



POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych
Instytut Telekomunikacji

PRACA DYPLOMOWA INŻYNIERSKA

Marcin Maciorowski

**Tworzenie narzędzi wspomagających projektowanie BPEL
w architekturze SOA**

Praca wykonana pod kierunkiem
dra inż. Andrzeja Ratkowskiego

.....
Ocena pracy

.....
Podpis Przewodniczącego Komisji

Warszawa, 2014

Życiorys

Urodziłem się 22 września 1988 roku w Radzynie Podlaskim. W 2004 roku rozpocząłem naukę w I Liceum Ogólnokształcącym w Radzynie Podlaskim, gdzie uczęszczałem do klasy o profilu matematyczno-fizyczno-informatycznym. Po uzyskaniu świadectwa dojrzałości w 2007 roku, rozpocząłem studia na Politechnice Warszawskiej na Wydziale Elektroniki i Technik Informatycznych. W trakcie studiów wybrałem specjalizację Systemy informacyjno-decyzyjne prowadzoną przez Instytut Automatyki i Informatyki Stosowanej.

Marcin Maciorowski

Streszczenie

BP EL.

Abstract

Implementation of (Font TNRoman 12 Normal).

This thesis includes the design and testing procedure of

Spis treści.

1. Wstęp.....	5
2. Cel pracy.....	6
3. Układ pracy	6
4. Wprowadzenie teoretyczne.	7
4.1. Język WSDL.	7
4.2. Język BPEL.....	7
4.2.1. Zmienna procesu BPEL (<i>Variable</i>).....	7
4.2.2. Instrukcja wywołania usługi (<i>Invoke</i>).	7
4.2.3. Instrukcja przepisania danych (<i>Assign</i>).....	7
4.3. EMF (Eclipse Modeling Framework).	7
4.3.1. TreeIterator.....	7
5. Wtyczka Eclipse BPEL Designer.	8
5.1. Interfejs użytkownika.....	8
5.1.1. Edytor.....	9
5.1.2. Widoki.....	10
5.2. Przykładowy proces BPEL.	14
6. Wtyczka generująca instrukcje kopiujące.	17
6.1. Transformacja procesu do postaci grafu.	19
6.2. Analiza grafu procesu.	24
6.3. Aktualizacja instrukcji kopiujących.....	27
6.4. Graficzny interfejs użytkownika.	28
6.5. Konfiguracja wtyczki (PDE).....	29
7. Testy funkcjonalne.	30
8. Podsumowanie.....	31
8.1. Napotkane problemy.....	31
8.2. Możliwości rozwoju.....	31
9. Bibliografia.....	32
10. Załączniki.	33
10.1. Płyta CD.....	33
10.2. Instrukcja instalacji wtyczki BPELag (BPEL assign generator).	33

1. Wstęp.

=====

We wstępie znajdzie się ogólne rozwinięcie streszczenia, czego praca dotyczy, z czym czytelnik się zetknie w kolejnych rozdziałach. Opisany układ dokumentu.

=====

Co to jest proces biznesowy, co to jest web service. WSDL. WS-BPEL opisuje w podejściu zorientowanym na usługi działanie procesów biznesowych w instytucji.

2. Cel pracy.

Celem pracy jest napisanie wtyczki do zintegrowanego środowiska programistycznego jakim jest Eclipse, która umożliwi automatyczne uzupełnienie zaprojektowanego procesu BPEL o instrukcje kopiujące dane. Wtyczka ma za zadanie dokonać analizy procesu oraz na podstawie wyników analizy wygenerować instrukcje kopiujące. Dalej umożliwiać użytkownikowi/projektantowi przegląd wyników analizy oraz dodatkowo manualną edycję wszystkich uczestniczących w procesie elementów przepisania danych (*Assign*), akceptację lub rezygnację z zapisu zmian do projektowanego procesu. Analizowany proces BPEL oraz uczestniczące w procesie usługi wykorzystują tę samą konwencję nazewnictwa.

3. Układ pracy

Opis układu pracy. Powstanie na końcu, gdy zostaną napisane już wszystkie rozdziały.

4. Wprowadzenie teoretyczne.

4.1. Język WSDL.

4.2. Język BPEL.

=====

Wstępny opis języka BPEL wprowadzający czytelnika w aspekty języka, których dotyczy niniejsza praca dyplomowa. Na pewno będą to: Struktura procesu, bloki assign, invoke.

=====

4.2.1. Zmienna procesu BPEL (*Variable*).

=====

Bla bla bla.

=====

4.2.2. Instrukcja wywołania usługi (*Invoke*).

=====

Bla bla bla.

=====

4.2.3. Instrukcja przepisania danych (*Assign*).

=====

Bla bla bla.

=====

4.3. EMF (Eclipse Modeling Framework).

4.3.1. TreeIterator

5. Wtyczka Eclipse BPEL Designer.

Eclipse BPEL Designer jest wtyczką integrującą się ze zintegrowanym środowiskiem programistyczne (ang. *Integrated Development Environment – IDE*) Eclipse dostarczając własnej perspektywy wspomagającej projektowanie procesów BPEL. Wtyczka dostarcza wsparcie w definiowaniu, edytowaniu, instalacji oraz testowaniu i debuggowaniu procesów WS-BPEL 2.0, czyli języka do definiowania procesów biznesowych opartego o usługi sieciowe, dostarczonego przez konsorcjum OASIS. Główne cechy wtyczki:

- *Designer* – edytor graficzny (oparty o GEF – Graphical Editing Framework) wprowadzający graficzne oznaczenia elementów procesu BPEL.
- *Model* – reprezentacja modelu BPEL (specyfikacja WS-BPEL 2.0) reprezentowana przez model oparty o EMF (Eclipse Modelling Framework).
- *Validation* – operujący na modelu EMF walidator informujący o błędach i ostrzeżeniach dotyczących procesu BPEL, wynikających ze specyfikacji.
- *Runtime Framework* – zestaw narzędzi umożliwiających instalację oraz wykonanie procesu BPEL.
- *Debug* – zestaw narzędzi umożliwiający śledzenie kolejnych kroków wykonywanego procesu oraz dostarczających obsługę przerw wywołania.

