiadająca mu podstronę. Niehierarchiczny diagram komunikacji traktowany jest jako pojedyncza strona bez żadnych zależności hierarchicznych od innych stron modelu. Dowolny hierarchiczny diagram komunikacji można zamienić na równoważny mu diagram niehierarchiczny [50], dlatego, aby uprościć rozważania teoretyczne, można ograniczyć się do diagramów niehierarchicznych.

Cele poniższej rozprawy doktorskiej nie wymagają analizy modeli hierarchicznych. Cała uwaga zostanie skupiona na niehierarchicznych modelach komunikacji.

2.3. Warstwa kodu

Warstwa kodu stosowana jest do definiowania zachowania agentów. Język modelowania Alvis dostarcza stosunkowo mały zbiór instrukcji [54], a w zakresie definiowania danych i manipulowaniu nimi wykorzystuje język funkcyjny Haskell [38]. Uproszczona składnia instrukcji języka Alvis dostępna dla warstwy systemowej α^1_{FPPS} zaprezentowana jest na listingu 2.1. W opisie składni zastosowano poniższe oznaczenia:

- `A` reprezentuje nazwę agenta,
- `p` reprezentuje nazwę portu,
- `x` reprezentuje parametr,
- `g`, `g1`, `g2`,... reprezentują warunki logiczne,
- `expression` reprezentuje wyrażenie języka funkcyjnego Haskell,
- `t` reprezentuje liczbę całkowitą (sumę jednostek czasu).

Listing 2.1: Uproszczona składnia instrukcji języka Alvis

```
1  critical {... null; }
2  delay t;
3  exec x = expression;
4  exit;
5  in p x;
6  in (t) p x;
7  in (t) p x {
8    success {...}
9    fail {...} }
10 jump label;
11 loop {...}
12 loop (g) {...}
13 loop (every t) {... null; }
14 null;
15 out p x;
16 out (t) p x;
17 out (t) p x {
18   success {...}
19   fail {...} }
20 proc (g) p {...}
21 select {
22   alt (g1) {...}
23   alt (g2) {...}  ... }
24 start A;
```

- (1) Sekcja krytyczna *critical* wymusza wykonanie sekwencji instrukcji w niej zawartych, bez względu na zaistnienie jakiegokolwiek przerwania zewnętrznego. Koniec sekcji krytycznej musi być wskazany przez instrukcję *null*, jako ostatnią z instrukcji zdefiniowanych w bloku ograniczonym nawiasami klamrowymi.
- (2) Instrukcja *delay* wstrzymuje egzekucję danego agenta na zdefiniowaną liczbę jednostek czasu t.
- (3) Instrukcja *exec* notyfikuje wykonywanie kalkulacji wyrażenia i przypisania wyniku do parametru. To słowa kluczowe może być pominięte bez wpływu na semantykę tej instrukcji.
- (4) Instrukcja *exit* oznacza zakończenie działania agenta aktywnego lub koniec wykonywania procedury agenta pasywnego.
- (5) Instrukcja *in* reprezentuje pobranie przez agenta wartości przez port p i przypisanie tej wartości do parametru x. Jeżeli instrukcje jest stosowana wyłącznie do pobrania sygnału sterującego (sygnał bez określonej wartości) to nazwa parametru nie jest stosowana. Jest to tzw. *wersja blokująca* tej instrukcji, co oznacza, że agent, który zainicjował tę komunikację, będzie oczekiwał tak długo, aż inny agent zewnętrzny dostarczy wartość na jego port wejściowy p.
- (6) Jest to tzw. *wersja nieblokująca* instrukcji *in*. Działa podobnie jak poprzednia, ale agent, który zainicjował tę komunikację, będzie oczekiwał na otrzymanie danej na swoim porcie p przez zadany okres czasu t wyrażony w jednostkach czasu. Jeśli podczas tego okresu t nie pojawi się żadna dana na porcie p, to agent zacznie wykonywać następną instrukcję w bloku swojego kodu.
- (7–9) jest to rozszerzona składnia nieblokującej instrukcji *in*. Rezultat wykonania tej instrukcji jest podobny do poprzedniego. Rozszerzenie polega na tym, że w zależności od sukcesu lub jego braku w realizacji komunikacji, wykonywany jest dodatkowy blok instrukcji (np. obsługa wyjątku). Każda z dodatkowych klauzul jest opcjonalna tj. można wykorzystać tylko klauzulę *success* lub tylko klauzulę *fail*.
- (10) Instrukcja *jump* jest instrukcją skoku. W wyniku jej wykonania sterowanie przenoszone jest do miejsca wskazanego przez etykietę. Następnie agent realizuje pierwszą instrukcję umieszczoną po etykiecie. Sama etykieta nie może być umieszczona bezpośrednio przed klamrą zamykającą.
- (11) *Bezwarunkowa* instrukcja *loop* wykonuje sekwencję instrukcji w niej zawartych nieskończoną liczbę razy.
- (12) *Warunkowa* instrukcja *loop* działa podobnie jak poprzednia, ale zawartość pętli jest wykonywana tak długo, jak długo spełniony jest warunek logiczny g. Warunek logiczny jest wyrażeniem logicznym zapisanym w języku funkcyjnym Haskell.
- (13) *Okresowa* instrukcja *loop* realizuje nieskończoną liczbę powtórzeń swojej zawartości, przy czym kolejne przebiegi pętli realizowane są co t jednostek czasu. Instrukcja *null* jest wymagana dla pętli okresowych i sygnalizuje koniec pojedynczej iteracji pętli.

- (14) Instrukcja *null* jest instrukcją *pustą*. Służy ona m.in. do notyfikowania o końcu pojedynczej iteracji pętli *loop every* oraz końcu sekcji krytycznej. Instrukcję pustą stosuje się również w przypadkach, gdy składnia języka Alvis wymaga jakiejkolwiek instrukcji, a nie chcemy umieszczać tam żadnego konkretnego kodu, np. po etykiecie instrukcji skoku umieszczonej na końcu głównego bloku definiującego dynamikę agenta.
- (15) Instrukcja *out* reprezentuje wysłanie przez agenta wartości parametru x przez port p. Jeżeli instrukcja jest stosowana wyłącznie do wysłania sygnału sterującego, to nazwa parametru nie jest stosowana. Jest to tzw. *wersja blokująca* tej instrukcji, co oznacza, że agent, który zainicjował tę komunikację, będzie oczekiwał tak długo, aż inny agent zewnętrzny odbierze wartość z jego portu wyjściowego p.
- (16) Jest to tzw. *wersja nieblokująca* instrukcji *out*. Działa podobnie jak poprzednia, ale agent, który zainicjował tę komunikację, będzie oczekiwał na pobranie danych ze swojego portu p przez zadany okres czasu t wyrażony w jednostkach czasu. Jeśli podczas tego okresu t inny agent nie pobierze danych, to agent zacznie wykonywać następną instrukcję w bloku swojego kodu.
- (17–19) jest to rozszerzona składnia nieblokującej instrukcji *out*, analogiczna do instrukcji z linii 7–9.
- (20) Instrukcja *proc* definiuje procedurę agenta pasywnego. Parametr p określa nazwę portu proceduralnego. Warunek logiczny g musi być spełniony, aby procedura była dostępna dla innych agentów. Warunek logiczny jest wyrażeniem logicznym napisanym w języku funkcyjnym Haskell.
- (21–23) Instrukcja *select* jest instrukcją wielowariantowego wyboru. Każda alternatywa (zwana *gałęzią*) reprezentowana jest poprzez słowo kluczowe *alt*. Wykonywana jest pierwsza (licząc w kolejności umieszczenia kodu) gałąź ze spełnionym warunkiem. Warunek logiczny jest wyrażeniem logicznym napisanym w języku funkcyjnym Haskell. Jeżeli żaden z warunków nie jest spełniony, realizowany jest kod po instrukcji *select*.
- (24) Instrukcja *start* daje możliwość uruchomienia innego agenta, który znajduje się w trybie *init*.

Listing 2.2: Struktura bloku agenta

```
agent AgentName (priority)
{
  -- declaration of parameters
  -- agent body
}
```

Warstwa kodu jest sekwencją bloków agentów (Listing 2.2). Jeden blok może być współdzielony przez wielu agentów. W tym przypadku nazwy tych agentów oddzielone przecinkami muszą być umieszczone po słowie kluczowym *agent*. Zakres priorytetów agenta należy do zbioru liczb całkowitych od 0 do 9. Zero jest najwyższym priorytetem.

Listing 2.3: Przykład definicji agenta aktywnego

```
agent A(0) {
  i :: Int = 0;
  loop {                          -- 1
    in a i;                       -- 2
    select {                      -- 3
      alt (i < 0) {
        exec i = 0;               -- 4
        out b i; }                -- 5
      alt (i >= 0) {
        out b i; }                -- 6
    }
  }
}
```

Rozważmy definicję aktywnego agenta A zaprezentowaną na listingu 2.3. Na początku definiowany jest parametr i typu całkowitego i deklarowana jest jego wartość początkowa, w tym wypadku równa zeru. Odbywa się to przy użyciu składni języka funkcyjnego Haskell. Pierwsza instrukcja, oznaczona w komentarzu jako `--1` jest egzekucją pętli *loop*. Podczas egzekucji drugiej instrukcji agent A oczekuje na swoim porcie wejściowym a na wartość, która w przypadku otrzymania zostanie przypisana do parametru i. W następnym kroku instrukcja *select*, biorąc pod uwagę wartość parametru i, określa, która alternatywa *alt* powinna zostać uruchomiona.

Listing 2.4: Przykład definicji agenta pasywnego

```
agent AddSub {
  flag :: Bool = True;
  buffer :: Int = 0;

  proc (flag == True) add {
    exec flag = False;            -- 1
    in add buffer;                -- 2
    exit;                         -- 3
  }

  proc (flag == False) sub {
    exec flag = True;             -- 4
    exec buffer = buffer - 1;     -- 5
```

```
    out sub buffer;              -- 6
    exit;                        -- 7
  }
}
```

Rozważmy definicję pasywnego agenta *AddSub* zaprezentowaną na listingu 2.4. Agent pasywny *AddSub* posiada dwa porty proceduralne port *add* i port *sub*. Na początku definiowane są dwa parametry *flag* i *buffer* oraz określane ich wartości początkowe. W zależności od wartości parametru *flag* możliwe jest wywołanie jednej lub drugiej procedury. W przypadku gdy wartość parametru *flag* jest równa *True*, wartość z portu wejściowego *add* nadpisuje poprzednią wartość parametru *buffer*. W przypadku, gdy wartość parametru *flag* jest równa *False*, wartość parametru *buffer* jest zmniejszana o 1 i wysyłana przez port proceduralny.

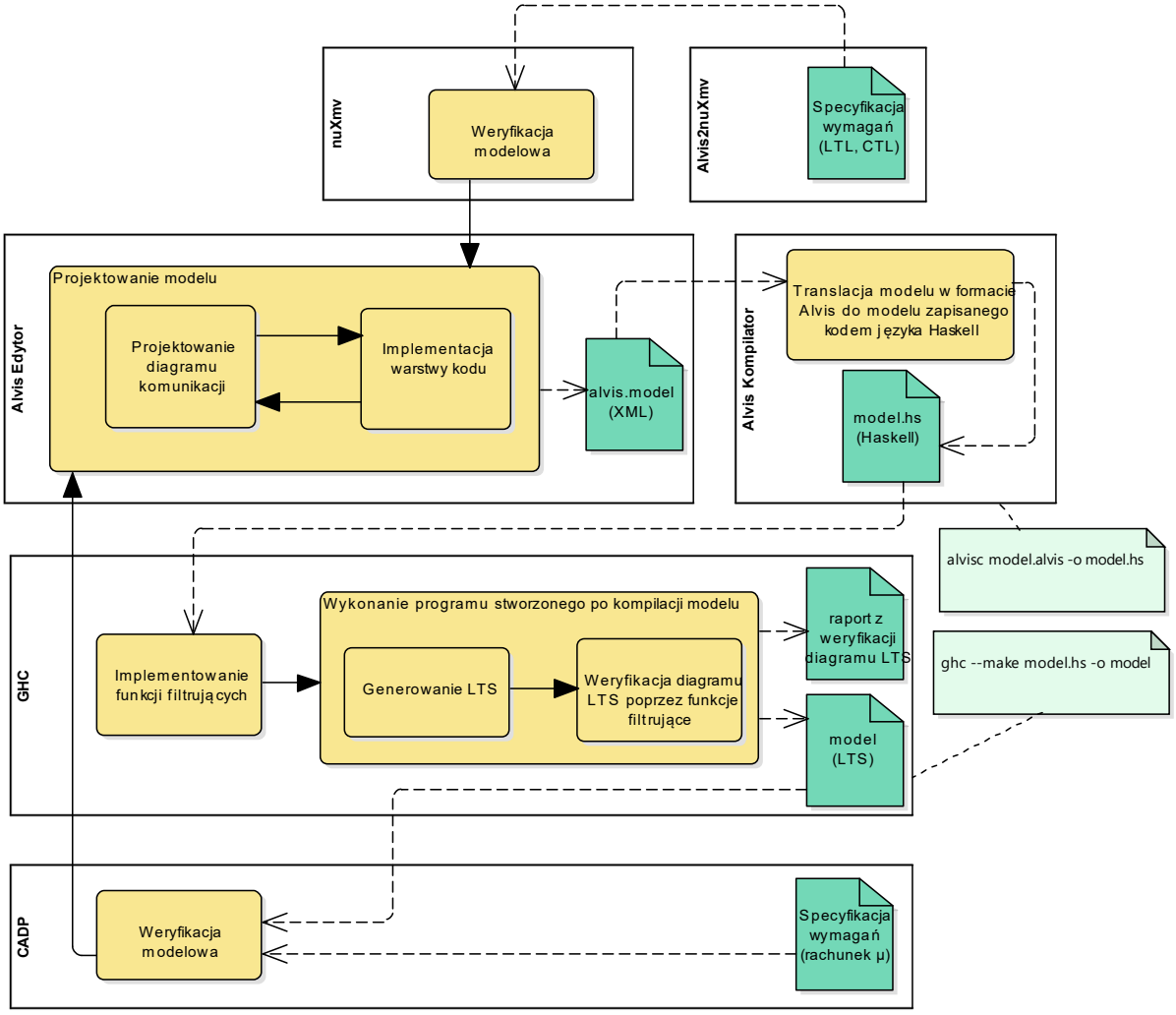
W przypadku modeli, w których uwzględniane są aspekty czasowe, niezbędne jest dostarczenie informacji o czasie trwania poszczególnych instrukcji. Użytkownik ma możliwość przypisania wartości całkowitej, określającej ten czas, do każdej instrukcji w warstwie kodu. Przypisanie to reprezentowane jest przez funkcję w języku Haskell `duration`. Listing 2.5 przedstawia przykład takie funkcji dla obu rozważanych agentów (numery instrukcji agenta podane są w komentarzach).

Listing 2.5: Przykład definicji funkcji duration

```
duration :: Agent -> Int -> Int
duration A 1 = 2
duration A 2 = 2
duration A 3 = 3
duration A 4 = 4
duration A 5 = 5
duration A 6 = 5
duration AddSub 1 = 3
duration AddSub 2 = 2
duration AddSub 3 = 3
duration AddSub 4 = 3
duration AddSub 5 = 2
```

2.4. Kompilacja modelu

Na rysunku 2.4 pokazano proces projektowania, kompilacji i weryfikacji modelu w języku Alvis. Przepływ wykonywanych czynności zaczyna się od zaprojektowania diagramu komunikacji i zaimplementowania warstwy kodu agentów w edytorze *Alvis Editor*. W wyniku tych czynności otrzymuje się model zapisany do pliku w formacie XML. Plik ten jest obiektem wejściowym dla kompilatora *Alvis Compiler*, który doko-



Rysunek 2.4: Kompilacja modelu

nuje translacji modelu stworzonego w Alvisie do kodu języka funkcyjnego Haskell – tzw. IHR *Intermediate Haskell Representation*.

Uzyskany kod źródłowy może być modyfikowany przez użytkownika. Możliwa jest modyfikacja algorytmów weryfikujących model, np. dodanie tzw. *funkcji filtrujących*, które definiują kryteria wyszukiwania stanów lub fragmentów grafu w grafie LTS. Rezultatem kompilacji kodu Haskella jest program wykonywalny, który domyślnie generuje graf LTS. To domyślne zachowanie może być zmienione przez opcje kompilatora i plik Haskella może być użyty do symulacji modelu lub jego weryfikacji z użyciem metod zaimplementowanych w Haskellu. Wygenerowany graf LTS można zapisać w jednym następujących formatów:

- *dot*
- *Aldebaran*
- *text*

lub zdefiniować własne funkcje eksportu do formatu wymaganego przez użytkownika.

Opracowane narzędzia zostały przygotowane przede wszystkim do współpracy z najpopularniejszymi pakietami do weryfikacji modelowej. Eksport grafu LTS do formatu *Aldebaran* pozwala na wykorzystanie środowiska CADP (*Construction and Analysis of Distributed Processes*) [24], [25]. W takim przypadku własności systemu specyfikuje się w logice μ [19], [32].

Możliwe jest również zastosowanie weryfikatora nuXmv [16]. W takim przypadku na podstawie grafu LTS wygenerowanego do formatu *dot* generowany jest model w formacie *SMV* (za pomocą konwertera *Alvis2nuXmv* [48]). W podejściu tym własności systemu można specyfikować w logikach LTL (*Linear Temporal Logic*) i CTL (*Computation Tree Logic*) [18], [4].

Listing 2.6: Proces transformacji modelu z języka Alvis do grafu LTS

```
alvisc model.alvis -o model.hs
ghc --make model.hs -o model
./model > model.dot
```

Na listingu 2.6 pokazane serię poleceń przekształcających model w języku Alvis w reprezentację jego przestrzeni stanów (graf LTS) w formacie *dot*. W pierwszej linii pokazano wywołanie kompilatora języka Alvis, w drugiej kompilatora języka Haskell, a na koniec uruchomienie uzyskanego programu, którego wynikiem jest graf LTS.

2.5. Podsumowanie

W rozdziale zaprezentowano podstawowe komponenty modeli w języku Alvis. Opisano przeznaczenie trzech warstw modelu: *warstwy graficznej*, *warstwy kodu* i *warstwy systemowej*. Omówiono koncepcje agenta, jego portów i kanałów komunikacyjnych pomiędzy agentami. Zaprezentowano również dwa możliwe typy agentów: aktywne i pasywne z uwzględnieniem różnic pomiędzy nimi.

Rozdział zawiera też definicję niehierarchicznego diagramu komunikacji, z uwzględnieniem ograniczeń jakie muszą spełniać takie diagramy oraz opis instrukcji języka wchodzących w skład *warstwy kodu* modelu. Na przykładach agentów aktywnych i pasywnych pokazano jak te instrukcje są używane i interpretowane. Wskazano również na możliwości określania czasu wykonania poszczególnych instrukcji, co jest jedną z kluczowych zagadnień w niniejszej rozprawy doktorskiej

W ostatniej części rozdziału przedstawiono proces transformacji od modelu w języku Alvis do reprezentacji przestrzeni stanów w postaci grafu LTS. Ta ostatnia stosowana jest przez narzędzia do weryfikacji modelowej do weryfikacji poprawności modelu.

3. Model i jego stan

Alvis, w zamyśle swojej definicji i celów przed nim stawianych, jest formalnym językiem modelowania systemów współbieżnych. Oznacza to, że niezbędnym jest jednoznaczne zdefiniowanie stanu modelowanego systemu i opisanie reguł, według których następuje zmiana tego stanu. Definicje te i reguły są podstawą dla algorytmu wygenerowania reprezentacji przestrzeni stanów, jaką stanowi graf LTS (*Labeled Transition System*), który jest podstawą formalnej weryfikacji modelu danego systemu.

3.1. Formalna definicja modelu

Jak wspomniano w poprzednim rozdziale, w niniejszej rozprawie rozważane są niehierarchiczne diagramy komunikacji. Formalna definicja modelu w języku Alvis zaczerpnięta jest z monografii [52] i prezentuje się w następujący sposób:

Definicja 3.1. *Model* w języku Alvis jest krotką $\mathbf{A} = (D, B, \varphi)$, gdzie:

- $D = (\mathcal{A}, \mathcal{C}, \sigma)$ jest *niehierarchicznym diagramem komunikacji*,
- B jest składniowo poprawną *warstwą kodu*,
- φ jest *warstwą systemową*.

Ponadto każdy agent X należący do diagramu D musi być zdefiniowany w warstwie kodu i każdy agent zdefiniowany w warstwie kodu musi mieć swoją reprezentację graficzną na diagramie komunikacji.

Składniowo poprawna warstwa kodu oznacza, że instrukcje użyte są we właściwy sposób i korespondują z diagramem komunikacji, co w praktyce oznacza, że dla przykładu, tylko porty wejściowe są argumentami dla instrukcji *in*, a porty wyjściowe mogą być użyte jako jedyny dopuszczalny argument dla instrukcji *out*. Poprawność warstwy kodu sprawdzana jest przez kompilator *Alvis Compiler* i w przypadku wystąpienia błędów generowany jest odpowiedni komunikat.

W dalszej części rozprawy doktorskiej rozważane będą wyłącznie niehierarchiczne modele systemów postaci $\mathbf{A} = (D, B, \alpha^1_{FPPS})$.

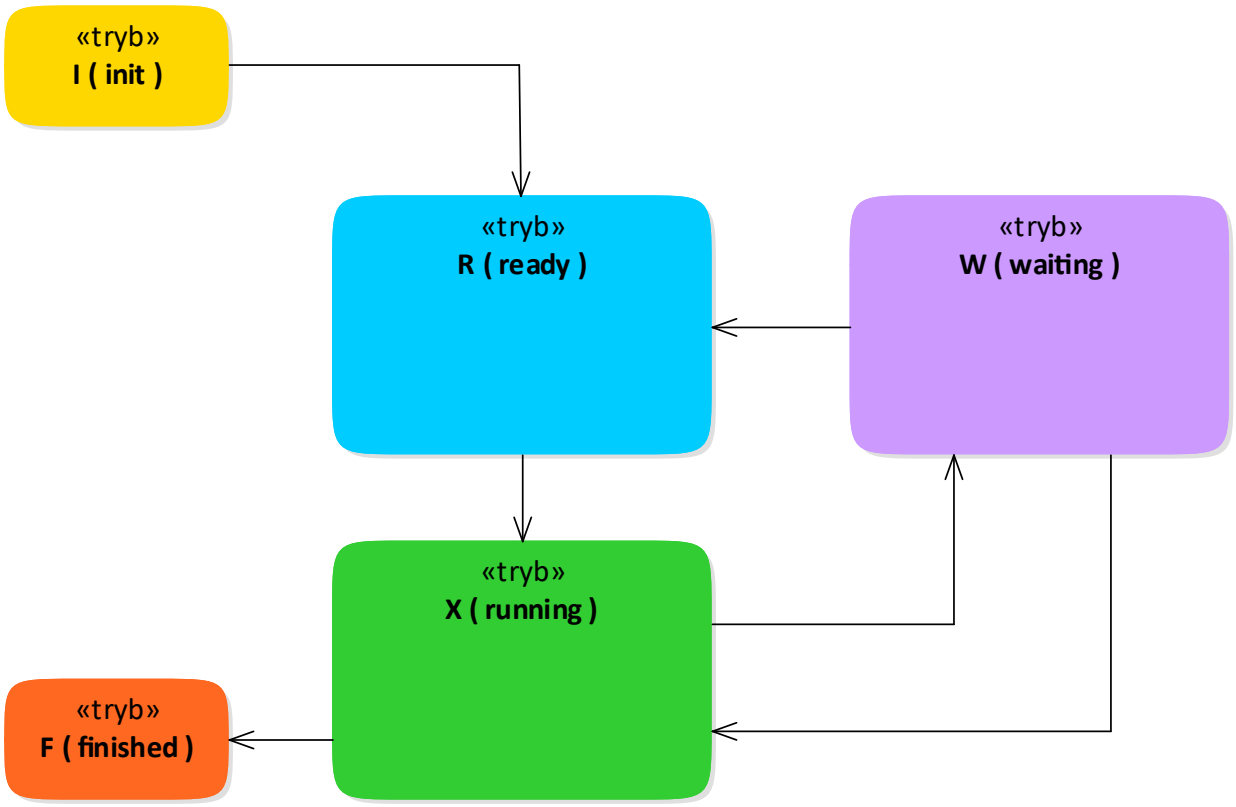
3.2. Stan agenta

Podczas prac nad projektem Alvis założono, że finalne środowisko musi być przyjazne i adoptowane do warunków sprzyjających pracy w środowisku inżynierskim. To samo założenie przyjęto w kwestii prezentowania modelu danego systemu, a co za tym idzie przestrzeni stanów dla takiego modelu (LTS). Nazwy elementów z których składa się stan danego agenta, tryby pracy agenta itp. zostały dobrane w ten sposób, aby nie odróżniały się od pojęć ogólnie przyjętych i znanych w powszechnych zastosowaniach inżynierskich.

Definicja 3.2. *Stanem agenta* X nazywamy krotkę: $S(X) = (am(X), pc(X), ci(X), pv(X))$, gdzie:

- $am(X)$ jest *trybem agenta*,
- $pc(X)$ jest *licznikiem rozkazów agenta*,
- $ci(X)$ jest *listą kontekstową agenta*,
- $pv(X)$ jest *krotką parametrów agenta*.

W zależności od kontekstu prezentowanych definicji elementy am, pc, ci, pv będą indeksowane przez S, S' itp. w celu wskazania właściwego stanu, do którego referują. Dla przykładu $am_S(X)$ oznacza tryb agenta X w stanie S, a zapis $am_{S'}(X)$ odnosi się do trybu tego samego agenta w stanie S'.



Rysunek 3.1: Możliwe tranzycje pomiędzy trybami pracy agenta aktywnego.

Rysunek 3.2: Możliwe tranzycje pomiędzy trybami pracy agenta pasywnego.

Tryb agenta określa bieżący typ aktywności danego agenta. Tryby w jakich może znajdować się agent aktywny zaprezentowano na rysunku 3.1 i przedstawiają się następująco:

- *init* (oznaczenie I) – reprezentuje tryb, w jakim znajdują się agenty aktywne, które nie są uruchamiane przy starcie systemu – wartość *False* funkcji aktywności σ.
- *running* (oznaczenie X) – reprezentuje tryb agenta, który aktualnie kontroluje zasoby procesora i realizuje swoje instrukcje zawarte w warstwie kodu;
- *waiting* (oznaczenie W) – reprezentuje tryb agenta, który jest wstrzymany i oczekuje na jakieś zdarzenie, np. na finalizację komunikacji przez innego agenta aktywnego lub na dostępność procedury agenta pasywnego.
- *finished* (oznaczenie F) – reprezentuje tryb agenta, który zakończył swoją pracę. W trybie tym agent zostanie do końca przetwarzania modelu dla danego systemu;
- *ready* (oznaczenie R) – reprezentuje tryb agenta, w którym znajdują się te agenty, które pretendują do przejęcia kontroli nad zasobami procesora, lecz nie zostały wyselekcjonowane przez algorytm szeregujący. Reguły według których agenty uzyskują dostęp do zasobów procesora, będą przedstawione w dalszej części tego rozdziału;

W celu zmiany trybu pracy agenta na *running*, agent musi choć na krótką chwilę znaleźć się w trybie *ready*. Tylko z trybu gotowości *ready* możliwa jest tranzycja agenta do trybu *running*. Ta zależność wynika wprost z definicji algorytmu szeregującego α^1_{FPPS} i będzie wyjaśniona w dalszej części rozdziału (zob. podrozdział 3.3).

W przypadku agentów pasywnych pod uwagę brane są jedynie dwa tryby (zob. rysunek 3.2):

- *waiting* (oznaczenie W) – reprezentuje tryb agenta, który jest nieaktywny i oczekuje, aż inny agent wywoła jedną z jego procedur;
- *taken* (oznaczenie T) – reprezentuje tryb agenta, który realizuje jedną ze swoich procedur na rzecz innego agenta.

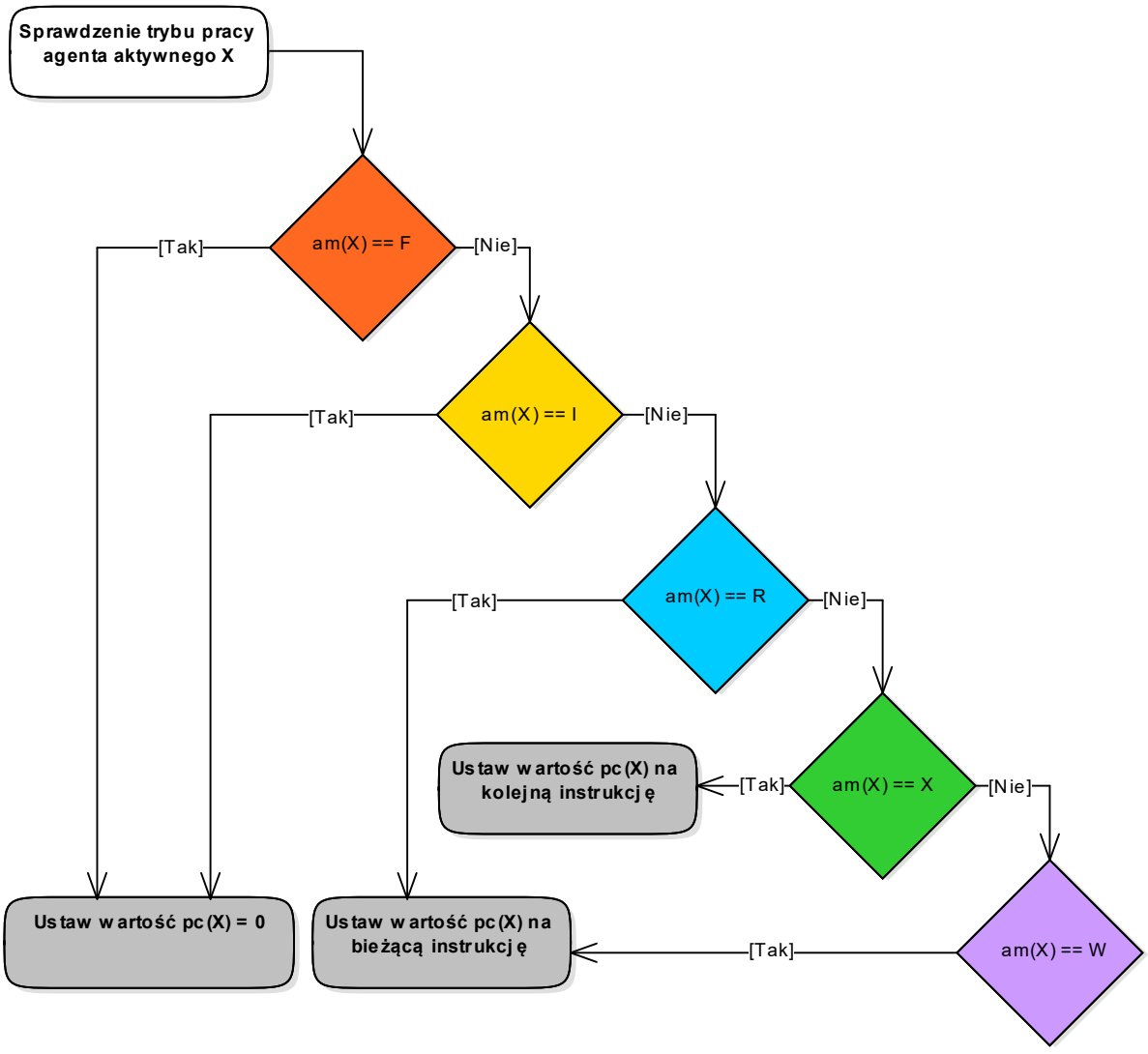
Wartość *licznika rozkazów* reprezentuje numer bieżącej instrukcji agenta, tj.:

- numer kolejnej instrukcji, którą będzie realizował agent, jeżeli wykonanie poprzedniej zostało zakończone;
- numer ostatnio realizowanej instrukcji, jeżeli do jej finalizacji wymagane są jeszcze jakieś zdarze-

nia np. zakończenie komunikacji przez innego agenta, upływ czasu zawieszenia agenta w ramach instrukcji *delay* itp.

Wyróżnia się trzy przypadki szczególne, gdy wartość licznika rozkazów danego agenta jest równa zeru:

- agent aktywny znajduje się w trybie *init*,
- agent aktywny znajduje się w trybie *finished*,
- agent pasywny znajduje się w trybie *waiting*.

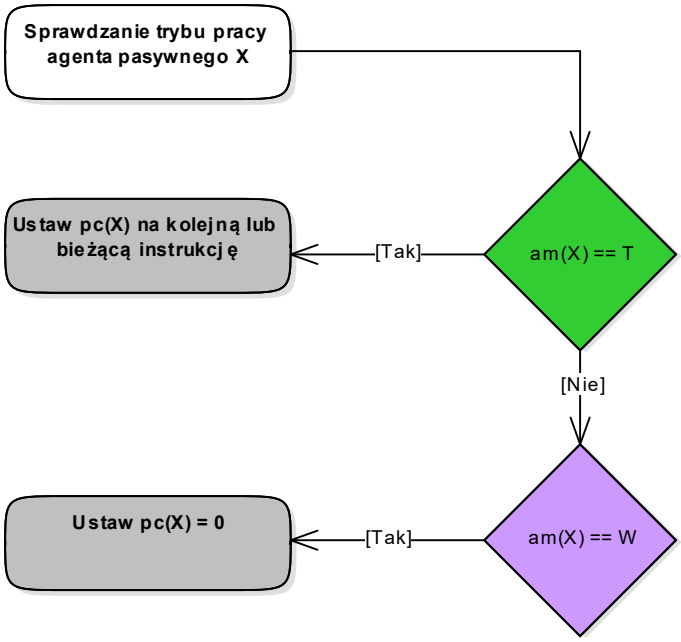


Rysunek 3.3: Obliczanie licznika rozkazów dla agenta aktywnego

Sposób w jaki obliczany jest licznik rozkazów zależy od aktualnego trybu agenta. Kalkulację dla agenta aktywnego i pasywnego przedstawiono kolejno na rysunku 3.3 i rysunku 3.4.

Wartość licznika rozkazów w następujący sposób wiązana jest z instrukcjami w kodzie:

- $pc(X)$ wskazuje na instrukcję *exec* (*exit*, *jump*, *null*, *start*, *delay*), jeśli następną instrukcją do wykonania jest *exec* (*exit*, *jump*, *null*, *start*, *delay*).



Rysunek 3.4: Obliczanie licznika rozkazów dla agenta pasywnego

- $pc(X)$ wskazuje na instrukcję *critical*, jeśli następną instrukcją do wykonania jest wejście do sekcji krytycznej *critical*.
- $pc(X)$ wskazuje na instrukcję *loop*, jeśli następną instrukcją do wykonania jest obliczenie warunku logicznego (jeżeli taki występuje) i wejście do pętli *loop*.
- $pc(X)$ wskazuje na instrukcję *loop every*, jeśli następną instrukcją do wykonania jest wejście do pętli*loop every*.
- $pc(X)$ wskazuje na instrukcję *select*, jeśli następną instrukcją do wykonania ewaluacja warunków w gałęziach tej instrukcji i ewentualnie wybór jednej z nich.
- $pc(X)$ wskazuje na instrukcję *in* lub *out*, jeśli następną instrukcją do wykonania jest *in* lub *out*, albo ostatnio wykonaną instrukcją była *in* lub *out* i agent oczekuje na zakończenie rozpoczętej komunikacji (dotyczy to tak samo agenta aktywnego jak i pasywnego).

Lista kontekstowa zawiera dodatkową informację odnośnie bieżącego stanu danego agenta. W przypadku modeli z warstwą systemową α^1_{FPPS} możliwe są następujące wpisy na liście kontekstowej agenta X:

- $critical$ – Agent realizuje instrukcje zawarte w sekcji krytycznej.
- $in(a)$ – Dla agenta pasywnego w trybie *waiting* wpis informuje o dostępnej procedurze wejściowej danego agenta. Dla agenta aktywnego w trybie *waiting* wpis informuje o nazwie portu użytego wraz z ostatnią instrukcją *in*.
- $out(a)$ – Dla agenta pasywnego w trybie *waiting* wpis informuje o dostępnej procedurze wyjściowej

danego agenta. Dla agenta aktywnego w trybie *waiting* wpis informuje o nazwie portu użytego wraz z ostatnią instrukcją *out*.

- $proc(Y.b)$ – Wpis informuje o nazwie wywołanej procedury.
- $sft(t)$ (*step finish time*) – Wpis zawiera informację o liczbę jednostek czasu jakie są niezbędne do ukończenia bieżącej instrukcji.
- $timer(t, n)$ – Wpis ten reprezentuje zegar powiązany z instrukcją o numerze n. Pierwszy parametr informuje o liczbie jednostek czasu, która pozostała do wygenerowania sygnału *timeout*, np. do zakończenia czasu zawieszenia po wykonaniu instrukcji *delay*.
- $timeout(n)$ – Wpis ten informuje o wystąpieniu sygnału *timeout* dla instrukcji o numerze n.
- $lock(a)$ – Wpis ten jest używany, gdy agent rozpoczął, ale nie zakończył komunikacji. Dotyczy to sytuacji takich jak wywołanie procedury agenta pasywnego lub kończenie komunikacji z innym agentem aktywnym. Za pomocą $lock$ agent „rezerwuje" na wyłączność drugiego agenta zaangażowanego w rozpoczętą komunikację.

Wyróżnia się dwa przypadki, w których lista kontekstowa danego agenta z definicji jest pusta:

- agent aktywny znajduje się w trybie *init*,
- agent aktywny znajduje się w trybie *finished*.

Krotka parametrów zawiera bieżące wartości parametrów agenta. Jeśli nie zdefiniowano żadnych parametrów dla danego agenta to krotka jest pusta.

Rozważmy definicję agenta A z listingu 2.3 (strona 23) i wybrane stany agenta A, jakie można wyodrębnić dla tego przykładu:

- $(\mathsf{I}, 0, [\,], (0))$ – Agent aktywny *A* jest w trybie *init*; *licznik rozkazów* nie wskazuje na żadną z instrukcji; lista kontekstowa agenta jest pusta; a parametr i jest równy zeru (wartość początkowa).
- $(\mathsf{X}, 2, [\,], (0))$ – Agent aktywny *A* jest w trybie *running*; *licznik rozkazów* wskazuje na drugą instrukcję, która właśnie będzie wykonana; lista kontekstowa agenta jest pusta; a parametr i ma wartość zero.
- $(\mathsf{W}, 2, [in(a)], (0))$ – Agent aktywny *A* jest w trybie *waiting*; *licznik rozkazów* wskazuje na drugą instrukcję, która została właśnie wykonana; lista kontekstowa agenta informuje, że agent czeka na dokończenie komunikacji ze strony innego agenta na swoim porcie wejściowym a; parametr i ma wartość zero.
- $(\mathsf{X}, 3, [sft(2)], (5))$ – Agent aktywny *A* jest w trybie *running*; *licznik rozkazów* wskazuje na trzecią instrukcję; lista kontekstowa agenta informuje, że instrukcja jest w trakcie realizacji, a do jej ukończenia potrzeba 2 jednostki czasu; parametr i ma wartość 5.

W podobny sposób rozważmy kilka potencjalnych stanów agenta pasywnego z listingu 2.4 (page 24):

- $(\mathsf{W}, 0, [in(add)], (True, 0))$ – Agent pasywny *AddSub* jest w trybie *waiting*; *licznik rozkazów* nie wskazuje na żadną instrukcję; lista kontekstowa wskazuje, że dostępna jest procedura *add*; parametry

flag i *buffer* mają wartości odpowiednio *True* i 0.

- $(\mathsf{T}, 1, [\,], (True, 0))$ – Agent pasywny *AddSub* jest w trybie *taken*; *licznik rozkazów* wskazuje, że realizowana będzie pierwsza instrukcja z pierwszej procedury; lista kontekstowa jest pusta; parametry mają wartości odpowiednio *True* i 0.
- $(\mathsf{T}, 4, [\,], (True, 5))$ – Agent pasywny *AddSub* jest w trybie *taken*; *licznik rozkazów* wskazuje, że realizowana będzie instrukcja 4 (druga instrukcja z drugiej procedury); lista kontekstowa jest pusta; parametry mają wartości odpowiednio *True* i 5.
- $(\mathsf{T}, 5, [sft(2)], (True, 5))$ – agent pasywny *AddSub* jest w trybie *taken*; *licznik rozkazów* wskazuje, że realizuje instrukcję numer 5; lista kontekstowa informuje, że do zakończenia bieżącej instrukcji potrzeba jeszcze 2 jednostki czasu; parametry mają wartości odpowiednio *True* i 5.

3.3. Algorytm szeregujący

Do pełnego opisu stanu modelu danego systemu niezbędne jest określenie stanów wszystkich agentów w nim zdefiniowanych. Ponieważ warstwa systemowa α^1_{FPPS} dopuszcza użycie tylko jednego procesora, trzeba wziąć pod uwagę, iż w danym momencie czasowym tylko jeden agent aktywny będzie w trybie *running*, a w konsekwencji tylko ten agent będzie mógł przejąć kontrolę nad zasobami procesora danego systemu. Aby spełnić powyższe założenie potrzebny jest algorytm szeregujący (ang. *scheduler*), który będzie zarządzał agentami w ten sposób, że jeden z nich będzie promowany do trybu aktywnego *running*. W tym podrozdziale zostanie przedstawiona koncepcja i definicja dla algorytmu szeregującego warstwy systemowej α^1_{FPPS}.