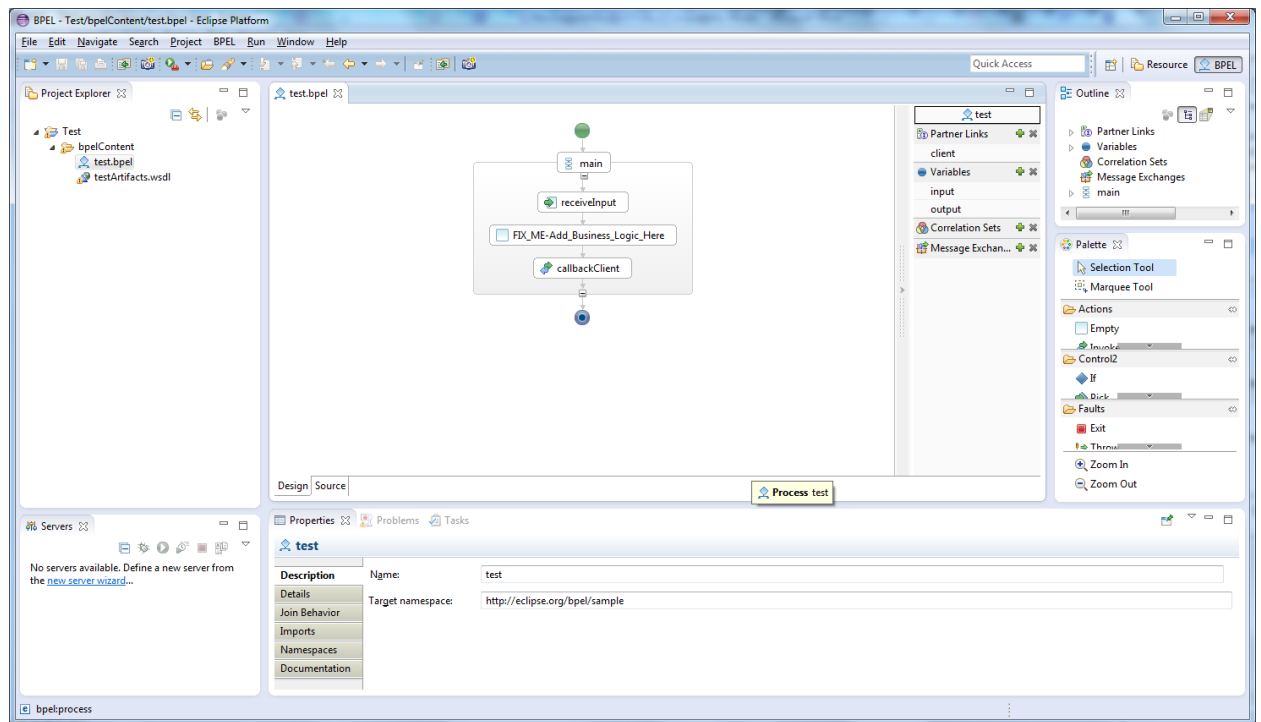
W niniejszej pracy wykorzystywana jest wtyczka Eclipse BPEL Designer w wersji 1.0.3.

5.1. Interfejs użytkownika.

Wtyczka Eclipse BPEL Designer dostarcza własnej perspektywy dodając:

- Okno edytora procesów BPEL (umożliwiający również edycję kodu BPEL).
- Widok *Palette* zawierający graficzne elementy będące reprezentacją znaczników języka BPEL biorących udział w procesie.
- Widok *Outline* - schemat procesu w postaci drzewa elementów.
- Widok *Properties* służący do modyfikacji konfiguracji poszczególnych elementów w procesie – specyficzny dla różnych elementów.

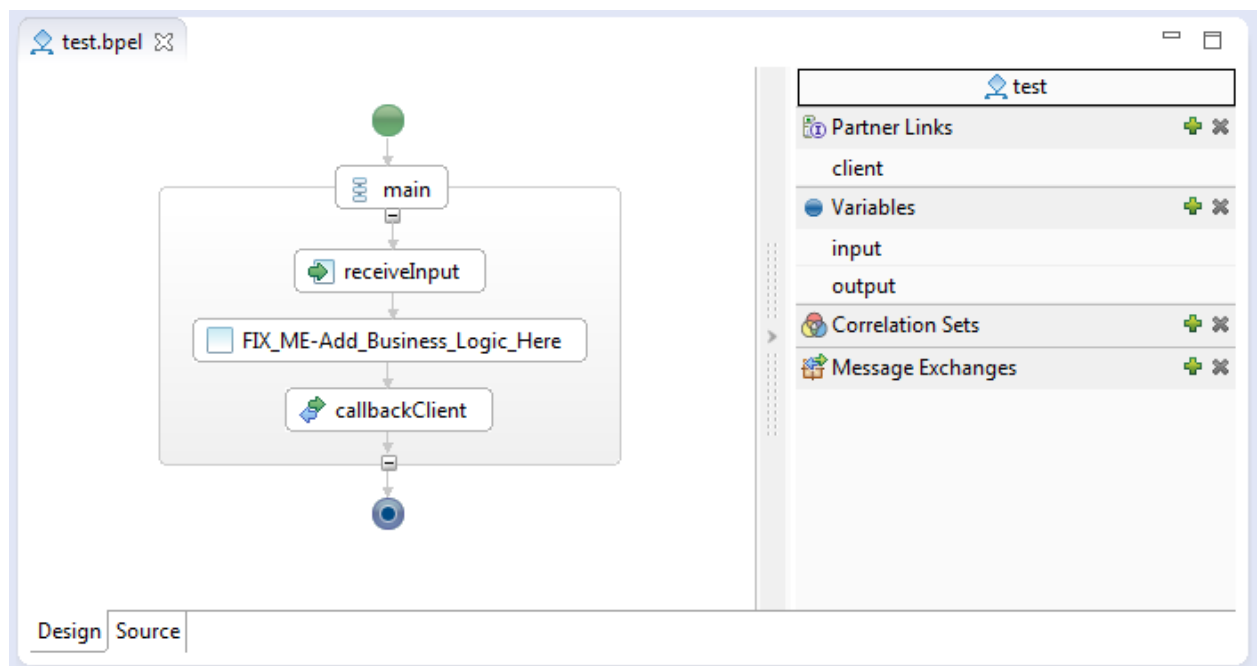
Na rys. 5.1 przedstawiono wygląd IDE Eclipse w perspektywie BPEL.



Rys. 5.1 UI wtyczki Eclipse BPEL Designer.

5.1.1. Edytor.

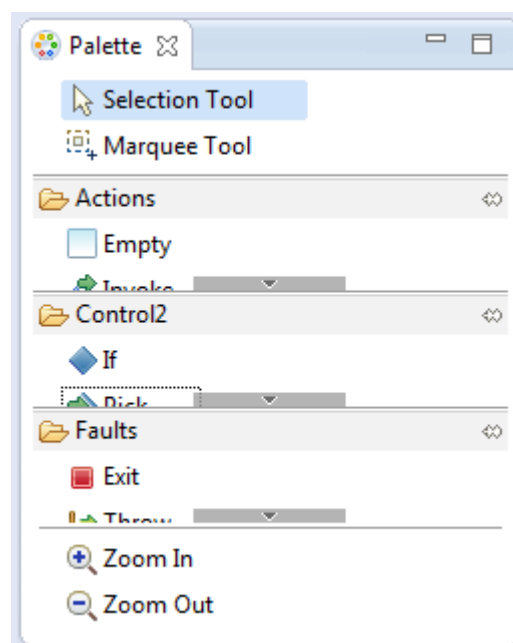
Edytor projektu umożliwia podgląd procesu oraz jego edycję przy użyciu edytora graficznego oraz standardowego edytora kodu BPEL. Edytor graficzny dostarcza możliwość modyfikacji reprezentacji graficznej procesu wizualizującej strukturę głównej sekwencji przebiegu przy użyciu dostarczonych przez wtyczkę Eclipse BPEL Designer graficznych reprezentacji aktywności (*Activity*) BPEL. Graficzny edytor umożliwia również edycję elementów wchodzących w skład procesu takich jak zmienne (*Variables*), połączenia z partnerami biorącymi udział w procesie (*Partner Links*), listę definicji korelacji (*Correlation Sets*) i definicje wiadomości używanych do komunikacji dostawca-konsument usługi (*Message Exchanges*). Ponadto proces może być również edytowany przy użyciu standardowego edytora kodu (zakładka *Source* na Rys. 5.2).



Rys. 5.2 Edytor procesu BPEL – reprezentacja graficzna.


5.1.2. Widoki.




















Jednym z widoków jakie dostarcza Eclipse BPEL Designer jest widok palety, zawierający elementy wykorzystywane do zbudowania struktury procesu w oknie edytora.




Rys. 5.3 Widok palety BPEL Designer

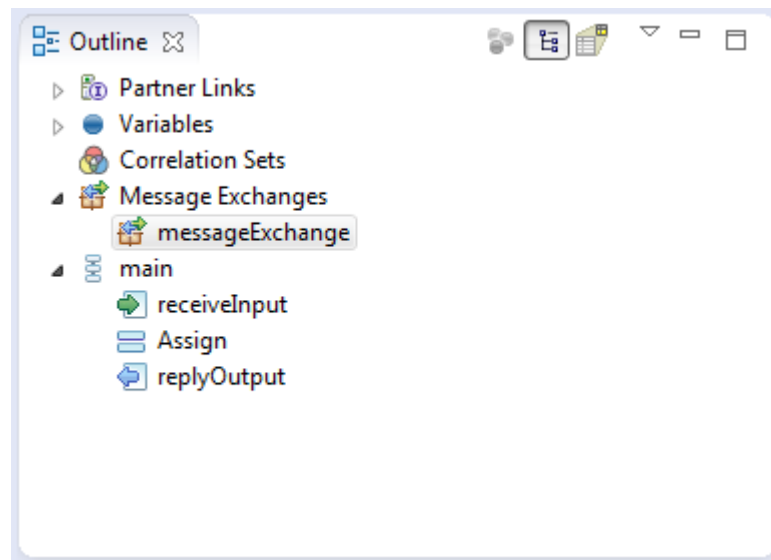
Tab. 5.1 Opis elementów palety BPEL Designer.