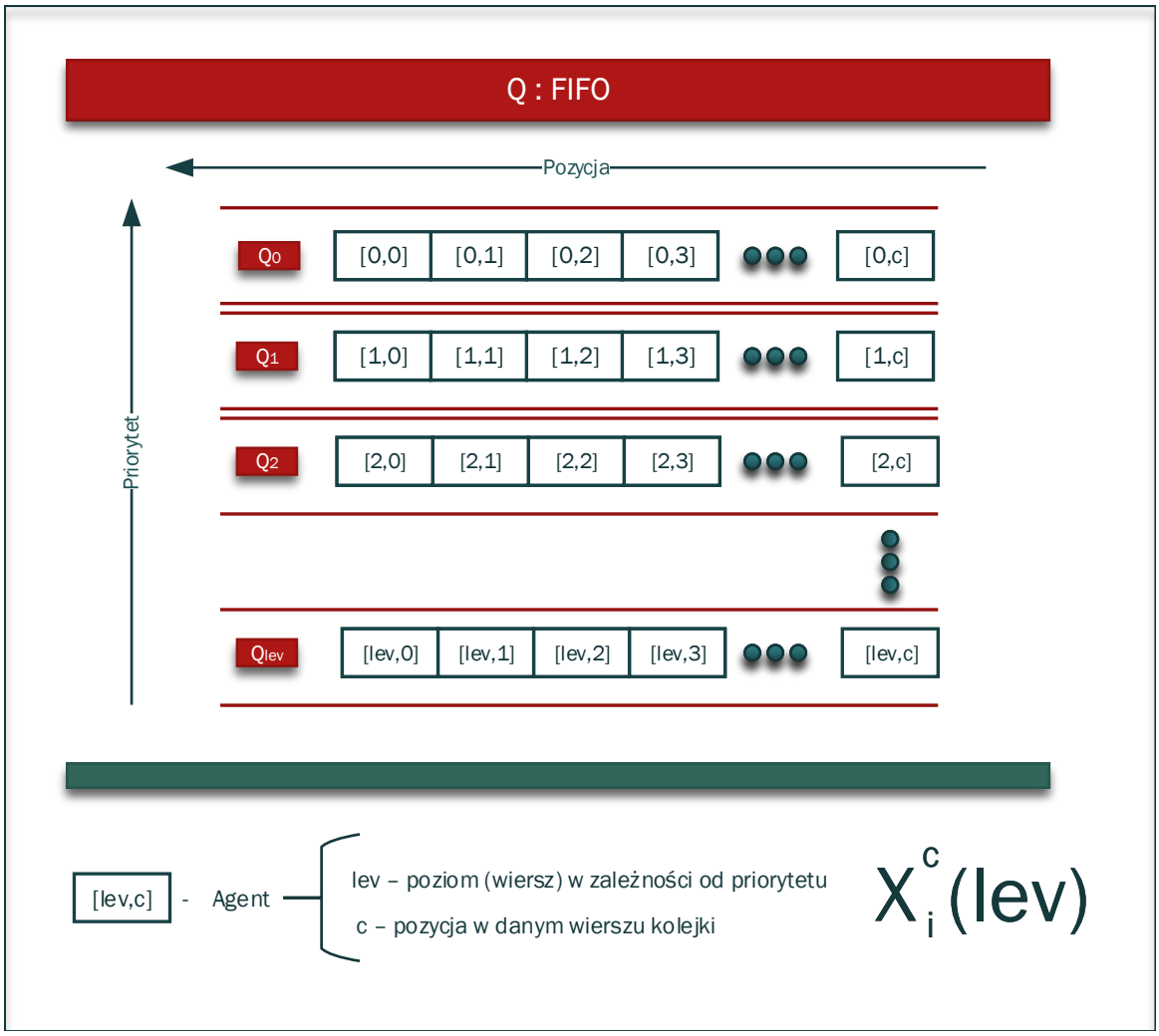
Algorytm szeregujący rozważany dla warstwy systemowej α^1_{FPPS} ma pełną nazwę *Fixed Priority Preemptive Scheduling* i jest oznaczony tym samym symbolem jak warstwa systemowa. Trzy główne założenia tego algorytmu są następujące:

- priorytet,
- wywłaszczanie,
- kolejka FIFO.

Zakłada się, że każdy agent zdefiniowany w danym modelu ma stałą i niezmienną wartość priorytetu (ang. *fixed priority*). Priorytety agentów przyjmują wartości od 0 do 9. Priorytet o wartości zero jest uznawany za najwyższy priorytet w rozważanym modelu. Algorytm szeregujący decydując, który z agentów ma zostać promowany do trybu aktywnego *running*, m.in. porównuje ich priorytety.

Każdy agent aktywny, który obecnie znajduje się w trybie *running*, może zostać wywłaszczony za pomocą algorytmu szeregującego przez agenta o wyższym priorytecie, znajdującego się w trybie *ready* (czyli zgłaszającego swą gotowość do pracy, a w konsekwencji do przejęcia kontroli nad zasobami procesora). Biorąc pod uwagę te założenia, algorytm szeregujący w celu wywłaszczenia bieżącego agenta bierze pod

uwagę agenty aktywne w trybie *ready* i o wyższym priorytecie niż posiada bieżący agent aktywny. Wypromowany do trybu *running* agent aktywny przejmuje kontrolę nad zasobami procesora i w rezultacie jest jedynym agentem w modelu, który wykonuje swoje instrukcje zdefiniowane w warstwie kodu.



Rysunek 3.5: Kolejka FIFO agentów aktywnych w trybie *ready*

Biorąc pod uwagę fakt, że w danym momencie czasowym więcej niż jeden agent aktywny może zgłaszać swoją gotowość do przejęcia zasobów procesora, algorytm szeregujący warstwy systemowej α^1_{FPPS} wprowadza do swojej konstrukcji dwuwymiarową kolejkę FIFO. Kolejka ta kolekcjonuje i porządkuje w swojej dwuwymiarowej strukturze agenty aktywne znajdujące się w trybie gotowości *ready*. Decyzja, czy dany agent powinien zostać wypromowany do trybu *running*, podejmowana jest na podstawie pozycji danego agenta we właściwym wierszu kolejki FIFO. Numer wiersza oznacza wartość priorytetu skolekcjonowanych tu agentów i tak dla przykładu, agenty z najwyższym priorytetem będą zawsze elementami wiersza o indeksie 0. Ogólną strukturę dwuwymiarowej kolejki FIFO przedstawiono na rysunku 3.5.

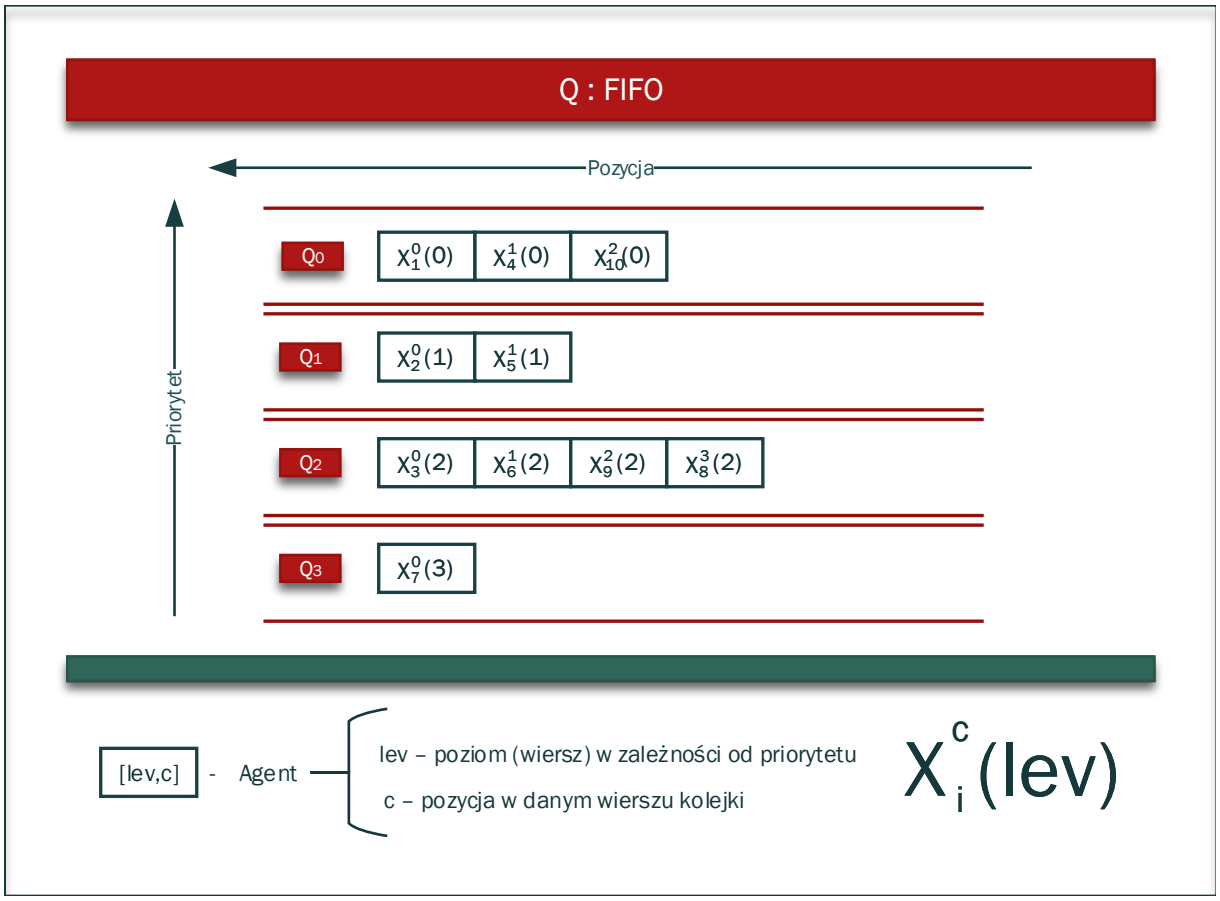
Niech dany będzie model w języku Alvis $\mathbf{A} = (D, B, \alpha^1_{FPPS})$ ze zbiorem agentów aktywnych $\mathcal{A}_A$. Zbiór agentów aktywnych $\mathcal{A}_A$ może być podzielony na zbiory rozłączne o tej samej wartości priorytetów.

Dla każdego takiego podzbioru przeznaczony jest jeden poziom (wiersz) Q_{lev} w dwuwymiarowej kolejce FIFO. Dodatkowo zakłada się, że na danym poziomie Q_{lev} kolejka kolekcjonuje jedynie agenty w trybie *ready*. Indeks lev określa wartość priorytetu agentów zgromadzonych na danym poziomie.

Niech Q oznacza zbiór wszystkich wierszy dwuwymiarowej kolejki FIFO $Q = \{Q_0, Q_1, \ldots, Q_{lev}\}$. Biorąc pod uwagę, że kolejka kolekcjonuje tylko agenty znajdujące się w trybie *ready*, ogólna definicja jej wiersza wygląda następująco:

$$Q_{lev} = \{X \in \mathcal{A}_A \colon am_S(X) = \mathsf{R} \wedge pr(X) = lev\} \tag{3.1}$$

Symbolem $X_i^c(lev)$ oznaczamy agenta aktywnego X_i, który jest elementem wiersza o indeksie lev i znajduje się pozycji c w tym samym wierszu.



Rysunek 3.6: Przykład dwuwymiarowej kolejki FIFO kolekcjonującej agenty aktywne w trybie *ready*

Biorąc pod uwagę powyższe założenia, rozważmy kolejkę przedstawioną na rysunku 3.6). Jej struktura będzie zapisana w następujący sposób:

$Q_0 = \{X_1^0(0), X_4^1(0), X_{10}^2(0)\}|^{lev=0}$,

$Q_1 = \{X_2^0(1), X_5^1(1)\}|^{lev=1}$,

$Q_2 = \{X_3^0(2), X_6^1(2), X_9^2(2), X_8^3(2)\}|^{lev=2}$,
$Q_3 = \{X_7^0(3)\}|^{lev=3}$.

W celu wypromowania agenta aktywnego w trybie gotowości *ready* do trybu *running* algorytm szeregujący wykonuje następujące czynności:

1. Wiersz zawierający agenty aktywne o najwyższym priorytecie Q_r wybierany jest spośród zbioru wszystkich wierszy Q dwuwymiarowej kolejki FIFO:

$$Q_r = \min_j \{Q_j \in Q \colon Q_j \neq \emptyset\}. \tag{3.2}$$

2. Z wiersza Q_r wyselekcjonowany zostaje agent o najwyższej pozycji w danym wierszu. Jeżeli jego priorytet jest większy lub równy priorytetowi agenta, który bieżąco kontroluje zasoby procesora, to jest on promowany przez algorytm szeregujący do trybu aktywnego i jego tryb pracy zamieniany jest na *running*. Agent, który do tej pory kontrolował zasoby procesora, jest wywłaszczany na rzecz agenta X_r i zostaje umieszczony na najniższej pozycji w wierszu odpowiadającym wartości jego priorytetu. Agent wywłaszczony zostaje skolejkowany w dwuwymiarowej kolejce FIFO w trybie gotowości *ready*.

$$X_r = \max_c(Q_r) \Rightarrow am_S(X_r) = \mathsf{X}. \tag{3.3}$$

3. Aby w przyszłości umożliwić innym agentom zmianę trybu na *running*, dopiero co wyselekcjonowany agent X_r jest usuwany z wiersza Q_r. W przypadku, gdy agent X_r będzie zakolejkowany powtórnie w wierszu Q_r, zostanie on umieszczony na najniższej pozycji tego wiersza:

$$Q_r = Q_r \setminus \{X_r\}. \tag{3.4}$$

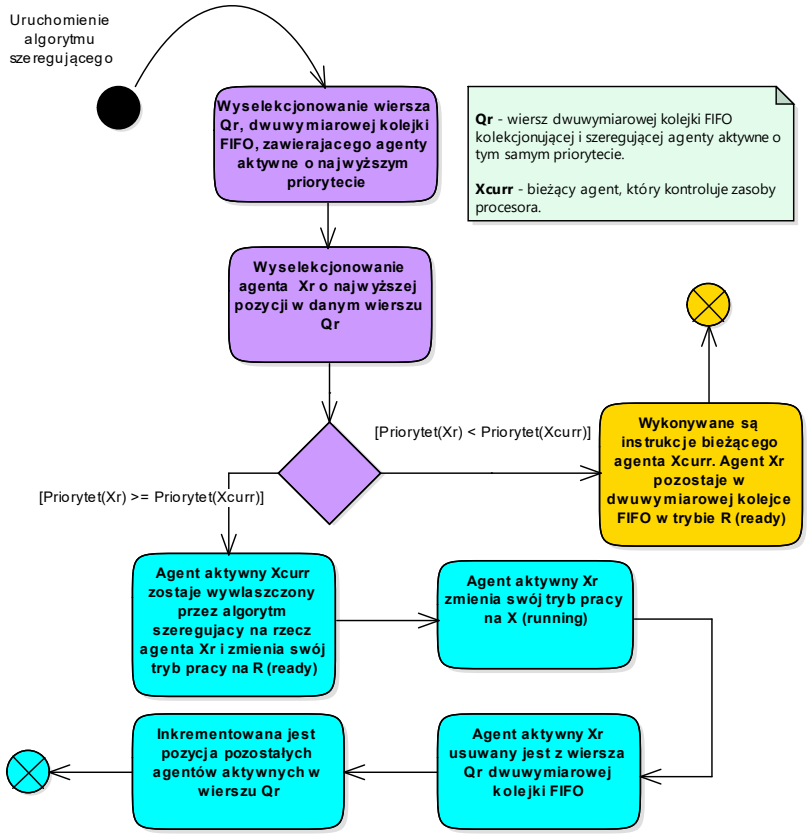
4. Jeśli wywłaszczony agent ponownie jest dodawany do kolejki, to pozycja każdego agenta w wierszu Q_r jest zwiększana o 1:

$$\forall_{X \in Q_r}\ c = c + 1. \tag{3.5}$$

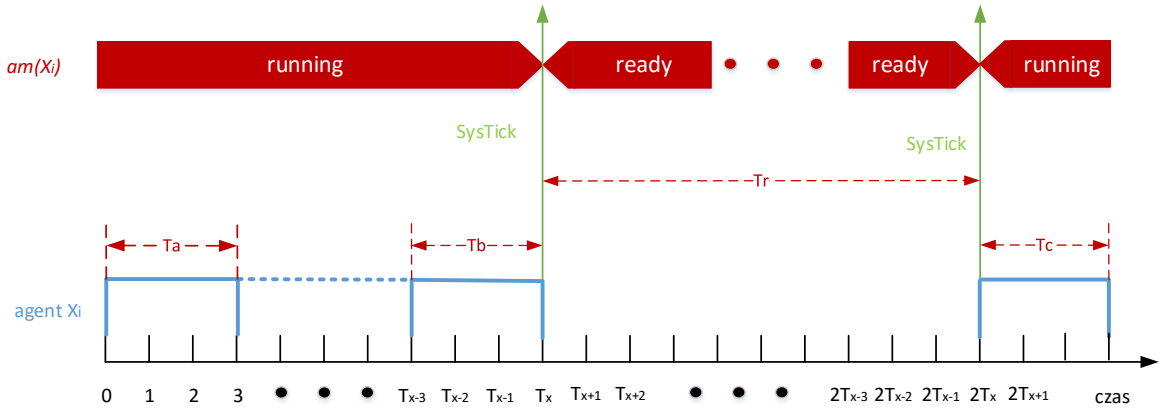
Algorytm szeregujący warstwy systemowej α^1_{FPPS} zaprezentowany został na rysunku 3.7.

Biorąc pod uwagę, że algorytm szeregujący warstwy systemowej α^1_{FPPS} w procesie promocji agenta do trybu *running* korzysta z mechanizmu wywłaszczania agentów, należy rozważyć poziom granulacji takiego wywłaszczania. Jak wspomniano wcześniej, instrukcja ma swój zdefiniowany czas realizacji. Problemem jaki pojawia się w jednoprocesorowych platformach jest sytuacja, w której agent zostaje wywłaszczony w momencie czasowym, w którym nie ukończył jeszcze bieżącej instrukcji.

Załóżmy, że agent aktywny X_i osadzony jest w algorytmie *round-robin* z wywołanie systemowym *SysTick* o okresie T_x (rysunek 3.8). W momencie czasowym T_x agent X_i kończy wykonywać instrukcję o czasie trwania T_b oraz dodatkowo następuje przerwanie systemowe, w wyniku którego agent X_i jest wywłaszczany na rzecz innego agenta i przechodzi do trybu *ready*. W stanie tym oczekuje na możliwość ponownego



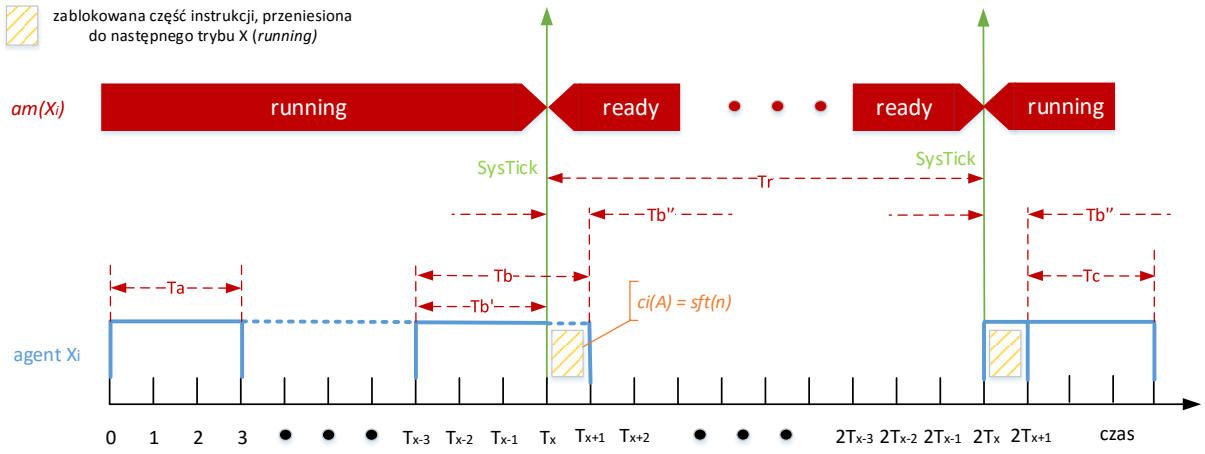
Rysunek 3.7: Algorytm szeregujący warstwy systemowej α^1_{FPPS}



Rysunek 3.8: Wykonywanie instrukcji przez agenta z punktu widzenia czasu w warstwie systemowej α^1_{FPPS}

przejęcia i kontroli zasobów procesora, co następuje w momencie czasowym $2T_x$. Agent X_i przechodzi do trybu *running* i zaczyna wykonywać kolejną instrukcję o czasie trwania T_c.

Załóżmy teraz, że instrukcja o czasie trwania T_b nie została ukończona przez agenta X_i w momencie przerwania systemowego T_x (rysunek 3.9). Niech T'_b określa liczbę jednostek czasu, w których instrukcja ta była wykonywana, a T''_b odpowiada liczbie tych jednostek czasu, które zostały do ukończenia instrukcji.



Rysunek 3.9: Granulacja instrukcji z punktu widzenia domeny czasu w warstwie systemowej α^1_{FPPS}

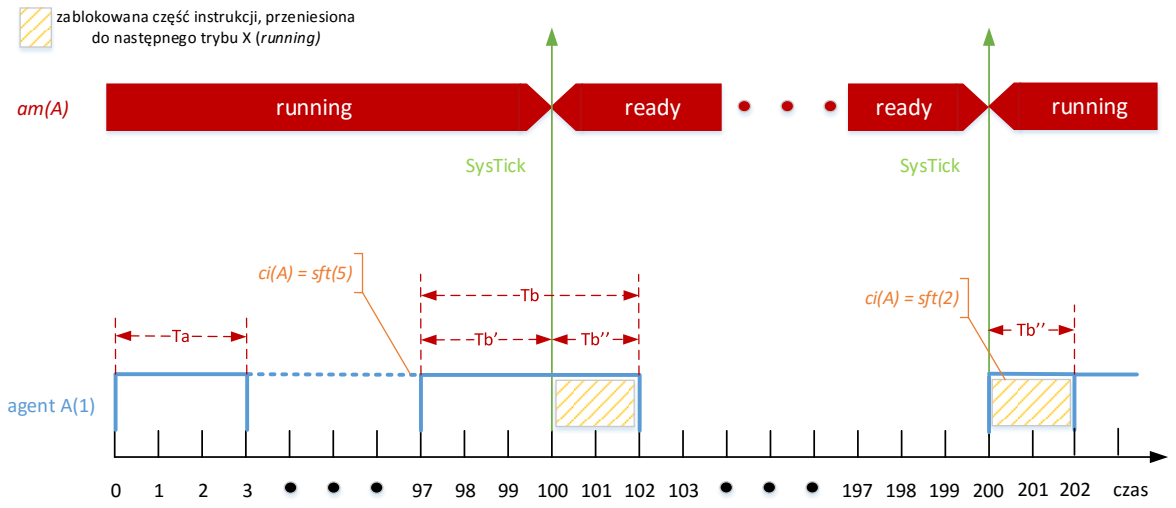
Istnieją dwie możliwości, określające sposób w jaki można dokończyć instrukcję: opóźnić konsekwencje przerwania systemowego (potencjalne wywłaszczenie agenta X_i) o czas T_b'' i dokończyć instrukcję lub przenieść dokończenie instrukcji do momentu, w którym agent X_i znów będzie w trybie *running*. W tej rozprawie doktorskiej założono, że przerwanie systemowe *SysTick* nie może zostać opóźnione i przerwana instrukcja zostaje dokończona po ponownym przejęciu zasobów procesora przez agenta X_i.

W ten sposób granulacja instrukcji jest ustalana na poziomie pojedynczej jednostki czasu, a nie czasu wykonywania całej instrukcji. W rezultacie tego założenia analiza warstwy systemowej α^1_{FPPS} musi uwzględniać stany agentów, w których poszczególne instrukcje nie zostały ukończone. Informacja o tym, że dana instrukcja potrzebuje jeszcze n jednostek czasu do ukończenia, zawarta jest na liście kontekstowej agenta w postaci wpisu $sft(n)$.

Rozważmy przykład pokazany na rysunku 3.10. Algorytm szeregujący warstwy systemowej α^1_{FPPS} wywłaszcza agenta aktywnego A z powodu przerwania systemowego *SysTick*, które wystąpiło w momencie czasowym $T = 100$. Instrukcja o czasie trwania T_b nadal potrzebuje dwóch jednostek czasu do ukończenia. W myśl przyjętych zasad w kwestii granulacji instrukcji, dokończenie procesowania zostaje przeniesione do czasu, aż agent A znów będzie w trybie *running*. W rozważanym przykładzie dzieje się to w momencie czasowym $T = 200$. Do tego czasu na liście kontekstowej agenta A widnieje wpis $sft(2)$ informujący o tym, że instrukcja nie została zakończona i potrzebuje ciągle dwóch jednostek czasu.

3.4. Stan modelu

W celu opisania bieżącego stanu dla danego modelu systemu współbieżnego potrzebne są stany wszystkich agentów zdefiniowanych w modelu oraz stan bieżący dwuwymiarowej kolejki FIFO.



Rysunek 3.10: Przykład granulacji instrukcji z punktu widzenia domeny czasu w warstwie systemowej α^1_{FPPS}

Definicja 3.3. Niech dany będzie model niehierarchiczny $\mathbf{A} = (D, B, \alpha^1_{FPPS})$, gdzie $D = (\mathcal{A}, \mathcal{C}, \sigma)$ i $\mathcal{A} = \{X_1, \ldots, X_n\}$. Niech Q będzie uporządkowaną dwuwymiarową kolejką FIFO dla warstwy systemowej $alpha^1_{FPPS}$ oraz niech t oznacza liczbę jednostek czasu do ponownego wywołania algorytmu szeregującego.

Stan modelu **A** jest krotką:

$$S = (S(X_1), \ldots, S(X_n), (Q, t)). \tag{3.6}$$

Dla każdego stanu modelu S i każdego agenta $X \in \mathcal{A}_A$ zachowane są następujące warunki:

$$critical \in ci(X) \Rightarrow \forall_{Y \in \mathcal{A}_A \setminus \{X\}} am(Y) \neq \mathsf{X}, \tag{3.7}$$

$$|\{X \in \mathcal{A}_A \colon am(X) = \mathsf{X}\}| \leq 1. \tag{3.8}$$

Zdefiniowanie stanu początkowego modelu wymaga określenia w jaki sposób uporządkowana jest kolejka Q w stanie początkowym, tj. w jakiej kolejności są wstawiane do kolejki agenty aktywne X_i i X_j, w przypadku, gdy mają taki sam priorytet. Przyjęcie założenia, że rozważane są wszystkie możliwe warianty w ramach uszeregowania agentów w poszczególnych wierszach kolejki Q, prowadzi do sytuacji, w której uzyskuje się wiele możliwych stanów początkowych. Z drugiej strony niektóre narzędzia do weryfikacji modelowej np. CADP wymagają, aby stan początkowy modelu był określony jednoznacznie. Z tych praktycznych względów przyjęto, że kompilator umieszcza agenty o tym samym priorytecie w kolejności ich definiowania w kodzie modelu. Użytkownik może zmodyfikować odpowiedni fragment kodu w Haskellu i określić inny stan początkowy. Oczywiście zastosowanie różnych priorytetów eliminuje ten problem.

Definicja 3.4. Niech dany będzie model niehierarchiczny $\mathbf{A} = (D, B, \alpha^1_{FPPS})$, gdzie $D = (\mathcal{A}, \mathcal{C}, \sigma)$ i $\mathcal{A} = \{X_1, \ldots, X_n\}$. Niech Q będzie uporządkowaną dwuwymiarową kolejką FIFO dla warstwy systemowej

$alpha^1_{FPPS}$ oraz t niech oznacza liczbę jednostek czasu do ponownego wywołania algorytmu szeregującego.

Stan początkowy modelu **A** jest krotką:

$$S_0 = (S_0(X_1), \ldots, S_0(X_n), (Q_0, t)) \tag{3.9}$$

której zawartość określana jest przez następujący algorytm:

1. $am_S(X) = \mathsf{R}$ dla agenta $X \in \mathcal{A}_A$ takiego, że $\sigma(X) = True$;
2. $am_S(X) = \mathsf{I}$ dla każdego agenta $X \in \mathcal{A}_A$ takiego, że $\sigma(X) = False$;
3. $am_S(X) = \mathsf{W}$ dla każdego agenta $X \in \mathcal{A}_P$;
4. $pc(X) = 1$ dla każdego agenta $X \in \mathcal{A}_A$ będącego w trybie *ready*;
5. $pc(X) = 0$ dla każdego agenta $X \in \mathcal{A}_A$ będącego w trybie *init*;
6. $pc(X) = 0$ dla każdego agenta $X \in \mathcal{A}_P$;
7. $ci(X) = [\,]$ dla każdego agenta $X \in \mathcal{A}_A$;
8. Dla każdego agenta $X \in \mathcal{A}_P$, $ci(X)$ zawiera nazwy wszystkich dostępnych procedur tego agenta X razem z kierunkiem przesyłania parametrów, np. $in(a)$, $out(b)$ itp.;
9. Dla każdego agenta $X \in \mathcal{A}$, $pv(X)$ zawiera wartości początkowe parametrów tego agenta;
10. Wszystkie agenty będące w trybie *ready* umieszczane są w kolejce Q zgodnie z wartością ich priorytetów. O pozycji agentów w poszczególnych wierszach decyduje użytkownik lub przyjmowany jest układ domyślnie generowany przez kompilator.
11. Agent X_r wyznaczony zgodnie ze wzorem (3.3) usuwany jest z kolejki Q, a jego tryb ustawiany jest na wartość *running*.

3.5. Podsumowanie

W niniejszym rozdziale skupiono się nad zagadnieniami związanymi z formalnym opisem stanu modelu, który wykorzystuje warstwę systemową α^1_{FPPS}. Przeniesienie semantyki języka Alvis na systemy jednoprocesorowe wymagało wprowadzenia istotnych zmian w opisie stanów agentów i modeli, w porównaniu z definicjami przedstawionymi w pracy [52]. Poza adaptacją wybranych pojęć (np. tryb agenta, lista kontekstowa) do potrzeb warstwy systemowej α^1_{FPPS}, przedstawiono koncepcję dwuwymiarowej kolejki szeregującej agenty oczekujące na przydział procesora. Zdefiniowana kolejka Q stanowi istotne rozszerzenie opisu stanu modelu, wprowadzone dla warstwy systemowej α^1_{FPPS}.

Do wkładu własnego autora rozprawy należy zaliczyć:

- Opracowanie koncepcji warstwy systemowej α^1_{FPPS}, w tym przedstawienie głównych założeń algorytmu szeregującego, sposobu kolejkowania i wywłaszczania agentów przy stałym okresie zgłaszania przerwania systemowego.

- Definicję dwuwymiarowej kolejki FIFO do szeregowania agentów i zarządzania nimi w przypadku promocji jednego z nich do trybu *running*.
- Opracowanie metody ustalania stanu początkowego dla modeli z warstwą systemową α^1_{FPPS}.

4. Zmiany stanów modelu

Koncepcja zaprezentowana w poprzednim rozdziale daje możliwość jednoznacznego opisu każdego ze stanów modelu $\mathbf{A} = (D, B, \alpha^1_{FPPS})$ utworzonego w języku Alvis. Końcowym wynikiem przygotowania do formalnej analizy jest wygenerowanie etykietowanego systemu przejść LTS. W celu przedstawienia kompletnego opisu semantyki modelu należny zdefiniować i dostarczyć zestaw reguł, które opisują zmiany stanów w modelu. W poniższym rozdziale przedstawiono reguły opracowane dla rozważanego środowiska jednoprocesorowego.

4.1. Tranzycje

Niech dany będzie model utworzony w języku Alvis postaci $\mathbf{A} = (D, B, \alpha^1_{FPPS})$. Wszystkie dostępne instrukcje, które mogą być użyte w warstwie kodu, zaprezentowane zostały na listingu 2.1 (str. 21). Z teoretycznego punktu widzenia, instrukcje te są reprezentowane poprzez tzw. *tranzycje*. Zaprezentowany poniżej zestaw tranzycji jest właściwy dla modeli postaci $\mathbf{A} = (D, B, \alpha^1_{FPPS})$. Niech X i Y będą agentami zawierającymi porty odpowiednio $X.p$ i $Y.q$.

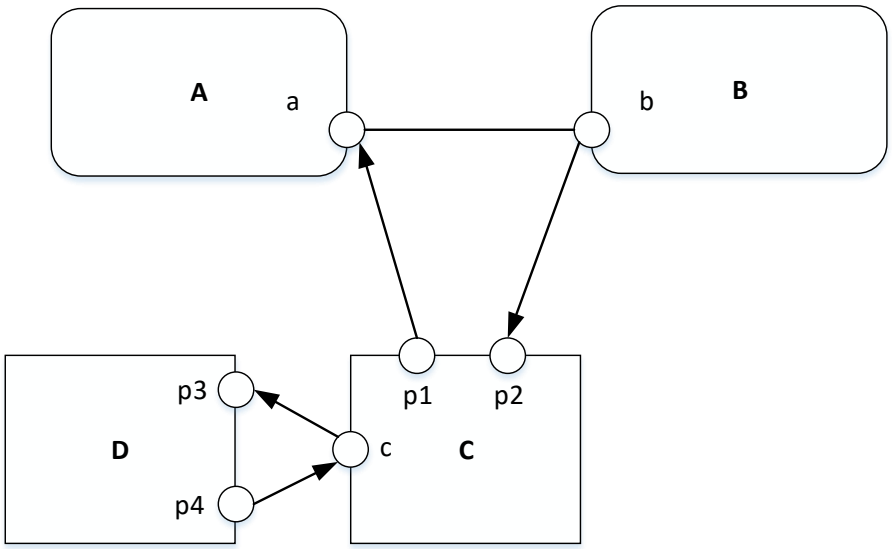
- $TCritical(X)$ – Reprezentuje wykonanie instrukcji *critical* przez agenta X – wejście do sekcji krytycznej.
- $TDelay(X)$ – Reprezentuje wykonanie instrukcji *delay* przez agenta X.
- $TExec(X)$ – Reprezentuje wykonanie instrukcji *exec* przez agenta X – ewaluacja wyrażenia i przypisanie jego wyniku do parametru.
- $TExit(X)$ – Reprezentuje wykonanie instrukcji *exit* przez agenta X – trwałe zatrzymanie pracy agenta aktywnego lub zakończenie procedury agenta pasywnego.
- $TIn(X.p)$ – Reprezentuje wykonanie instrukcji *in* przez aktywnego agenta X (port $X.p$), rozumianej jako inicjalizowanie komunikacji z innym agentem.
- $TInF(X.p, Y.p)$ – Reprezentuje wykonanie instrukcji *in* przez aktywnego agenta X (port $X.p$), rozumianej jako finalizacja komunikacji z agentem Y (port $Y.q$).
- $TInAP(X.p, Y.q)$ – Reprezentuje wykonanie instrukcji *in* przez aktywnego agenta X (port $X.p$), rozumianej jako wywołanie dostępnej procedury wyjściowej $Y.q$.

- $TInPP(X.p, Y.q)$ – Reprezentuje wykonanie instrukcji *in* przez pasywnego agenta X (port $X.p$, rozumianej jako wywołanie dostępnej procedury wyjściowej $Y.q$.
- $TJump(X)$ – Reprezentuje wykonanie instrukcji *jump* przez agenta X.
- $TLoop(X)$ – Reprezentuje wykonanie instrukcji *loop* (z lub bez warunku logicznego) przez agenta X – ewaluacja warunku pętli (jeśli występuje) i ewentualne przejście do jej wnętrza.
- $TLoopEvery(X)$ – Reprezentuje wykonanie instrukcji *loop every* przez agenta X – wejście do wnętrza pętli okresowej.
- $TNull(X)$ – Reprezentuje wykonanie instrukcji *null* przez agenta X.
- $TOut(X.p)$ – Reprezentuje wykonanie instrukcji *out* przez aktywnego agenta X (port $X.p$), rozumianej jako inicjalizowanie komunikacji z innym agentem.
- $TOutF(X.p, Y.p)$ – Reprezentuje wykonanie instrukcji *out* przez aktywnego agenta X (port $X.p$), rozumianej jako finalizacja komunikacji z agentem Y (port $Y.q$).
- $TOutAP(X.p, Y.q)$ – Reprezentuje wykonanie instrukcji *out* przez aktywnego agenta X (port $X.p$), rozumianej jako wywołanie dostępnej procedury wejściowej $Y.q$.
- $TOutPP(X.p, Y.q)$ – Reprezentuje wykonanie instrukcji *out* przez aktywnego agenta X (port $X.p$), rozumianej jako wywołanie dostępnej procedury wejściowej $Y.q$.
- $TSelect(X)$ – Reprezentuje wykonanie instrukcji *select* przez agenta X – ewaluacja warunków poszczególnych alternatyw i ewentualne wybranie jednej z nich.
- $TStart(X)$ – Reprezentuje wykonanie instrukcji *start* przez agenta X.

Słowo kluczowe *proc* użyte jest jedynie w celu organizacji kodu agent pasywnego i jest pomijane przy numerowaniu instrukcji kodu, dlatego też nie przewiduje się żadnej tranzycji w tym przypadku.

Rozważmy model, którego diagram komunikacji zaprezentowano na rysunku 4.1. W zależności od instrukcji użytych w warstwie kodu, w modelu takim mogą wystąpić następujące tranzycje związane z komunikacją między agentami:

- $TIn(A.a)$ – Reprezentuje wykonanie instrukcji *in* przez agenta A (port $A.a$), jako inicjalizowanie komunikacji. Tak rozpoczęta komunikacja może zostać zakończona przez agenta B lub C. Ten drugi przypadek ma miejsce, gdy w trakcie wykonywania instrukcji *in* procedura $C.p1$ była niedostępna, ale została udostępniona później i realizacja tej procedury sfinalizowała komunikację.
- $TIn(B.b)$ – Reprezentuje wykonanie instrukcji *in* przez agenta B (port $B.b$), jako inicjalizowanie komunikacji, która może być sfinalizowana tylko przez agenta A.
- $TInF(A.a, B.b)$ – Reprezentuje wykonanie instrukcji *in* przez agenta aktywnego A (port $A.a$), jako finalizacja komunikacji z aktywnym agentem B (port $B.b$). Agent B musiał wcześniej zainicjalizować komunikację wykonując instrukcję *out* na porcie $B.b$.
- $TInF(B.b, A.a)$ – Reprezentuje wykonanie instrukcji *in* przez agenta aktywnego B (port $B.b$), jako



Rysunek 4.1: Przykład diagramu komunikacji dla modelu stworzonego w języku Alvis

finalizacja komunikacji z aktywnym agentem A (port $A.a$). Agent A musiał wcześniej zainicjalizować komunikację wykonując instrukcję *out* na porcie $A.a$.

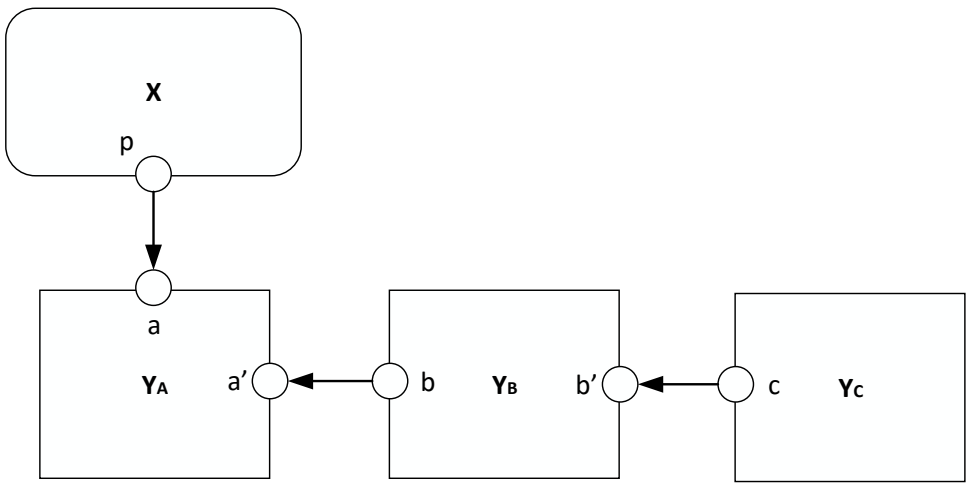
- $TInAP(A.a, C.p1)$ – Reprezentuje wykonanie instrukcji *in* przez agenta aktywnego A (port $A.a$), jako wywołanie dostępnej wyjściowej procedury $C.p1$.
- $TInPP(C.c, D.p4)$ – Reprezentuje wykonanie instrukcji *in* przez agenta pasywnego C (port $C.c$), jako wywołanie dostępnej wyjściowej procedury $D.p4$.
- $TOut(A.a)$ – Reprezentuje wykonanie instrukcji *out* przez agenta aktywnego A (port $A.a$), jako inicjalizowanie komunikacji z aktywnym agentem B (w tym modelu nie ma innej możliwości).
- $TOut(B.b)$ – Reprezentuje wykonanie instrukcji *out* przez agenta aktywnego B (port $B.b$), jako inicjalizowanie komunikacji. Tak rozpoczęta komunikacja może zostać zakończona przez agenta A lub C. Ten drugi przypadek ma miejsce, gdy w trakcie wykonywania instrukcji *out* procedura $C.p2$ była niedostępna, ale została udostępniona później i realizacja tej procedury sfinalizowała komunikację.
- $TOutF(A.a, B.b)$ – Reprezentuje wykonanie instrukcji *out* przez agenta aktywnego A (port $A.a$), jako finalizacja komunikacji z aktywnym agentem B (port $B.b$). Agent B musiał wcześniej zainicjalizować komunikację wykonując instrukcję *in* na porcie $B.b$.
- $TOutF(B.b, A.a)$ – Reprezentuje wykonanie instrukcji *out* przez agenta aktywnego B (port $B.b$), jako finalizacja komunikacji z aktywnym agentem A (port $A.a$). Agent A musiał wcześniej zainicjalizować komunikację wykonując instrukcję *in* na porcie $A.a$.
- $TOutAP(B.b, C.p2)$ – Reprezentuje wykonanie instrukcji *out* przez agenta aktywnego B (port $B.b$), jako wywołanie dostępnej procedury wejściowej $C.p2$.

- $TOutPP(C.c, D.p3)$ – Reprezentuje wykonanie instrukcji *out* przez agenta pasywnego C (port $C.c$), jako wywołanie dostępnej wejściowej procedury $D.p3$.

Zachowanie modelu utworzonego w Alvisie jest rozważane na poziomie pojedynczych tranzycji. Każda tranzycja realizowana jest w kontekście jednego agenta aktywnego. Dotyczy to także procedur agentów pasywnych, które realizowane są w kontekście agenta aktywnego, który dana procedurę wywołał. W celu opisania tranzycji pomiędzy stanami modelu pomocne będzie wprowadzenie funkcji $context(X)$ i $caller(X)$.