Element palety	Opis
 Empty	instrukcja pusta (<i><empty></i>)

 Invoke	wywołanie zewnętrznej usługi sieciowej (<i><invoke></i>)
 Receive	odebranie komunikatu wywołania procesu (<i><receive></i>)
 Reply	odpowiedź wysyłana do obiektu wywołującego process (<i><reply></i>)
 Opaque Activity	
 Assign	instrukcja przypisania wartości zmiennej (<i><assign></i>)
 Validate	walidacja wartości zmiennej na podstawie definicji typu (<i><validate></i>)
 If	warunkowy wybór jednej instrukcji do wykonania (<i><if></i>)
 Pick	oczekiwanie na nadejście wiadomości lub przekroczenie czasu oczekiwania (<i><pick></i>)
 While	pętla wykonywana dopóki warunek jest spełniony (<i><while></i>)
 For Each	pętla wykonywana określoną ilość razy (<i><forEach></i>)
 Repeat Until	pętla wykonywana do spełnienia warunku (<i><repeatUntil></i>)
 Wait	oczekiwanie przez określony czas lub do określonego momentu (<i><wait></i>)
 Sequence	zbiór instrukcji uruchamianych sekwencyjnie (<i><sequence></i>)
 Scope	definicja nowej aktywności (<i><scope></i>)
 Flow	zbiór instrukcji wykonywanych jednocześnie (<i><flow></i>)
 Exit	natychmiastowe zakończenie wykonywanego procesu (<i><exit></i>)
 Throw	instrukcja generująca wystąpienie błędu w procesie (<i><throw></i>)
 Rethrow	instrukcja generująca wystąpienie już obsługiwanego błędu (<i><rethrow></i>)
 Compensate	odwrócenie działania wszystkich zakończonych wewnętrznych instrukcji procesu (<i><compensate></i>)

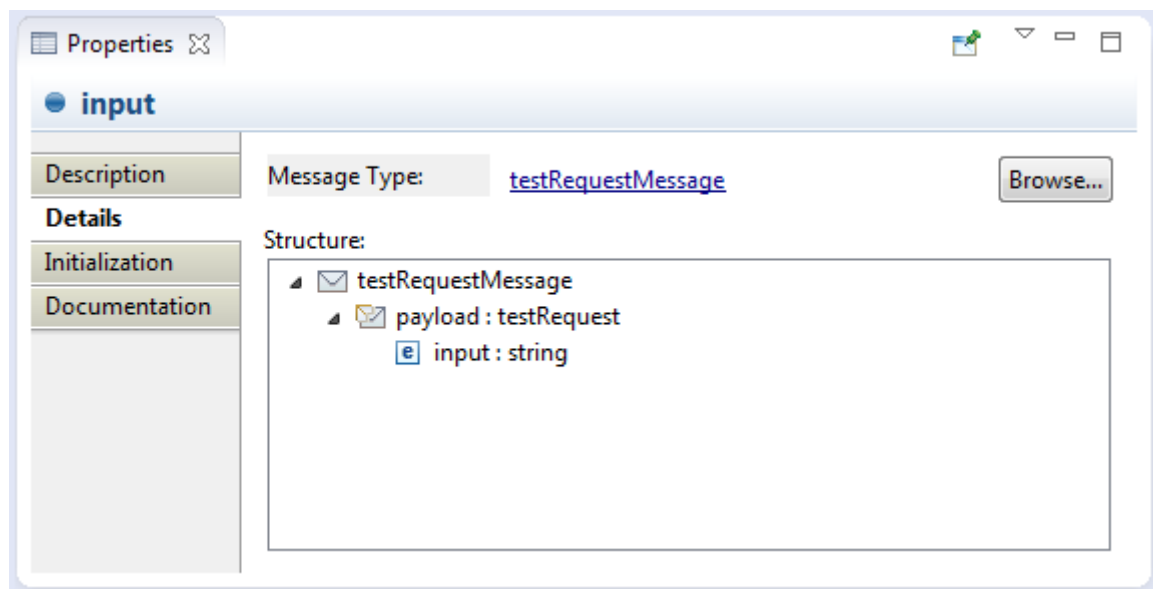
 CompensateScope	odwrócenie działania określonej wewnętrznej instrukcji procesu (<code><compensate></code>)
---	---

Kolejny widok jakim dysponuje Eclipse BPEL Designer jest schemat procesu widoczny na rysunku Rys. 5.4. Schemat przy użyciu struktury drzewiastej obrazuje strukturę zagnieżdżeń wszystkich elementów w całym procesie.



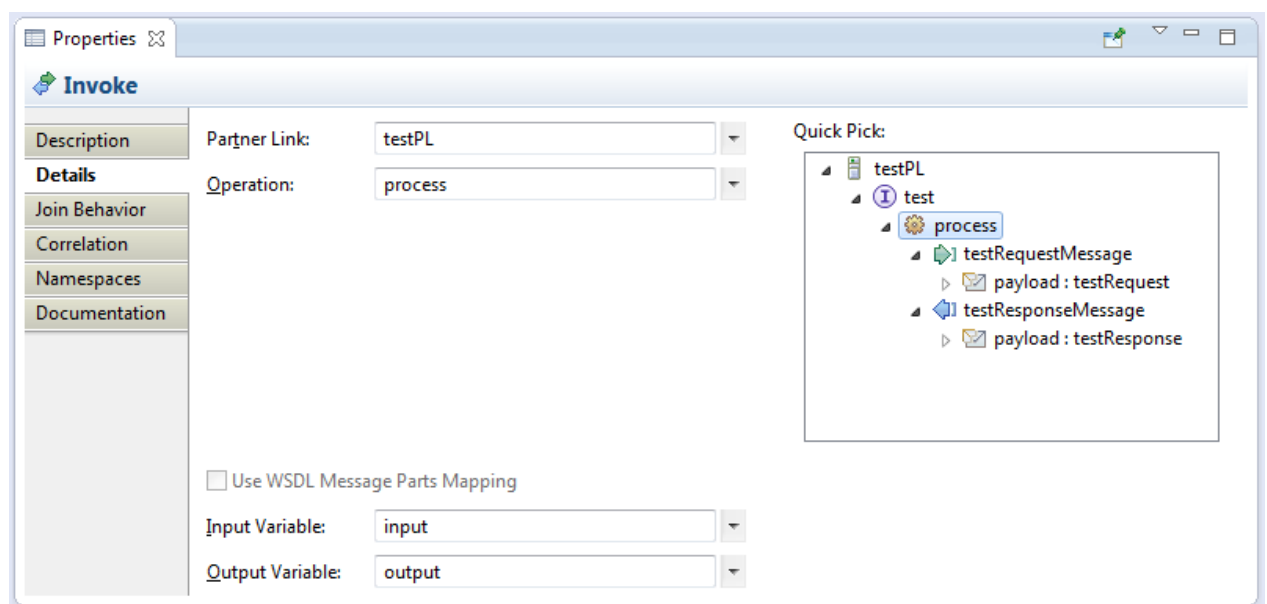
Rys. 5.4 Widok schematu procesu.

Widok konfiguracji (*Properties*) poszczególnych elementów procesu jest ściśle związany z rodzajem edytowanego elementu. Każdy widok konfiguracyjny zawiera elementy podwidoki wspólne dla wszystkich elementów procesu oraz podwidoki specyficzne dla aktualnie edytowanego procesu. Na rysunku Rys. 5.5 przedstawiono szczegółową konfigurację zmiennej procesu umożliwiającą określenie typu danych konfigurowanej zmiennej, który może być typem prostym lub złożonym. Po wyborze typu wyświetlana jest jego struktura, przy czym w przypadku wyboru typu prostego ogranicza się ona do jednego poziomu.



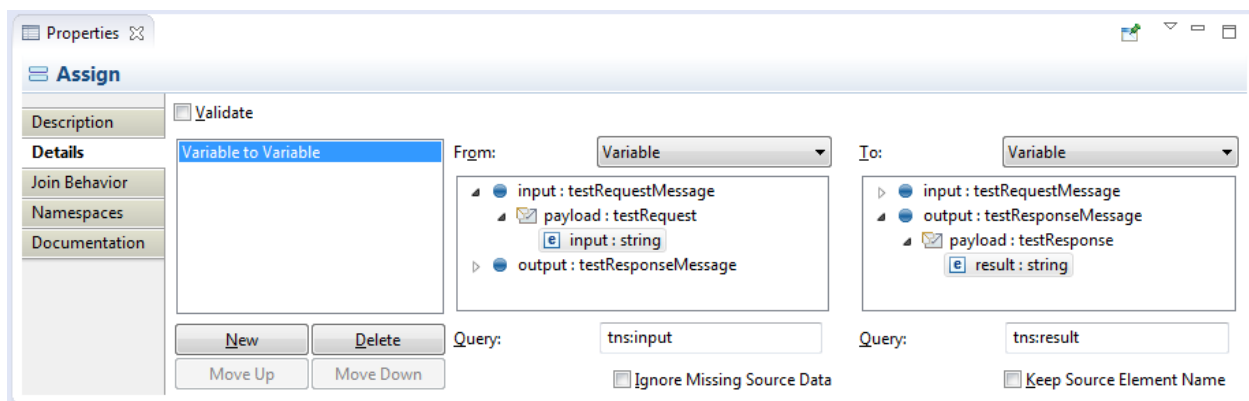
Rys. 5.5 Szczegółowa konfiguracja zmiennej procesu (*Variable*).

Na rysunku Rys. 5.6 przedstawiono widok szczegółowej konfiguracji instrukcji wywołania usługi zewnętrznej umożliwiającą zdefiniowanie partnera procesu oraz operacji wywołanej przez instrukcję. Dokonanie wyboru możliwe jest poprzez sekwencyjne określenie partnera procesu z listy rozwijanej, a następnie operacji – z listy operacji publikowanych przez usługę – z listy rozwijanych, lub przy użyciu sekcji szybkiego wyboru. Po określeniu operacji usługi zewnętrznej wywołanej w konfigurowanej instrukcji konieczne jest również określenie zmiennych procesu, które powinny zostać użyte do wywołania usługi (*Input Variable*) oraz do której zostanie zapisany wynik wywołania operacji usługi zewnętrznej (*Output Variable*).



Rys. 5.6 Szczegółowa konfiguracja instrukcji wywołania usługi zewnętrznej (*Invoke*).