Formalnym parametrem funkcji $context(X)$ jest nazwa agenta pasywnego, a wartością zwracaną przez tą funkcję jest nazwa agenta aktywnego, w kontekście którego została wywołana procedura tego agenta pasywnego. Za pomocą tej funkcji można się więc dowiedzieć, który z agentów aktywnych przejął danego agenta pasywnego poprzez wywołanie na nim procedury. Posługując się przykładem zaprezentowanym na rysunku 4.1 można wyobrazić sobie sytuację, w której agent aktywny A wywołał procedurę $C.p1$, w ramach której wywołana została procedura $D.p1$ pasywnego agenta D. W takiej sytuacji procedura $D.p1$ agenta pasywnego D jest realizowana w kontekście agenta aktywnego A, tj. $context(D) = A$.

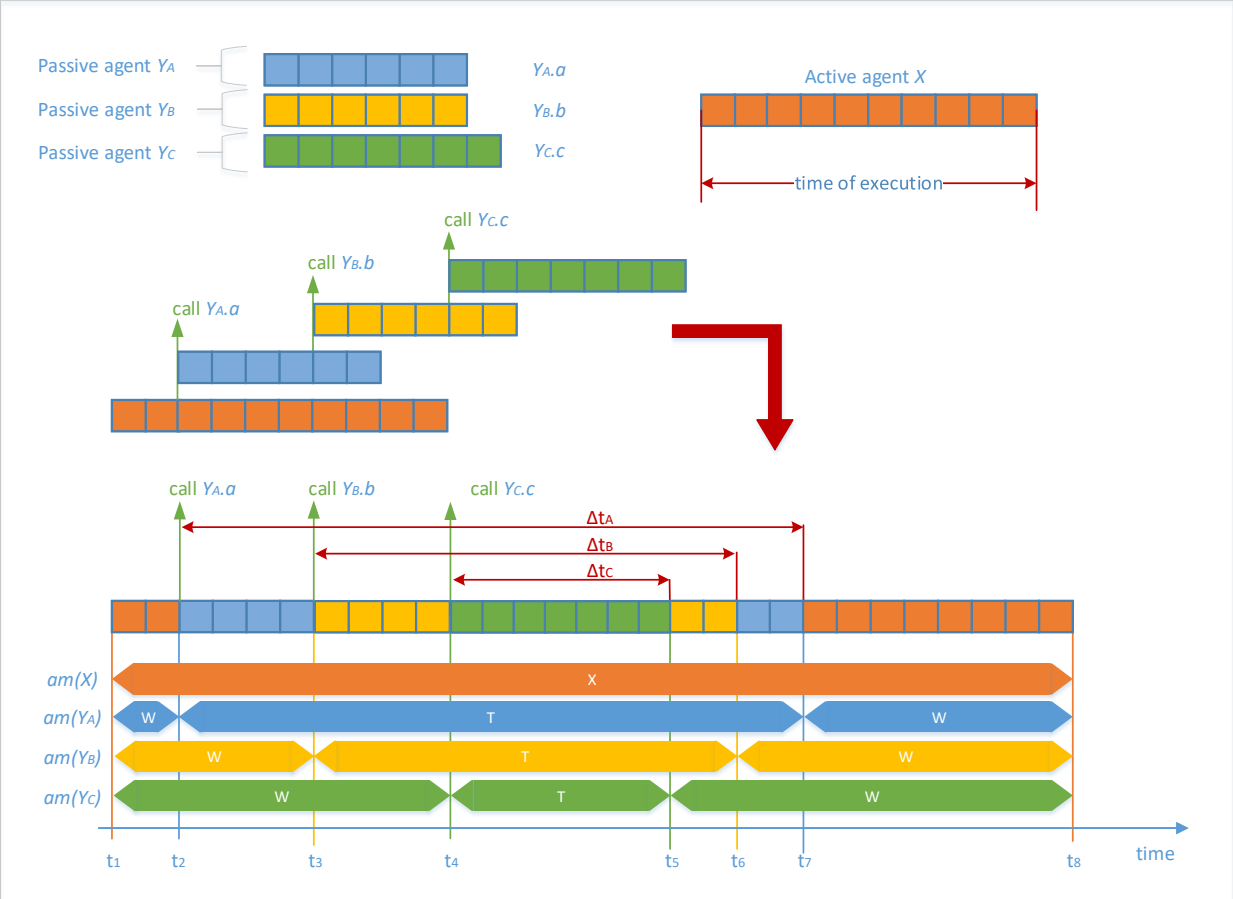
Używając funkcji $caller(X)$ można uzyskać informację, który agent bezpośrednio wywołał daną procedurę. Formalnym parametrem tej funkcji jest nazwa agenta pasywnego, a wartością zwracaną nazwa agenta, który w sposób bezpośredni wywołał procedurę danego agenta pasywnego. Posługując się ponownie przykładem zaprezentowanym na rysunku 4.1 można zaobserwować, że agent pasywny C wywołuje bezpośrednio procedurę $D.p3$ pasywnego agenta D, dlatego rezultatem wykonania funkcji $caller$ będzie: $caller(D) = C$. Jak wspomniano funkcja $caller$ może również zwracać nazwy agentów aktywnych i taki przypadek można zaobserwować w analizowanym przykładzie, gdy aktywny agent B wywołuje bezpośrednio procedurę $C.p2$ pasywnego agenta C, w takim przypadku $caller(C) = B$.



Rysunek 4.2: Przykład diagramu komunikacji pomiędzy agentem aktywnym i połączoną strukturą agentów pasywnych

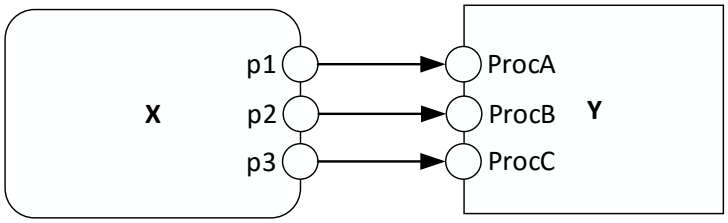
Rozważmy teraz przypadek, gdzie agent aktywny X wywołuje procedurę agenta pasywnego, który to komunikuje się poprzez kanał komunikacyjny z innym agentem pasywnym, ten agent pasywny z kolejnym itd. Przykład takiej komunikacji pokazany jest na rysunku 4.2, a aspekty czasowe dla powyższego przykładu zaprezentowano na rysunku 4.3.

Aktywny agent X wywołuje procedurę $Y_A.a$ pasywnego agenta Y_A dlatego można powiedzieć, że $context(Y_A) = X$, co obrazuje, że agent pasywny Y_A został wywołany w kontekście aktywnego agenta X. Rezultat stwierdzający, iż $caller(Y_A) = X$ pokazuje, że aktywny agent X bezpośrednio wywołał procedurę pasywnego agenta Y_A. W ten sam sposób, kolejne rezultaty dla funkcji $context$ i $caller$ mogą być przedstawione następująco: $caller(Y_B) = Y_A$, $context(Y_B) = X$, $caller(Y_C) = Y_B$, $context(Y_C) = X$. W końcowym rezultacie można zaobserwować, że każdy agent pasywny jest uruchamiany w kontekście agenta aktywnego X.

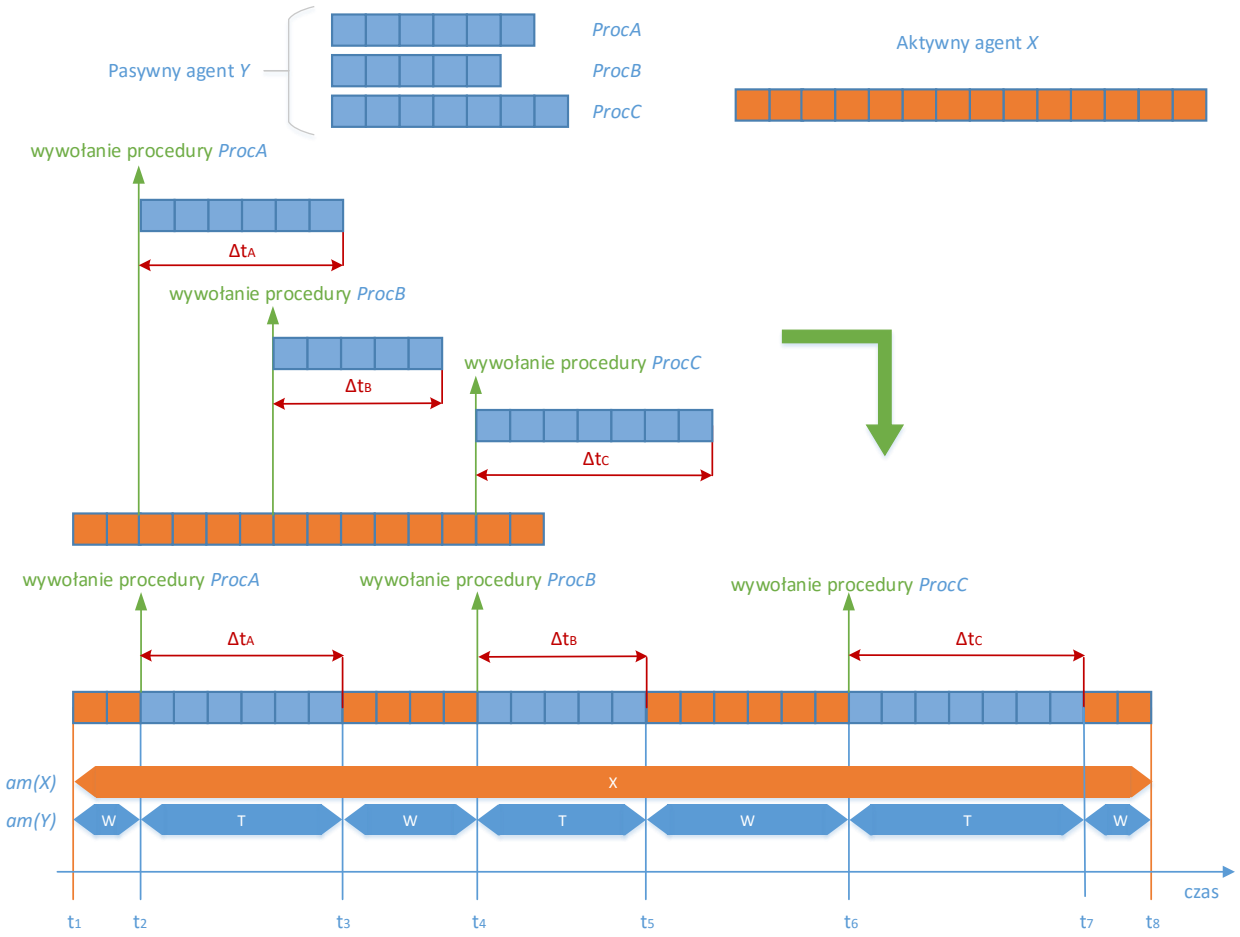


Rysunek 4.3: Aspekty czasowe dla przykładu komunikacji pomiędzy agentem aktywnym i połączoną strukturą agentów pasywnych

Biorąc pod uwagę aspekty czasowe, należy zauważyć, że czas egzekucji danego agenta aktywnego X wzrasta o sumę czasów konsumowanych przez agenty pasywne, które to zostały przez danego agenta aktywnego uruchomione: Y_A, Y_B i Y_C.



Rysunek 4.4: Diagram komunikacji prezentujący wywołanie procedur przez agenta aktywnego

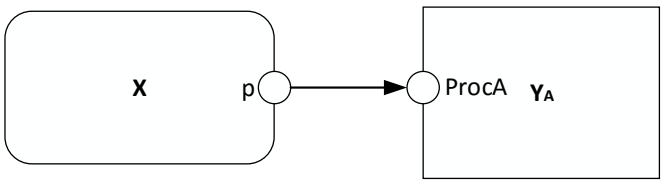


Rysunek 4.5: Przykład konsumpcji czasu w przypadku wywołania procedur przez agenta aktywnego

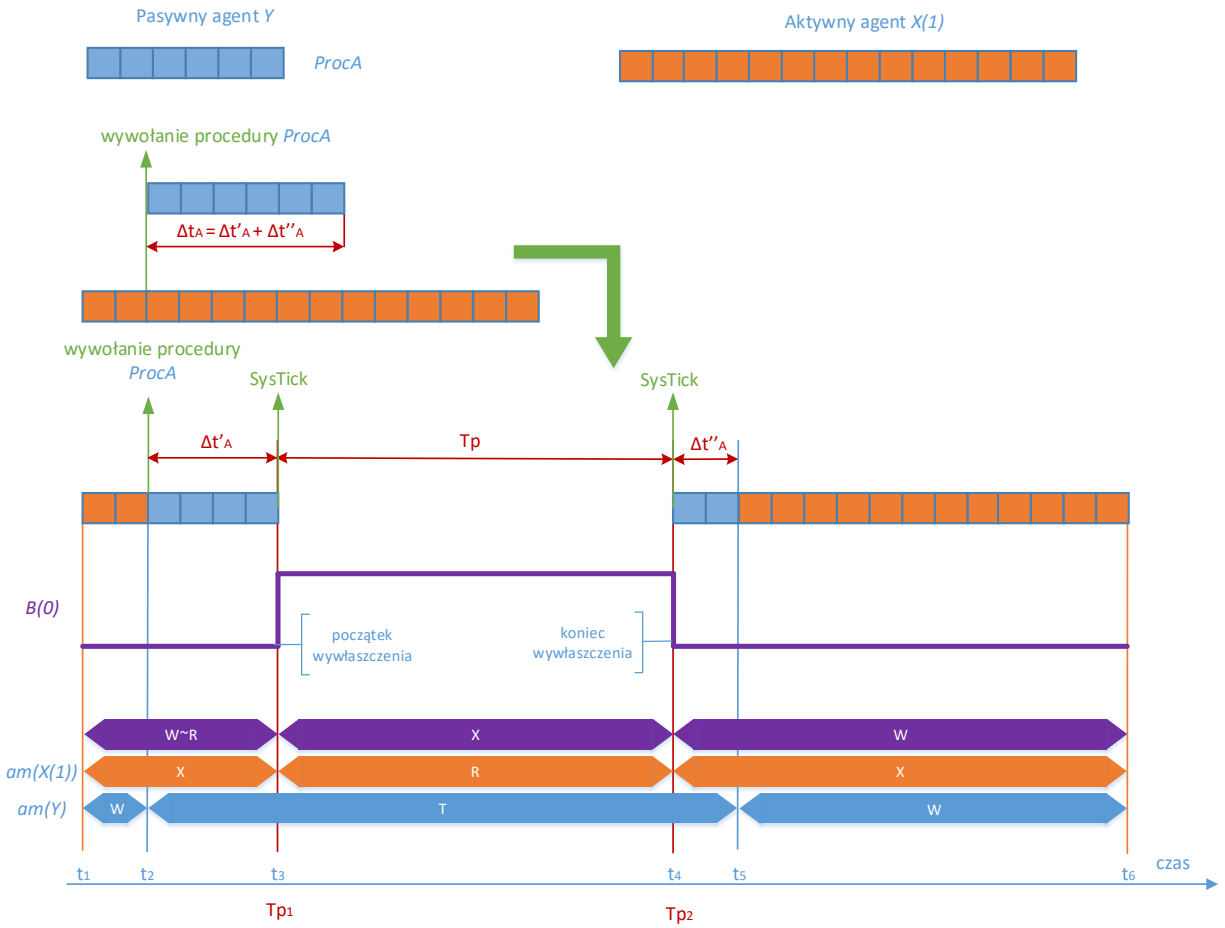
Zależności czasowe podczas komunikacji pomiędzy agentem aktywnym i pasywnym są także widoczne na przykładzie zademonstrowanym na rysunku 4.5. Diagram komunikacyjny dla tego przykładu został przedstawiony na rysunku 4.4. Agent aktywny X wywołuje procedury $ProcA$, $ProcB$ and $ProcC$ pasywnego agenta Y. W konsekwencji czas egzekucji tych procedur zwiększa czas egzekucji agenta X.

W przypadku warstwy systemowej α^1_{FPPS} należy również rozważyć przypadki, w których może nastą-

pić wywłaszczenie agenta aktywnego w czasie, gdy w jego kontekście uruchomiony jest agent pasywny. Sytuacja taka może wydarzyć się podczas przerwania systemowego $SysTick$, po którym dany agent aktywny zostaje wywłaszczony na rzecz innego agenta aktywnego. Przeanalizujmy przykład zaprezentowany na rysunku 4.6.



Rysunek 4.6: Przykład wywołania procedury agenta pasywnego przez agenta aktywnego



Rysunek 4.7: Przykład wywłaszczenia agenta aktywnego podczas wywołania procedury agenta pasywnego

Aspekty czasowe zostały zobrazowane graficznie na rysunku 4.7. W analizowanym przykładzie agent aktywny X jest wywłaszczany na rzecz innego agenta aktywnego B z większym priorytetem. W momencie czasowym T_{p1} następuje przerwanie systemowe $SysTick$, w konsekwencji czego wywoływany jest algo-

rytm szeregujący. W rezultacie agent B jest promowany do trybu *running*, a agent aktywny X przechodzi do trybu gotowości *ready*. Ponieważ agent X wywołał wcześniej, jeszcze przed przerwaniem systemowym, procedurę $Y_A.ProcA$, dlatego też agent Y zostaje w trybie *taken*. Dochodzi tutaj do sytuacji, że od tego konkretnego momentu czasowego T_{p1}, aż do przywrócenia agenta X do trybu *running* i zwolnienia przez niego agenta Y, żadna procedura tego agenta (w tym przypadku $Y_A.ProcA$) nie może być wywołana przez innego agenta aktywnego. W momencie czasowym T_{p2} agent aktywny X ponownie znajduje się w trybie *running* i procedura wywołana w jego kontekście może zostać dokończona.

Do opisu tranzycji przyjmiemy następujące oznaczenia i zdefiniujemy pewne funkcje:

- S reprezentuje bieżący stan modelu, w którym wykonywana jest dana tranzycja.
- S' reprezentuje stan modelu, jaki jest osiągany po wykonaniu danej tranzycji.
- $statement(n)$ jest funkcją, która zwraca nazwę instrukcji w warstwie kodu danego agenta na podstawie jej numeru n.
- $nextpc_S(X)$ jest to funkcja, która zwraca numer kolejnej instrukcji w odniesieniu do bieżącego stanu S. Wartym wspomnienia jest przypadek, w którym wykonywana jest tranzycja t_{loop} z warunkiem logicznym g. Istnieją tu dwie możliwości w zależności od tego, czy warunek logiczny g jest spełniony, czy też nie. W pierwszym wypadku funkcja $nextpc_S(X)$ zwróci numer pierwszej instrukcji znajdującej się w bloku pętli. W drugim wypadku, ponieważ warunek logiczny nie został spełniony, funkcja $nextpc_S(X)$ wskaże numer pierwszej instrukcji znajdującej się zaraz po w bloku pętli (nie w jej środku). Specjalny przypadek wystąpi również, gdy instrukcja *loop* jest ostatnia instrukcją w kodzie agenta i warunek logiczny g nie został spełniony. W takim układzie funkcja $nextpc_S(X)$ zwróci wartość zero.

Listing 4.1: Przykład agenta aktywnego z pętlą *loop*

```
agent OrSe(0) {
  serviceType :: String = "";
  serviceUUID :: Int = 0;

  choose_service_type:
  in service_type serviceType;                                    -- 1
  loop (serviceType == "application" || serviceType == "support") { -- 2
    select {                                                      -- 3
      alt (serviceType == "application") {
        serviceUUID = 100; }                                      -- 4
      alt (serviceType == "support") {
        serviceUUID = 200; }                                      -- 5
    }
  }
```

```
  jump choose_service_type;                                    -- 6
}
```

W przykładzie prezentowanym na listingu 4.1 możliwe są dwie sytuacje. Pierwsza wystąpi wtedy, gdy na porcie wejściowym *service_type* zostanie dostarczona, poprzez kanał komunikacji, wartość typu string równa *„application”* lub *„support”*. W tym wypadku warunek logiczny instrukcji *loop* zostanie spełniony i funkcja $nextpc_S(X)$ zwróci wartość równą 3 – numer pierwszej instrukcji zawartej w jej bloku. Druga możliwość wystąpi wtedy, gdy na porcie wejściowym *service_type* agent aktywny $OrSe$ otrzyma inną wartość niż *„application”* lub *„support”*. W tym wypadku warunek logiczny instrukcji *loop* nie będzie spełniony i funkcja $nextpc_S(X)$ zwraca wartość równą 6 – numer pierwszej instrukcji znajdującej się zaraz za blokiem pętli $loop$. W tym konkretnym przypadku jest to instrukcja *jump*, która przenosi wykonywanie instrukcji ponownie do miejsca, w którym agent $OrSe$ oczekuje na wiadomość mówiącą o typie serwisu, niezbędną do przypisania właściwego numeru UUID.

- Ponieważ *lista kontekstowa* agenta jest traktowana jako zbiór w ujęciu matematycznym, dlatego operacje na liście będą zapisywane z użyciem symboli: $\cup, \setminus, \in$.
- $arg_1(c)$, $arg_2(c)$, $arg_3(c)$, ..., $arg_n(c)$ – oznaczają kolejno pierwszy, drugi, trzeci i n-ty argument instrukcji c. Wyjątkiem jest tutaj instrukcja *null*, która nie ma argumentów.

Listing 4.2: Przykład instrukcji agenta z argumentami

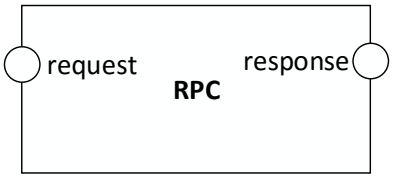
```
agent ISS(0) {
  analogData :: Int = 0;
  analog :: Int = 0;

  loop (every 200) {                          -- 1
    in (10) getData analog;                   -- 2
    analogData = analogData + analog;         -- 3
    null;                                     -- 4
  }
}
```

Rozpatrzmy kod agenta ISS zaprezentowany na listingu 4.2. Instrukcja in ma trzy argumenty, kolejno $arg_1(in) = 10$, $arg_2(in) = getData$ and $arg_3(in) = analog$. W instrukcji $exec$ wyodrębnia się dwa argumenty. Pierwszym jest nazwa parametru po lewej stronie znaku $=$, a drugim jest wartość wyrażenia po prawej stronie znaku $=$. Biorąc powyższe pod uwagę, można powiedzieć, że $arg_1(exec) = analogData$ i $arg_2(exec)$ jest wartością otrzymaną z wyrażenia $analogData + analog$. Instrukcja $null$, jak wspomniano wcześniej, nie posiada żadnych argumentów.

- $procedures_S(X)$ jest funkcją, która zwraca listę wszystkich dostępnych procedur agenta pasywnego

X, które są dostępne w danym stanie modelu S. Istnieją tu dwie możliwości w zależności od tego, w jakim obecnie trybie znajduje się dany agent pasywny. Pierwszy przypadek dotyczy sytuacji, w której agent pasywny znajduje się w trybie *waiting*, który oznacza, że żaden agent zewnętrzny nie wywołał żadnej jego procedury. W takim przypadku funkcja $procedures_S(X)$ zwróci listę wszystkich procedur, których warunki logiczne są spełnione (brak warunku logicznego domyślnie oznacza, że jest on spełniony). Druga możliwość dotyczy sytuacji, gdy agent pasywny jest w trybie *taken*, co oznacza, że jedna z jego procedur została wywołana przez innego agenta. W takiej sytuacji funkcja $procedures_S(X)$ zwróci pustą listę, gdyż agent pasywny nie może być uruchomiony w kontekście więcej niż jednego agenta.



Rysunek 4.8: Przykład agenta pasywnego z dostępnymi procedurami w trybie *waiting*

Listing 4.3: Przykład definicji agenta pasywnego z procedurami

```
agent RPC {
  typeOfMessage :: String = "typeRequest";
  requestMessage :: String = "";
  responseMessage :: String = "";

  proc (typeOfMessage == "typeRequest") request {
    typeOfMessage = "typeResponse";                    -- 1
    in request requestMessage;                         -- 2
    exit;                                              -- 3
  }

  proc (typeOfMessage == "typeResponse") response {
    typeOfMessage = "typeRequest";                     -- 4
    responseMessage = "response on the request: "      -- 5
         + requestMessage;
    out response responseMessage;                      -- 6
    exit;                                              -- 7
  }
}
```

Rozważmy przykład z listingu 4.3. Graficzna reprezentacja agenta pasywnego RPC pokazana jest na rysunku 4.8. Agent RPC ma dwa porty proceduralne: wejściowy *request* i wyjściowy *response*. Gdy agent RPC jest w trybie *waiting* i parametr $typeOfMessage$ jest ustawiony na $typeRequest$, wtedy funkcja $procedures_S(X)$ zwraca procedurę $request$ jako dostępną. Gdy agent RPC jest w trybie *waiting* i parametr $typeOfMessage$ jest ustawiony na $typeResponse$, funkcja $procedures_S(X)$ zwraca procedurę $response$ jako dostępną w danym momencie czasowym. W przypadku, gdy agent RPC będzie pracował w trybie *taken*, wtedy funkcja $procedures_S(X)$ zwróci pustą listę.

Niech $pv_S(X)|_{v=a}$ oznacza listę wartości parametrów $pv_S(X)$, taką że parametrowi v przypisywana jest nowa wartość a.

Na tym etapie opisu dynamiki modeli rozważone zostaną zostaną zmiany stanu wynikające jedynie z wykonywania tranzycji, bez uwzględnienia działania algorytmu szeregującego oraz upływu czasu. Aspekty te opisane zostały w kolejnych rozdziałach niniejszej rozprawy doktorskiej.

Wymagania dotyczące aktywności są takie same dla większości tranzycji, dla przykładu tranzycja $TCritical(X)$ jest możliwa do wykonania wtedy i tylko wtedy, gdy:

- Jeżeli $X \in \mathcal{A}_A$, to $am_S(X) = \mathsf{X}$ i $statement(pc_S(X)) = critical$,
- Jeżeli $X \in \mathcal{A}_P$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{X}$, i $statement(pc_S(X)) = critical$,

Podobnie, wymagania dotyczące aktywności są definiowane dla: $TDelay(X)$, $TExec(X)$, $TExit(X)$, $TJump(X)$, $TLoop(X)$, $TLoopEvery(X)$, $TNull(X)$, $TSelect(X)$, and $TStart(X)$. Jedyną różnicą jest prawa strona ostatniego warunku (zarówna dla agentów aktywnych, jak i pasywnych), gdzie zawarta jest nazwa odpowiedniej instrukcji.

Wymagania dotyczące aktywności dla innych tranzycji zostaną przedstawione razem z opisem zmian stanu jakie one powodują. W celu uproszczenia zapisu przyjęto, że c oznacza nazwę bieżącej instrukcji tj. $c = statement(pc_S(X))$.

► $TCritical(X)$

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$, $ci_{S'}(X) = ci_S(X) \cup \{critical\}$

► $TDelay(X)$

Modyfikacje stanu:

Jeśli $X \in \mathcal{A}_A$, to $am_{S'}(X) = \mathsf{W}$, $ci_{S'}(X) = ci_S(X) \cup \{timer(pc_S(X), arg_1(c))\}$.

Jeśli $X \in \mathcal{A}_P$, to $am_{S'}(context(X)) = \mathsf{W}$, $ci_{S'}(X) = ci_S(X) \cup \{timer(pc_S(X), arg_1(c))\}$.

► $TExec(X)$

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$, $pv_{S'}(X) = pv_S(X)|_{agr_1(c)=arg_2(c)}$

Ponadto, jeśli $X \in \mathcal{A}_A$ i $nextpc_S(X) = 0$, wtedy $am_{S'}(X) = \mathsf{F}$, $ci_{S'}(X) = [\,]$.

► $TExit(X)$

Modyfikacje stanu:

Jeśli $X \in \mathcal{A}_A$, to $am_{S'}(X) = \mathsf{F}$, $pc_{S'}(X) = 0$, $ci_{S'}(X) = [\,]$.

Jeśli $X \in \mathcal{A}_P$, to $am_{S'}(X) = \mathsf{W}$, $pc_{S'}(X) = 0$, $ci_{S'}(X) = procedures_{S'}(X)$, $pc_{S'}(Y) = nextpc_S(Y)$, i $ci_{S'}(Y) = ci_S(Y) \setminus \{proc(X.p)\}$, gdzie $X.p$ jest kończoną procedurą i $Y = caller(X)$.

Ponadto, jeśli $caller(X) = context(X)$ i $nextpc_S(Y) = 0$ wtedy $am_{S'}(Y) = \mathsf{F}$, $ci_{S'}(Y) = [\,]$.

► $TInF(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X, Y \in \mathcal{A}_A$, $(Y.q, X.p) \in \mathcal{C}$[1], $am_S(X) = \mathsf{X}$, $c = in$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$,
$arg_1(c) = X.p$ lub $arg_2(c) = X.p$, $am_S(Y) = \mathsf{W}$, i $out(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$, $pc_{S'}(Y) = nextpc_S(Y)$, $am_{S'}(Y) = \mathsf{R}$,
$ci_{S'}(Y) = ci_S(Y) \setminus \{out(Y.q), timer(pc_S(Y), \_)\}$, gdzie znak podkreślenia oznacza liczbę całkowitą (aktualny parametr zegara).

Ponadto:

Jeśli stosowana jest komunikacja z przesyłaniem danych i jest to komunikacja blokująca, wtedy $pv_{S'}(X) = pv_S(X)|_{agr_1(c)=arg_2(c)}$

Jeśli stosowana jest komunikacja z przesyłaniem danych i jest to komunikacja nieblokująca, wtedy $pv_{S'}(X) = pv_S(X)|_{agr_2(c)=arg_3(c)}$

Jeśli $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$, i $ci_{S'}(X) = [\,]$.

Jeśli $nextpc_S(Y) = 0$, to $am_{S'}(Y) = \mathsf{F}$, i $ci_{S'}(Y) = [\,]$.

► $TInAP(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X \in \mathcal{A}_A$, $Y \in \mathcal{A}_P$, $(Y.q, X.p) \in \mathcal{C}$, $am_S(X) = \mathsf{X}$, $c = in$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$,
$arg_1(c) = X.p$ lub $arg_2(c) = X.p$, $am_S(Y) = \mathsf{W}$ i $out(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$ci_{S'}(X) = ci_S(X) \cup \{proc(Y.q)\}$, $am_{S'}(Y) = \mathsf{T}$, $pc_{S'}(Y)$ przypisuje numer pierwszej instrukcji wewnątrz procedury $Y.q$, i $ci_{S'}(Y) = [\,]$.

► $TInPP(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X, Y \in \mathcal{A}_P$, $(Y.q, X.p) \in \mathcal{C}$, $X.p \notin \mathcal{P}_{proc}$, $am_S(X) = \mathsf{T}$, $c = in$, $am_S(context(X)) = \mathsf{X}$,
$\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$, $arg_1(c) = X.p$ lub $arg_2(c) = X.p$, $am_S(Y) = \mathsf{W}$ i $out(Y.q) \in ci_S(Y)$.

[1] $\mathcal{C}$ jest relacją komunikacji (Def. 2.1, strona 18).

Modyfikacje stanu:

$ci_{S'}(X) = ci_S(X) \cup \{proc(Y.q)\}$, $am_{S'}(Y) = \mathsf{T}$, $pc_{S'}(Y)$ przypisuje numer pierwszej instrukcji wewnątrz procedury $Y.q$ i $ci_{S'}(Y) = [\,]$.

► $TIn(X.p)$

Wymagania dotyczące aktywności:

Jeśli $X \in \mathcal{A}_A$, to $am_S(X) = \mathsf{X}$, $c = in$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$, $arg_1(c) = X.p$ lub $arg_2(c) = X.p$ i tranzycje $TInF(X.p, \_)$, $TInAP(X.p, \_)$ nie są aktywne w stanie S, gdzie znak podkreślenia określa tu dowolny port połączony z $X.p$.

Jeśli $X \in \mathcal{A}_P$ i $X.p \in \mathcal{P}_{proc}$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{X}$ i $c = in$

Jeśli $X \in \mathcal{A}_P$ i $X.p \notin \mathcal{P}_{proc}$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{X}$, $c = in$, $arg_1(c) = X.p$ lub $arg_2(c) = X.p$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$ i tranzycja $TInPP(X.p, \_)$ nie jest aktywna w stanie S.

Modyfikacje stanu:

- Jeśli $X \in \mathcal{A}_A$ i użyta jest blokująca instrukcja *in*, to $am_{S'}(X) = \mathsf{W}$ i $ci_{S'}(X) = ci_S(X) \cup \{in(X.p)\}$.
- Jeśli $X \in \mathcal{A}_A$, użyta jest nieblokująca instrukcja *in* i $arg_1(c) = 0$, to $pc_{S'}(X) = nextpc_S(X)$.
  Ponadto, jeśli $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.
- Jeśli $X \in \mathcal{A}_A$, użyta jest nieblokująca instrukcja *in* i $arg_1(c) > 0$, to $am_{S'}(X) = \mathsf{W}$
  i $ci_{S'}(X) = ci_S(X) \cup \{in(X.p), timer(pc_S(X), arg_1(c))\}$.
- Jeśli $X \in \mathcal{A}_P$ i $X.p \in \mathcal{P}_{proc}$, to $pc_{S'}(X) = nextpc_S(X)$.
  Ponadto, jeśli użyta jest komunikacja z przesyłaniem wartości, to $pv_{S'}(X) = pv_S(X)|_{agr_1(c)=arg_2(c)}$.
- Jeśli $X \in \mathcal{A}_P$, $X.p \notin \mathcal{P}_{proc}$ i użyta jest blokująca instrukcja *in*, to $am_{S'}(context(X)) = \mathsf{W}$
  i $ci_{S'}(X) = ci_S(X) \cup \{in(X.p)\}$.
- Jeśli $X \in \mathcal{A}_P$, $X.p \notin \mathcal{P}_{proc}$, użyta jest nieblokująca instrukcja *in* i $arg_1(c) = 0$, to $pc_{S'}(X) = nextpc_S(X)$.
- Jeśli $X \in \mathcal{A}_P$, $X.p \notin \mathcal{P}_{proc}$, użyta jest nieblokująca instrukcja *in* i $arg_1(c) > 0$, to
  $am_{S'}(context(X)) = \mathsf{W}$ i $ci_{S'}(X) = ci_S(X) \cup \{in(X.p), timer(pc_S(X), arg_1(c))\}$.

► $TJump(X)$

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$

► $TLoop(X)$

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$

Ponadto, jeśli $X \in \mathcal{A}_A$ i $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.

► $TLoopEvery(X)$

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$ i $ci_{S'}(X) = ci_S(X) \cup \{timer(pc_S(X), arg_1(c) - d)\}$, gdzie d jest czasem trwania instrukcji *loop every*.

► $TNull(X)$

Modyfikacje stanu:

- Jeśli $X \in \mathcal{A}_A$ i c jest ostatnią instrukcją w środku instrukcji *loop every* wtedy $am_{S'}(X) = \mathsf{W}$ i $pc_{S'}(X)$ jest przypisany numer instrukcji *loop every*.
- Jeśli $X \in \mathcal{A}_P$ i c jest ostatnią instrukcją w środku instrukcji *loop every*, wtedy $am_{S'}(context(X)) = \mathsf{W}$ i $pc_{S'}(X)$ jest przypisany numer instrukcji *loop every*.
- W każdym innym wypadku $pc_{S'}(X) = nextpc_S(X)$

  Ponadto:

  Jeśli c jest ostatnią instrukcją w bloku *critical*, to $ci_{S'}(X) = ci_S(X) \setminus \{critical\}$.

  Jeśli $X \in \mathcal{A}_A$ i $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.

► $TOutF(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X, Y \in \mathcal{A}_A$, $(X.p, Y.q) \in \mathcal{C}$, $am_S(X) = \mathsf{X}$, $c = out$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$,

$arg_1(c) = X.p$ lub $arg_2(c) = X.p$, $am_S(Y) = \mathsf{W}$ i $in(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$, $pc_{S'}(Y) = nextpc_S(Y)$, $am_{S'}(Y) = \mathsf{R}$,

$ci_{S'}(Y) = ci_S(Y) \setminus \{in(Y.q), timer(pc_S(Y), \_)\}$, gdzie znak podkreślenia oznacza liczę całkowitą (aktualny parametr zegara).

Ponadto:

Jeśli stosowana jest komunikacja z przesyłaniem danych i jest to komunikacja blokująca, wtedy $pv_{S'}(Y) = pv_S(Y)|_{agr_1(c)=arg_2(c)}$

Jeśli stosowana jest komunikacja z przesyłaniem danych i jest to komunikacja nieblokująca, wtedy $pv_{S'}(Y) = pv_S(Y)|_{agr_2(c)=arg_3(c)}$

Jeśli $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.

Jeśli $nextpc_S(Y) = 0$, to $am_{S'}(Y) = \mathsf{F}$ i $ci_{S'}(Y) = [\,]$.

► $TOutAP(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X \in \mathcal{A}_A$, $Y \in \mathcal{A}_P$, $(X.p, Y.q) \in \mathcal{C}$, $am_S(X) = \mathsf{X}$, $c = out$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$,

$arg_1(c) = X.p$ lub $arg_2(c) = X.p$, $am_S(Y) = \mathsf{W}$, i $in(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$ci_{S'}(X) = ci_S(X) \cup \{proc(Y.q)\}$, $am_{S'}(Y) = \mathsf{T}$, $pc_{S'}(Y)$ przypisuje numer pierwszej instrukcji w środku procedury $Y.q$ i $ci_{S'}(Y) = [\,]$.

► $TOutPP(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X, Y \in \mathcal{A}_P$, $(X.p, Y.q) \in \mathcal{C}$, $X.p \notin \mathcal{P}_{proc}$, $am_S(X) = \mathsf{T}$, $c = out$, $am_S(context(X)) = \mathsf{X}$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$, $arg_1(c) = X.p$ lub $arg_2(c) = X.p$, $am_S(Y) = \mathsf{W}$ i $in(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$ci_{S'}(X) = ci_S(X) \cup \{proc(Y.q)\}$, $am_{S'}(Y) = \mathsf{T}$, $pc_{S'}(Y)$ przypisuje numer pierwszej instrukcji wewnątrz procedury $Y.q$ i $ci_{S'}(Y) = [\,]$.

► $TOut(X.p)$

Wymagania dotyczące aktywności:

Jeśli $X \in \mathcal{A}_A$, to $am_S(X) = \mathsf{X}$, $c = out$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$, $arg_1(c) = X.p$ lub $arg_2(c) = X.p$ i tranzycje $TOutF(X.p, \_)$, $TOutAP(X.p, \_)$ nie są aktywne w stanie S, gdzie znak podkreślenia określa tu dowolny port połączony z $X.p$.

If $X \in \mathcal{A}_P$ i $X.p \in \mathcal{P}_{proc}$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{X}$ i $c = out$

If $X \in \mathcal{A}_P$ i $X.p \notin \mathcal{P}_{proc}$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{X}$, $c = out$, $arg_1(c) = X.p$ lub $arg_2(c) = X.p$, $\forall_{Z.r \in \mathcal{P}_{proc}}\ proc(Z.r) \notin ci_S(X)$ i tranzycja $TOutPP(X.p, \_)$ nie jest aktywna w stanie S.

Modyfikacje stanu:

- Jeśli $X \in \mathcal{A}_A$ i użyta jest blokująca instrukcja *out*, to $am_{S'}(X) = \mathsf{W}$ i $ci_{S'}(X) = ci_S(X) \cup \{out(X.p)\}$.
- Jeśli $X \in \mathcal{A}_A$, użyta jest nieblokująca instrukcja *out* i $arg_1(c) = 0$, to $pc_{S'}(X) = nextpc_S(X)$.
  Ponadto, jeśli $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.
- Jeśli $X \in \mathcal{A}_A$, użyta jest nieblokująca instrukcja *out* i $arg_1(c) > 0$, wtedy $am_{S'}(X) = \mathsf{W}$ i $ci_{S'}(X) = ci_S(X) \cup \{out(X.p), timer(pc_S(X), arg_1(c))\}$.
- Jeśli $X \in \mathcal{A}_P$ i $X.p \in \mathcal{P}_{roc}$, to $pc_{S'}(X) = nextpc_S(X)$.
  Ponadto, jeśli użyta jest komunikacja z przesyłaniem wartości, to aktualizowany jest parametr użyty w wywołaniu procedury po stronie agenta wywołującego.
- Jeśli $X \in \mathcal{A}_P$, $X.p \notin \mathcal{P}_{proc}$ i użyta jest blokująca instrukcja *out*, to $am_{S'}(context(X)) = \mathsf{W}$ i $ci_{S'}(X) = ci_S(X) \cup \{out(X.p)\}$.
- Jeśli $X \in \mathcal{A}_P$, $X.p \notin \mathcal{P}_{proc}$, użyta jest nieblokująca instrukcja *out* i $arg_1(c) = 0$, to $pc_{S'}(X) = nextpc_S(X)$.
- Jeśli $X \in \mathcal{A}_P$, $X.p \notin \mathcal{P}_{proc}$, użyta jest nieblokująca instrukcja *out* i $arg_1(c) > 0$, to $am_{S'}(context(X)) = \mathsf{W}$ i $ci_{S'}(X) = ci_S(X) \cup \{out(X.p), timer(pc_S(X), arg_1(c))\}$.

► $TSelect(X)$

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$

Ponadto, jeśli $X \in \mathcal{A}_A$ i $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.

► $TStart(X)$

Modyfikacje stanu:

$pc_{S'}(X) = nextpc_S(X)$

Ponadto:

Jeśli $am_S(Y) = \mathsf{I}$, to $am_{S'}(Y) = \mathsf{R}$ i $pc_{S'}(Y) = 1$, gdzie $Y = arg_1(c)$.

Jeśli $X \in \mathcal{A}_A$ i $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.

4.2. Tranzycje systemowe

Pełen opis procesu zmian stanów modelu wymaga także opisania tzw. *tranzycji systemowych*. Przykładem tranzycji systemowej jest tranzycja reprezentująca przerwanie systemowe $SysTick$, w wyniku którego uruchamiany jest algorytm szeregujący dla warstwy systemowej α^1_{FPPS}. Dodatkowo tranzycje systemowe występują w sytuacjach takich jak: koniec trwania zawieszenia agenta po wykonaniu instrukcji *delay*, przeterminowanie związane z oczekiwaniem na komunikację zapoczątkowaną instrukcją nieblokującą, koniec okresu pętli *loop every* itp.