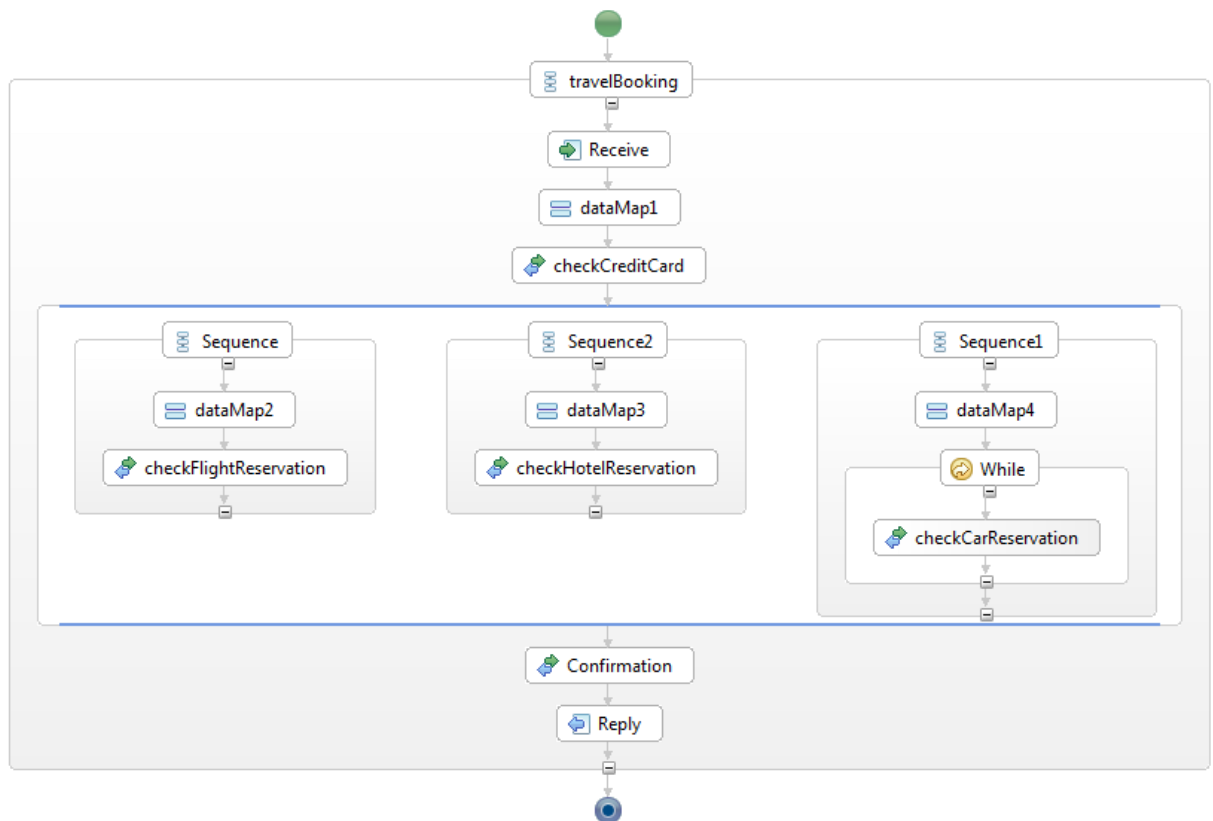
Na rysunku Rys. 5.6 przedstawiono widok szczegółowej konfiguracji instrukcji przepisania wartości zmiennej/zmiennych w procesie. Widok dostarcza możliwość tworzenia/usuwania/zmiany kolejności instrukcji kopiujących w instrukcji przepisania. Możliwe jest również określenie rodzaju elementu z którego oraz do którego wartość ma zostać skopiowana, a następnie określenie tych elementów. Na rysunku zaprezentowano instrukcję przepisania wartości z jedną instrukcją kopiującą. Wybrany typ elementu źródłowego to zmienna procesu, element źródłowy to zmienna typu prostego wchodząca w skład typu złożonego wybranej zmiennej procesu. Podobnie jest w przypadku elementu docelowego.



Rys. 5.6 Szczegółowa konfiguracja instrukcji przepisania wartości (Assign).

5.2. Przykładowy proces BPEL.

Na Rys. 5.1 przedstawiony został przykładowy proces BPEL utworzony przy użyciu Eclipse BPEL Designera, na podstawie procesu rezerwacji wycieczki [2]. W momencie wywołania procesu rezerwacji, zostają mu przekazane informacje dotyczące karty kredytowej, celu oraz okresie podróży. Poprzez wywołanie zewnętrznych usług następuje najpierw sprawdzenie dostępności środków – na karcie kredytowej, następnie równolegle rezerwacja lotu, hotelu oraz samochodu. Po zakończeniu równoległych przebiegów do konsumenta usługi trafia żądanie potwierdzenia rezerwacji, po którym zostaje wysłana informacja o poprawnym zakończeniu procesu.