- $STInAP(X.p, Y.q)$, $STInPP(X.p, Y.q)$, $STOutAP(X.p, Y.q)$, $STOutPP(X.p, Y.q)$ – Te tranzycje systemowe reprezentują sytuacje, gdzie po upływie oczekiwania na niedostępne procedury agent, który zainicjalizował komunikację (w tym wypadku X), jest budzony przez środowisko systemu wtedy, gdy procedura, na którą czekał, staje się dostępna.
- $STDelayEnd(X)$ – Tranzycja systemowa reprezentuje działania systemu mające na celu wybudzenie agenta X, zaraz po tym jak upłynął okres zawieszenia agenta, będący wynikiem wykonania instrukcji *delay*.
- $STLoopEnd(X)$ – Tranzycja systemowa reprezentuje działania systemu mające na celu wybudzenie agenta X, związane z zakończeniem bieżącego okresu instrukcji *loop every*.
- $STInEnd(X)$, $STOutEnd(X)$ – Te tranzycje systemowe reprezentują działania systemu mające na celu wybudzenie agenta X, zaraz po tym jak upłynął okres oczekiwania na finalizację komunikacji, która nie nastąpiła.
- $STTime$ – Tranzycja systemowa reprezentuje upływ czasu. Używana jest ona do opisywania zmian stanów wynikających wyłącznie z upływu czasu. Sytuacja taka ma miejsce, gdy np. wszystkie agenty są w trybie pracy *waiting*, lecz po upływie pewnego czasu co najmniej jeden z nich może zmienić tryb na *ready*.
- $STSysTick$ – Tranzycja systemowa reprezentująca działania systemu podczas przerwania systemowego.

W celu zaprezentowania wymagań dotyczących aktywności oraz modyfikacji stanu, zostaną użyte te same notacje, które zostały użyte dla tranzycji wynikających z wykonania instrukcji (podrozdział 4.1).

► $STInAP(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X \in \mathcal{A}_A$, $Y \in \mathcal{A}_P$, $(Y.q, X.p) \in \mathcal{C}$, $am_S(X) = \mathsf{W}$, $c = in$, $in(X.p) \in ci_S(X)$, $am_S(Y) = \mathsf{W}$ i $out(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$am_{S'}(X) = \mathsf{R}$, $ci_{S'}(X) = ci_S(X) \setminus \{in(X.p), timer(pc_S(X), \_)\} \cup \{proc(Y.q)\}$, $am_{S'}(Y) = \mathsf{T}$, $pc_{S'}(Y)$ przypisuje numer pierwszej instrukcji wewnątrz procedury $Y.q$ i $ci_{S'}(Y) = [\,]$.

► $STInPP(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X, Y \in \mathcal{A}_P$, $(Y.q, X.p) \in \mathcal{C}$, $X.p \notin \mathcal{P}_{proc}$, $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{W}$, $c = in$, $in(X.p) \in ci_S(X)$, $am_S(Y) = \mathsf{W}$ i $out(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$am_{S'}(context(X)) = \mathsf{R}$, $ci_{S'}(X) = ci_S(X) \setminus \{in(X.p), timer(pc_S(X), \_)\} \cup \{proc(Y.q)\}$, $am_{S'}(Y) = \mathsf{T}$, $pc_{S'}(Y)$ przypisuje numer pierwszej instrukcji wewnątrz procedury $Y.q$ i $ci_{S'}(Y) = [\,]$.

► $STOutAP(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X \in \mathcal{A}_A$, $Y \in \mathcal{A}_P$, $(X.p, Y.q) \in \mathcal{C}$, $am_S(X) = \mathsf{W}$, $c = out$, $out(X.p) \in ci_S(X)$, $am_S(Y) = \mathsf{W}$ i $in(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$am_{S'}(X) = \mathsf{R}$, $ci_{S'}(X) = ci_S(X) \setminus \{out(X.p), timer(pc_S(X), \_)\} \cup \{proc(Y.q)\}$, $am_{S'}(Y) = \mathsf{T}$, $pc_{S'}(Y)$ przypisuje numer pierwszej instrukcji wewnątrz procedury $Y.q$ i $ci_{S'}(Y) = [\,]$.

► $STOutPP(X.p, Y.q)$

Wymagania dotyczące aktywności:

$X, Y \in \mathcal{A}_P$, $(X.p, Y.q) \in \mathcal{C}$, $X.p \notin \mathcal{P}_{proc}$, $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{W}$, $c = out$, $out(X.p) \in ci_S(X)$, $am_S(Y) = \mathsf{W}$ i $in(Y.q) \in ci_S(Y)$.

Modyfikacje stanu:

$am_{S'}(context(X)) = \mathsf{R}$, $ci_{S'}(X) = ci_S(X) \setminus \{out(X.p), timer(pc_S(X), \_)\} \cup \{proc(Y.q)\}$, $am_{S'}(Y) = \mathsf{T}$, $pc_{S'}(Y)$ przypisuje numer pierwszej instrukcji wewnątrz procedury $Y.q$ i $ci_{S'}(Y) = [\,]$.

► $STDelayEnd(X)$

Wymagania dotyczące aktywności:

Jeśli $X \in \mathcal{A}_A$, to $am_S(X) = \mathsf{W}$, $c = delay$, $timeout(pc_S(X)) \in ci_S(X)$.

Jeśli $X \in \mathcal{A}_P$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{W}$, $c = delay$, $timeout(pc_S(X)) \in ci_S(X)$.

Modyfikacje stanu:

- Jeśli $X \in \mathcal{A}_A$, to $am_{S'}(X) = \mathsf{R}$, $pc_{S'}(X) = nextpc_S(X)$,

$ci_{S'}(X) = ci_S(X) \setminus \{timeout(pc_S(X))\}$.

Ponadto, jeśli $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$, $ci_{S'}(X) = [\,]$.

- Jeśli $X \in \mathcal{A}_P$, to $am_{S'}(context(X)) = \mathsf{R}$, $pc_{S'}(X) = nextpc_S(X)$ i $ci_{S'}(X) = ci_S(X) \setminus \{timeout(pc_S(X))\}$.

► $STLoopEnd(X)$

Wymagania dotyczące aktywności:

Jeśli $X \in \mathcal{A}_A$, to $am_S(X) = \mathsf{W}$, $c = loop\_every$, $timeout(pc_S(X)) \in ci_S(X)$.

Jeśli $X \in \mathcal{A}_P$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{W}$, $c = loop\_every$, $timeout(pc_S(X)) \in ci_S(X)$.

Modyfikacje stanu:

Jeśli $X \in \mathcal{A}_A$, to $am_{S'}(X) = \mathsf{R}$, $ci_{S'}(X) = ci_S(X) \setminus \{timeout(pc_S(X))\}$.

Jeśli $X \in \mathcal{A}_P$, to $am_{S'}(context(X)) = \mathsf{R}$, $ci_{S'}(X) = ci_S(X) \setminus \{timeout(pc_S(X))\}$.

► $STInEnd(X)$

Wymagania dotyczące aktywności:

Jeśli $X \in \mathcal{A}_A$, to $am_S(X) = \mathsf{W}$, $c = in$, $timeout(pc_S(X)) \in ci_S(X)$.

Jeśli $X \in \mathcal{A}_P$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{W}$, $c = in$, $timeout(pc_S(X)) \in ci_S(X)$.

Modyfikacje stanu:

- Jeśli $X \in \mathcal{A}_A$, to $am_{S'}(X) = \mathsf{R}$, $pc_{S'}(X) = nextpc_S(X)$, $ci_{S'}(X) = ci_S(X) \setminus \{in(X.p), timeout(pc_S(X))\}$.

  Ponadto, jeśli $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.
- Jeśli $X \in \mathcal{A}_P$, to $am_{S'}(context(X)) = \mathsf{R}$, $pc_{S'}(X) = nextpc_S(X)$, $ci_{S'}(X) = ci_S(X) \setminus \{in(X.p), timeout(pc_S(X))\}$.

► $STOutEnd(X)$

Wymagania dotyczące aktywności:

Jeśli $X \in \mathcal{A}_A$, to $am_S(X) = \mathsf{W}$, $c = out$, $timeout(pc_S(X)) \in ci_S(X)$.

Jeśli $X \in \mathcal{A}_P$, to $am_S(X) = \mathsf{T}$, $am_S(context(X)) = \mathsf{W}$, $c = out$, $timeout(pc_S(X)) \in ci_S(X)$.

Modyfikacje stanu:

- Jeśli $X \in \mathcal{A}_A$, to $am_{S'}(X) = \mathsf{R}$, $pc_{S'}(X) = nextpc_S(X)$, $ci_{S'}(X) = ci_S(X) \setminus \{out(X.p), timeout(pc_S(X))\}$.

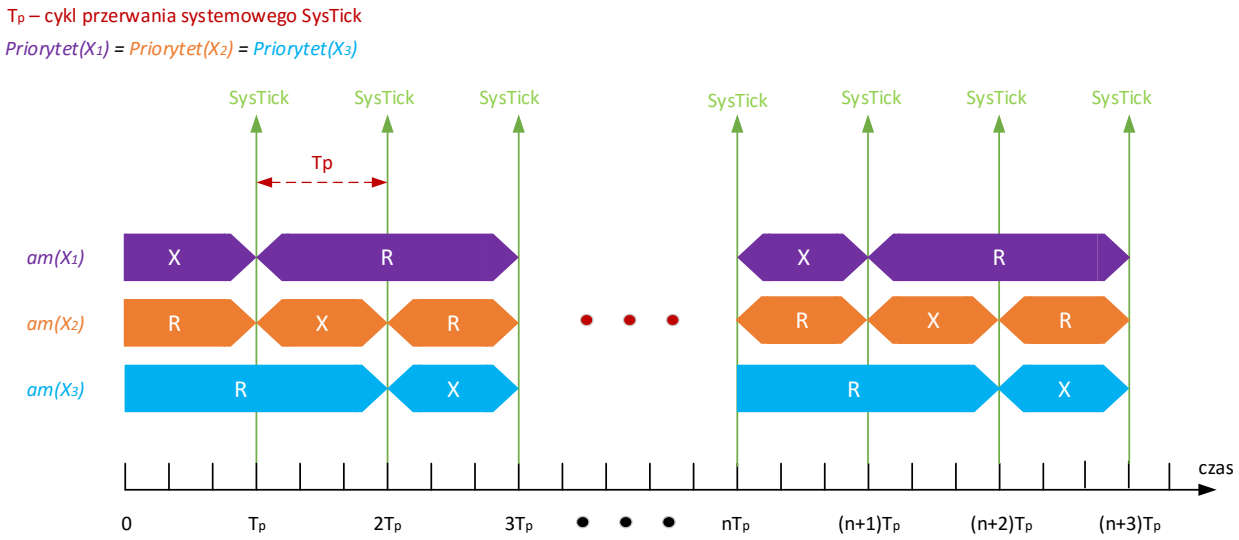
  Ponadto, jeśli $nextpc_S(X) = 0$, to $am_{S'}(X) = \mathsf{F}$ i $ci_{S'}(X) = [\,]$.
- Jeśli $X \in \mathcal{A}_P$, to $am_{S'}(context(X)) = \mathsf{R}$, $pc_{S'}(X) = nextpc_S(X)$, $ci_{S'}(X) = ci_S(X) \setminus \{out(X.p), timeout(pc_S(X))\}$.

Tranzycje systemowe *STTime* i *STSysTick* przedstawiono w rozdziale 5.

4.3. Wywołanie algorytmu szeregującego

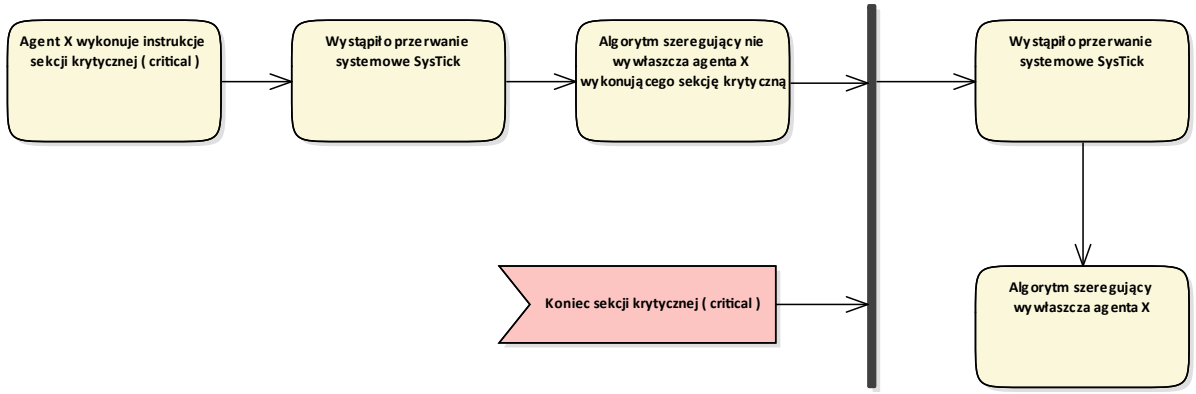
Algorytm szeregowania warstwy systemowej α^1_{FPPS} uruchamiany jest w celu wypromowania agenta aktywnego z trybu gotowości *ready* do trybu aktywnego *running*. W rezultacie tej promocji agent przejmuje kontrolę na zasobami procesora i wykonuje swoje instrukcje. Reszta agentów aktywnych, które również zabiegają o przejęcie zasobów procesora, pozostaje w trybie *ready*, jako skolejkowane agenty dwuwymiarowej kolejki FIFO.

$SysTick$ jest przerwaniem systemowym, które występuje periodycznie z okresem T_p, jak pokazano w przykładzie na rysunku 4.9. Trzy agenty: X_1, X_2 i X_3 są zdefiniowane z tą samą wartością priorytetu. Przerwanie $SysTick$ uruchamia algorytm szeregujący i w konsekwencji promuje agenta do trybu *running* zgodnie z definicją dla warstwy systemowej α^1_{FPPS}.



Rysunek 4.9: Przykład działania przerwania systemowego $SysTick$

Algorytm szeregujący warstwy systemowej α^1_{FPPS} jest uruchamiany jedynie w przypadku wystąpienia przerwania systemowego $SysTick$, co oznacza, że jedynie podczas tego zdarzenia może nastąpić promocja agenta do trybu *running*. Wykonanie pozostałych tranzycji systemowych sprawia, że pozostałe agenty mogą zostać jedynie zakolejkowane w dwuwymiarowej kolejce FIFO i będą oczekiwać na promocję do trybu trybu *running*. Agent aktywny może zostać wypromowany do trybu *running* jedynie poprzez algorytm szeregujący uruchamiany podczas przerwania systemowego. Dla przykładu, jeśli agent który do tej pory kontrolował zasoby procesora zwolni je, to żaden inny agent aktywny oczekujący na promocję do trybu *running* nie zmieni trybu, aż do chwili kiedy wystąpi przerwanie systemowe $SysTick$ i uruchomiony algorytm szeregujący o tym nie zadecyduje. Upływ czasu, podobnie jak pozostałe tranzycje systemowe, kolejkuje jedynie agentów do dwuwymiarowej kolejki FIFO.



Rysunek 4.10: Przerwanie systemowe $SysTick$ w przypadku sekcji krytycznej *critical*

Agent realizujący sekcję krytyczną (instrukcja *critical*) nie może zostać wywłaszczony, aż do zakończenia takiej sekcji. Sytuacja ta została przedstawiona na rysunku 4.10.

Jak pokazano w rozdziale 3, stan modelu definiujemy jako $S = (S(X_1), \ldots, S(X_n), (Q, t))$, gdzie Q oznacza dwuwymiarową kolejkę, zawierająca agenty w trybie *ready* pogrupowane według ich priorytetów. Przyjmijmy, że zdefiniowane są następujące funkcje operujące na kolejce Q:

- $qHighestPosition(Q)$ – Zwraca nazwę agenta, który zajmuje najwyższą pozycję w niepustej kolejce Q.
- $qRemove(Q, X)$ – Usuwa agenta X z kolejki Q.
- $qInsert(Q, X)$ – Wstawia agenta X do kolejki Q (najniższa pozycja na właściwym poziomie) i modyfikuje pozycję agentów, które znajdują się na tym samym poziomie co agent X.

Wystąpienie przerwania systemowego jest reprezentowane przez tranzycję systemową $STSysTick$. Tranzycja ta z definicji ma najwyższy priorytet w modelu (nawet od tranzycji agentów o priorytecie 0). Niech n oznacza przyjęty okres (częstotliwość wywoływania przerwania systemowego).

► $STSysTick$

Wymagania dotyczące aktywności:

$t = 0$.

Modyfikacje stanu:

Jeżeli agent $X \in \mathcal{A}_A$ znajduje się w trybie *running* i w sekcji krytycznej, to $t = n$.

Jeżeli agent $X \in \mathcal{A}_A$ znajduje się w trybie *running* i kolejka Q jest niepusta oraz priorytet agenta X jest mniejszy bądź równy priorytetowi $qHighestPosition(Q)$, to: $am_{S'}(X) = \mathsf{R}$,

$am_{S'}(qHighestPosition(Q)) = \mathsf{X}$, $t = n$ i $Q = qInsert(qRemove(Q, qHighestPosition(Q)), X)$.

Jeżeli priorytet agenta X jest większy od priorytetu $qHighestPosition(Q)$, to wywłaszczenie nie nastąpi.

Jeżeli żaden agent nie znajduje się w trybie *running* i kolejka Q jest niepusta, to:

$am_{S'}(qHighestPosition(Q)) = \mathsf{X}$, $t = n$ i $Q = qRemove(Q, qHighestPosition(Q))$.

Jeżeli kolejka Q jest pusta, to $t = n$.

4.4. Upływ czasu

Warstwa systemowa α^1_{FPPS} w swojej definicji jest zorientowana na architekturę jednoprocesorową, co w praktyce oznacza, że w danym momencie czasowym tylko jeden agent może kontrolować zasoby procesora. Przy tak przyjętym założeniu, może zaistnieć sytuacja, w której wszystkie agenty danego modelu będą znajdowały się w trybie oczekującym *waiting* lub pozostaną w trybie gotowości *ready*. Dopiero upływ czasu spowoduje przejście jednego z nich z trybu *waiting* do trybu *ready* a w konsekwencji, na skutek wystąpienia przerwania systemowego $SysTick$, do trybu *running*. Upływ czasu jest kolejną tranzycją systemową, która zmienia stan agenta i co za tym idzie stan całego modelu. Upływ czasu będzie miał znaczenie przy instrukcjach natury czasowej, takich jak:

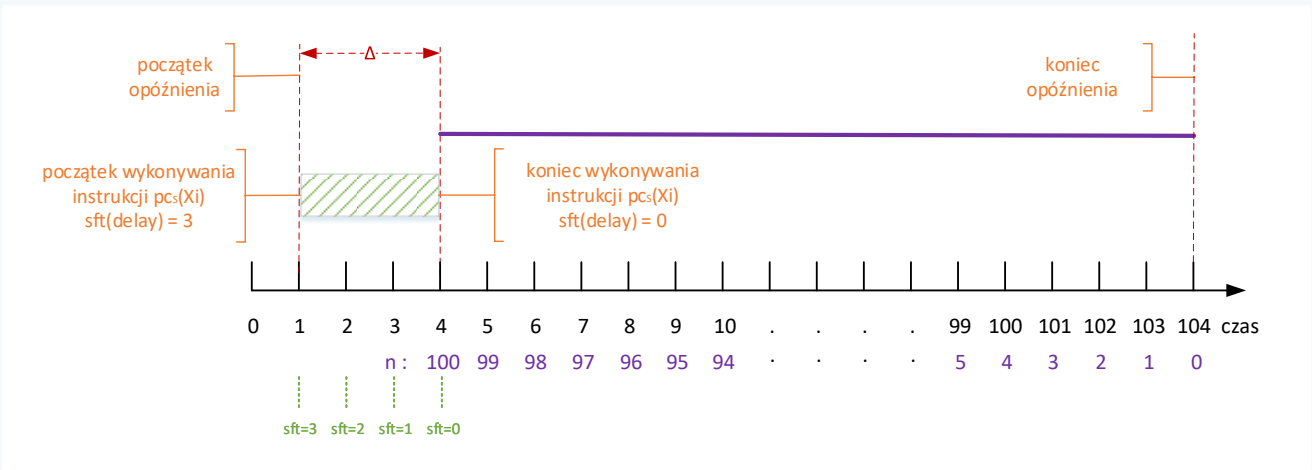
- *delay*;
- *loop every*;
- nieblokująca komunikacja *in*;
- nieblokująca komunikacja *on*.

W praktyce możliwe są też konstrukcje kodu agenta, w których będą zagnieżdżone instrukcje czasowe w instrukcjach czasowych, np. instrukcja *delay* lub instrukcje dotyczące nieblokujących komunikacji w pętli *loop every*. Zważywszy, że w warstwie systemowej α^1_{FPPS} agenty mogą być wywłaszczane z uwagi na obecność w systemie tylko jednego procesora, bieżąca instrukcja może nie zostać zakończona. W takiej sytuacji na liście kontekstowej agenta umieszczany jest wpis sft (ang. *step finish time*), którego argument zawiera informację, ile jednostek czasu koniecznych jest jeszcze do zakończenia bieżącej instrukcji.

W przypadku instrukcji czasowych zachodzi problem, jak wiązać czas potrzebny na wykonanie samej instrukcji z czasem, jaki ta instrukcja wnosi ze względu na swoją funkcjonalność do systemu. Dla przykładu załóżmy, że czas wykonywania instrukcji czasowej *delay* zdefiniowany jest na 3 jednostki czasu i ponadto sama instrukcja zdefiniowana jest z opóźnieniem 100 jednostek czasu: `delay 100` (rysunek 4.11).

Przyjęto, że czas wykonywania instrukcji, w tym wypadku 3 jednostki czasu dla instrukcji `delay`, nie jest wliczany do opóźnienia, czyli w tym przykładzie 100 jednostek czasu opóźnienia będzie odliczanych po zakończeniu wykonywania instrukcji *delay*. Czas ten odliczany jest z użyciem lokalnego dla danego agenta zegara. Zegar taki reprezentowany jest przez wpis *timer* na liście kontekstowej. Wpis ten przyjmuje dwa argumenty. Pierwszym jest numer instrukcji, a drugim czas pozostały do końca odliczania. W podobny sposób uwzględniane są aspekty czasowe wnoszone przez instrukcje *in* oraz *out* dla nieblokującej komunikacji pomiędzy agentami.

W sposób wyjątkowy traktowana jest instrukcja *loop every*. W jej przypadku czas wykonania instrukcji (wejścia do pętli) jest uwzględniany przez zegar. Przykładowo, jeżeli czas wykonania instrukcji *loop every*



Rysunek 4.11: Czas trwania instrukcji $delay$ zagnieżdżony w czasie opóźnienie systemu

wynosi 2 jednostki czasu, a zdefiniowany okres 10, to po wykonaniu instrukcji wpis *timer* odnoszący się do tej instrukcji będzie miał drugi argument równy 8, a nie 10.

Na liście kontekstowej agenta może być w danym momencie więcej niż jeden wpis *timer*, np. po wykonaniu instrukcji *delay*, która jest umieszczona wewnątrz pętli *loop every*. W momencie gdy odliczanie zostaje zakończone (drugi argument równy 0), wpis *timer* zastępowany jest przez *timeout* (jedynym argumentem jest tutaj numer instrukcji, do której ten wpis się odnosi).

Przedstawiając w podrozdziałach 4.1 i 4.2 tranzycje opisujące dynamikę modelu, ominęliśmy zmiany zmiany związane z aktualizacją wpisów *sft* i *timer*. Pierwszy z tych wpisów jest aktualizowany tylko dla agenta, który aktualnie realizuje swoją instrukcję. Wpisy *timer* (łącznie z ewentualną zmianą na *timeout*) wszystkich agentów są aktualizowane za każdym razem, gdy wykonywana jest jakakolwiek tranzycja, której wykonanie trwa co najmniej 1 jednostkę czasu. Ponadto rozważana jest specjalna tranzycja systemowa $STTime$, która reprezentuje wyłącznie upływ czasu.

- $STTime$

Wymagania dotyczące aktywności:

$t > 0$ i żaden agent nie jest w trybie *running*

Modyfikacje stanu:

Aktualizacja wpisów *timer* wszystkich agentów.

Aktualizacja polega na zmniejszeniu liczby jednostek czasu, które pozostały do odliczenia. Wartość ta jest zmniejszana o tyle jednostek czasu, o ile tranzycja $STTime$ „przesuwa” globalny zegar. Maksymalna wartość przesunięcia to t, ale może się zdarzyć, że przesunięcie jest mniejsze.

4.5. Podsumowanie

W rozdziale zaprezentowano listę dostępnych tranzycji w języku modelowania Alvis ze szczególną analizą tych tranzycji, które uwzględniają aspekty czasowe i systemowe, jak chociażby przerwanie systemowe $SysTick$ i tranzycje związane z upływem czasu. Opisując poszczególne instrukcje, przeanalizowano wymagania dotyczące ich aktywności, a także ich wpływ na stany poszczególnych agentów. Podczas omawiania poszczególnych tranzycji pokazano również dodatkowe zależności pomiędzy agentami pasywnymi i aktywnymi, zwracając uwagę na to, jak agenty pasywne uruchamiane są w kontekście akcji wykonywanych przez agenty aktywne.

Do wkładu własnego autora niniejszej rozprawy należy zaliczyć:

- dopasowanie warunków aktywności i wykonania tranzycji do modeli z warstwą systemową α^1_{FPPS};
- wprowadzenie do zestawu tranzycji nowej tranzycji systemowej $SysTick$ i opisanie zasad jej funkcjonalności;
- dopasowanie funkcjonowania tranzycji systemowej STTime do potrzeb warstwy systemowej α^1_{FPPS}.

5. Algorytm generowania etykietowanego systemu przejść

Rozdział ten przedstawia formalną definicję oraz proces generowania grafu LTS (ang. *Labelled Transition Systems* [4]) dla modeli czasowych systemów współbieżnych uruchomionych na jednoprocesorowych platformach sprzętowych. Jak wspomniano w poprzednich rozdziałach etykietowany system przejść danego modelu jest punktem wejściowym do jego analizy i weryfikacji. W przypadku języka Alvis jest ona prowadzona przede wszystkim z użyciem metod weryfikacji modelowej i uznanych narzędzi takich jak nuXmv [16], [12] and CADP [24], [25], [13].

Etykietowany system przejść stanowi grafową reprezentację przestrzeni osiągalnych stanów. Węzły takiego skierowanego grafu reprezentują stany, a krawędzie akcje powodujące zmiany tych stanów. W przypadku czasowych modeli języka Alvis (zarówno dla wersji systemowej α^0 jak i $alpha^1_{FPPS}$) etykiety łuków wygenerowanego grafu LTS oraz listy kontekstowe poszczególnych agentów w węzłach grafu LTS zawierają informacje dotyczące aspektów czasowych. Podstawą pomiaru upływającego czasu jest *zegar systemowy*, będący odpowiednikiem *zegara globalnego*, spotykanego w innych formalizmach takich, jak automaty czasowe [3], [8], [58] lub sieci Petriego. Algorytm generowania grafu LTS bazuje na funkcjach *fire* i *enable*, których idea została zaczerpnięta z podejścia stosowanego w sieciach Petriego [37], [46]. Ich definicje i role jakie spełniają w Alvisie zostaną opisane w kolejnych częściach tego rozdziału. W dalszej części pracy zamiennie będziemy używac terminów etykietowany system przejśc, graf LTS lub krótko LTS.

5.1. Grafy LTS dla modeli z warstwą systemową α^1_{FPPS}

Graf LTS dla modeli z dowolną warstwą systemową definiujemy następująco:

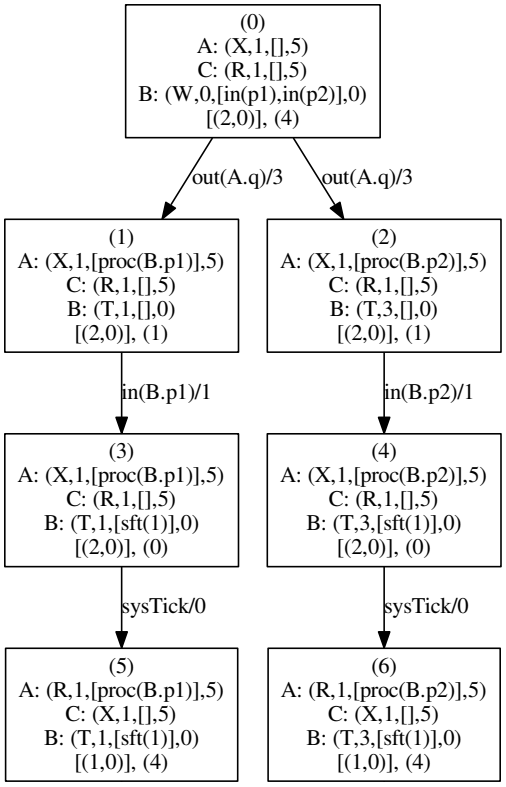
Definicja 5.1. *Etykietowanym systemem przejść* (grafem LTS, LTSem) nazywamy graf skierowany $\mathrm{LTS} = (\mathcal{S}, E, L, S_0)$, gdzie:

- $\mathcal{S}$ jest zbiorem *węzłów*,
- L jest zbiorem *etykiet*,
- $E \subseteq S \times L \times S$ jest zbiorem *łuków*,
- S_0 jest *stanem początkowym*.

W przypadku modeli z warstwą systemową α^1_{FPPS} stan modelu zawiera poza stanami poszczególnych agentów, zawartość kolejki priorytetowej i czas pozostały do wywołania algorytmu szeregującego (patrz wzór (3.6), str. 40). Każdy element ze zbioru S reprezentuje jeden stan modelu, dodatkowo opatrzony unikalnym numerem porządkowym. *Stan początkowy* reprezentowany jest przez węzeł o numerze 0.

Jeżeli stany S i S' są połączone krawędzią w grafie LTS, to oznacza to, że stan S' jest *bezpośrednio osiągalny* ze stanu S. Krawędź taka etykietowana jest nazwą wykonywanej tranzycji i czasem jaki upływa w związku z przejściem ze stanu S do S'. Tak więc elementy zbioru L są parami – (tranzycja, liczba całkowita nieujemna). Warto tutaj przypomnieć, że w modelach czasowych w języku Alvis przejście ze stanu S do bezpośrednio osiągalnego stanu S' nie musi oznaczać „pełnego" zrealizowania tranzycji (akcji modelu). Akcja taka może być zrealizowana tylko częściowo, a czas dalej wymagany do jej sfinalizowania jest reprezentowany przez odpowiedni wpis na liście kontekstowej. Innymi słowy, pełny opis realizacji tranzycji, np. ewaluacji wyrażenia (*exec*) może być reprezentowany przez kilka krawędzi grafu LTS.

Na rys. 5.1 przestawiono fragment grafu LTS dla modelu zawierającego 3 agenty,w tym 2 aktywne A, C i jeden pasywny B. Sama wizualizacja grafu uzyskana została automatycznie na podstawie pliku w formacie *dot* [23] – jeden z formatów wspieranych przez środowisko modelowania Alvis Toolkit.



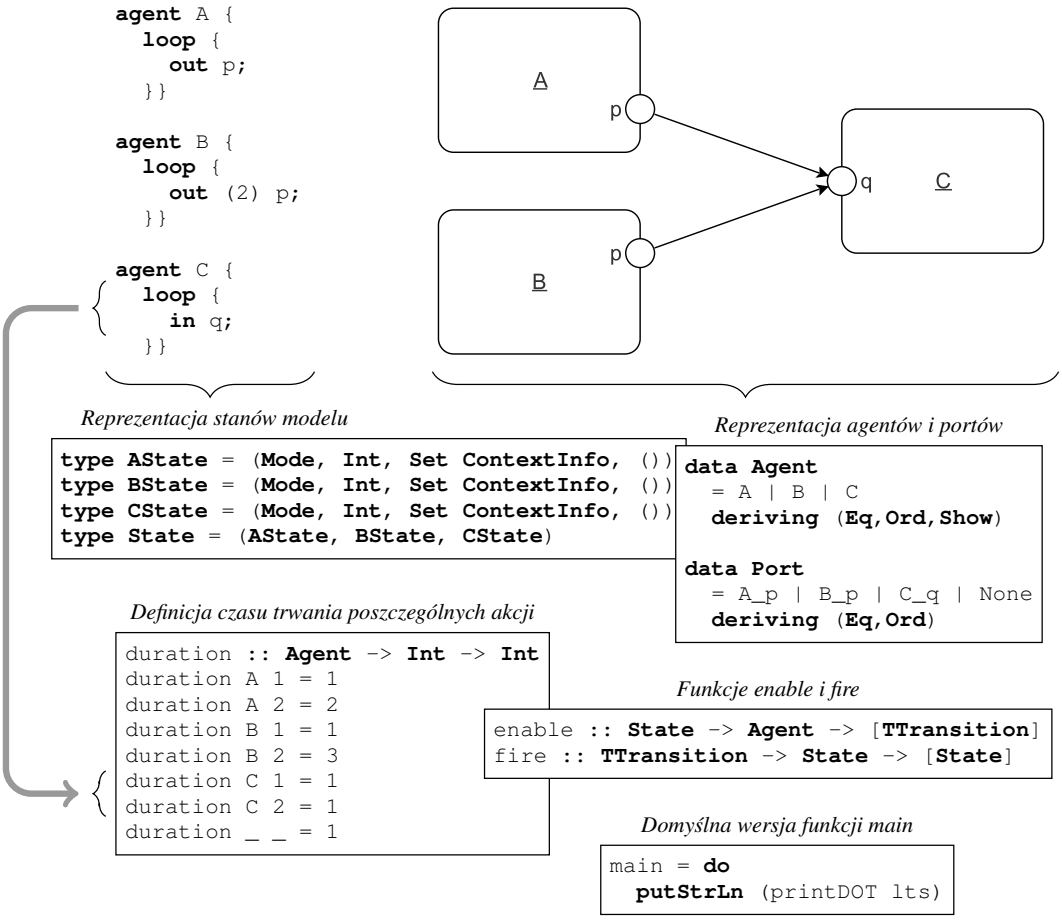
Rysunek 5.1: Fragment grafu LTS dla modelu z warstwą systemową α^1_{FPPS}

Dla przykładu krawędź prowadząca od węzła początkowego do węzła 1 wskazuje na pełne wykonanie instrukcji *out* przez agenta A, co zajmuje 3 jednostki czasu (argumentem instrukcji był port $A.q$). W wyniku

wykonania tej instrukcji wywołana została procedura $B.p1$, co wynika ze stanów agentów A i B (węzeł 1). Natomiast w przypadku krawędzi prowadzącej od węzła 1 do węzła 3 mamy do czynienia tylko z częściowym wykonaniem instrukcji *in* po stronie agenta B – w stanie 3 widoczny jest wpis $sft(1)$ na liście kontekstowej agenta B, który wskazuje, że do dokończenia bieżącej instrukcji pozostała jedna jednostka czasu. Pełne wykonanie tej instrukcji nie było możliwe ze względu na konieczność wywołania algorytmu szeregującego po 1 jednostce czasu licząc od stanu w węźle 1.

5.2. Reprezentacja modeli w języku Haskell (IHR)

Modele w języku Alvis są budowane z wykorzystaniem środowiska *Alvis Editor*. Środowisko zawiera w sobie zarówno edytor dla diagramów komunikacji, jak również edytor warstwy kodu. Kompletny model zapisany w formacie XML przetwarzany jest następnie przez *kompilator* języka Alvis. *Alvis Compiler* jest aplikacją napisaną w języku Java, a jego podstawowym zadaniem jest translacja modelu z języka Alvis do języka Haskell [38]. Wynikiem tej kompilacji jest pośrednia reprezentacja modelu określana jako *Intermediate Haskell Representation* (IHR) [55].

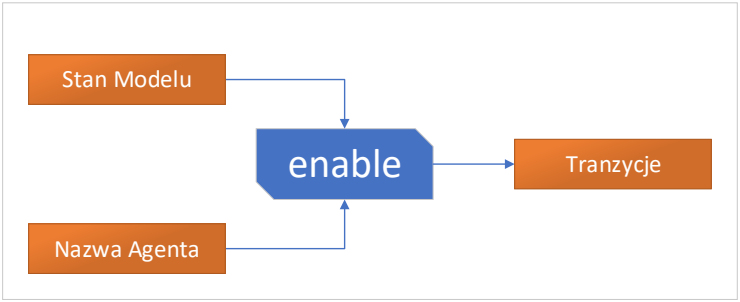


Rysunek 5.2: Fragmenty reprezentacji IHR dla modelu w języku Alvis ([55])

Na rys. 5.2 pokazano kluczowe elementy reprezentacji IHR dla modelu w języku Alvis. Uzyskany plik Haskella może służyć do wygenerowania grafu LTS, symulacji pracy modelu lub realizacji zadań określonych przez użytkownika, który ma dostęp do kodu źródłowego wygenerowanej reprezentacji i może ją dowolnie zmieniać. Zawartość generowanego kodu zależy również od użytych opcji kompilatora, które pozwalają m.in. określić warstwę systemową, która ma zostać użyta. Niezależnie od użytych opcji i wprowadzonych zmian, rdzeń reprezentacji IHR pozostaje wspólny dla wszystkich modeli. Agenty i porty modelu reprezentowane są przez generowane przez kompilator typy *Agent* i *Port*. Dla każdego agenta generowany jest typ *AgentNameState*, który reprezentuje stan danego agenta. Typy te generowane są indywidualnie ze względu na czwarty element krotki, tj. zestaw parametrów (lokalnych zmiennych) agenta, który może być różny dla różnych agentów. Ostatecznie definiowany jest typ *State* reprezentujący stan modelu, ale wyłącznie jako krotka krotek reprezentujących stany poszczególnych agentów.

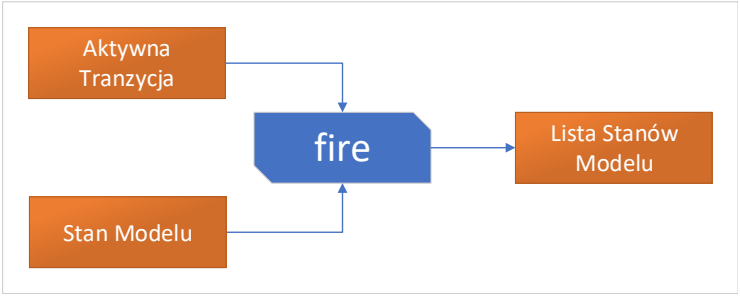
Dla modeli czasowych generowana jest funkcja *duration*, która określa czas trwania poszczególnych akcji. Domyślnie wszystkie wartości są ustawione na 1 i odpowiedniej zmiany należy dokonać w pliku źródłowym Haskella.

Z punktu widzenia opisu dynamiki modelu najważniejsze są funkcje *enable* i *fire* [53], [55]. Funkcja *enable* (rys. 5.3) pobiera jako swoje argumenty stan modelu i nazwę agenta i zwraca listę wszystkich tranzycji (akcji) agenta, które są możliwe do wykonania w tym stanie.



Rysunek 5.3: Funkcja *enable*

Funkcja *fire* (rys. 5.4) pobiera jako argumenty aktywną tranzycję i stan, i zwraca listę nowych stanów będących konsekwencją wykonania rozważanej tranzycji (zazwyczaj jest to lista jednoelementowa).



Rysunek 5.4: Funkcja *fire*

Ostatnim pokazanym na rysunku elementem jest funkcja *main*, która w wersji domyślnej generuje graf LTS i zapisuje go w formacie *dot*. Inne bezpośrednio wspierane formaty to *Aldebaran* (na potrzeby weryfikacji z użyciem CADP [25]) i CSV na potrzeby stosowania podejść opartych na analizie danych (np. z użyciem języków Python lub R).

Listing 5.1: Wybrane typy i funkcje IHR

```
1  data Mode = ...
2  data ContextInfo = ...
3  data TTransition = ...
4  data Agent = ...
5  data Port = ...
6
7  transAgent :: TTransition -> Agent
8  transNumber :: TTransition -> Int
9  agentNumber :: Agent -> Int
10 procAgent :: ContextInfo -> Agent
11 fstPort :: TTransition -> Port
12 sndPort :: TTransition -> Port
13 sndAgent :: TTransition -> Agent
14 isInOutTransition :: TTransition -> Bool
15 isCommTransition :: TTransition -> Bool
16 isSystemTransition  :: TTransition -> Bool
17 enableInState :: State -> [TTransition]
18 agentName :: Port -> Agent
19 agentPriority :: Agent -> Int
20 duration :: Agent -> Int -> Int
21 takeam :: Int -> State -> Mode
22 takepc :: Int -> State -> Int
23 takeci :: Int -> State -> Set ContextInfo
24 takepvString :: Int -> State -> String
25 updateContextInfo :: Int -> (Set ContextInfo -> Set ContextInfo) -> State -> State
26 updateAllContextInfo :: (Set ContextInfo -> Set ContextInfo) -> State -> State
```

W rzeczywistości IHR zawiera również definicje wielu innych typów danych oraz kilkudziesięciu funkcji, które pozwalają na zarządzanie stanami i tranzycjami modelu oraz na dostęp do wybranych informacji związanych ze stanami i tranzycjami. Przykładowe typy danych i funkcje przedstawiono na listingu 5.1 (wyłącznie nazwy typów i nagłówki funkcji). Typ danych *Mode* definiuje wszystkie dostępne tryby pracy agentów, a typ *ContextInfo* wszystkie dopuszczalne wpisy na listach kontekstowych. Typ *TTransition* za-

wiera definicje wszystkich dostępnych tranzycji. Dla przykładu funkcja *transAgent* zwraca agenta, który realizuje daną tranzycję a funkcja *enableInState* listę wszystkich tranzycji aktywnych w danym stanie (dla wszystkich agentów).

Warto zwrócić uwagę, że funkcje *enable* i *fire* ignorują zależności czasowe i priorytety agentów. Pozwala to na uzyskanie uniwersalnego rdzenia dla reprezentacji IHR, który jest następnie rozszerzany dla konkretnej warstwy systemowej z użyciem dodatkowych modułów implementowanych w Haskellu.

5.3. Rozszerzenie reprezentacji IHR dla warstwy systemowej α^1_{FPPS}

Rozszerzenie reprezentacji IHR dla warstwy systemowej α^1_{FPPS} wymaga rozbudowania kodu w Haskellu o elementy niewystępujące w rdzeniu IHR (np. kolejki priorytetowe) oraz dopasowanie funkcji *enable*, *fire*, algorytmu generowania grafu LTS oraz eksportu wyników do środowiska jednoprocesorowego.

Stan modelu

Oprócz wartości typu *State* do opisu pełnego stanu modelu zastosowano reprezentację kolejki priorytetowej i czasu do najbliższego wywołania algorytmu szeregowania. Kolejka reprezentowana jest przez listę par liczb typu *Int* – `[(Int,Int)]`. Pierwszy element pary reprezentuje numer agenta, a drugi jego priorytet. Dla uproszczenia implementacji agent, który aktualnie ma dostęp do procesora, znajduje się na początku tej kolejki, ale nie jest „drukowany” w momencie zapisywania grafu LTS. Czas do wygenerowania najbliższego zdarzenia *SysTick* reprezentowany jest przez wartość typu *Int*. Budowę kolejki priorytetowej oraz funkcje służące do jej zbudowania przedstawiono na listingu 5.2

Listing 5.2: Kolejka priorytetowa agentów aktywnych

```
1 q0 :: [(Int,Int)]
2 q0 = createPriorityQueue s0 1 modelSize []
3
4 createPriorityQueue :: State -> Int -> Int -> [(Int,Int)] -> [(Int,Int)]
5 insertIntoQueue :: (Int,Int) -> [(Int,Int)] -> [(Int,Int)]
```

Funkcja *enable*

Do ustalania tranzycji aktywnych w poszczególnych stanach zaimplementowana została funkcja *alpha1EnableTransitions*. Funkcja ta obudowuje funkcję *enable*, a dokładniej mówiąc funkcję *enableInState*. Typ funkcji przedstawiono na listingu 5.3:

Listing 5.3: Typ funkcji *alpha1EnableTransitions*

```
alpha1EnableTransitions :: State              -- ^ current state
                        -> [(Int,Int)]        -- ^ current priority queue
                        -> Int                -- ^ time to SysTick
                        -> [TTransition]      -- ^ active transitions
```

Funkcja *enableInState* dostarcza listę wszystkich tranzycji aktywnych w danym stanie (bez *STTime* i *STSysTick*, które obsługujemy sami na poziomie opisywanego rozszerzenia). Funkcja *alpha1EnableTransitions* określa, co tak naprawdę pozostanie aktywne, gdy rozważamy uwarunkowania związane z warstwą α^1_{FPPS}. Przedstawione przypadki rozpatruje się w kolejności ich przedstawienia, przy czym punkt $n+1$ stosujemy tylko, jeżeli nie zastosowano punktu n.

1. Jeżeli pozostało 0 jednostek czasu do wywołania algorytmu szeregującego, to jedyną aktywną tranzycją jest *STSysTick*, reprezentująca pracę dyspozytora (ang. *scheduler*).
2. Jeżeli nie ma aktywnych tranzycji i nie ma wpisów kontekstowych wskazujących na możliwe sensowne przesunięcie w czasie, np. do momentu końca zawieszenia agenta będącego w trybie *waiting* po wykonaniu instrukcji *delay*, to wynikiem funkcji jest *lista pusta*. Wskazuje ona na osiągnięcie *stanu końcowego* (węzeł terminalny w grafie LTS).
3. Jeżeli są aktywne jakieś systemowe tranzycje, to wszystkie inne odrzucamy i dajemy im pierwszeństwo. Tranzycje systemowe odpowiadają przede wszystkim za „budzenie" agentów z trybu *waiting* do trybu *ready*, a także obsługują przesunięcia w czasie (tranzycja *STTime*).
4. Szukamy tranzycji agenta aktywnego, który ma dostęp do zasobów procesora lub ewentualnie agenta pasywnego pracującego w kontekście tego bieżącego agenta i tylko te tranzycje pozostają jako aktywne.

Funkcja *fire*

Analogicznie jak w poprzednim przypadku zaimplementowano funkcję, będącą obudową dla funkcji *fire* z rdzenia IHR. Funkcja *timeFire* realizuje wykonanie bieżącej tranzycji w realiach warstwy sprzętowej jednoprocesorowej. Typ funkcji przedstawiono na listingu 5.4:

Listing 5.4: Typ funkcji *timeFire*

```
timeFire :: Int                           -- ^ time shift
         -> TTransition                   -- ^ transition
         -> State                         -- ^ current state
         -> [(Int,Int)]                   -- ^ current queue
         -> Int                           -- ^ time to SysTick
         -> [(State,[(Int,Int)],Int)]     -- ^ new states
```

Funkcja *timeFire* w pierwszej części obsługuje tranzycję *STSysTick*. Ta część jest zaimplementowana następująco:

1. Jeżeli kolejka ma długość 1 (czyli tak naprawdę 0, bo tym jedynym agentem jest agent mający dostęp do zasobów procesora), to *SysTick* nic nie zmienia. Tak samo zachowuje się algorytm szeregowania jeżeli agent z dostępem do procesora zawiera wpis *critical* na liście kontekstowej.
2. W innym przypadku głowa jest usuwana z kolejki i drugi agent w kolejce staje się bieżącym. Usunięta głowa jest wstawiana z powrotem do kolejki, jeżeli ma w bieżącym stanie aktywną tranzycję, lub jest kontekstem dla agenta pasywnego z aktywną tranzycją.

W dalszej części zaimplementowana jest tranzycja *STTime*. Wykonanie tej tranzycji sprowadza się wyłącznie do aktualizacji (zmniejszenia) wartości zegarów (wpisów *timer*) na listach kontekstowych agentów.

Ostatnia część funkcji obsługuje wykonanie pozostałych tranzycji podzielonych na grupy zależnie od:

- czy wykonanie rozważanej tranzycji może wpłynąć na aktywność tranzycji innych agentów czy nie;
- czy przesunięcie czasowe jest równe 0, czy większe od 0;
- czy konieczna jest modyfikacja kolejki priorytetowej, czy nie;
- czy wykonujemy (ewentualnie kończymy) tranzycję, czy też realizujemy jej część (która nie jest ostatnią);
- czy lista kontekstowa agenta zawiera już wpis *sft*, czy nie.

Warto zaznaczyć, że właściwa funkcja *fire* jest wykorzystywana wewnątrz *timeFire*, ale tylko w przypadkach, gdy mamy do czynienia z dokończeniem lub realizacją całej tranzycji. Ponadto poza zmianami wynikającymi z definicji tej funkcji, konieczne jest wprowadzanie zmian związanych z aspektami czasowymi.

Przesunięcie w czasie

Tak jak zostało już powiedziane, etykietą krawędzi w grafie LTS jest para, której drugiej elementem jest liczba całkowita wskazująca przesunięcie w czasie. Jednym z kluczowych elementów algorytmu jest wyznaczenie tego przesunięcia. Podstawowe założenie jest następujące: *chcemy się przesunąć możliwie najdalej, ale nie tracąc żadnej informacji*. Warunek ten oznacza, że w miarę możliwości chcemy unikać przesuwania się o 1 jednostkę czasu, jeżeli możliwe jest większe przesunięcie. Z drugiej strony, jeżeli w między czasie pojawia się inna zmiana niż modyfikacja zegarów, to musi być ona uwzględniona jako oddzielny węzeł w grafie LTS.

Za wyznaczenie możliwego przesunięcia odpowiada funkcja *timeShift*. Typ funkcji przedstawiono na listingu 5.5:

Funkcja przyjmuje 3 argumenty: bieżący stan, tranzycję którą chcemy wykonać i ile czasu zostało do wygenerowania zdarzenia *SysTick*. Funkcja działa następująco:

Listing 5.5: Typ funkcji *alpha1EnableTransitions*

```
1 timeShift :: State           -- ^ current state
2           -> TTransition     -- ^ current transition
3           -> Int             -- ^ time to SysTick
4           -> Int             -- ^ time shift for the considered transition
```

1. Jeżeli rozważaną tranzycją jest *STSysTick* to wynikiem jest 0 (zerowy czas wykonania tej tranzycji).
2. Jeżeli bieżącą tranzycją jest *STTime*, to wynikiem jest argument tej tranzycji. Wynika to z innej funkcji, która decyduje o tym kiedy *STTime* jest aktywna. Argument tej tranzycji jest wtedy już wyznaczany jako to maksymalne możliwe przesunięcie.
3. Dla każdej innej tranzycji rozważane są 4 wartości i na podstawie tego, które z nich mają określoną wartość (w dwóch przypadkach wartość może być nieokreslona). Te cztery wartości to
   - maksymalne możliwe przesunięcie wynikające z zegarów (wpisów *timer* na listach kontekstowych agentów;
   - wartość wpisu *sft* dla agenta wykonującego tranzycję (ile jednostek czasu zostało do jej dokończenia);
   - pełny czas wykonania tej tranzycji,
   - czas pozostały do zdarzenia *SysTick*.

   Wyznaczane jest *minimum* z określonych ww wartości.

Generowanie grafu LTS

Węzeł grafu LTS reprezentowany jest poprzez typ *Node*:

```
type Node = (Int, State, [(Int,Int)], Int, [(TTransition, Int, Int)])
```

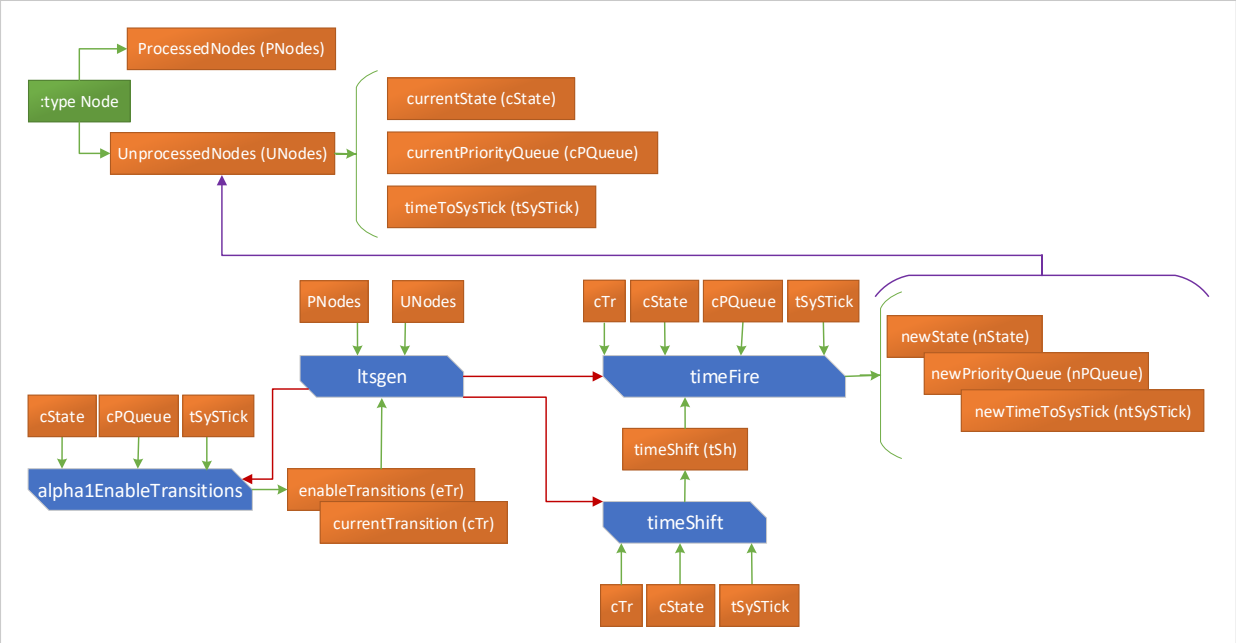
W typie tym uwzględnia się kolejno: numer węzła, stan modelu, kolejkę priorytetową, czas do wygenerowania najbliższego zdarzenia *SysTick* oraz wychodzące z danego węzła łuki. Każdy taki łuk zawiera informację o tranzycji, przesunięciu w czasie i numerze docelowego węzła.

Funkcja *ltsgen*, przedstawiona na listingu 5.6, jest główną funkcją generującą graf LTS. W tym celu korzysta z dwóch list węzłów. Pierwsza lista to węzły „do przetworzenia", druga to wszystkie już wygenerowane węzły. Na początku obie listy zawierają tylko węzeł początkowy, reprezentujący stan początkowy modelu. Dodatkowym parametrem formalnym funkcji *ltsgen* jest lista możliwych do wykonania tranzycji ze stanu pierwszego węzła na liście węzłów „do przetworzenia".

Listing 5.6: Funkcja *ltsgen*

```
1  ltsgen :: [Node]            -- ^ List of unprocessed nodes.
2                              --   We focus on the first node in the list.
3         -> [TTransition]     -- ^ List of transitions enable in the first state
4                              --   of the unprocessed nodes.
5         -> [Node]            -- ^ Auxiliary list of already generated nodes.
6         -> [Node]            -- ^ Final LTS graph.
7
8  -- | LTS graph.
9  lts = ltsgen [(0,s0,q0,t0,[])] (alpha1EnableTransitions s0 q0 t0)
10   [(0,s0,q0,t0,[])]
```

Działanie funkcji *ltsgen* polega na tym, że w początkowym etapie brany jest pierwszy element z listy „nieprzeprocesowanych" węzłów. Następnie za pomocą funkcji *alpha1EnableTransitions* szukane są wszystkie tranzycje aktywne w stanie danego węzła. Kolejna akcja polega na wyznaczeniu nowych możliwych do osiągnięcia stanów, które są rezultatem wykonania tranzycji określonych w poprzednim kroku. Każdy nowo wyznaczony stan jest porównywany z zawartością drugiej listy. Pozwala to na sprawdzenie, czy taki węzeł już wystąpił w przeszłości. Jeśli tak, to wtedy do grafu LTS zostaje dodana tylko krawędź do znalezionego węzła. W przeciwnym wypadku, jeśli takiego stanu jeszcze nie było, to dodajemy i węzeł i krawędź. Każdy taki „nowy" stan jest dołączany na koniec obu list. Za odpowiednią modyfikację obu tych list odpowiada funkcja *update*.



Rysunek 5.5: Zależności pomiędzy funkcjami

Funkcja *ltsgen* korzysta z trzech innych kluczowych dla algorytmu α^1_{FPPS} funkcji: *timeShift*, *timeFire* i wspomnianej już *alpha1EnableTransitions*. Zależności pomiędzy tymi funkcjami a funkcją *ltsgen* pokazano na rysunku 5.5.

Funkcja *alpha1EnableTransitions* przyjmuje jako swoje argumenty formalne: bieżący stan *cState*, bieżącą kolejkę priorytetową *cPQueue* oraz czas *tSysTick* reprezentujący okres czasu do wystąpienia najbliższego przerwania systemowego *SysTick*. Lista aktywnych tranzycji *eTr*, która jest zwracana przez tą funkcję, stanowi parametr wejściowy dla funkcji *ltsgen*. Dodatkowo bieżąca tranzycja *cTr*, rozpatrywana w funkcji *ltsgen*, jest parametrem wejściowym dla funkcji *timeShift* oraz *timeFire*. Funkcja *timeShift* jako swoje parametry formalne przyjmuje dodatkowo bieżący stan *cState* i *tSysTick*, czyli czas do wystąpienia kolejnego przerwania *SysTick*. Zwracaną wartością jest przesunięcie w czasie *tSh*. Parametr ten staje się parametrem wejściowym dla funkcji *timeFire*. Oprócz tego funkcja *timeFire* pobiera jeszcze bieżącą tranzycję *cTr*, bieżący stan *cState*, bieżącą kolejkę priorytetową cPQueue oraz czas do wystąpienia przerwania *SysTick* – *tSysTick*.

5.4. Podsumowanie

Podstawą formalnej weryfikacji modeli konstruowanych w języku Alvis jest grafowa reprezentacja przestrzeni stanów, zwana tutaj grafem LTS. Uzyskanie takiego grafu wymaga kompilacji modelu w języku Alvis do wykonywalnej postaci w języku Haskell. Generowana przez kompilator języka Alvis haskellowa reprezentacja modelu zwana IHR (ang. *Intermediate Haskell Representation*) nie jest przystosowana do warstwy systemowej α^1_{FPPS}. W niniejszym rozdziale przedstawiono kluczowe dla praktycznego korzystania z warstwy systemowej α^1_{FPPS} rozszerzenia IHR.

Do istotnych z punktu widzenia tez pracy osiągnięć, które opisano w tym rozdziale można zaliczyć:

- rozszerzenie reprezentacji stanu modelu tak, aby uwzględniona została kolejka priorytetowa i czas do najbliższego wywołania algorytmu szeregującego;
- opracowanie rozszerzeń funkcji *enable* i *fire*, które dostosowują je do potrzeb warstwy systemowej α^1_{FPPS};
- adaptację algorytmu generowania grafu LTS do potrzeb modeli z warstwą systemową α^1_{FPPS}.

6. Studium przypadków

W przykładach opisujących wykorzystanie języka Alvis do modelowania systemów czasu rzeczywistego, starano się przedstawić te jego cechy, które są związane z warstwą systemową α^1_{FPPS}. Należą do nich wszelkie te aspekty, które uwzględniają zależności czasowe oraz wzajemne relacje pomiędzy agentami. Algorytm generowania grafu LTS bierze pod uwagę czas trwania wykonywanych instrukcji, wartości ich parametrów czasowych, jak okres pętli synchronicznej *loop every*, lub też czas oczekiwania na podjęcie komunikacji w przypadku nieblokujących operacji na portach agentów. Istotnym elementem jest również czas, jaki pozostał do wystąpienia kolejnego przerwania systemowego *SySTick*. W rezultacie uwzględnienia tych wszystkich zależności, graf LTS przedstawia:

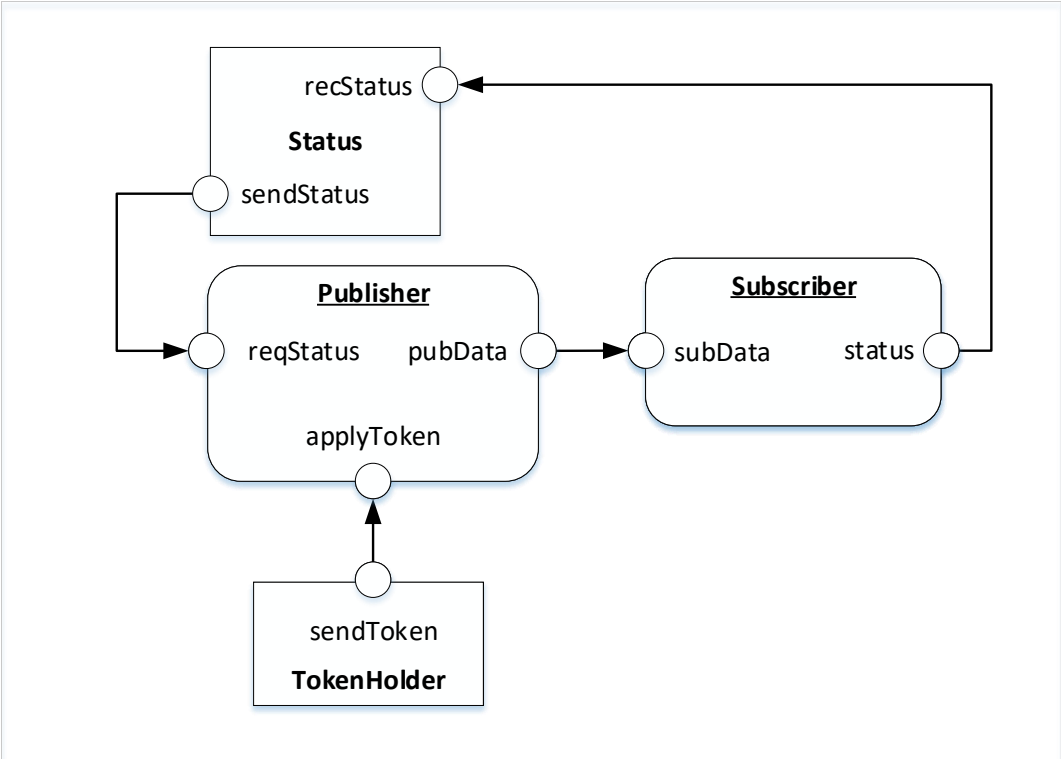
- czasy wykonania poszczególnych instrukcji na łukach łączących stany modelu;
- uaktualnione wpisy czasowe na listach kontekstowych agentów, takie jak, np. *timer* lub *timeout*;
- zmiany stanów agentów oraz ich wywłaszczanie przez algorytm szeregowania warstwy α^1_{FPPS}.

6.1. Publikator i subskrybent

Prezentowane studium przypadku opisuje przykład modelowania wzorca publikatora i subskrybenta (ang. *publisher-subscriber*), powszechnie znanego i szeroko używanego w implementacji oprogramowania. W praktyce wzorzec ten jest używany do redystrybucji danych w kierunku od publikatora do zarejestrowanych na te dane subskrybentów. Liczba publikatorów i subskrybentów może być dowolna, ale skończona. W tym przykładzie ograniczono się do jednego publikatora i jednego subskrybenta. W rozważanym modelu publikatorem jest agent `Publisher`, którego główne zadanie polega na wysyłaniu danych do swojego subskrybenta, którym jest agent `Subscriber`. Przed wysłaniem jakiejkolwiek danej agent `Publisher` musi pobrać `token` od agenta pasywnego `TokenHolder`. *Token* ten uprawnia agenta `Publisher` do komunikacji ze swoim subskrybentem. Diagram komunikacji dla danego przykładu pokazano na rysunku 6.1.

Warstwa kodu

Odbiorca (subskrybent) odbiera dane na swoim porcie wejściowym `subData`. Otrzymana wartość dodawana jest do bufora danego subskrybenta. Następnie bufor jest wysyłany do agenta pasywnego `Status`,



Rysunek 6.1: Model wzorca publikatora z subskrybentami

w celu obliczenia statusu danego subskrybenta. Status subskrybenta brany jest pod uwagę przez agenta `Publisher`. Na tej podstawie agent `Publisher` decyduje o tym, czy powiększyć (ang. *increment*) bufor agenta `Subscriber`, czy też go pomniejszyć (ang. *decrement*). W przypadku otrzymania statusu `Lower`, agent `Publisher` inkrementuje dany bufor, a w przypadku statusu `Bigger` dekrementuje go.

Listing 6.1: Definicja agenta aktywnego *Publisher*

```
agent Publisher (0) {
    increment :: Int = 1;
    decrement :: Int = -1;
    status :: String = "";
    dataMessagePub :: Int = 0;
    token :: Char = ' ';

    loop (every 100) {                                    -- 1
        in applyToken token;                              -- 2
        in reqStatus status;                              -- 3
        select {                                          -- 4
            alt (token == 'T' && status == "Lower") {
                dataMessagePub = increment;               -- 5
            }
```

```
            alt (token == 'T' && status == "Bigger") {
                dataMessagePub = decrement;                    -- 6
            }
        }
        out pubData dataMessagePub;                            -- 7
        null;                                                  -- 8
    }
}
```

Warstwę kodu agenta aktywnego `Publisher` zaprezentowano na listingu 6.1. Priorytet tego agenta jest zadeklarowany jako 0, co oznacza, że jest to najwyższa z możliwych wartości.

Agent `Publisher` używa następujące parametry (lokalne zmienne):

- `increment :: Int = 1` – parametr typu całkowitego o wartości początkowej 1. Wartość tego parametru zostanie użyta do inkrementacji parametru `buffer` dla danego subskrybenta.
- `decrement :: Int = -1` – parametr typu całkowitego o wartości początkowej -1. Wartość tego parametru zostanie użyta do dekrementacji parametru `buffer` dla danego subskrybenta.
- `status :: String = ""` – status subskrybenta zdefiniowany jako parametr typu **`String`**. Parametr ten zostanie użyty przez agenta `Publisher` podczas wywoływania procedury `sendStatus` agenta pasywnego `Status`.
- `dataMessagePub :: Int = 0` – parametr typu całkowitego, reprezentuje wartość wysyłanej danej. Wartość ta zostanie dodana do parametru `buffer` danego subskrybenta.
- `token :: Char = ' '` – parametr typu znakowego o wartości początkowej `' '`. W praktyce będzie równy `'T'`, co będzie świadczyło o pobraniu tokena przez agenta `Publisher` niezbędnego do komunikacji z subskrybentem `Subscriber`.

Agent `Publisher` realizuje swoje obliczenia w nieskończonej pętli, której zawartość powtarzana jest co 100 jednostek czasu (j.cz.). Wewnątrz pętli agent odpytuje o wartość *tokena* i przypisuje ją do parametru `token` (instrukcja 2) oraz pobiera status subskrybenta `Subscriber` (parametr `status`, instrukcja 3). W zależności od wartości parametrów `token` i `status` określana jest wartość parametru `dataMessagePub`, która na koniec wysyłana jest przez port `pubData`. Zastosowana instrukcja **`null`** jest niezbędna jako sygnalizator końca zawartości pętli **`loop every`**.

Fragment funkcji `duration`, określającej czas trwania poszczególnych instrukcji dla agenta aktywnego `Publisher`, przedstawiono na listingu 6.2.

Listing 6.2: Definicja funkcji *duration* dla agenta aktywnego *Publisher*

```
duration :: Agent -> Int -> Int
duration Publisher 1 = 1
duration Publisher 2 = 2
```

```
duration Publisher 3 = 2
duration Publisher 4 = 1
duration Publisher 5 = 3
duration Publisher 6 = 3
duration Publisher 7 = 2
duration Publisher 8 = 1
```

Definicja warstwy kodu dla agenta aktywnego `Subscriber` jest zaprezentowana na listingu 6.3. Priorytet tego agenta został zadeklarowany jako 1, co oznacza, że jest to niższy priorytet od priorytetu agenta `Publisher`. W przypadku przerwania systemowego *SysTick* agent aktywny `Subscriber` zostanie wywłaszczony na rzecz, ubiegającego się o zasoby procesora agenta `Publisher`.

Listing 6.3: Definicja aktywnego agenta *subscriber*

```
agent Subscriber (1) {
    buffer :: Int = 0;
    dataMessageSub :: Int = 0;

    loop {                                              -- 1
        in subData dataMessageSub;                      -- 2
        buffer = buffer + dataMessageSub;               -- 3
        out status buffer;                              -- 4
    }
}
```

Deklaracja i definicja parametrów dla agenta aktywnego `Subscriber` wygląda następująco:

- `buffer :: Int = 0` – parametr typu całkowitego o wartości początkowej równej 0. Wartości danych otrzymywanych od agenta aktywnego `Publisher` inkrementują lub dekrementują wartość parametru `buffer`.
- `dataMessageSub :: Int = 0` – parametr typu całkowitego, przygotowany na otrzymanie danych od agenta aktywnego `Publisher`.

Agent `Subscriber` pracuje w nieskończonej pętli (ale nie okresowej), w której pobiera wartość przez port `subData`, aktualizuje wartość parametru `buffer` i wysyła ją przez port `status`.

Część funkcji `duration`, deklarująca długość egzekucji poszczególnych instrukcji dla agenta aktywnego `Subscriber`, jest przedstawiona na listingu 6.4.

Listing 6.4: Definicja funkcji *duration* dla agenta aktywnego *Subscriber*

```
duration :: Agent -> Int -> Int
duration Subscriber 1 = 1
```

```
duration Subscriber 2 = 2
duration Subscriber 3 = 3
duration Subscriber 4 = 2
```

Agent pasywny `Status` ma zdefiniowane dwa porty proceduralne: wejściowy `recStatus` i wyjściowy `sendStatus`. Nazwy procedur zdefiniowane w warstwie kodu danego agenta są dokładnie takie same jak odpowiadające im porty. Definicję dynamiki agenta pasywnego `Status` pokazano na listingu 6.5.

Listing 6.5: Definicja agenta pasywnego *Status*

```
agent Status {
    bufferValue :: Int = 0;
    statusMessage :: String = "";

    proc recStatus {
        in recStatus bufferValue;                    -- 1
        exit;                                        -- 2
    }

    proc sendStatus {
        select {                                     -- 3
            alt (bufferValue < 1) {
                statusMessage = "Lower";             -- 4
            }
            alt (bufferValue >= 1) {
                statusMessage = "Bigger";            -- 5
            }
        }
        out sendStatus statusMessage;                -- 6
        exit;                                        -- 7
    }
}
```

Definicje parametrów dla pasywnego agenta `Status` wyglądają następująco:

- `bufferValue :: Int = 0` – parametr typu całkowitego definiowany z wartością początkową równą 0. Wartość ta jest przesyłana przez agenta aktywnego `Suscriber`. Na podstawie tej wartości agent `Status` oblicza status agenta `Suscriber`.
- `statusMessage :: String = ""` – wartość typu łańcuchowego przedstawia bieżący status agenta `Suscriber`.

Pierwsza z procedur służy wyłącznie zapisaniu nowej wartości do parametru`bufferValue` w celu późniejszego użycia do obliczenia statusu aktywnego agenta `Suscriber`. Druga z procedur służy do pobrania odpowiedniego komunikatu, określającego modyfikację bufora.

Fragment funkcji `duration` zaprezentowany na listingu 6.6 zawiera definicje czasów trwania poszczególnych instrukcji dla pasywnego agenta `Status`.

Listing 6.6: Definicja funkcji *duration* dla pasywnego agenta *Status*

```
duration :: Agent -> Int -> Int
duration Status 1 = 2
duration Status 2 = 1
duration Status 3 = 1
duration Status 4 = 3
duration Status 5 = 3
duration Status 6 = 2
duration Status 7 = 1
```

Jak wspomniano na początku opisu danego przykładu, agent `Publisger` przed wysłaniem danych do agenta `Subscriber` musi wpierw pobrać *token* od agenta pasywnego `TokenHolder`. Agent ten posiada procedurę `sendToken`, która jest wywoływana na jego porcie o tej samej nazwie. Definicję dynamiki agenta `TokenHolder` zaprezentowano na listingu 6.7, a odpowiedni fragment funkcji `duration` na listingu 6.8.

Listing 6.7: Definicja agenta pasywnego *TokenHolder*

```
agent TokenHolder {
    token :: Char = 'T';

    proc sendToken {
        out sendToken token;                    -- 1
        exit;                                   -- 2
    }
}
```

Listing 6.8: Definicja funkcji *duration* dla pasywnego agenta *TokenHolder*

```
duration :: Agent -> Int -> Int
duration TokenHolder 1 = 2
duration TokenHolder 2 = 1
```

Kompletny kod przykładu jest zamieszczony w dodatku A.1 (str. 121).

Analiza grafu LTS

Rozszerzenie modułów IHR przedstawione w rozdziale 5 pozwala na automatyczne wygenerowanie grafu LTS dla rozważanego przykładu. Graf ten zawiera 103 stany. Poniżej przedstawiono wybrane fragmenty, które są najciekawsze z punktu widzenia modelowania systemów dla platformy jednoprocesorowej.

Dla celów analizy grafu przyjęto, następujące zapisy:

- (n) – oznacza numer danego stanu;
- $(n) \rightarrow (m)$: l oznaczać będzie przejście ze stanu n do stanu m asygnowane etykietą l, znajdującą się na łuku pomiędzy tymi stanami.

Graf LTS został wygenerowany przy założeniu, że przerwanie systemowe *SysTick* będzie generowane z okresem równym dziesięciu jednostkom czasu. Odpowiedni wpis został dokonany w pliku *PubSubStatus.hs*, który jest pośrednią reprezentacją modelu określoną jako *Intermediate Haskell Representation* (IHR):

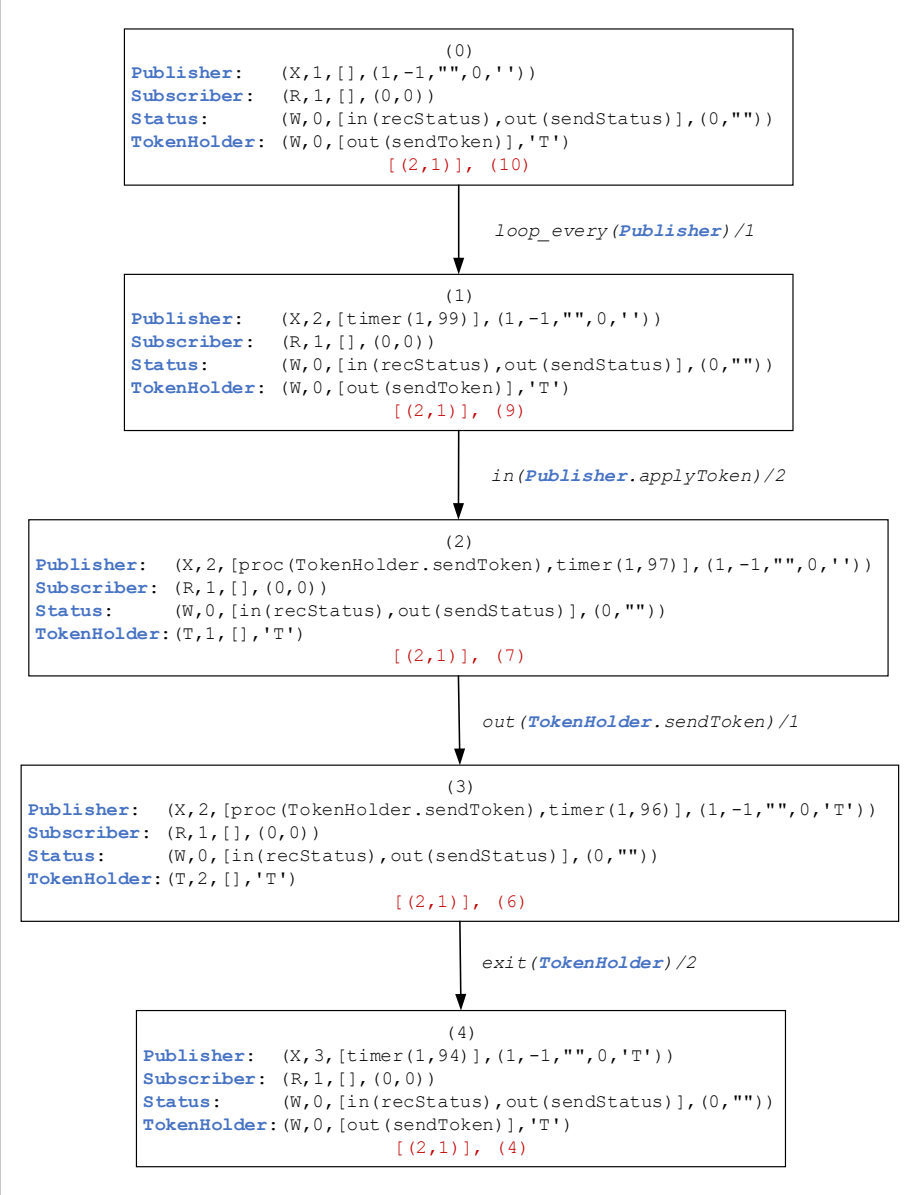
Listing 6.9: Definicja okresu generowania przerwania systemowego *SysTick*

```
-- | Defines the SysTick period
t0 :: Int
t0 = 10
```

W początkowej fazie działania systemu aktywny agent `Publisher` przejmuje zasoby procesora i jako jedyny jest uaktywniany w trybie *running* (X). Na rysunku 6.2 pokazano fragment grafu LTS, którego stany przedstawiają moment pobrania tokena przez agenta `Publisher`.

Analiza tego fragmentu wygląda następująco:

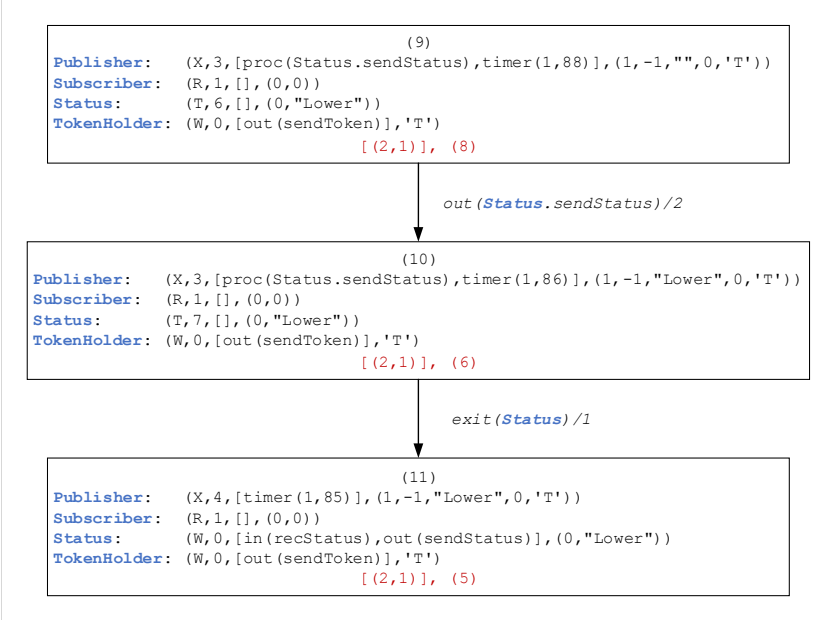
- (0) – stan początkowy modelu. Wpis $[(2,1)], (10)$ świadczy o tym, że kolejnym agentem do pobrania z kolejki priorytetowej jest agent `Subscriber` z numerem 2 i wartością priorytetu 1. Kolejną informacją w tym wpisie jest to, że zostało 10 jednostek czasu do wystąpienia przerwania systemowego *SysTick*. Agent `Subscriber` jest w trybie *ready* (R), co oznacza jego gotowość do przejęcia zasobów procesora. Agenty pasywne `Status` i `TokenHolder` znajdują się w trybie oczekiwania na wywołanie ich procedur (W - *waiting*). Na ich listach kontekstowych widnieją wpisy świadczące o dostępności tych procedur: `[in(recStatus),out(sendStatus)]` oraz `[out(sendToken)]`.
- $(0) \rightarrow (1)$: `loop_every(Publisher)/1` – Agent `Publisher` wykonał swoją pierwszą instrukcję (wejście do pętli), która trwała 1 j.cz.
- (1) – Wpis `[timer(1,99)]` na liście kontekstowej agenta `Publisher` świadczy o tym, że do ukończenia pętli *loop every* zostało jeszcze 99 j.cz.
- $(1) \rightarrow (2)$: `in(Publisher.applyToken)/2` – Agent `Publisher` wywołał procedurę wyjściową `sendToken` pasywnego agenta `TokenHolder` (trwało to 2 j.cz.).
- (2) – Wywołanie procedury przez agenta `Publisher` jest oznaczone wpisem na jego liście kontek-



Rysunek 6.2: LTS (frament): pobranie *tokena* przez agenta *Publisher*

stowej: `proc(TokenHolder.sendToken)`. Agent pasywny `TokenHolder` znajduje się w trybie *taken* (T), a jego lista kontekstowa jest pusta, co świadczy o tym, że w bieżącym stanie realizuje już procedurę i nie można wywołać kolejnej. Wartość parametru przechowującego token wynosi $'T'$. Wszystkie te zależności wyrażone są następującym wpisem: `TokenHolder: (T,1,[],'T')`.

- $(2) \rightarrow (3)$: `out(TokenHolder.sendToken)/1` – Agent `TokenHolder` wykonał swoją pierwszą instrukcję (trwała 1 j.cz.) Instrukcja ta dotyczyła wysłania *tokena* do aktywnego agenta `Publisher`.
- $(3) \rightarrow (4)$: `exit(TokenHolder)/2` – Zakończenie realizacji procedury przez agenta `TokenHolder` (trwało 2 j.cz.).
- (4) – Stan wskazuje, że agent `Publisher` realizuje kolejną swoją instrukcję (3). Agent pasywny `TokenHolder` jest ponownie w trybie *waiting* i oferuje dostępną procedurę `sendToken`.



Rysunek 6.3: LTS (fragment): pobranie statusu agenta *Subscriber* przez agenta *Publisher* z wartością *Lower*

W ten sam sposób wywoływane są procedury `sendStatus` oraz `recStatus` agenta `Status` przez odpowiednio agenty `Publisher` i `Subsciber`. Fragment grafu LTS, w który aktywny agent `Publisher` pobiera status agenta `Subscriber` przedstawiono na rysunku 6.3

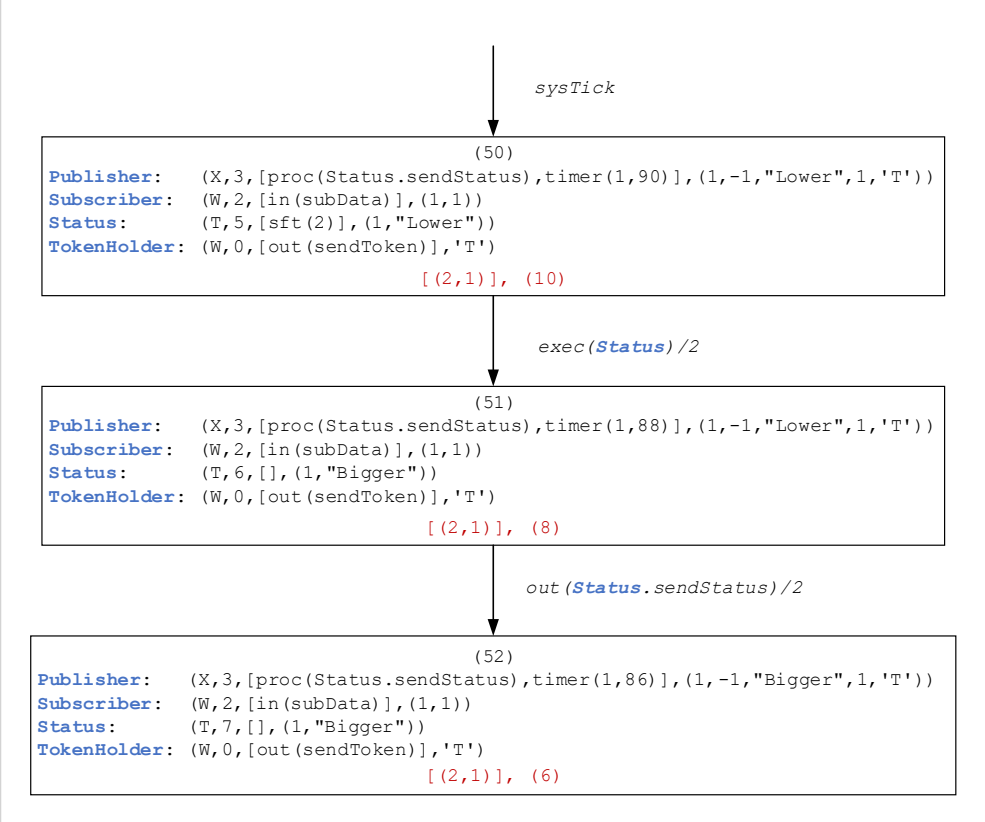
Analiza stanów modelu dla tego fragmentu wygląda następująco:

- (10) – Agent `Publisher` wywołał procedurę `sendStatus` agenta pasywnego `Status` o czym świadczy odpowiedni wpis na liście kontekstowej. Z powyższego wpisu można również dostrzec, że obliczony status przyjmuje wartość `Lower`.
- (11) – Agent pasywny `Status` jest ponownie w trybie *waiting* i oferuje dwie dostępne procedury (widoczne na liście kontekstowej).