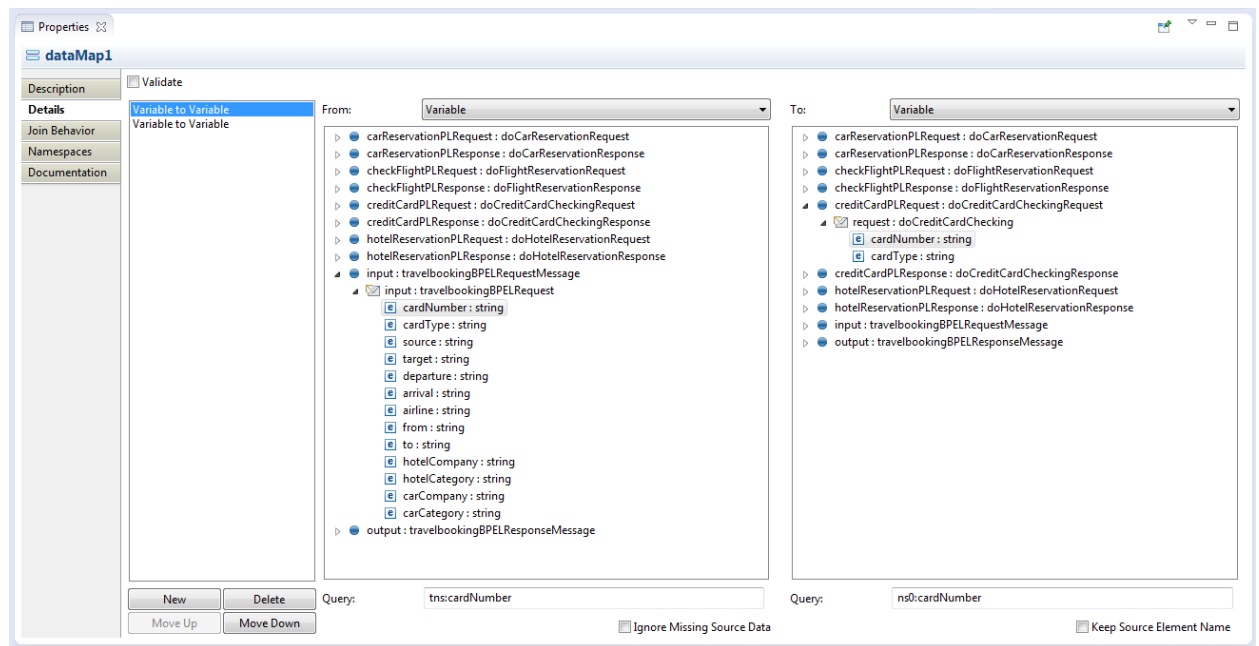


Rys. 5.1 Przykładowy proces BPEL utworzony w Eclipse BPEL Designer – *travelBooking*

W przedstawionym procesie występują trzy bloki przepisania danych (*Assign*):

- *dataMap1* – zawiera instrukcje kopiujące odpowiednie wartości wejściowe procesu do zmiennych będących elementami parametru wywołania usługi *checkCreditCard*.
- *dataMap2* – analogicznie do *dataMap1* dla usługi *checkFlightReservation*.
- *dataMap3* – analogicznie do *dataMap1* dla usługi *checkHotelReservation*.
- *dataMap4* – analogicznie do *dataMap1* dla usługi *checkCarReservation*.

Na Rys. X przedstawiona została konfiguracja instrukcji kopiujących dane na przykładzie bloku przepisywania danych *dataMap1*. Sekcja zawiera listę instrukcji kopiujących opisanych jako para typów (elementu źródłowego oraz elementu docelowego dla instrukcji kopiowania) oraz dwie listy zmiennych o zasięgu nie mniejszym niż aktualnie konfigurowany blok *Assign*. W obu listach *From* oraz *To* zaznaczono zmienne odpowiednio źródłowa i docelowa.



Rys. 3. Sekcja *Properties* bloku przepisywania danych *dataMap1*, zakładka z listą instrukcji kopiujących.