W podobny sposób wygląda analiza stanów modelu, w których wartość statusu agenta `Subscriber` obliczana jest jako `Bigger` (rysunek 6.4).

Kolejnym ciekawym fragmentem grafu LTS dla przykładu *publikator i subskrybent* jest moment zakończenia pętli *loop every*. Sytuacja ta przedstawiona została na rysunku 6.5.

- (81) – Agent `Publisher` na skutek ukończenia instrukcji wewnątrz bloku *loop every* pozostaje w trybie oczekiwania *waiting* (W).
- (81) → (82): `timeout(Publisher)/0` – Realizowana jest tranzycja $STLoopEnd$, reprezentowana tu przez etykietę *timeout*, która „wybudza” agenta `Publisher`.
- (82) – Agent `Publisher` ponownie wykonuje swoją pierwszą instrukcję, którą jest wejście do pętli *loop every*.
- (82) → (83): `loop_every(Publisher)/1` – Wykonanie instrukcji *loop every*.



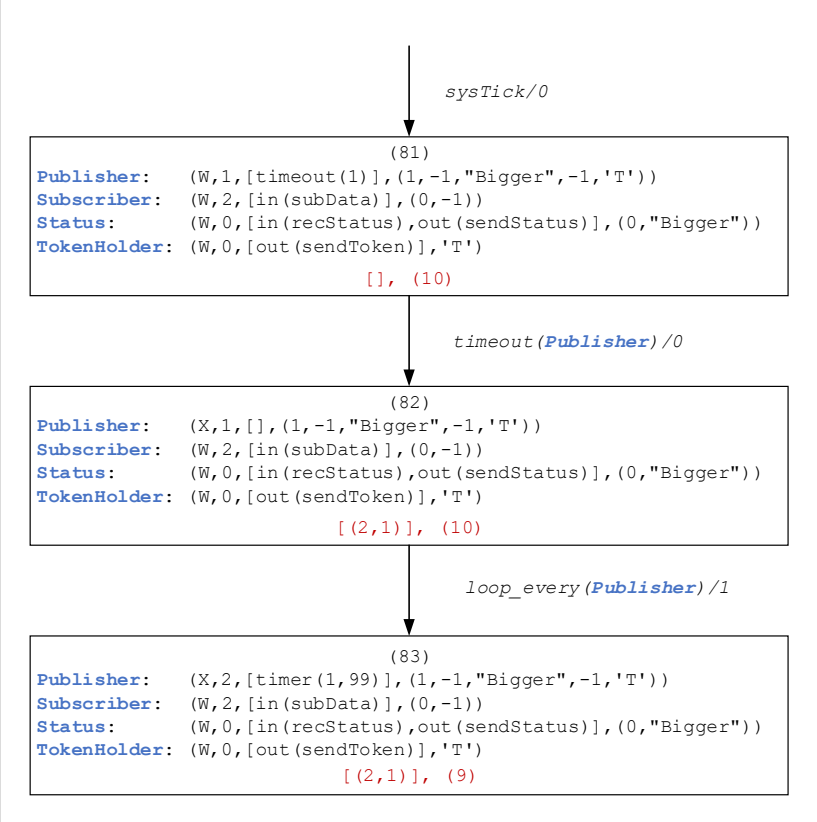
Rysunek 6.4: LTS (fragment): pobranie statusu agenta *Subscriber* przez agenta *Publisher* z wartością *Bigger*

Bardzo ważną cechą, z punktu widzenia założeń niniejszej rozprawy, jest możliwość uruchamiania agentów o niższych priorytetach w chwili, gdy agenty o wyższych wartościach priorytetów przechodzą lub znajdują się w trybie oczekiwania *waiting* (W). Sytuację taką zawiera fragment grafu LTS pokazany na rysunku 6.6.

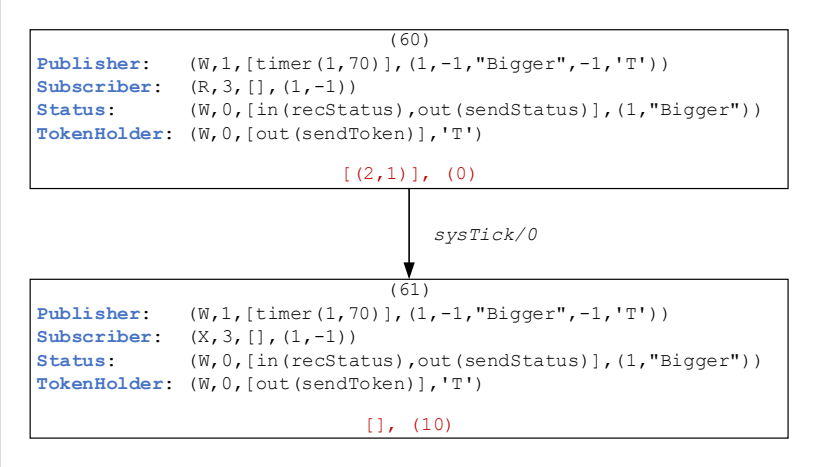
- (60) – Agent `Publisher` jest w trybie oczekiwania *waiting* (W), na skutek ukończenia wszystkich instrukcji zawartych w bloku pętli *loop every*. Pętla ta została zadeklarowana w warstwie kodu agenta z okresem czasowym równym 100 j.cz. Ponieważ suma czasów trwania instrukcji w niej zawartych nie przekracza tej wartości, to agent zawierający tą pętlę przechodzi do trybu *waiting*, umożliwiając tym samym przejęcie zasobów procesora innemu agentowi aktywnemu – w tym przykładzie jest to agent `Subscriber`.
- (60) → (61): `sysTick/0` – przerwanie systemowe *SysTick*.
- (61) – Agent o niższym priorytecie `Subscriber` znajduje się w trybie *running* (X).

6.2. Obserwator

Kolejne studium przypadku przedstawia przykład modelowania wzorca obserwator (ang. *observer*), należący do grupy wzorców czynnościowych. Główny element tego wzorca, czyli obserwator reprezentowany



Rysunek 6.5: LTS (frament): koniec pętli *loop every* agenta *Publisher*

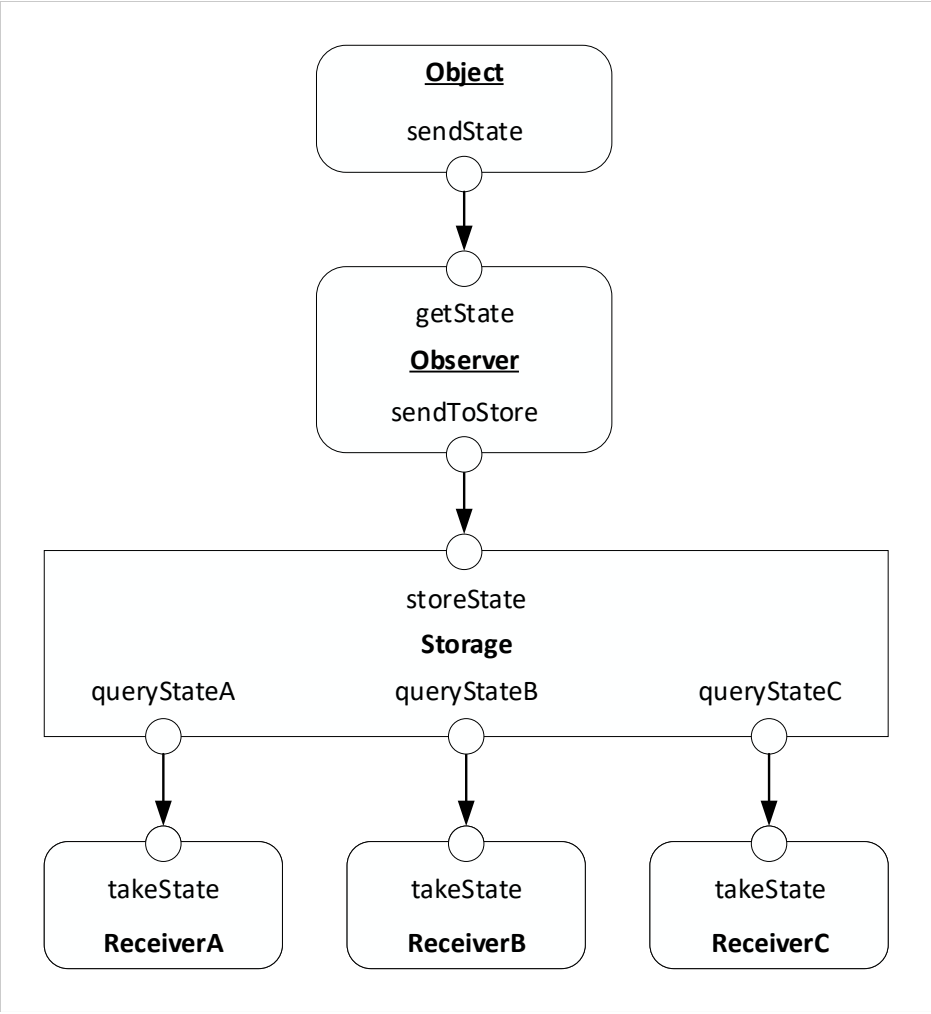


Rysunek 6.6: LTS (fragment): uruchomienie agenta *Subscriber* na skutek przerwania systemowego *SysTick*.

w modelu za pomocą agenta aktywnego `Observer`, analizuje bieżący stan danego obiektu i przesyła go dalej wszystkim zainteresowanym aktorom występującym w systemie. W modelu obserwowany obiekt jest przedstawiony za pomocą agenta aktywnego `Object`. Przyjęto, że jego stan jest zapisywany do agenta pasywnego `Storage`, za pomocą wywołania procedury `sendToStore`. Agent `Storage` modeluje element pamięciowy, występujący w analizowanym systemie. Elementami, które są informowane o stanie agenta `Object`, są odpowiednio agenty aktywne `PublisherA`, `PublisherB` oraz `PublisherC`. Każdemu z nich

dedykowany jest inny stan agenta `Object`. W celu pobrania tego stanu wywołują one dedykowane im procedury wyjściowe agenta pasywnego `Storage`.

Istotnym założeniem w prezentowanym przykładzie jest to, że każdy agent aktywny kończąc swoje zadanie do wykonania, kończy zarazem swoją pracę i przechodzi do trybu *finished* (F). W konsekwencji ostatni stan modelu jest stanem *martwym*, z którego nie ma możliwości wykonania jakiejkolwiek tranzycji do następnego stanu. Dodatkowo założono, że agenty `PublisherA`, `PublisherB` oraz `PublisherC` nie są uruchamiane podczas startu systemu i pozostają w trybie *init*. Ich aktywacja następuje w chwili wysłania wszystkich możliwych stanów agenta `Object` do agenta pasywnego `Storage`.



Rysunek 6.7: Model wzorca obserwator

Warstwa kodu

Głównym zadaniem agenta aktywnego `Object` jest wysłanie do obserwatora swoich stanów o wartościach odpowiednio A, B i C. Odbiorcą wartości tych stanów jest agent `Observer`. Komunikacja pomiędzy tymi agentami odbywa się za pomocą portu `sendState` po stronie agenta `Object` oraz portu `getState`

po stronie agenta `Observer`. Agent `Object` wysyłając swoje stany, czeka odpowiednio 10 jednostek czasu na podjęcie komunikacji ze strony agenta `Observer`. Jeżeli ta komunikacja nie zostanie podjęta, wtedy wykonywana jest instrukcja skoku zawarta w poszczególnych blokach **`fail`**. Skok do odpowiedniej etykiety powoduje ponowną próbę nawiązania komunikacji z agentem `Observer`. W przypadku gdy komunikacja zostanie podjęta, agent `Object` wysyła kolejny swój stan w bloku **`success`**. Po wysłaniu ostatniego stanu, aktywny agent `Object` przechodzi do trybu *finished*, na skutek wykonania instrukcji **`exit`**.

Listing 6.10: Definicja agenta aktywnego *Object*

```
agent Object (0) {
  stateA :: Char = 'A';
  stateB :: Char = 'B';
  stateC :: Char = 'C';

  state_a:
  out (20) sendState stateA {              -- 1
    success {
      state_b:
      out (20) sendState stateB {          -- 2
        success {
          state_c:
          out (20) sendState stateC {      -- 3
            success {
                exit;                      -- 4
            }
            fail { jump state_c; }         -- 5
          }
        }
        fail { jump state_b; }             -- 6
      }
    }
    fail { jump state_a; }                 -- 7
  }
}
```

Warstwę kodu agenta aktywnego `Object` zaprezentowano na listingu 6.10. Priorytet tego agenta jest zadeklarowany jako 0, co oznacza, że jest to najwyższa z możliwych wartości.

Agent `Object` używa następujące parametry (lokalne zmienne):

- `stateA :: Char = 'A'` – parametr typu znakowego o wartości początkowej A. Wartość tego pa-

rametru określa stan agenta `Object`.

- `stateB :: Char = 'B'` – parametr typu znakowego o wartości początkowej B. Podobnie jak powyżej, wartość tego parametru określa stan agenta `Object`.
- `stateC :: Char = 'C'` – parametr typu znakowego o wartości początkowej C. Jest to kolejna z możliwych wartości stanu agenta `Object`.

Część funkcji `duration`, deklarująca długość egzekucji poszczególnych instrukcji dla agenta aktywnego `Object`, jest przedstawiona na listingu 6.11.

Listing 6.11: Definicja funkcji *duration* dla agenta aktywnego *Object*

```
duration :: Agent -> Int -> Int
duration Object 1 = 2
duration Object 2 = 2
duration Object 3 = 2
duration Object 4 = 1
duration Object 5 = 1
duration Object 6 = 1
duration Object 7 = 1
```

Agent `Observer` odbiera informację o stanie agenta `Object` i przesyła ją do agenta pasywnego `Storage`, wywołując procedurę `storeState`. Podobnie jak w definicji dynamiki agenta `Object`, tutaj także zastosowano komunikację nieblokującą pomiędzy agentami. Agent `Observer` czeka 10 jednostek czasu na informację o stanie agenta `Object`. Ponadto wraz z każdą otrzymaną wartością stanu, inkrementowana jest wartość lokalnego licznika `counter`. Jeżeli przekroczy ona wartość równą 3, agent `Observer` aktywuje agenty aktywne `ReceiverA`, `ReceiverB` oraz `ReceiverC`, a sam przechodzi do trybu *finished* (F).

Listing 6.12: Definicja agenta aktywnego *Observer*

```
agent Observer (0) {
  state :: Char = ' ';
  counter :: Int = 0;

  loop {                                    -- 1
    in (20) getState state {                -- 2
      success {
        counter = counter + 1;              -- 3
        out sendToStore state;              -- 4
      }
    }
```

```
    select {                                -- 5
      alt (counter == 3) {
        start ReceiverA;                    -- 6
        start ReceiverB;                    -- 7
        start ReceiverC;                    -- 8
        exit;                               -- 9
      }
    }
  }
}
```

Warstwa kodu agenta aktywnego `Observer` przedstawia listing 6.12. Priorytet tego agenta jest zadeklarowany jako 0, czyli jest to najwyższy priorytet w systemie.

Agent `Observer` używa następujące parametry (lokalne zmienne):

- `state :: Char = ' '` – parametr typu znakowego zadeklarowany z wartością spacji, która jest wartością początkową. Parametr ten jest kontenerem dla wartości stanu agenta `Object`.
- `counter :: Int = 0` – parametr typu całkowitego zadeklarowany z wartością zerową. Jest to licznik, zliczający liczbę otrzymanych wartości dla stanu agenta `Object`.

Część funkcji `duration`, zawierającej deklaracje długości egzekucji poszczególnych instrukcji dla agenta aktywnego `Observer`, przedstawiono na listingu 6.13.

Listing 6.13: Definicja funkcji *duration* dla agenta aktywnego *Observer*

```
duration :: Agent -> Int -> Int
duration Observer 1 = 1
duration Observer 2 = 2
duration Observer 3 = 3
duration Observer 4 = 2
duration Observer 5 = 1
duration Observer 6 = 1
duration Observer 7 = 1
duration Observer 8 = 1
duration Observer 9 = 1
```

Agent pasywny `Storage` stanowi w modelu odpowiednik elementu systemu, który zapamiętuje stany obiektu. Służy do tego procedura wejściowa `storeState`. Za jej pomocą agent aktywny `Observer` przekazuje poszczególne wartości stanów agenta `Object`. Kolejne procedury `queryStateA`, `queryStateB` i `queryStateC` służą do wyciągania poszczególnych wartości stanu przez agenty aktywne `ReceiverA`, `ReceiverB` oraz `ReceiverC`.

Listing 6.14: Definicja agenta pasywnego *Storage*

```
agent Storage {
  storedStateA :: Char = ' ';
  storedStateB :: Char = ' ';
  storedStateC :: Char = ' ';
  state :: Char = ' ';

  proc storeState {
    in storeState state;                                    -- 1
    select {                                                -- 2
      alt (state == 'A') { storedStateA = state; }          -- 3
      alt (state == 'B') { storedStateB = state; }          -- 4
      alt (state == 'C') { storedStateC = state; }}         -- 5
    exit;                                                   -- 6
  }

  proc queryStateA {
    out queryStateA storedStateA;                           -- 7
    exit;                                                   -- 8
  }

  proc queryStateB {
    out queryStateB storedStateB;                           -- 9
    exit;                                                   -- 10
  }

  proc queryStateC {
    out queryStateC storedStateC;                           -- 11
    exit;                                                   -- 12
  }
}
```

Warstwa kodu agenta pasywnego `Storage` została zaprezentowana na listing 6.14.

Agent `Storage` używa następujące parametry (lokalne zmienne):

- `storedStateA :: Char = ' '` – parametr typu znakowego zadeklarowany z wartością spacji, która jest wartością początkową. Parametr ten jest kontenerem dla wartości A stanu agenta `Object`.
- `storedStateA :: Char = ' '` – parametr typu znakowego zadeklarowany z wartością spacji, która jest wartością początkową. Parametr ten jest kontenerem dla wartości B stanu agenta `Object`.

- `storedStateA :: Char = ' '` – parametr typu znakowego zadeklarowany z wartością spacji, która jest wartością początkową. Parametr ten jest kontenerem dla wartości C stanu agenta `Object`.
- `state :: Char = ' '` – parametr typu znakowego zadeklarowany z wartością spacji, która jest wartością początkową. Parametr ten jest kontenerem dla wartości stanu agenta `Object`.

Część funkcji `duration`, posiadającej deklaracje długości egzekucji poszczególnych instrukcji dla agenta pasywnego `Storage`, przedstawiono na listingu 6.15.

Listing 6.15: Definicja funkcji *duration* dla agenta pasywnego *Storage*

```
duration :: Agent -> Int -> Int
duration Storage 1 = 2
duration Storage 2 = 1
duration Storage 3 = 3
duration Storage 4 = 3
duration Storage 5 = 3
duration Storage 6 = 1
duration Storage 7 = 2
duration Storage 8 = 1
duration Storage 9 = 2
duration Storage 10 = 1
duration Storage 11 = 2
duration Storage 12 = 1
```

Zadaniem agentów aktywnych `ReceiverA`, `ReceiverB` oraz `ReceiverC`, należących do grupy odbiorców, jest pobranie odpowiedniej wartości stanu agenta `Object`. W przypadku, gdy poszczególny odbiorca pobierze dedykowaną mu wartość stanu, przechodzi do trybu *finished* i jego praca zostaje zakończona. Z uwagi na podobieństwo agentów modelujących odbiorców, ograniczono się do analizy jednego z nich, czyli agenta aktywnego `ReceiverA`. Warstwa kodu tego agenta została zaprezentowana na listing 6.16.

Listing 6.16: Definicja agenta aktywnego *ReceiverA*

```
agent ReceiverA (1) {
  receivedState :: Char = ' ';

  loop {                                          -- 1
    in takeState receivedState;                   -- 2
    select {                                      -- 3
      alt (receivedState == 'A') { exit; }        -- 4
    }
  }}
```

Agenta `ReceiverA` posiada tylko jeden parametr (lokalną zmienną) w definicji swojej dynamiki:

- `receivedState :: Char = ' ';` – jest to parametr typu znakowego zadeklarowany z wartością spacji, która jest wartością początkową. Parametr ten jest kontenerem dla otrzymanej wartości stanu agenta `Object`. W przypadku tego agenta będzie to wartość A.

Fragment funkcji `duration`, posiadającej deklaracje długości egzekucji poszczególnych instrukcji dla agenta aktywnego `ReceiverA`, przedstawiono na listingu 6.17.

Listing 6.17: Definicja funkcji *duration* dla agenta aktywnego *ReceiverA*

```
duration :: Agent -> Int -> Int
duration ReceiverA 1 = 1
duration ReceiverA 2 = 2
duration ReceiverA 3 = 1
duration ReceiverA 4 = 1
```

Kompletny kod przykładu jest zamieszczony w dodatku A.2 (str. 123).

Analiza grafu LTS

Podobnie jak w poprzednim przykładzie posłużono się rozszerzeniem modułów IHR, przedstawionym w rozdziale 5, które pozwala na automatyczne wygenerowanie grafu LTS dla rozważanego przykładu. Graf ten zawiera 78 stanów. W poniższym opisie przedstawiono wybrane fragmenty, które są najciekawsze z punktu widzenia modelowania systemów dla platformy jednoprocesorowej.

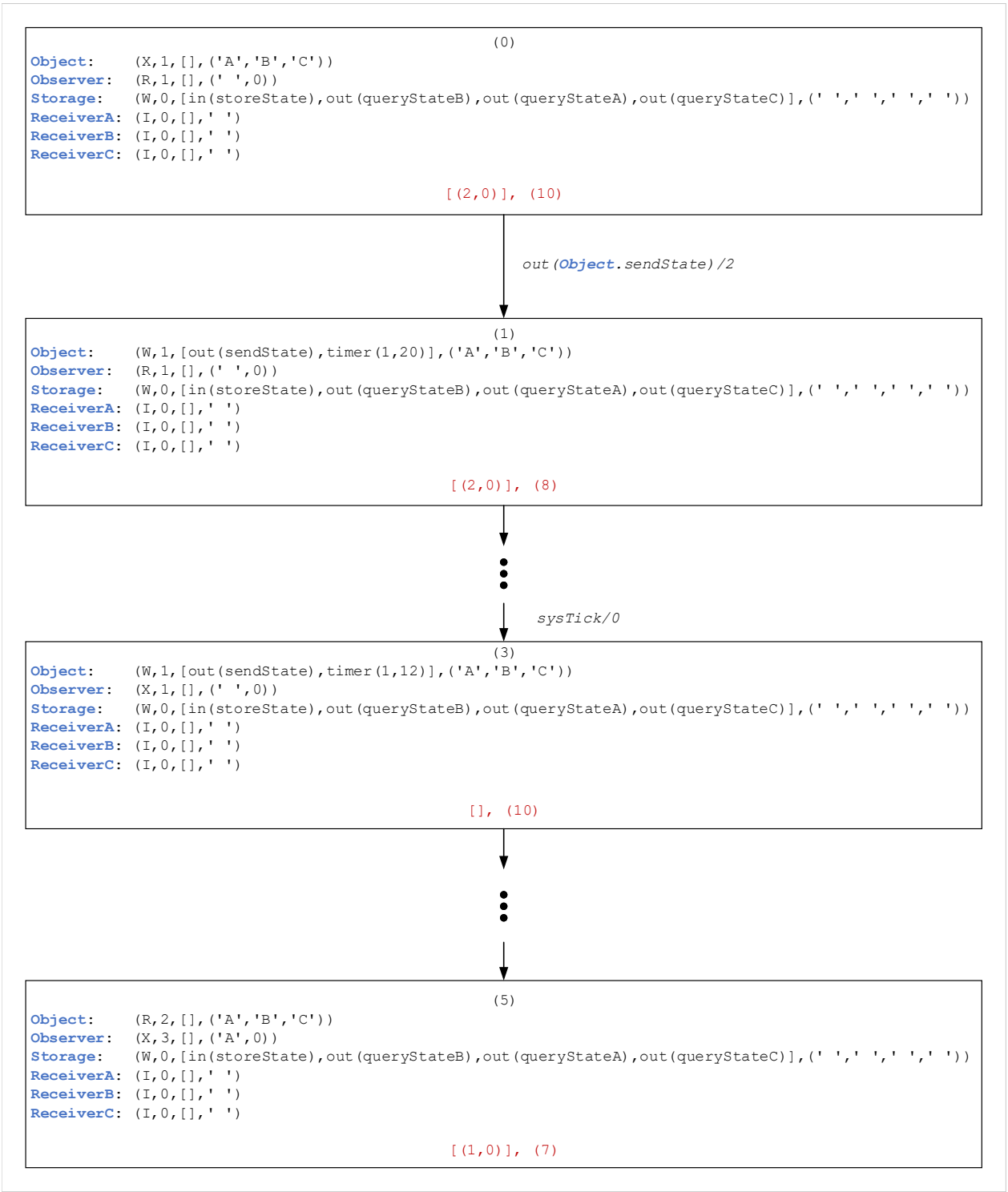
W celu analizy poszczególnych fragmentów grafu przyjęto (jak w poprzednim przykładzie), następujące zapisy:

- (n) – oznacza numer danego stanu;
- $(n) \rightarrow (m)$: l oznaczać będzie przejście ze stanu n do stanu m asygnowane etykietą l, znajdującą się na łuku pomiędzy tymi stanami.

Z punktu widzenia celów i założeń niniejszej rozprawy istotnym zagadnieniem jest *nieblokująca* komunikacja pomiędzy agentami. Ten rodzaj komunikacji uzależnia wymianę danych od aspektów związanych z upływem czasu. Operacje na portach agenta mają ściśle zdefiniowany czas oczekiwania na reakcję ze strony innego agenta. W modelach systemów czasu rzeczywistego tego typu uwarunkowanie sprawia, że agent inicjalizujący komunikację nie jest zależny od dynamiki innego agenta i może ponownie ubiegać się o zasoby procesora, nie będąc ”zawieszonym” na swoim porcie komunikacyjnym.

W poniższym przykładzie komunikacja nieblokująca została wykorzystana m.in. pomiędzy agentami `Object` i `Observer`. Na rysunku 6.8 pokazano fragment grafu LTS, który przedstawia stany modelu podczas komunikacji nieblokującej.

- (0) – agent `Object`, posiadając zasoby procesora, zaczyna nieblokującą komunikację z agentem



Rysunek 6.8: LTS (fragment): komunikacja *nieblokująca* pomiędzy agentami aktywnymi *Object* i *Observer*.

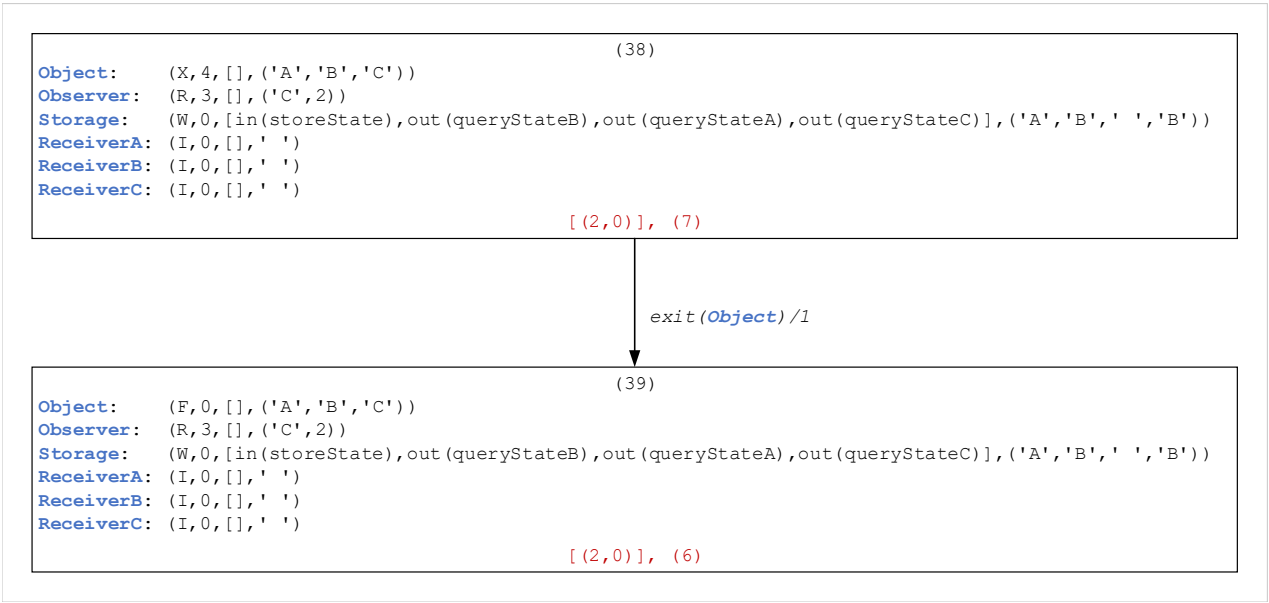
`Observer`. Czas oczekiwania na reakcję ze strony agenta `Observer` został zdefiniowany na 20 jednostek czasu. Agent `Observer` jest w trybie *ready* (R), co oznacza jego gotowość do przejęcia zasobów procesora, i co za tym idzie, podjęcie komunikacji z agentem `Object`.

- (1) – w tym stanie agent `Object` przechodzi do trybu oczekiwania na podjęcie komunikacji ze strony

agenta `Observer`. Na jego liście kontekstowej widać wpis `timer(1,20)`, co oznacza, że od tego momentu będzie odmierzany czas 20 jednostek czasu. Agent `Observer` zmienia swój tryb na *ready* (R), poprzez co zgłasza swoją gotowość do przejęcia zasobów procesora. Tryby agentów nie zmieniają się do stanu 3.

- (2) → (3): `sysTick/0` – wystąpiło przerwanie systemowe *SysTick*, które promuje agenta `Observer` do przejęcia zasobów procesora.
- (3) – agent `Observer` przejmuje zasoby procesora i zaczyna procesować swoje instrukcje.
- (5) – agent `Observer` pobrał już wartość stanu agenta agent `Object` i teraz przekazuje ją do agenta pasywnego agent `Storage`. Na liście kontekstowej agenta `Object` nie ma już wpisu, informującego o czasie oczekiwania na podjęcie komunikacji.

Moment, w którym agent aktywny `Object` przechodzi do trybu *finished* (F) i tym samym kończy swoją pracę, widoczny jest w stanach modelu 38 i 39 (rys. 6.9).



Rysunek 6.9: LTS (fragment): przechodzenie do trybu *finished* przez agenta aktywnego *Object*.

- (38) → (39): **`exit`**`(Object)/1` – agent `Object` wykonuje swoją czwartą instrukcję, **`exit`**.
- (39) – w tym stanie agent `Object` jest już w stanie *finished* (F). Jego licznik rozkazów przyjmuje wartość 0, czyli nie wskazuje na kolejną do wykonania instrukcję. Lista kontekstowa jest pusta.

W prezentowanym przykładzie zachodzi sytuacja, w której agent aktywny `Observer` inicjalizuje pracę agentów aktywnych `ReceiverA`, `ReceiverB` oraz `ReceiverC`. Przedstawiają to stany modelu od 50 do 53 (rys. 6.10).

- (50) → (51): **`start`**`(Observer)/1` – agent `Observer` inicjalizuje pracę agenta `ReceiverA`.
- (51) → (52): **`start`**`(Observer)/1` – agent `Observer` inicjalizuje pracę agenta `ReceiverB`.
- (52) → (53): **`start`**`(Observer)/1` – agent `Observer` inicjalizuje pracę agenta `ReceiverC`.

- (56) – w tym stanie agent `ReceiverA` przejmuje kontrolę nad zasobami procesora, na skutek wystąpienia przerwania systemowego *SysTick*.

6.3. Podsumowanie

W rozdziale zaprezentowano dwa przykłady modelowania systemów czasu rzeczywistego za pomocą języka Alvis. Poza prezentacją modeli skupiono się przede wszystkim na analizie wybranych fragmentów grafów LTS, generowanych z użyciem algorytmu opisanego w rozdziale 5.

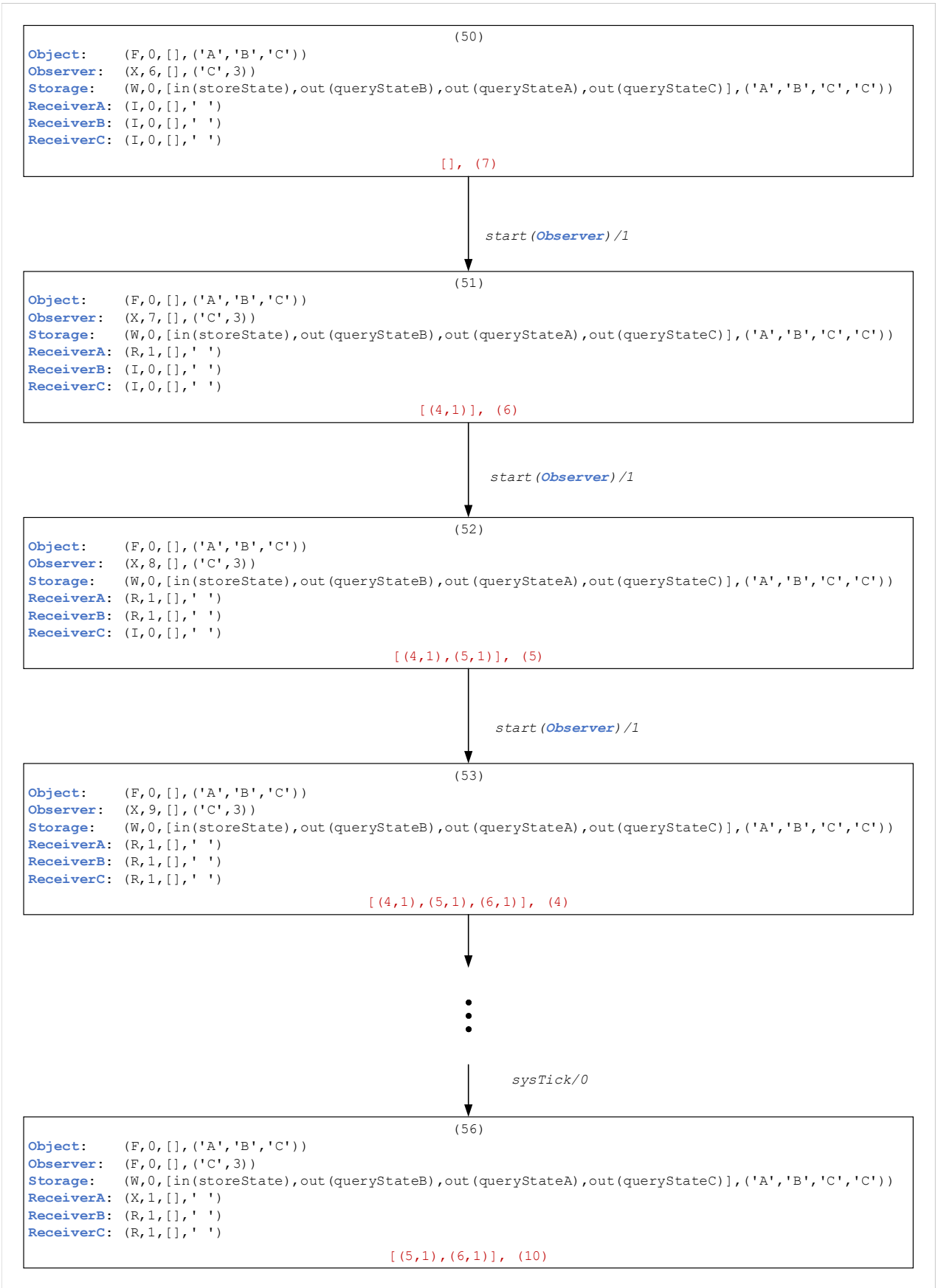
Pierwszy przykład dotyczył popularnego wzorca implementacyjnego *publikator i subskrybent*. Na jego przykładzie pokazano tematy silnie związane z celami niniejszej rozprawy, takie jak:

- komunikacja pomiędzy agentem aktywnym i pasywnym;
- obsługa pętli okresowej *loop every*;
- przełączanie zasobów procesora pomiędzy agentami na skutek wystąpienia przerwania systemowego *SysTick*.

Drugi przykład obrazował wykorzystanie kolejnego wzorca projektowego, jakim jest *obserwator*. Pokazano tutaj:

- komunikację nieblokującą pomiędzy agentami aktywnymi;
- kończenie pracy przez agenty aktywne i zmianę ich trybu na *finished*;
- inicjalizowanie pracy innych agentów aktywnych, przez agenta aktywnego posiadającego kontrolę nad zasobami procesora.

Uzyskanie kompletnego grafu LTS jest kluczowe dla weryfikacji modelu. Jeżeli specyfikacja wymagań została określona z użyciem logiki temporalnej, to graf LTS używany jest do sprawdzenia czy model spełnia weryfikację wymagań. W przypadku języka Alvis możemy to zrobić korzystając z nuXmv [16] i CADP toolbox [24]. Można również zaimplementować dodatkowe funkcje eksportujące graf LTS do formatów wejściowych innych narzędzi służących do weryfikacji modelowej.



Rysunek 6.10: LTS (fragment): inicjalizowanie innych agentów aktywnych przez agenta aktywnego *Observer*.

7. Metody formalne dla systemów z czasem

Próba opracowania wyczerpującego przeglądu metod formalnych stosowanych do modelowania systemów informatycznych byłaby zadaniem niezwykle trudnym i czasochłonnym. Jeżeli przeanalizujemy chociażby zawartość portalu *Formal Methods Wiki*, to znajdziemy tam informacje o:

- 36 organizacja powiązanych z metodami formalnymi,
- 59 korporacjach stosujących w jakimś zakresie metody formalne, wśród nich m.in: Bell Labs, IBM i Rolls Royce,
- **108** notacjach, językach lub narzędziach.

Z drugiej strony analizując liczbę publikacji związanych z metodami formalnymi można odnieść wrażenie, że zaledwie kilka z tych podejść zyskało dużą popularność. Niewątpliwym liderem są tutaj sieci Petriego [40], [37], [30], [46], [45]. Różnorodność typów sieci Petriego pozwala na stosunkowo łatwe dobranie klasy sieci do konkretnego zastosowania. Ważną grupę stanowią formalizmy zaliczane do *algebr procesów* [11], np.: CCS (Calculus of Communicating Systems [35], [1]), CSP (Communicating Sequential Processes [27]), algebra LOTOS (*Language Of Temporal Ordering Specification* [29], ACP (*Algebra of Communicating Processes* [10]), czy też rachunek π (π-*calculus* [36]). Z kolei patrząc na publikacje związane z weryfikacja modelową [4] stosunkowo często rozważanym formalizmem są automaty czasowe [2], [3], [57]. Jedną z popularniejszych klas są *bezpieczne automaty czasowe* [26], dla których opracowano środowisko UPPALL [9] wspierające projektowanie i weryfikację modeli.

Biorąc pod uwagę tematykę niniejszej rozprawy, w rozdziale tym przedstawiono porównanie języka Alvis z automatami czasowymi i czasowymi kolorowanymi sieciami Petriego. Te dwa formalizmy wydaję się być najbliżej języka Alvis, jeżeli chodzi o podejście do modelowania systemów wbudowanych Poprzez zaprezentowane przykłady spróbowano określić, jakie cechy Alvisa są analogiczne z cechami prezentowanych formalizmów oraz jakie właściwości jego konstrukcji, semantyki i przyjętej definicji odróżniają ten język w sposób zdecydowany i stanowią swoistą wartość dodaną.

Biorąc pod uwagę, że tematyka i cele niniejszej rozprawy silnie skupione są na domenie czasu w systemach współbieżnych, starano się podkreślić i porównać te aspekty Alvisa, które odnoszą się do zagadnień związanych z takimi tematami jak:

[0] `http://formalmethods.wikia.com/wiki`

- modelowanie czasu, w sensie gdzie jest on umieszczany w odniesieniu do konkretnego formalizmu;
- odniesienie aspektów czasowych modelu, jak choćby zmienne czasowe, czas trwania akcji lub instrukcji (Alvis) do wzorca czasu, jakim w przypadku automatów czasowych jest *zegar globalny* a dla Alvisa *zegar systemowy*;
- upływ czasu i jego wpływ na stan modelu;
- warunki czasowe decydujące o dynamice danego modelu w odniesieniu do czasu;

7.1. Automaty czasowe

Podobnie jak język Alvis automaty czasowe służą do modelowania systemów współbieżnych z uwzględnieniem aspektów czasowych. Dostarczają one model matematyczny, który wykorzystywany jest następnie do modelowania zachowań w systemach zależnych od czasu.

Weryfikacja poprawności danego systemu realizowana jest z użyciem technik weryfikacji modelowej. Składa się ona z kilku etapów, z których pierwszym jest przedstawienie rozważanego systemu w postaci etykietowanego systemu tranzycyjnego. Stany osiągalne tego systemu oraz tranzycie pomiędzy nimi tworzą model, który podlega weryfikacji. Następny krok to przedstawienie specyfikacji wymagań danego systemu współbieżnego w postaci formuł logicznych [56]. Do specyfikowania wymagań używane są logiki temporalne. Najczęściej są to: LTL (ang. *Linear Temporal Logic* [18], [4]), CTL (ang. *Computation Tree Logic* [18], [4]), TCTL (ang. *Timed Computation Tree Logic*, [4]), RTCTL (ang. *Real-Timed Computation Tree Logic* [20]) lub jakiś podzbiór tych logik [59]. Zazwyczaj rozważany czas ma charakter logiczny, a nie ilościowy, tj. własności opisują kolejność osiągania poszczególnych stanów, ale nie pozwalają ilościowo określać np. czasu przejścia między dwoma wskazanymi stanami.

Zegary

Zegary są zmiennymi rzeczywistymi, które modelują aspekty czasowe danego systemu. Mogą być zerowane w dowolnym momencie działania systemu. Zakłada się, że podczas startu systemu wszystkie jego zegary są zerowane. W późniejszym czasie działania systemu zegary mogą być zerowane niezależnie od siebie w zależności od specyfikacji danego miejsca systemu i akcji, jaka jest wykonywana w danym momencie czasowym. W graficznej reprezentacji modelowej danego systemu zegary przedstawia się najczęściej za pomocą liter alfabetu, jak x, y, z itp. Proces zerowania zegara wyraża się w następujący sposób: $x := 0$. Zapis ten oznacza wyzerowanie zegara x. W celu zamodelowania warunków, jakie nakłada się na dziedzinę czasu, stosuje się ograniczenia dla zegarów. Są one wyrażane za pomocą symboli relacji: $\leq$, $<$, $==$, $>$, $\geq$. Przykładowo warunek iż upływający czas w kontekście danego zegara x, wyrażony w jednostkach czasu, ma być mniejszy bądź równy dwóm jednostkom, wyrażony zostanie w następujący sposób: $x \leq 2$.

Lokacje

Lokacje w teorii automatów czasowych symbolizują stany osiągalne danego systemu. Modelowane są za pomocą kół wypełnionych etykietami, określającymi nazwę danej lokacji. W modelu w sposób zasadniczy wyróżnia się lokację początkową, od której zaczyna się działanie danego systemu i w którym to miejscu zerowane są wszystkie zegary. Lokację, oprócz etykiety określającej nazwę, mogą też posiadać niezmienniki lokacji. Niezmiennik lokacji określa wartość wybranych zegarów, dla których system pozostanie w danej lokacji. Przykładowo lokacja z zapisem: $y \leq 10$ oznacza, że system może pozostać w tej lokacji tak długo jak długo prawdziwy jest niezmiennik czyli maksymalnie 10 jednostek czasu. W wyniku zapisu specyfikacji dla danego systemu może się okazać, że obecna lokacja jest ostatnią z możliwych do osiągnięcia przez ten system.

Akcje

Zmiana pomiędzy lokacjami jest możliwa na skutek wykonywania akcji. Przyjmuje się, że z danej lokacji może wychodzić więcej niż jedna akcja. Dodatkowo akcja może zaczynać się i kończyć w tej samej lokacji. W tej sytuacji mamy do czynienia z procesem wykonywania pętli w systemie. Akcje, podobnie jak lokacje, są etykietowane nazwami, które najczęściej określają rodzaj wykonywanej akcji. Akcje mogą również warunkować wartości zegarów systemowych. Przykładowo, i jest to przypadek najczęstszy, zegary mogą być zerowane w wyniku wykonania pewnej akcji. Akcje mogą też posiadać warunki niezbędne do swojego wykonania. Najczęściej są to wyrażenia logiczne na zmiennych zegarowych i określają relacje liczbowe tych zegarów.

Definicja automatu czasowego

Definicja 7.1. *Automatem czasowym* nazywamy krotkę $\mathbf{TA} = (\Sigma, L, l^0, E, X, I)$, [58] gdzie:

- Σ jest *alfabetem*, czyli skończonym zbiorem *akcji* ;
- L jest skończonym zbiorem *lokacji*;
- $l^0 \in L$ jest *lokacją początkową*;
- $E \subseteq L \times \Sigma \times C(X) \times 2^X \times L$ jest *relacją przejścia* lub *zbiorem tranzycji*;
- X jest skończonym zbiorem *zegarów*;
- $I \colon L \to C(X)$ jest funkcją etykietującą lokację ograniczeniem na zegary nazywanym *niezmiennikiem*;

Przejście z dowolnej lokacji l do następnej l' jest zawsze etykietowane następującymi wpisami:

- akcją a, która modeluje konkretną czynność w systemie;
- zbiorem zegarów $Z \subseteq X$, które zostaną wyzerowane po wykonaniu danego przejścia;

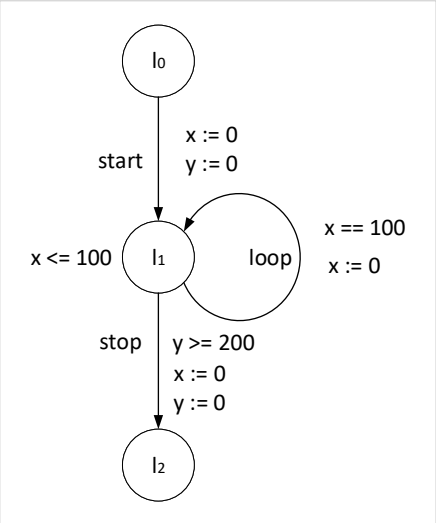
- warunkiem $cond$, jaki musi być spełniony, aby dane przejście było możliwe;

Mając zdefiniowane powyższe składowe przejście e, jako element zbioru E, można opisać zależnością:

$e = l \xrightarrow{a,Z,cond} l'$

Na rysunku 7.1 zaprezentowano przykład automatu czasowego, dla którego ogólna definicja sprowadza się do następujących zapisów:

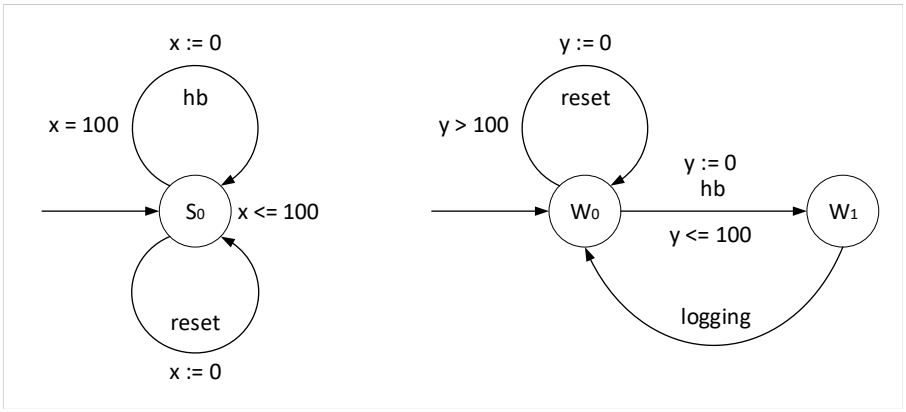
- $\Sigma = (start, loop, stop)$;
- $L = \{l_0, l_1, l_2\}$;
- l^0 jest stanem początkowym;
- $E = \{l_0 \xrightarrow{start,\{x,y\}} l_1, l_1 \xrightarrow{loop,x==100,\{x\}} l_1, l_1 \xrightarrow{stop,y\geq 100,\{x,y\}} l_2\}$;
- $X = \{x, y\}$;
- $I(l_0) = true$, $I(l_1) = x \leq 100$, $I(l_2) = true$



Rysunek 7.1: Przykład automatu czasowego

Przykład systemu *WatchDog* zamodelowanego za pomocą automatów czasowych

Przykład zamodelowania systemu $WatchDog$ za pomocą automatów czasowych został zaprezentowany na rysunku 7.2. Model podzielono na dwie części. Pierwsza z nich odpowiada serwisowi, którego aspektami czasowymi zarządza zegar x. Model serwisu zawiera tylko jedną lokację s_0, w której to znajduje się zaraz po starcie systemu. Niezmiennikiem miejsca jest tutaj warunek zegara $x <= 100$, co oznacza, że odczeka 100 jednostek czasu zanim wyśle sygnał hb (ang. *heartbeat*) do układu nadzorującego. Po wysłaniu tego sygnału zegar x jest resetowany i ponownie zaczyna się proces odliczania 100 jednostek czasu. Niezależnie od wykonywanych akcji serwis może otrzymać polecenie wykonania resetu. W takiej sytuacji zegar x jest również resetowany i procedura odliczania 100 jednostek czasu rozpoczyna się ponownie, niezależnie od tego jaki był upływ czasu do tego momentu.



Rysunek 7.2: Przykład systemu *WatchDog* zamodelowanego za pomocą automatów czasowych

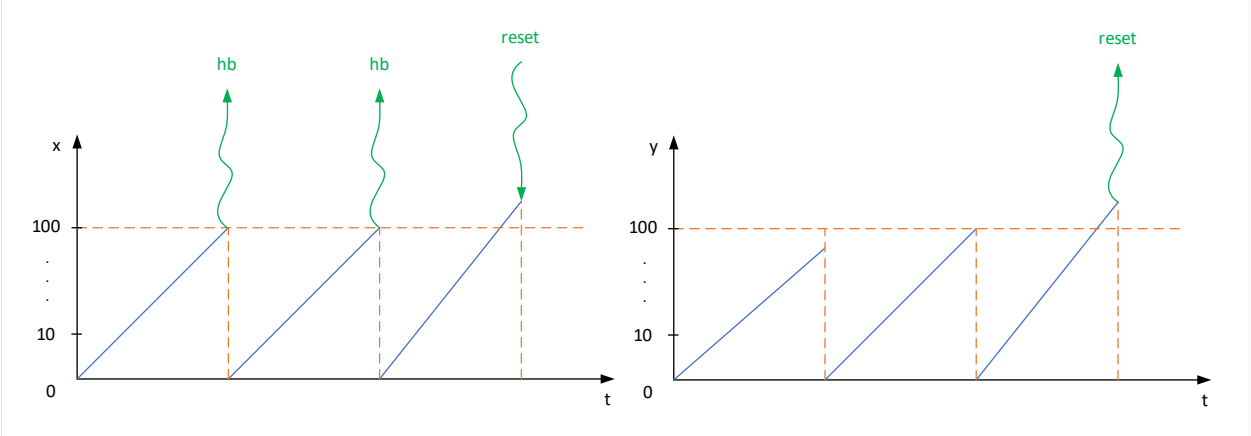
W odniesieniu do definicji automatów czasowych moduł ten prezentuje się następująco:

- $\Sigma = \{hb, reset\}$;
- $L = \{s_0\}$;
- s_0 jest stanem początkowym;
- $X = \{x\}$;
- $E = s_0 \xrightarrow{hb, x=100, \{x\}} s_0, s_0 \xrightarrow{reset, -, \{x\}} s_0$;
- $I(s_0) = x <= 100$

W drugiej części modelu, odpowiedzialnej za sam układ kontrolujący otrzymywanie sygnałów hb oraz resetowanie serwisu w przypadku braku takiego sygnału, zależności czasowe zarządzane są przez zegar y. Wyróżnia się tu dwie lokacje. W pierwszej lokacji w_0 element nadzorujący oczekuje przez 100 jednostek czasu na nadejście sygnału hb. Jeżeli sygnał ten dotrze do układu resetowany jest zegar y i sterowanie przechodzi do drugiej lokacji w_1. W drugim możliwym przypadku, jeżeli sygnał z jakiegoś powodu po czasie 100 jednostek czasu nie zostanie odebrany przez układ nadzorujący, następuje wysłanie sygnału $reset$ do serwisu i wyzerowanie zegara y. W przypadku poprawnego odebrania sygnału hb układ kontrolujący, znajdujący się w lokacji w_1, natychmiast bez żadnych warunków czasowych loguje to zdarzenie i zmienia lokację na początkową, w której ponownie odmierzany jest czas 100 jednostek za pomocą zegara y. Odnosząc powyższe zależności do definicji automatów czasowych, otrzymuje się następujące zapisy:

- $\Sigma = \{hb, reset, logging\}$;
- $L = \{w_0, w_1\}$;
- w_0 jest stanem początkowym;
- $X = \{y\}$;
- $E = w_0 \xrightarrow{hb, y<=100, \{y\}} w_1, w_0 \xrightarrow{reset, y>100, \{y\}} w_0, w_1 \xrightarrow{logging, -, -} w_0$
- $I(s_0) = x <= 100$

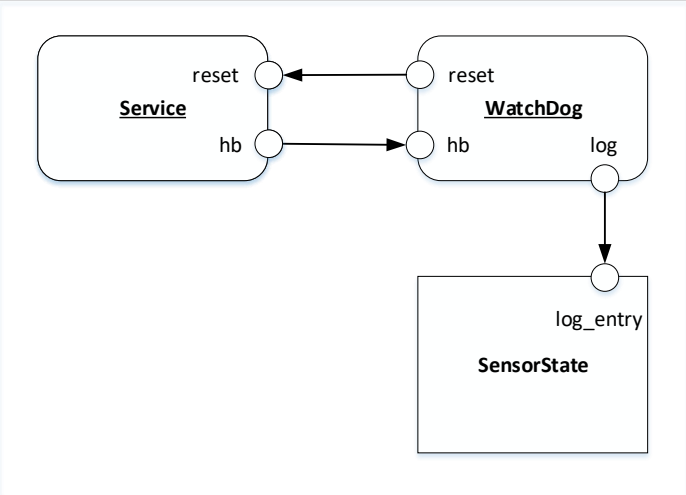
Zależności czasowe zegarów x i y przedstawiono na rysunku 7.3.



Rysunek 7.3: Zależności czasowe zegarów dla modelu *WatchDog*

Przykład systemu *WatchDog* zamodelowanego za pomocą języka Alvis

W celu pokazania analogii pomiędzy automatami czasowymi i Alvisem rozważmy model tego samego systemu, ale przedstawiony w języku Alvis. Warstwa komunikacji, pokazana na rysunku 7.4, wyróżnia dwa agenty $Service$ i $WatchDog$, które modelują odpowiednio serwis i jednostkę kontrolującą komunikację z danym serwisem. Kontrola ta odbywa się na zasadzie monitorowania otrzymywania sygnału hb (ang. *heartbeat*). Dodatkowo wyodrębniono agenta pasywnego Log, jako funkcję do logowania zdarzenia polegającego na poprawnym otrzymaniu sygnału hb lub też jego braku. Użytkownik, bazując na zalogowanych wpisach, może śledzić te momenty, w których sygnał hb nie został wysłany przez serwis. W takim podejściu system jako całość służy do diagnostyki danego serwisu.



Rysunek 7.4: Przykład systemu *WatchDog* zamodelowanego za pomocą języka Alvis

Warstwa kodu definiuje dynamikę rozważanych agentów. Kod agenta $Service$ przedstawia listing 7.1. Zasadniczą jego częścią jest nieskończona pętla wykonywana co 100 jednostek czasu. W pętli tej dokonywane jest przełączenie pomiędzy dwoma możliwymi ścieżkami, w zależności od wartości otrzymanej na

porcie wejściowym *reset*. W przypadku nieotrzymania sygnału *reset*, wysyłany jest sygnał na port wyjściowy *hb*. W przeciwnym przypadku flaga `reset_` ustawiana jest na `True`, co świadczy o wykonaniu resetu przez serwis. W przedstawionym przykładzie procedura resetowania nie została zamodelowana.

Listing 7.1: Logika agenta aktywnego Service

```
1  agent Service (0) {
2      reset_ :: Bool = False;
3
4      loop (every 100) {
5          htbt:
6          in reset reset_;
7          select {
8              alt (reset_ == True) { jump htbt; }
9              alt (reset_ == False) { out hb; }
10         }
11         null;
12     }
13 }
```

Agent *WatchDog*, którego kod pokazano na listingu 7.2, poczeka 100 jednostek czasu na sygnał *hb* na swoim porcie wejściowym *hb*. W przypadku otrzymania sygnału loguje to zdarzenia do funkcji logującej poprzez port wyjściowy *log* i wysyła na zewnątrz informację do serwisu o braku konieczności resetowania. W przypadku braku sygnału agent *WatchDog* wysyła żądanie resetu poprzez port wyjściowy *reset* i loguje to zdarzenie.

Listing 7.2: Logika agenta aktywnego WatchDog

```
1  agent WatchDog (0) {
2      loop {
3          in (100) hb {
4              success {
5                  out reset False;
6                  out log "OK"; }
7              fail {
8                  out reset True;
9                  out log "NOK"; }
10         }
11     }
12 }
```

Jak już wspomniano wcześniej, agent pasywny *Log* modeluje funkcję do logowania zdarzeń otrzymania sygnału *hb*. Dla potrzeb niniejszego przykładu przyjęto, że logowanie polega na inkrementowaniu odpowiednio zmiennej `log_ok_` w przypadku otrzymania sygnału i zmiennej `log_nok_` w przypadku nieotrzymania. Kod agenta pasywnego *Log* przedstawiono na listingu 7.3

Listing 7.3: Logika agenta pasywnego Log

```
1  agent Log {
2      log_ :: String = "null";
3      log_ok_ :: Int = 0;
4      log_nok_ :: Int = 0;
5
6      proc log_entry {
7          in log_entry log_;
8          select {
9              alt (log_ == "OK") { exec log_ok_ = log_ok_ + 1; }
10             alt (log_ == "NOK") { exec log_ok_ = log_ok_ + 1; }
11         }
12         exit;
13     }
14 }
```

Porównanie obu formalizmów

Porównując oba sposoby modelowania tego samego przykładu można dostrzec pewne analogie pomiędzy rozważanymi formalizmami. Pierwszą, niezwykle ważną z punktu widzenia tej rozprawy, jest modelowanie dziedziny czasu i wszelkich aspektów z nią związanych. W automatach czasowych czas kontrolowany jest za pomocą zegarów, które warunkują wykonanie akcji, pozostanie w danej lokacji oraz podlegają resetowaniu ich do zera. W przypadku Alvisa odniesienie do czasu polega na przypisaniu do każdej instrukcji z kodu danego agenta, niezerowej, całkowitej oraz skończonej wartości czasu, jaki jest potrzebny do wykonania danej instrukcji. Czas ten odnosi się do globalnego zegara i ma wpływ na pracę innych agentów umieszczonych w danym modelu. Ten wpływ uwypukla się dzięki priorytetom, które są przypisane odpowiednio do każdego z agentów. Priorytety brane są pod uwagę przez algorytm szeregujący, który zarządza całością i decyduje o tym, który agent może w danej chwili przejąć zasoby procesora i poprzez to wykonywać swoje instrukcje. Dodatkowo instrukcje same z siebie, poprzez ich egzekucję, mogą realnie wpływać na stan danego agenta, a co za tym idzie na proces egzekucji jego logiki. Przykładowo agent *WatchDog* podczas oczekiwania na sygnał *heartbeat* jest w trybie oczekiwania *waiting* i wszelkie jego instrukcje są wstrzymane.

W automatach czasowych czas trwania akcji, odpowiednika alvisowych instrukcji, nie jest zdefiniowany. Poprzez zegary określa się warunki wykonania danej akcji, ale nie czas jej trwania. Alvis wspiera również czasowe warunki wykonania danej instrukcji, chociażby poprzez instrukcje in i out z czasem, lub periodyczne pętle z zadanym okresem.

Dzięki zdefiniowaniu czasu trwania każdej instrukcji możliwe jest rozróżnienie ich czasowo poprzez porównanie czasu ich egzekucji. Podejście takie sprawia, że instrukcje, o których wiadomo, że zabierają więcej cyklów procesora, mogą być modelowane z większym czasem trwania w odróżnieniu od tych, które potrzebują mniej tego czasu. Dodatkowym atutem jest to, że w ten sposób widoczna jest ich rozpiętość czasowa, poprzez co realnie wpływają na walidację danego systemu.

W automatach czasowych można zamodelować wymuszenie pozostania w danej lokacji przez odpowiednio wymagany czas i w ten sposób niejako wpłynąć na czas, za jaki wykona się dana akcja, lecz nie ma możliwości bezpośrednio przypisania czasu trwania danej akcji. Czas wykonania akcji można również zamodelować poprzez zdefiniowanie odpowiedniego warunku czasowego dla zegara lecz takie podejście również nie przypisuje akcji czasu jej trwania w sposób bezpośredni.

Dodatkowo w języku Alvis w wersji czasowej α^1_{FPPS} można przerwać wykonywanie pewnej instrukcji bez utraty naliczonego już czasu na jej egzekucję. Dzieje się tak w przypadku wywłaszczenia agenta poprzez algorytm szeregujący na rzecz innego agenta aktywnego. Wywłaszczony agent „pamięta", że wykonał pewną część instrukcji i spożytkował na to określony czas. Jeżeli w cyklu działania systemu uda mu się ponownie przejąć zasoby procesora, to zaczyna dokładnie od tego miejsca i pośrednio czasu, w którym skończył procesowanie w momencie wywłaszczenia. Może zdarzyć się też tak, że dana instrukcja miała swoje uwarunkowanie czasowe w stosunku do zegara globalnego i potrzeba jej egzekucji już dawno się przedawniła. Sytuacje takie najczęściej wynikają z błędnej architektury systemu i są uwypuklane podczas weryfikacji modelu na podstawie zapisów w warstwie systemowej. Chodzi tutaj głównie o informacje zawarte w listach kontekstowych agentów. W automatach czasowych nie sposób jest przerwać wykonywanie danej akcji, by później, po odpowiednio skończonym czasie, znów do niej powrócić. Można od nowa zacząć ją procesować z lokacji poprzedzającej tą akcję, ale nie z jakiejś konkretnej części tej akcji.

Język Alvis w wersji czasowej pozwala łatwo modelować zjawisko przeterminowania i definiować alternatywne zestawy akcji zależnie od tego, czy w zadanym czasie udało się zrealizować komunikację czy też nie. W tym celu stosowane są nieblokujące operacje wejścia/wyjścia. Ich działanie polega na tym, że przez określoną liczbę jednostek czasu agent czeka na komunikację ze strony drugiego agenta, po czym wykonuje akcje związane z faktem nawiązania komunikacji $success$ lub jej braku $fail$. Za pomocą automatów czasowych można zamodelować podobną sytuację, stosując warunki czasowe nałożone na zegary oraz oczekując na dany sygnał.

Kolejną cechą wspólną dla obu formalizmów jest możliwość etykietowania. W przypadku automatów

czasowy robione jest to dla przejść pomiędzy lokacjami, wskazując wykonywaną akcję, zegary i warunki czasowe dla nich. Dla systemu zamodelowanego za pomocą języka Alvis informację taką dostaje się w warstwie systemowej, gdzie widoczne są poszczególne stany modelu oraz wykonywane instrukcje i zależności czasowe.

7.2. Czasowe kolorowane sieci Petriego

Czasowe kolorowane sieci Petriego [30], [46] stanowią jedną z najpopularniejszych klas sieci Petriego. W porównaniu do zwykłych sieci kolorowanych wprowadzają aspekt czasowy w modelowaniu systemów współbieżnych. Ograniczenia czasowe rozszerzają definicje znaczników sieci, na zasadzie przypisania im, oprócz wartości odpowiedniego typu (*koloru*), tzw. *pieczątek czasowych*. To uwarunkowanie czasowe znacznika określa, kiedy dany znacznik może być dostępny dla przejść sieci, czyli innymi słowy uaktywnia dane przejście. Czas modelowany jest za pomocą zegara globalnego i jego wartość decyduje o dostępności danego znacznika.

Wykonanie aktywnego przejścia t usuwa odpowiednie znaczniki z miejsc wejściowych tego przejścia, a znaczniki dodawane do miejsc wyjściowych mają wartości pieczątek czasowych ustawiane na podstawie wartości zegara globalnego w momencie wykonania przejścia. Dodatkowo wartość ta może być modyfikowana z użyciem operatora $@+$, co pozwala na „przesunięcie" wartości pieczątki czasowej w przyszłość w porównaniu do wartości zegara. Użycie wyrażenia $@ + n$ oznacza, że dodany znacznik będzie dostępny dla przejść sieci dopiero po kolejnych n jednostkach czasu.

Czasowe kolorowane sieci Petriego dopuszczają dwa rodzaje typów miejsc ze względu na ich relację do czasu. Są to *typy czasowe*, dla miejsc, które nadają swym znacznikom pieczątki czasowe oraz *typy niecza-sowe*, które używane są dla tych miejsc sieci, w których nie stosujemy pieczątek czasowych – znaczniki są zawsze dostępne.

Koncepcja wielozbiorów

Pojęcie *wielozbiorów czasowych* jest kluczowym elementem definicji oraz opisu dynamiki czasowych kolorowanych sieci Petriego (CP-sieci). Wielozbiór czasowy, w kontekście sieci Petriego, różni się tym od zwykłego wielozbioru, że oprócz elementów danego typu, które w przypadku wielozbiorów mogą się powtarzać, zawierają również pieczątki czasowe definiowane dla każdego znacznika. W rezultacie możliwe jest określenie liczby znaczników, jak również czasu z jakim dane znaczniki w danym miejscu staną się dostępne w sieci w odniesieniu do wartości czasu zegara globalnego.

Załóżmy, że dany jest zbiór zawierający wszelkie możliwe rezultaty wykonania danej procedury w systemie. Dodatkowo, dla celów diagnostycznych, wymagane jest śledzenie rezultatu z wykonania danej pro-

cedury. Wartości możliwych rezultatów definiuje zbiór $Result = \{ok, nok, error\}$, czyli dana procedura może się zakończyć sukcesem (czyli zakładanym rezultatem), niepowodzeniem lub błędem spowodowanym potencjalnymi wyjątkami. Przyjmijmy, że rezultaty z wykonania procedury są zgłaszane na zasadzie zdarzeń w systemie. Zdarzenia te dla celów diagnostycznych powinny więc posiadać również swoje odniesienie do dziedziny czasu. W tym celu dla każdego zdarzenia zdefiniujmy skończoną, całkowitą pieczątkę czasową, mówiącą o tym, w którym momencie czasowym dane zdarzenie wystąpiło. Przykłady wielozbiorów czasowych nad zbiorem $Result$ mogą wyglądać następująco:

$Result^*_{t1} = \{(ok, 1), (ok, 3), (ok, 16), (nok, 2), (nok, 7), (error, 12), (error, 13)\}$
$Result^*_{t2} = \{(ok, 1), (ok, 5), (error, 14)\}$

możliwy jest również zapis algebraiczny powyższych wielozbiorów:

$Result^*_{t1} = 3 \cdot ok[1, 3, 16] + 2 \cdot nok[2, 7] + 2 \cdot error[12, 13]$
$Result^*_{t2} = 2 \cdot ok[1, 5] + error[14]$

Do podstawowych operacji na wielozbiorach, które są używane w czasowych kolorowanych sieciach Petriego, należą relacja mniejszości pomiędzy dwoma wielozbiorami oraz różnica dwóch wielozbiorów.

Relacja mniejszości stanowi, że jeżeli rozmiar danego wektora pieczątek czasowych jest mniejszy bądź równy w stosunku do rozmiaru porównywanego wektora oraz jeżeli wartości jego pieczątek są równe lub większe od pieczątek czasowych tegoż porównywanego wektora, to wektor ten uznajemy za mniejszy od porównywanego. Na tej zasadzie można powiedzieć, że $Result^*_{t2} \leq Result^*_{t1}$.

Przy odejmowaniu dwóch wielozbiorów czasowych stosuje się zasadę usuwania znaczników o najniższych pieczątkach czasowych (najstarszych). Wynikiem operacji odejmowania będzie wielozbiór:
$Result^*_{t1} - Result^*_{t2} = ok[16] + 2nok[2, 7] + error[13]$.

Rodzinę wszystkich możliwych wielozbiorów czasowych zbudowanych nad zbiorem X oznacza się symbolem X_{TMS}, zaś wielozbiorów nieczasowych X_{MS}.

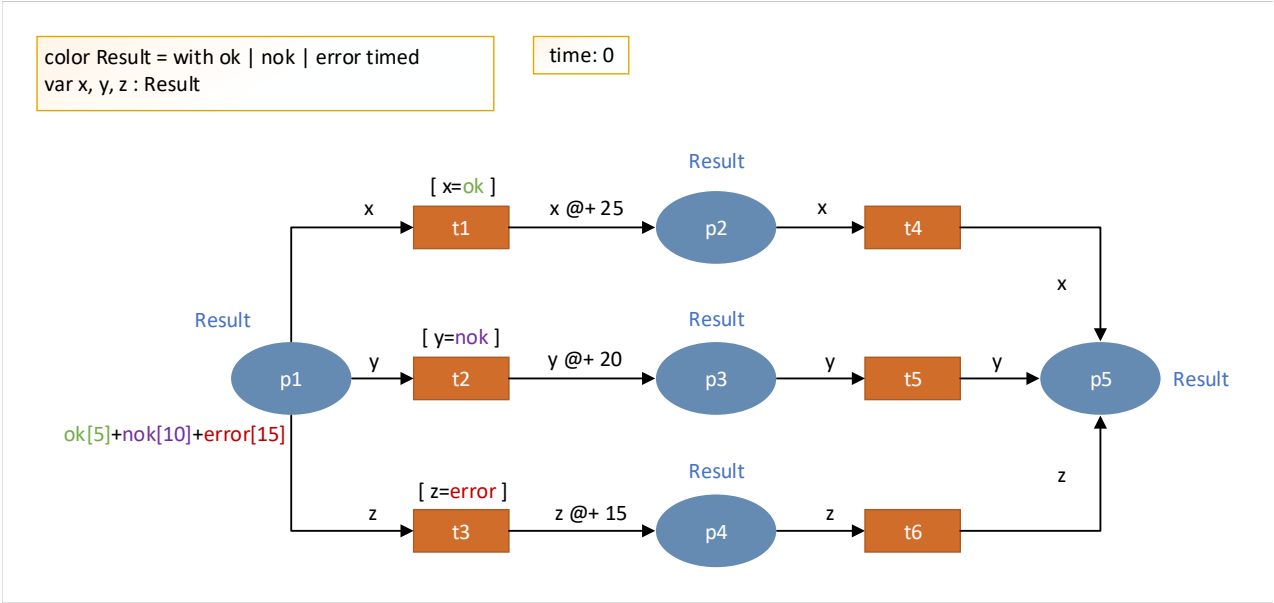
Definicja czasowej kolorowanej sieci Petriego

W celu zdefiniowania czasowych kolorowanych sieci Petriego istnieje konieczność przyjęcia pewnego języka inskrypcji do definiowania wyrażeń łuków, znakowań początkowych miejsc, zastrzeżeń przejść itd. Najpopularniejsze narzędzie do modelowania z wykorzystaniem CP-sieci, CPN Tools [30], stosuje w tym celu język funkcyjny CPN-ML. Dla potrzeb definicji sieci wystarczy przyjąć założenie, że taki język inskrypcji istnieje. Niech $EXPR$ oznacza zbiór wszystkich wyrażeń dostarczanych przez tej język. Przyjmujemy następujące oznaczenia:

- $Type[e]$ – typ wyrażenia $e \in EXPR$, czyli to, jak jest ono wartościowane na podstawie typów występujących w nim zmiennych;

- $Var[e]$ – zbiór zmiennych wolnych występujących w wyrażeniu e;
- $Type[v]$ – typ zmiennej v
- $EXPR_{V'}$ – zbiór wyrażeń $e \in EXPR$ takich, że $Var[e] \subseteq V'$

Podczas definiowania oraz analizy dynamiki czasowych kolorowanych sieci Petriego posłużono się przykładem zaprezentowanym na rysuku 7.5.



Rysunek 7.5: Przykład czasowej kolorowanej sieci Petriego *Diagnostic*

Powyższy przykład przedstawia wycinek czasowy, w którym wysyłane są raporty diagnostyczne z wykonania procedury. Zakłada się, że procedura może się zakończyć powodzeniem *ok*, niepowodzeniem *nok* lub błędem *error*. Raport diagnostyczny stanowi zestawienie możliwych wystąpień powyższych rezultatów w odcinku czasowym równym 30 jednostkom czasu. Oznacza to, że po tym okresie wysyłany jest raport diagnostyczny, zawierający informację o tym, że wystąpiły poszczególne rezultaty zdefiniowane zbiorem $Result = \{ok, nok, error\}$ lub nie. W przykładzie przyjęto sytuację, że raport kończy się dostarczeniem wszystkich rodzajów informacji, tj. ok, nok i $error$ w momencie czasowym równym 30 jednostkom czasu. W przykładzie zdefiniowano jeden obowiązujący typ (kolor):

`color Result = with ok | nok | error timed` – jest to typ czasowy określający możliwe rezultaty wykonania procedury;

Definicja 7.2. Uporządkowaną strukturę $\mathbf{N} = (P, T, A, \Sigma, V, C, G, E, I)$ nazywamy *niehierarchiczną czasową kolorowaną siecią Petriego* [30], gdzie:

- P jest skończonym zbiorem *miejsc*.

  W rozważanym przykładzie $P = \{p1, p2, p3, p4, p5\}$.
- T jest skończonym zbiorem *przejść* takim, że $P \cap T = \emptyset$.

  W rozważanym przykładzie $T = \{t1, t2, t3, t4, t5, t6\}$

- $A \subseteq P \times T \cup T \times P$ jest zbiorem *łuków skierowanych*.

  W rozważanym przykładzie

  $A = \{(p1, t1), (p1, t2), (p1, t3), (t1, p2), (t2, p3), (t3, p4), (p2, t4), (p3, t5), (p4, t6), (t4, p5), (t5, p5), (t6, p5)\}$.
- Σ jest niepustym, skończonym zbiorem *typów (kolorów)*, z których każdy jest zbiorem niepustym. Zbiór Σ może zawierać zarówno typy czasowe jak i nieczasowe.

  W rozważanym przykładzie $\Sigma = \{Result\}$.
- V jest skończonym zbiorem *zmiennych* takich, że dla dowolnej zmiennej $v \in V$, $Type[v] \in \Sigma$.

  W rozważanym przykładzie $V = \{x, y, z\}$.
- $C\colon P \rightarrow \Sigma$ jest *funkcją typów (kolorów)* określającą, jakiego typu znaczniki każde z miejsc może zawierać.

  W rozważanym przykładzie: $C(p1) = C(p2) = C(p3) = C(p4) = C(p5) = Result$.
- $G\colon T \rightarrow EXPR_V$ jest *funkcją zastrzeżeń (dozorów)* przypisującą każdemu przejściu t wyrażenie takie, że $Type[G(t)] = Bool$, gdzie $Bool$ jest typem logicznym.

  W rozważanym przykładzie występują następujące dozory:

  $G(t1) =$ `[x=ok]`

  $G(t2) =$ `[y=nok]`

  $G(t3) =$ `[z=error]`.

  Wyrażenie te wartościują się jako logicznie poprawne w zależności od dostępności odpowiednich typów znaczników, które są wymagane dla danego przejścia.
- $E\colon A \rightarrow EXPR_V$ jest *funkcją wag łuków* przypisującą każdemu łukowi a wyrażenie, takie że $Type[E(a)]$ jest wielozbiorem (odpowiednio czasowym lub nieczasowym) nad typem $C(p)$, gdzie p jest miejscem, z którym połączony jest łuk a.

  W rozważanym przykładzie wszystkie wyrażenia zawierają pojedyncze zmienne, więc ich wartością może być dowolna wartość należąca do typu *Result*. Biorąc jednak pod uwagę dozory tranzycji i znaczniki, które mogą się znaleźć w różnych miejscach sieci, za zmienną x zawsze będzie podstawiana wartość ok, za y – nok, a za z – $error$. Dodatkowo w przypadku łuków wychodzących tranzycji $t1$, $t2$ i $t3$, znaczniki będą wyposażone w pieczątką czasową, której wartość będzie przesunięta względem czasu wykonania tranzycji o wartość odpowiedniego wyrażenia czasowego. Dla łuków wyjściowych tranzycji $t4$, $t5$ i $t6$ pieczątka czasowa będzie równa czasowy wykonania tranzycji.
- $I\colon P \rightarrow EXPR_\emptyset$ jest *funkcją inicjalizującą*, która przypisuje każdemu miejscu niezawierające zmiennych wyrażenie takie, że $Type[I(p)]$ jest wielozbiorem (odpowiednio czasowym lub nieczasowym) nad typem $C(p)$.

  W rozważanym przykładzie $I(p1) = ok[5] + nok[10] + error[15]$, zaś w pozostałych przypadkach $I(p) = \emptyset$.

Analiza dynamiki czasowej kolorowanej sieci Petriego

Wraz z realizacją sieci ulegają zmianie znakowania jej poszczególnych miejsc. *Znakowanie sieci* N określa jakie znaczniki znajdują się w jej poszczególnych miejscach. Znakowanie M jest funkcją przypisującą każdemu z miejsc $p \in P$ wielozbiór $M(p) \in C(p)_{MS}$, jeżeli typ $C(p)$ jest typem nieczasowym i $M(p) \in C(p)_{TMS}$, jeżeli typ $C(p)$ jest typem czasowym.

Aktywność przejścia związana jest z dostępnością wymaganych znaczników w miejscach wejściowych tego przejścia. Liczbę i wartości wymaganych znaczników określają wagi łuków prowadzących od miejsc wejściowych do rozważanego przejścia. O dostępności znaczników decyduje aktualna wartość zegara globalnego – dostępne są znaczniki, dla których wartość pieczątki czasowej jest nie większa niż wartość zegara. Ponieważ wyrażenia przypisane łukom, jak również wyrażenia przypisane przejściom (dozory) mogą zawierać zmienne, to konieczne jest rozważanie aktywności przejść przy określonym wartościowaniu tych zmiennych.

Do pełnego opisu stanu sieci konieczne jest podanie bieżącej wartości zegara. Parę (M, t^*), gdzie M jest znakowaniem, zaś t^* wartością globalnego zegara nazywamy *znakowaniem czasowym* (lub krótko *stanem*). *Początkowym znakowaniem czasowym* nazywamy parę $(M_0, 0)$, gdzie znakowanie początkowe M_0 jest wynikiem ewaluacji wyrażeń inicjalizujących $I(p)$.

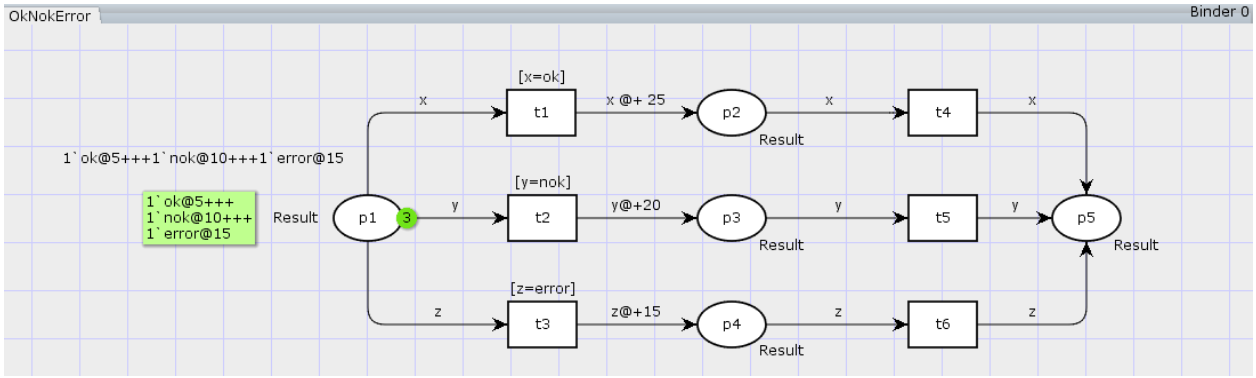
Niech $Var(t)$ oznacza zbiór zmiennych wolnych występujących w zastrzeżeniu przejścia t i wyrażeniach łuków wejściowych i wyjściowych przejścia t. *Wiązaniem* b nazywamy funkcję, która każdej zmiennej ze zbioru $Var(t)$ przypisuje wartość należącą do typu tej zmiennej i taką, że wynikiem wartościowania zastrzeżenia jest stała $True$. Wynik wartościowania wyrażenia $E(a)$, łuku a, przy wiązaniu b oznaczamy symbolem $E(a)\langle b\rangle$.

Przejście t jest *aktywne* przy wiązaniu b w stanie (M, t^*), jeżeli każde z jego miejsc wejściowych zawiera wymaganą liczbę znaczników o odpowiednich wartościach oraz, w przypadku miejsc czasowych, znaczniki te są dostępne. W wyniku jego wykonania otrzymujemy nowe znakowanie (M', t^*) takie, że

$$M(x) = \begin{cases} M(p) - E(p,t)\langle b\rangle & \text{gdy } p \in In(t) - Out(t); \\ M(p) + E(t,p)\langle b\rangle & \text{gdy } p \in Out(t) - In(t); \\ M(p) - E(p,t)\langle b\rangle + E(t,p)\langle b\rangle & \text{gdy } p \in In(t) \cap Out(t); \\ M(p) & \text{w pozostałych przypadkach} \end{cases}$$

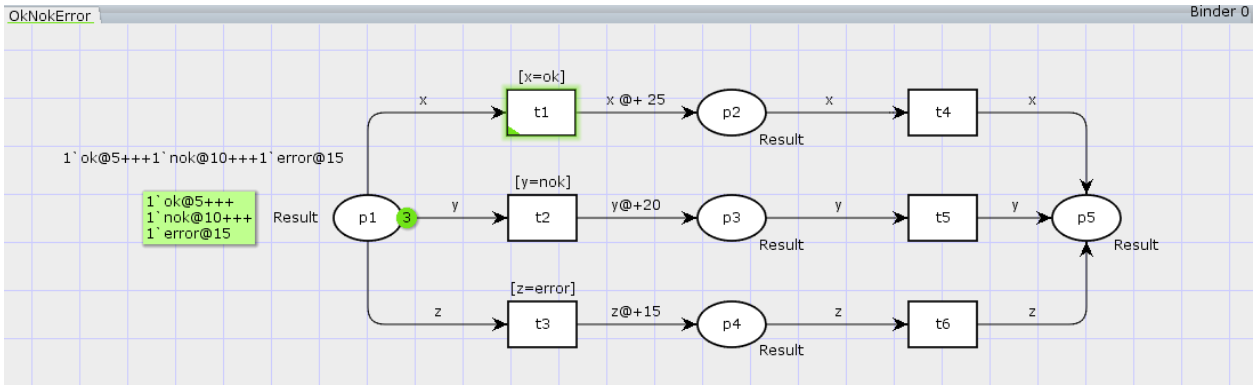
gdzie $In(t)$ i $Out(t)$ oznaczają odpowiednio zbiór miejść wejściowych i wyjściowych przejścia t.

Dla rozważanego przykładu z rysunku 7.5 stan początkowy można zapisać jako:
$(M_0, 0) = (ok[5] + nok[10] + error[15], \emptyset, \emptyset, \emptyset, \emptyset), 0)$.
W stanie tym nie jest aktywne żadne przejście wobec czego wszystkie znaczniki pozostają w miejscu $p1$. Krok z symulacji przykładu dla stanu $(M_0, 0)$ przedstawia rysunek 7.6.



Rysunek 7.6: Symulacja przykładu *Diagnostic* – stan $(M_0, 0)$

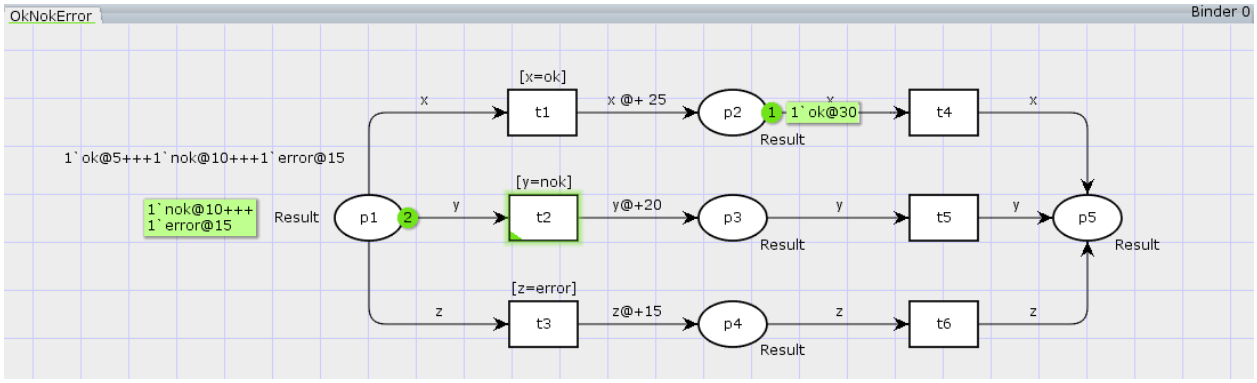
Po upływie 5 jednostek czasu, z uwagi na wartość pieczątki czasowej znacznika ok, aktywne staje się przejście $t1$. Możliwe są trzy wiązania: (ok/x), (nok/x) i $(error/x)$. Z uwagi na dozór przejścia $t1$ tylko wiązanie (ok/x) będzie się wartościowało jako logicznie poprawne. Obecny stan przyjmuje postać:
$(M_0, 5) = ((ok[5] + nok[10] + error[15], \emptyset, \emptyset, \emptyset, \emptyset), 5)$.
Krok z symulacji przykładu dla stanu $(M_0, 5)$ zaprezentowano na rysunku 7.7.



Rysunek 7.7: Symulacja przykładu *Diagnostic* – stan $(M_0, 5)$

Z uwagi na aktywność przejścia $t1$ znacznik ok usuwany jest z miejsca $p1$ i przenoszony jest do miejsca $p2$, gdzie obliczana jest jego nowa dostępność czasowa. Brana jest tu pod uwagę wartość zegara (5) oraz funkcja wagowa dla łuku $E(t1, p2)$. Zawarty w niej operator @+ po swojej prawej stronie przewiduje dodanie, czyli odczekanie dodatkowych 25 jednostek czasu. W rezultacie znacznik ok będzie dostępny w momencie czasowym równym 30 jednostkom czasu.

Po kolejnych 5 jednostkach czasu aktywne staje się także przejście $t2$. Możliwe są trzy wiązania: (ok/y), (nok/y) i $(error/y)$. Biorąc pod uwagę dozór przejścia $t2$ tylko wiązanie (nok/y) będzie się wartościowało jako logicznie poprawne. Obecny stan przyjmuje postać:
$(M_1, 10) = ((nok[10] + error[15], ok[30], \emptyset, \emptyset, \emptyset), 10)$.
Krok z symulacji przykładu dla stanu $(M_1, 10)$ prezentuje rysunek 7.8.



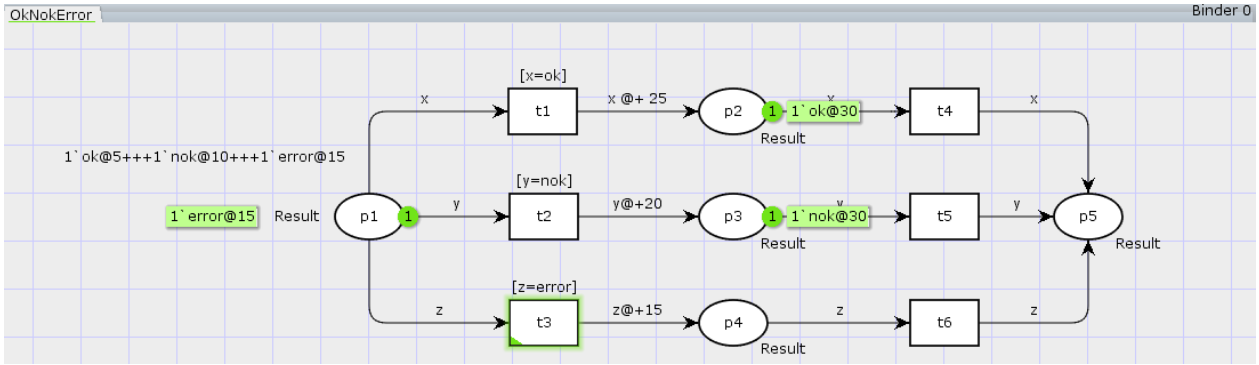
Rysunek 7.8: Symulacja przykładu *Diagnostic* – stan $(M_1, 10)$

Ponieważ aktywne jest przejście $t2$, znacznik nok usuwany jest z miejsca $p1$ i przenoszony jest do miejsca $p3$, gdzie również obliczana jest jego nowa dostępność czasowa. Brana jest tu pod uwagę wartość zegara (10) oraz funkcja wagowa dla łuku $E(t2, p3)$. Podobnie jak w przypadku znacznika ok, operator $@+$ po swojej prawej stronie dodaje dodatkowe jednostki czasu – w tym wypadku 20. W rezultacie znacznik nok będzie dostępny w momencie, gdy zegar globalny osiągnie wartość równą 30 jednostkom czasu.

Po kolejnych 5 jednostkach czasu aktywne staje się także przejście $t3$. Podobnie jak poprzednio możliwe są trzy wiązania: (ok/z), (nok/z) i $(error/z)$. Biorąc pod uwagę dozór przejścia $t3$, tylko wiązanie $(error/z)$ będzie się wartościowało jako logicznie poprawne. Obecny stan modelu zostanie określony następująco:

$(M_2, 15) = ((error[15], ok[30], nok[30], \emptyset, \emptyset), 15)$.

Krok z symulacji przykładu odnoszący się do stanu $(M_2, 15)$ prezentuje rysunek 7.9.



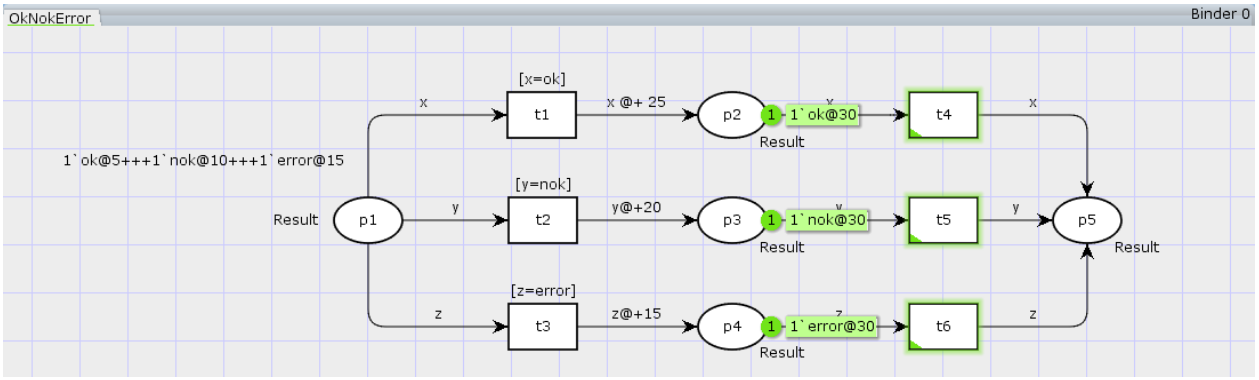
Rysunek 7.9: Symulacja przykładu *Diagnostic* – stan $(M_2, 15)$

Kolejny krok to przede wszystkim usunięcie znacznika $error$ z miejsca $p1$ i przeniesienie go do miejsca $p4$. W ten sam sposób, jak to miało miejsce dla znaczników ok oraz nok, operator $@+$ po swojej prawej stronie dodaje dodatkowe jednostki czasu dla pieczątki czasowej znacznika $error$. W tym wypadku będzie to dodatkowe 15 jednostek czasu. W rezultacie znacznik $error$ będzie dostępny w momencie gdy zegar globalny osiągnie wartość równą 30 jednostkom czasu.

W momencie, gdy zegar globalny osiągnie wartość równą 30 jednostkom czasu, aktywne będą przejścia $t4$, $t5$ oraz $t6$. Stan modelu określony jest wówczas następująco:

$(M_3, 30) = ((\emptyset, ok[30], nok[30], error[30], \emptyset), 30)$.

Krok z symulacji przykładu dla stanu $(M_3, 30)$ przedstawia rysunek 7.10.

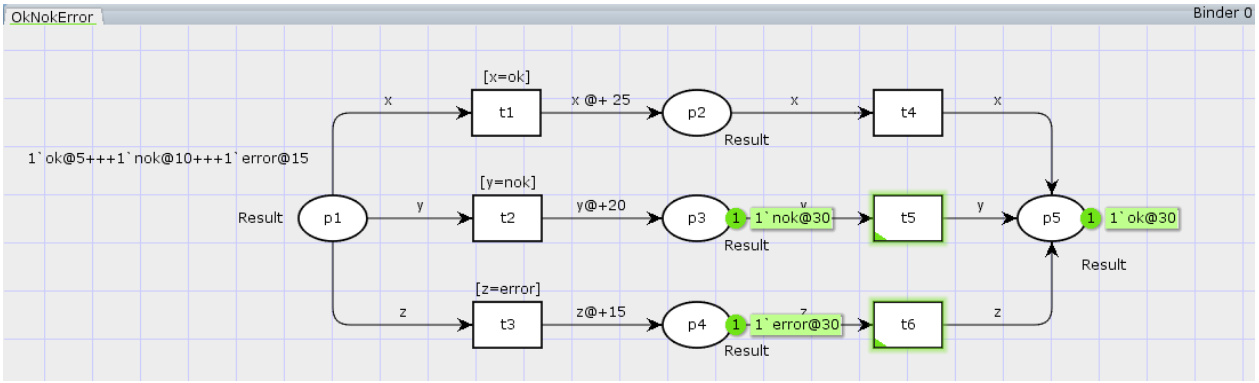


Rysunek 7.10: Symulacja przykładu *Diagnostic* – stan $(M_3, 30)$

Z uwagi na aktywność przejścia $t4$ znacznik ok usuwany jest z miejsca $p2$ i przenoszony jest do miejsca $p5$. Stan modelu określony jest odpowiednio:

$(M_4, 30) = ((\emptyset, \emptyset, nok[30], error[30], ok[30]), 30)$.

Symulację tego kroku prezentuje rysunek 7.11.



Rysunek 7.11: Symulacja przykładu *Diagnostic* – stan $(M_4, 30)$

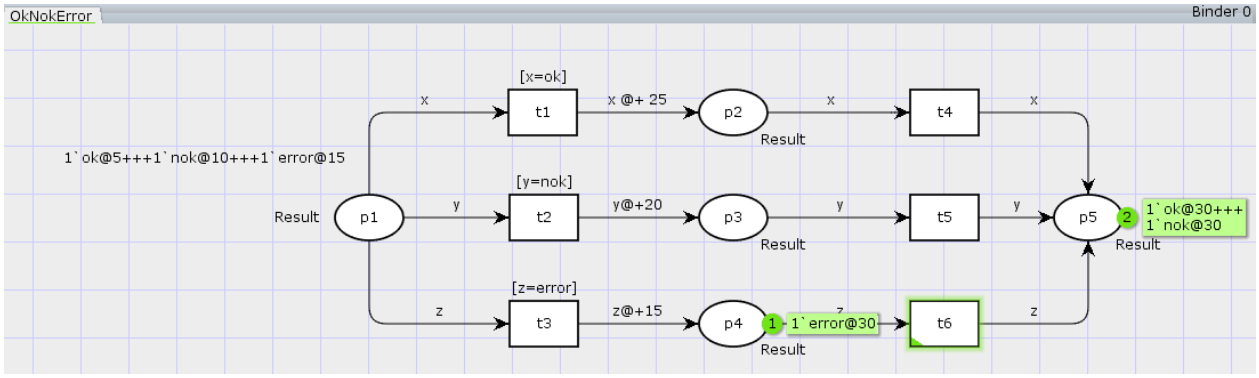
W podobny sposób jak powyżej aktywność przejścia $t5$ sprawi, że znacznik nok zostanie usunięty z miejsca $p3$ i przeniesiony do miejsca $p5$. W wyniku tego stan modelu zostanie zmieniony na następujący:

$(M_5, 30) = ((\emptyset, \emptyset, \emptyset, error[30], ok[30] + nok[30]), 30)$.

Symulację tego kroku prezentuje rysunek 7.12.

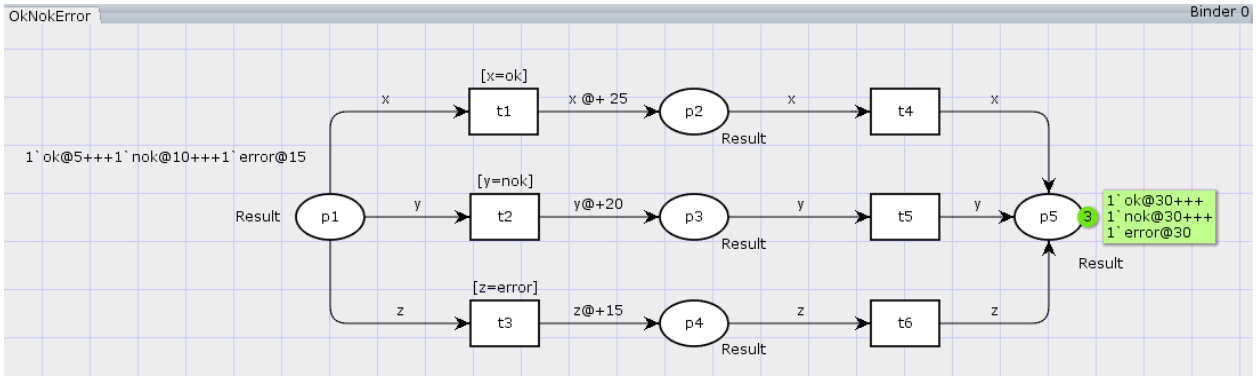
Analogicznie aktywność przejścia $t6$ sprawi, że znacznik $error$ zostanie usunięty z miejsca $p4$ i przeniesiony do miejsca $p5$. Wykonanie przejścia $t6$ sprawi, że model osiągnie stan martwy, z którego nie będzie już możliwe żadne przejście. Stan ten opiszemy w następujący sposób:

$(M_6, 30) = ((\emptyset, \emptyset, \emptyset, \emptyset, ok[30] + nok[30] + error[30]), 30)$.



Rysunek 7.12: Symulacja przykładu *Diagnostic* – stan $(M_5, 30)$

Symulację tego kroku prezentuje rysunek 7.13.



Rysunek 7.13: Symulacja przykładu *Diagnostic* – stan $(M_6, 30)$

W analizie dynamiki czasowych kolorowanych sieci Petriego ważnym elementem jest zjawisko zmian związanych z upływem czasu. Stan modelu w sieciach nieczasowych zmienia się poprzez aktywowanie danych przejść, co z kolei powoduje przemieszczanie się znaczników pomiędzy miejscami sieci. W sieciach czasowych zmiana stanu danego modelu zależy również od upływu czasu. Znaczniki ze swoimi pieczątkami czasowymi oczekują w miejscach wejściowych danego przejścia na odpowiednią wartość zegara globalnego, która umożliwiłaby ich pobranie.

Porównanie modelowania w języku Alvis z modelowaniem za pomocą czasowych kolorowanych sieci Petriego

Modelowanie systemów czasu rzeczywistego za pomocą języka Alvis znajduje wiele wspólnych analogii, jak też i różnic w porównaniu z czasową wersją kolorowanych sieci Petriego. Cechą niewątpliwie wspólną dla obu formalizmów jest pojęcie wzorca czasu, czyli zegara do którego odnoszą się wszystkie zależności czasowe danego modelu. W przypadku *TCPN* przyjmuje się *zegar globalny*, gdzie wartość odmierzanego czasu decyduje o dostępności znaczników w poszczególnych miejscach. W Alvisie czas wyko-

nywanych instrukcji i co za tym idzie tranzycji pomiędzy stanami modelu odnosi się do *zegara systemowego*, który jest odniesieniem dla agentów zdefiniowanych w danym modelu oraz dla algorytmu szeregującego, który zarządza dostępnością zasobów procesora.

Pojęcie przejścia w czasowych kolorowanych sieciach Petriego jest przeniesione do języka modelowania Alvis pod postacią tranzycji, która dokonuje się wraz z wykonaniem poszczególnej instrukcji danego agenta. Aspekt czasowy w języku Alvis jest usytuowany w założeniu, że tranzycja posiada określoną liczbę jednostek czasowych przeznaczonych na jej wykonanie. Inaczej te zależności czasowe rozumiane są w *TCPN*, gdzie opóźnienia czasowe są rezultatem dostępności znaczników danego miejsca. W Alvisie dostępne są również specjalne instrukcje czasowe, które nie tylko poprzez czas ich wykonywania, ale również przez wzgląd na swoją funkcjonalność, wprowadzają opóźnienia czasowe w odniesieniu do zegara systemowego (np. instrukcja *delay*).

Wspólną cechą obu formalizmów jest możliwość definiowania wyrażeń logicznych, których wynik wartościowania decyduje o dostępności danej procedury, pętli, alternatywy w języku Alvis lub dozoru przejścia w przypadku czasowych kolorowanych sieci Petriego. Dodatkowo w Alvisie możliwa jest komunikacja nieblokująca w zależności od spełnienia warunku czasowego. Dotyczy to sytuacji, w której przykładowy agent oczekuje przez skończony okres czasu na podjęcie komunikacji przez innego agenta.

Cechą wyróżniającą Alvisa w wersji czasowej α^1_{FPPS}, jest możliwość przerwania wykonywania danej instrukcji i ponowne wznowienie po odzyskaniu dostępu do zasobów procesora przez agenta, który daną instrukcję zawiera. Sytuacje takie mają miejsce w momencie wywłaszczania jednego agenta na rzecz drugiego. Wersja czasowa Alvisa α^1_{FPPS} wprowadza konieczność szeregowania wykonywanych zadań, czym różni się od *TCPN*, gdzie można rozważać równoległe wykonanie wielu przejść sieci. W odróżnieniu od tego podejścia algorytm szeregujący α^1_{FPPS} dopuszcza działanie tylko jednego agenta w danym momencie czasowym. Wszelkie konflikty pomiędzy agentami są rozstrzygane na etapie procedowania tego algorytmu.

Oba formalizmy dopuszczają możliwość wykonywania poszczególnych zadań w nieskończonych pętlach. Alvis dodatkowo wprowadza pętle, których zawartość procesowana jest periodycznie w pewnych zdefiniowanych skończonych okresach czasu. Język modelowania Alvis daje również możliwość uruchomienia kolejnego procesu (agenta) z procesu już uruchomionego. Jest to cecha dość powszechnie używana w systemach czasu rzeczywistego. Dodatkowo w odróżnieniu od *TCPN* istnieje możliwość skoku z bieżącej instrukcji do miejsca określonego etykietą.

Podobnie jak czasowe kolorowane sieci Petriego język modelowania Alvis uwzględnia zmiany związane z upływem czasu. Odbywa się to w sytuacjach, kiedy to stan modelu zmienia się wraz z upływem czasu. Niezależnie od tego czy agent posiada kontrolę nad zasobami w danym momencie czasowym, czy też znajduje się w stanie gotowości, bądź też oczekiwania, uaktualniana jest jego lista kontekstowa, która zawiera wpisy dotyczące aspektów czasowych (*timer*).

Dodatkową cechą odróżniającą Alvisa w wersji czasowej α^1_{FPPS} od czasowych kolorowanych sieci Petriego jest cykliczne występowanie przerwań systemowych. Jest to uwarunkowane definicją algorytmu szeregującego. Przerwania te dają możliwość przełączenia agentów nad zasobami procesora, co sprawia, że zasoby te nie są blokowane przez jednego agenta przez nieskończony okres czasu. Język modelowania Alvis wprowadza również pojęcie sekcji krytycznych, dla których warunki zewnętrzne, w tym również czasowe, stają się nieistotne, a agent wykonujący taką sekcję przestaje być argumentem dla algorytmu szeregującego, który mógłby go pozbawić dostępu do zasobów procesora.

Podstawą weryfikacji modelu w języku Alvis jest etykietowana przestrzeń stanów LTS, która jest odpowiednikiem grafów osiągalności z kolorowanych sieci Petriego.

Alvis wprowadza ponadto *listę kontekstową*, która poprzez wpisy w niej zawarte daje możliwość podglądu, co rzeczywiście w danej chwili dzieje się z rozważanym agentem, czyli mamy pełen podgląd na to w jakim znajduje się trybie, którą instrukcję wykonuje, jaką komunikację proceduje z innymi agentami, jak procesują się jego instrukcje czasowe oraz jakie są bieżące wartości jego parametrów.

7.3. Podsumowanie

Język modelowania systemów współbieżnych Alvis nie odstaje koncepcyjnie od innych formalizmów zdefiniowanych do tego celu. Dodatkowo definiuje wiele rozwiązań, które są wynikiem praktycznego doświadczenia jego autorów.

Koncentrując się na domenie czasu, Alvis w wersji czasowej α^1_{FPPS} wprowadza pojęcia oraz podejścia zaczerpnięte z praktyk stosowanych do modelowania i analizy systemów wbudowanych czasu rzeczywistego. Całość stanowi narzędzie przyjazne i warsztatowo zrozumiałe dla inżyniera zajmującego się systemem od strony jego wymagań, analizy, implementacji oraz weryfikacji.

Niewątpliwie mocną stroną modelu stworzonego w Alvise jest jego trójwarstwowość. Dzięki tej właściwości odseparowuje się warstwę jego definicji, czyli kodu i komunikacji, od warstwy systemowej, która stwarza możliwość weryfikacji danego modelu.

Z założenia język modelowania Alvis nie zawęża się do jednej dozwolonej wersji czasowej, a zamiast tego umożliwia definiowanie innych, potrzebnych i odpowiadających wymaganiom klienta wersji. Wersje te uwzględniają ilość dostępnych procesorów na danej platformie sprzętowej oraz szczegóły związane z wymaganiami dotyczącymi algorytmu szeregowania. Z tego punktu widzenia Alvis został opracowany jako środowisko otwarte pod względem adaptacji możliwych rozwiązań, które odpowiadają modelowanym systemom czasu rzeczywistego.

8. Podsumowanie

Historia języka Alvis sięga roku 2009, gdy rozpoczęto prace nad koncepcją formalnego języka modelowania „przyjaznego dla inżynierów". Zespół pracujący nad formułą nowego języka składał się z osób, które miały spore doświadczenie w uczeniu i stosowaniu metod formalnych takich jak sieci Petriego, czy algebry procesów. Jako efekt tych doświadczeń rozwijano język i narzędzia do modelowania systemów współbieżnych (z czasem lub bez). Stąd też domyślna wersja warstwy systemowej języka zakłada, że każdy agent aktywny ma dostęp do własnego procesora, a obliczenia realizowane są współbieżnie. Również pośrednia reprezentacja IHR generowana przez opracowany i zaimplementowany kompilator języka Alvis stosuje założenie o współbieżności obliczeń – w rzeczywistości kod generowany przez kompilator nie używa nawet trybu *ready*. Efektem tych prac jest język modelowania, który może rywalizować z sieciami Petriego, automatami czasowymi, czy też algebrami procesów. Żaden z tych formalizmów nie oferuje jednak narzędzi, które bezpośrednio wspierałyby modelowanie systemów jednoprocesorowych, w których współbieżne z natury procesy muszą rywalizować ze sobą o dostęp do jednego procesora.

Celem niniejszej rozprawy było opracowanie i implementacja warstwy systemowej, która umożliwiłaby bezpośrednie wykorzystanie języka do modelowania systemów jednoprocesorowych. Dodatkowo postawiono wymagania, aby opracowane koncepcje i implementacja wpisywały się w już istniejące rozwiązania. Oznaczało to m.in. ograniczenie do minimum ingerencji w istniejący kompilator języka oraz wykorzystanie uniwersalnej reprezentacji IHR, zaimplementowanej w oryginale dla systemów wieloprocesorowych (o nieograniczonej liczbie procesorów). Ponieważ z założenia narzędzia w pakiecie Alvis Toolkit współpracują z zewnętrznymi pakietami do weryfikacji modelowej (np. nuXmv, CADP), celem pośrednim pracy było opracowanie i implementacja algorytmu generującego reprezentację przestrzeni stanów modelu w postaci etykietowanego systemu przejść (LTS).

Zadanie zrealizowano poprzez opracowanie i implementację warstwy systemowej określonej jako α^1_{FPPS}, która stanowi rozszerzenie języka modelowania Alvis dla systemów jednoprocesorowych.

Do najważniejszych wyników niniejszej rozprawy należy zaliczyć:

1. Opracowanie koncepcji warstwy systemowej α^1_{FPPS}, w tym przedstawienie głównych założeń algorytmu szeregującego, sposobu kolejkowania i wywłaszczania agentów przy stałym okresie zgłaszania przerwania systemowego.

2. Definicję i implementację dwuwymiarowej kolejki FIFO do szeregowania agentów i zarządzania nimi w przypadku promocji jednego z nich do trybu *running*.
3. Opracowanie i implementację metody ustalania stanu początkowego dla modeli z warstwą systemową α^1_{FPPS}.
4. Dopasowanie warunków aktywności i wykonania tranzycji do modeli z warstwą systemową α^1_{FPPS} oraz implementację rozszerzeń funkcji *enable* i *fire* na potrzeby modeli z warstwą systemową α^1_{FPPS}.
5. Wprowadzenie do zestawu tranzycji nowej tranzycji systemowej $SysTick$, opisanie zasad jej funkcjonowania i implementację w języku Haskell.
6. Dopasowanie funkcjonowania tranzycji systemowej STTime do potrzeb warstwy systemowej α^1_{FPPS}.
7. Adaptację algorytmu generowania grafu LTS do potrzeb modeli z warstwą systemową α^1_{FPPS}.
8. Opracowanie dwóch modeli systemów czasu rzeczywistego w języku Alvis, które wykorzystano do zilustrowania typowych sytuacji, które występują w grafach LTS dla modeli z warstwą systemową α^1_{FPPS}.

Podsumowując, należy uznać, że realizacja powyższych zadań czyni zadość ww wymaganiom, jak również tezom pracy.

Przygotowana rozprawa na pewno nie wyczerpuje tematów związanych z rozwojem języka Alvis, jako środowiska służącego do formalnej analizy i weryfikacji systemów współbieżnych. Z punktu widzenia samego modelu zaobserwowano możliwości rozwoju, dotyczące następujących zagadnień:

- Zaimplementowane warstwy systemowe wielo- i jednoprocesorowa nie wyczerpują wszystkich możliwości. Po pierwsze można rozważać inne algorytmy szeregowania dla warstwy jednoprocesorowej. Po drugie można wprowadzić do Alvisa definicje innych warstw systemowych zawierających ustaloną, ale skończoną liczbę procesorów.
- Niezależnie od stosowanej warstwy systemowej modele w języku Alvis często wykorzystują te same konstrukcje. Ciekawym i praktycznym rozwiązaniem mogłoby być opracowanie pewnych *wzorców projektowych* (ang. *design pattern*), które stwarzałyby i opisywały reguły tworzenia poszczególnych rozwiązań. Przykładem może być tutaj modelowanie kolejki, szablonu producenta i konsumentów, fabryki, budowniczego, obserwatora, dekoratora, fasady, mostu, różnego rodzaju rozwiązań dotyczących współdzielenia zasobów, komunikacji i wielu innych, często spotykanych w rozwiązaniach implementacyjnych.
- Systemy informatyczne na ogół zanurzone są w pewnych dedykowanych im środowiskach. Dla modelu stworzonego w Alvisie stanowią one świat, jaki widziany jest na zewnątrz poza modelem. Stworzenie definicji formalnej takiego środowiska, przyczyniłoby się w sposób praktyczny do uwzględnienia wszelakich aspektów związanych z otoczeniem wewnątrz modelu. Wyeliminowałoby również konieczność bezpośredniego modelowania otoczenia jako części rozwijanego modelu.

A. Definicje warstw kodu dla przykładów opisanych w studium przypadków

A.1. Publikator i subskrybent

Listing A.1: Definicja warstwy kodu dla przykładu *publikator i subskrybent*

```
agent Publisher (0) {
    increment :: Int = 1;
    decrement :: Int = -1;
    status :: String = "";
    dataMessagePub :: Int = 0;
    token :: Char = ' ';

    loop (every 100) {                                          -- 1
        in applyToken token;                                    -- 2
        in reqStatus status;                                    -- 3
        select {                                                -- 4
            alt (token == 'T' && status == "Lower") {
                dataMessagePub = increment;                     -- 5
            }
            alt (token == 'T' && status == "Bigger") {
                dataMessagePub = decrement;                     -- 6
            }
        }
        out pubData dataMessagePub;                             -- 7
        null;                                                   -- 8
    }
}

agent Subscriber (1) {
    buffer :: Int = 0;
```

```
    dataMessageSub :: Int = 0;

    loop {                                                    -- 1
        in subData dataMessageSub;                            -- 2
        buffer = buffer + dataMessageSub;                     -- 3
        out status buffer;                                    -- 4
    }
}

agent Status {
    bufferValue :: Int = 0;
    statusMessage :: String = "";

    proc recStatus {
        in recStatus bufferValue;                             -- 1
        exit;                                                 -- 2
    }

    proc sendStatus {
        select {                                              -- 3
            alt (bufferValue < 1) {
                statusMessage = "Lower";                      -- 4
            }
            alt (bufferValue >= 1) {
                statusMessage = "Bigger";                     -- 5
            }
        }
        out sendStatus statusMessage;                         -- 6
        exit;                                                 -- 7
    }
}



agent TokenHolder {
    token :: Char = 'T';

    proc sendToken {
        out sendToken token;                                  -- 1
```

```
        exit;                                                    -- 2
    }
}
```

A.2. Obserwator

Listing A.2: Definicja warstwy kodu dla przykładu *obserwator*

```
agent Object (0) {
  stateA :: Char = 'A';
  stateB :: Char = 'B';
  stateC :: Char = 'C';

  state_a:
  out (20) sendState stateA {                    -- 1
    success {
      state_b:
      out (20) sendState stateB {                -- 2
        success {
          state_c:
          out (20) sendState stateC {            -- 3
            success {
                exit;                            -- 4
            }
            fail { jump state_c; }               -- 5
          }
        }
        fail { jump state_b; }                   -- 6
      }
    }
    fail { jump state_a; }                       -- 7
  }
}

agent Observer (0) {
  state :: Char = ' ';
  counter :: Int = 0;
```

```
  loop {                                    -- 1
    in (20) getState state {                -- 2
      success {
        counter = counter + 1;              -- 3
        out sendToStore state;              -- 4
      }
    }
    select {                                -- 5
      alt (counter == 3) {
        start ReceiverA;                    -- 6
        start ReceiverB;                    -- 7
        start ReceiverC;                    -- 8
        exit;                               -- 9
      }
    }
  }
}

agent Storage {
  storedStateA :: Char = ' ';
  storedStateB :: Char = ' ';
  storedStateC :: Char = ' ';
  state :: Char = '';

  proc storeState {
    in storeState state;                                    -- 1
    select {                                                -- 2
      alt (state == 'A') { storedStateA = state; }          -- 3
      alt (state == 'B') { storedStateB = state; }          -- 4
      alt (state == 'C') { storedStateC = state; }          -- 5
    }
    exit;                                                   -- 6
  }

  proc queryStateA {
    out queryStateA storedStateA;                           -- 7
    exit;                                                   -- 8
  }
```

```
  proc queryStateB {
    out queryStateB storedStateB;                    -- 9
    exit;                                            -- 10
  }

  proc queryStateC {
    out queryStateC storedStateC;                    -- 11
    exit;                                            -- 12
  }
}

agent ReceiverA (1) {
  receivedState :: Char = ' ';

  loop {                                             -- 1
    in takeState receivedState;                      -- 2
    select {                                         -- 3
      alt (receivedState == 'A') { exit; }           -- 4
    }
  }
}

agent ReceiverB (1) {
  receivedState :: Char = ' ';

  loop {                                             -- 1
    in takeState receivedState;                      -- 2
    select {                                         -- 3
      alt (receivedState == 'B') { exit; }           -- 4
    }
  }
}

agent ReceiverC (1) {
  receivedState :: Char = ' ';

  loop {                                             -- 1
    in takeState receivedState;                      -- 2
    select {                                         -- 3
```

```
      alt (receivedState == 'C') { exit; }    -- 4
    }
  }
}
```

Bibliografia

[1] L. Aceto, A. Ingófsdóttir, K.G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, Cambridge, UK, 2007.

[2] R. Alur and D. Dill. Automata for modelling real-time systems. In *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *LNCS*, pages 322–335. Springer-Verlag, 1994.

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, London, UK, 2008.

[5] K. Balicki and M. Szpyrka. Formal definition of XCCS modelling language. *Fundamenta Informaticae*, 93(1–3):1–15, 2009.

[6] J. Baniewicz and M. Szpyrka. Detekcja zakleszczenia w systemie ABS z zastosowaniem języka Alvis. In L. Trybus and S. Samolej, editors, *Projektowanie, analiza i implementacja systemów czasu rzeczywistego*, volume 300 of *N/A*, chapter 6, pages 77–86. Wydawnictwo Komunikacji i Łączności, 2011.

[7] J. Baniewicz and M. Szpyrka. Priority inversion detection in Alvis model. In *Proc. of Mixdes 2011, the 18th International Conference Mixed Design of Integrated Circuits and Systems*, page 162, Gliwice, Poland, June 16–18 2011.

[8] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. *Lecture Notes on Concurrency and Petri Nets*, 3098, 2004.

[9] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. *Lecture Notes on Concurrency and Petri Nets*, 3098, 2004.

[10] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[11] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science, Upper Saddle River, NJ, USA, 2001.

[12] A. Biernacka, J. Biernacki, and M. Szpyrka. State-based verification of RTCP-nets with nuXmv. In *International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2015)*, volume 1702 of *AIP Conference Proceedings*, pages 100010–1–100010–4, Athens, Greece, March 20-23 2015. AIP Publishing.

[13] J. Biernacki, A. Biernacka, and M. Szpyrka. Action-based verification of RTCP-nets with CADP. In *International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2015)*, volume 1702 of *AIP Conference Proceedings*, pages 100011–1–100011–4, Athens, Greece, March 20-23 2015. AIP Publishing.

[14] Ch.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[15] A. Burns and A. Wellings. *Concurrent and real-time programming in Ada 2005*. Cambridge University Press, 2007.

[16] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[17] A. M. K. Cheng. *Real-time Systems. Scheduling, Analysis, and Verification*. Wiley Interscience, New Jersey, 2002.

[18] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[19] E. A. Emerson. Model checking and the Mu-calculus. In N. Immerman and P. G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1997.

[20] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Syst.*, 4(4):331–352, December 1992.

[21] Ada Europe. *Ada Reference Manual ISO/IEC 8652:2007(E), Ed. 3*. N/A, N/A, 2007.

[22] C. Fencott. *Formal Methods for Concurrency*. International Thomson Computer Press, Boston, MA, USA, 1995.

[23] E. Gansner, E. Koutsofios, and S. North. *Drawing graphs with* dot, 2006.

[24] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Computer Aided Verification (CAV'2007)*, volume 4590 of *LNCS*, pages 158–163, Berlin, Germany, 2007. Springer.

[25] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):89–107, 2013.

[26] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proc. of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.

[27] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[28] G. Hutton. *Programming in Haskell*. Cambridge University Press, Cambridge, United Kingdom, 2016.

[29] Z. Huzar. *LOTOS – język formalnych specyfikacji systemów informatycznych*. Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 2007.

[30] K. Jensen and L.M. Kristensen. *Coloured Petri Nets. Modelling and Validation of Concurrent Systems*. Springer, Berlin Heidelberg, 2009.

[31] M. Lipovaca. *Learn You a Haskell for Great Good*. Random House Lcc Us, Random House Lcc Us, No Starch Pres, N/A, 2018.

[32] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free μ-calculus. Technical Report 3899, INRIA, 2000.

[33] P. Matyasik. *Modelowanie i analiza systemów wbudowanych z zastosowaniem algebry procesów XCCS*. PhD thesis, Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki AGH, Kraków, 2009. (promotor: Marcin Szpyrka).

[34] P. Matyasik, M. Szpyrka, M. Wypych, and J. Biernacki. Communication between agents in Alvis language. In *Proc. of Mixdes 2016, the 23nd International Conference Mixed Design of Integrated Circuits and Systems*, Łódź, Poland, June 23–25 2016.

[35] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[36] R. Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[37] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[38] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Sebastopol, CA, USA, 2008.

[39] D.L. Parnas. Really rethinking 'formal methods'. *Computer*, 43(1):28–34, January 2010.

[40] C. A. Petri. Communication with automata. Technical report, New York, 1965. English translation of *Kommunikation mit Automaten*, PhD Dissertation, University of Bonn, 1962.

[41] I. Sommerville. *Software Engineering*. Addison-Wesley, 2011.

[42] T. Szmuc and G. Motet. *Specyfikacja i projektowanie oprogramowania systemów czasu rzeczywistego*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne AGH, Kraków, 2000.

[43] T. Szmuc and M. Szpyrka. Metody formalne w inżynierii oprogramowania systemów czasu rzeczywistego. WNT, Warszawa, 2010.

[44] T. Szmuc and M. Szpyrka. Formal methods – support or scientific decoration in software development. In *Proc. of Mixdes 2015, the 22nd International Conference Mixed Design of Integrated Circuits and Systems*, pages 24–31, Toruń, Poland, June 25–27 2015.

[45] M. Szpyrka. *Modelowanie i analiza systemów wbudowanych z zastosowaniem RTCP-sieci*, volume 165 of *Rozprawy Monografie*. AGH Uczelniane Wydawnictwa Naukowo-Dydaktyczne, Kraków, 2007.

[46] M. Szpyrka. *Sieci Petriego w modelowaniu i analizie systemów współbieżnych*. Wydawnictwa Naukowo-Techniczne, Warszawa, 2008.

[47] M. Szpyrka. *Modelowanie systemów współbieżnych w języku Alvis*. Wydawnictwa AGH, Kraków, 2013.

[48] M. Szpyrka, J. Biernacki, and A. Biernacka. Tools and methods for RTCP-nets modelling and verification. *Archives of Control Sciences*, 26(3):339–365, 2016.

[49] M. Szpyrka and P. Matyasik. Formal modelling and verification of concurrent systems with XCCS. In *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008)*, pages 454–458, Krakow, Poland, July 1-5 2008.

[50] M. Szpyrka, P. Matyasik, J. Biernacki, A. Biernacka, M. Wypych, and L. Kotulski. Hierarchical communication diagrams. *Computing and Informatics*, 35(1):55–83, 2016.

[51] M. Szpyrka, P. Matyasik, and R. Mrówka. Alvis – modelling language for concurrent systems. In P. Bouvry, H. Gonzalez-Velez, and J. Kołodziej, editors, *Intelligent Decision Systems in Large-Scale Distributed Environments*, volume 362 of *Studies in Computational Intelligence*, chapter 15, pages 315–341. Springer-Verlag, 2011.

[52] M. Szpyrka, P. Matyasik, R. Mrówka, and L. Kotulski. Formal description of Alvis language with α^0 system layer. *Fundamenta Informaticae*, 129(1-2):161–176, 2014.

[53] M. Szpyrka, P. Matyasik, and M. Wypych. Generation of labelled transition systems for Alvis models using Haskell model representation. In *Proceedings of the 22nd International Workshop on Concurrency, Specification and Programming (CS&P 2013)*, volume 1032, pages 409–420, Warsaw, Poland, 2013. CEUR Workshop Proceedings.

[54] M. Szpyrka, P. Matyasik, M. Wypych, J. Biernacki, and Ł. Podolski. *Alvis modelling language*. AGH University of Science and Technology, v. 0.13 edition, 2017.

[55] M. Szpyrka, M. Wypych, Ł. Podolski, P. Matyasik, and J. Biernacki. Alvis toolkit in a nutshell. In *submitted to Safeprocess 2018 Conference*, 2018.

[56] B. Woźna-Szcześniak and A. Półrola. Weryfikacja modelowa. In T. Szmuc and M. Szpyrka, editors, *Metody formalne w inżynierii oprogramowania systemów czasu rzeczywistego*, pages 311–373. WNT, Warszawa, 2010.

[57] B. Woźna-Szcześniak and M. Szpyrka. Automaty czasowe. In T. Szmuc and M. Szpyrka, editors, *Metody formalne w inżynierii oprogramowania systemów czasu rzeczywistego*, pages 114–128. WNT, Warszawa, 2010.

[58] B. Woźna-Szczęśniak and M. Szpyrka. Automaty czasowe. In T. Szmuc and M. Szpyrka, editors, *Metody formalne w inżynierii oprogramowania systemów czasu rzeczywistego*, volume N/A of *N/A*, chapter 3, pages 114–128. Wydawnictwa Naukowo-Techniczne, 2010.

[59] A. Zbrzezny. *Wybrane metody weryfikacji modelowej wykorzystujące testery SAT i SMT*. PhD thesis, Instytut Podstaw Informatyki PAN, Warszawa, 2017. (promotor: Bożena Woźna-Szcześniak).