Na Rys. 4. Przedstawiono wygenerowany przez Eclipse BPEL Designer kod w języku BPEL odpowiadający konfiguracji przedstawionej na Rys. 3. dla bloku *dataMap1*.

```

69<bpel:assign validate="no" name="dataMap1">
70  <bpel:copy>
71    <bpel:from part="input" variable="input">
72      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
73        <![CDATA[tns:cardNumber]]>
74      </bpel:query>
75    </bpel:from>
76    <bpel:to part="request" variable="creditCardPLRequest">
77      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
78        <![CDATA[ns0:cardNumber]]>
79      </bpel:query>
80    </bpel:to>
81  </bpel:copy>
82  <bpel:copy>
83    <bpel:from part="input" variable="input">
84      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
85        <![CDATA[tns:cardType]]>
86      </bpel:query>
87    </bpel:from>
88    <bpel:to part="request" variable="creditCardPLRequest">
89      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
90        <![CDATA[ns0:cardType]]>
91      </bpel:query>
92    </bpel:to>
93  </bpel:copy>
94</bpel:assign>

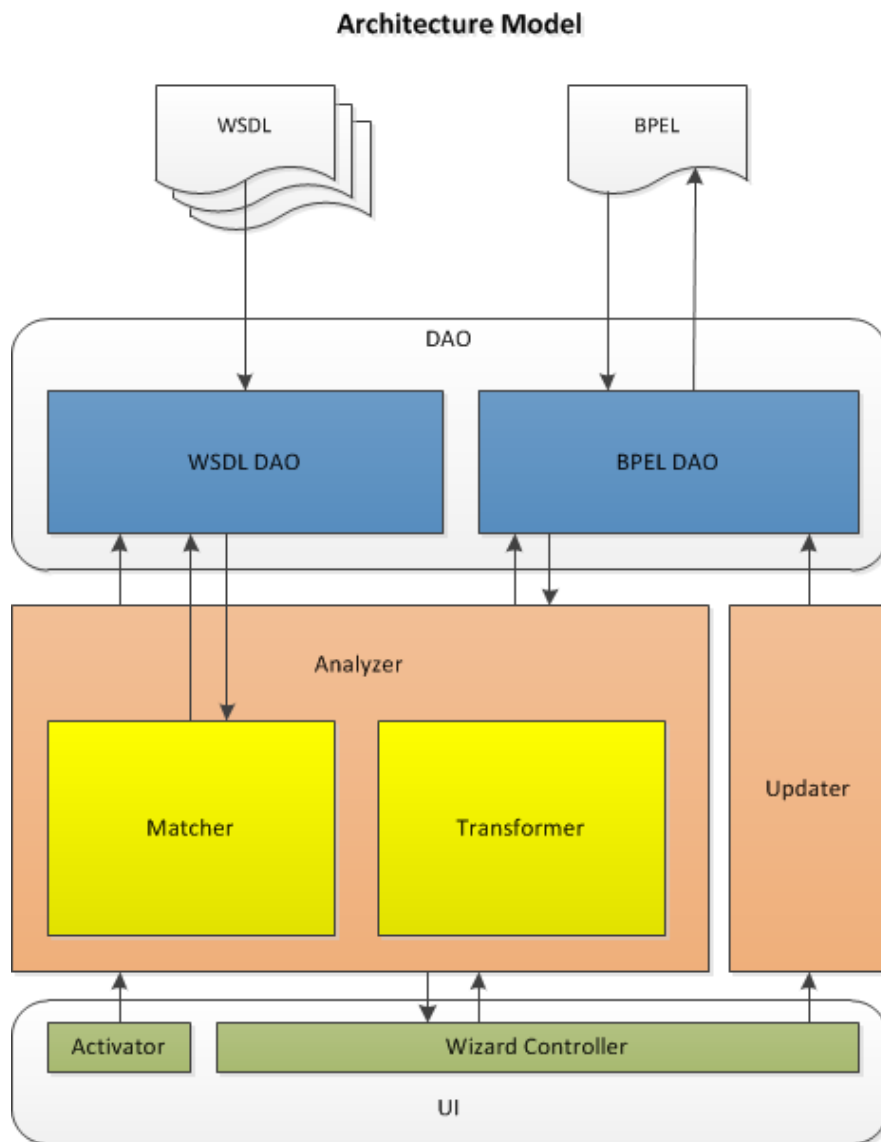
```

Rys. 4. Kod BPEL bloku *dataMap1*.

6. Wtyczka generująca instrukcje kopiujące.

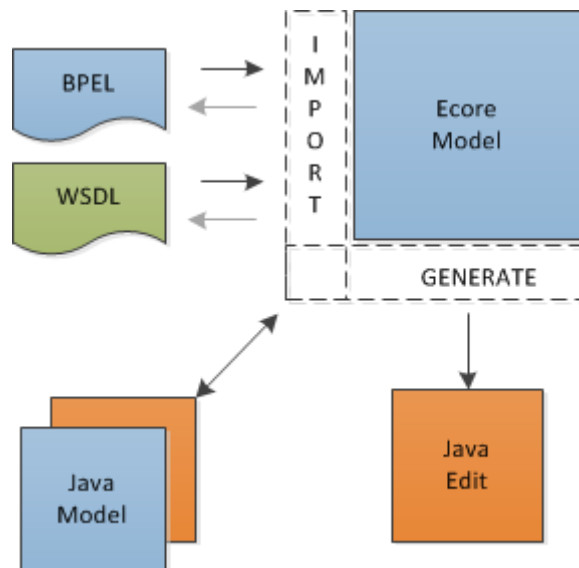
W ramach niniejszej pracy dyplomowej inżynierskiej została opracowana wtyczka generująca instrukcje kopiujące dla instrukcji przepisania wartości zmiennych w procesie BPEL. Wtyczka stanowi rozszerzenie IDE Eclipse bazując na wyniku analizy procesu przetransformowanego z postaci EMF – w jakiej znajduje się po wczytaniu – do postaci grafu. W podstawowym scenariuszu istnieje plik procesu BPEL, który zostaje wczytany przy użyciu obiektu wczytującego. Następnie wczytany proces zostaje przetransformowany do postaci grafu. Graf poddany zostaje analizie przeprowadzonej przez analizator. Wyniki analizy w postaci listy instrukcji kopiujących zmapowanych na konkretne instrukcje przepisania wartości zmiennych występujących w procesie zostają wykorzystane przez aktualizator do załadowania do procesu.

Na rysunku Rys. 6.1 przedstawiono model architektury omawianej wtyczki. Dostępem do danych – opublikowane pliki opisujące usługi sieciowe oraz proces biznesowy BPEL – zajmują się obiekty DAO (*Data Access Object*). Komponenty DAO zostały zastosowane w celu oddzielenia warstwy dostępu do danych od logiki znajdującej w Analizatorze oraz Aktualizatorze i warstwy prezentacji przedstawionych w modelu architektury.



Rys. 6.1 Model architektury wtyczki generującej instrukcje kopiujące.

Zaprezentowany schemat architektury przedstawia dwa obiekty dostępu do danych. BPEL DAO użyty jest do wczytania oraz zapisu procesu biznesowego z i do pliku zawierającego jego definicję (*.bpel) do modelu BPEL Designer przy użyciu modelu EMF. Drugim obiektem dostępu do danych jest WSDL DAO używany do wczytania plików definiujących opis usług sieciowych biorących udział w procesie BPEL. Na rysunku Rys. 6.2 przedstawiono schemat importu plików *.bpel oraz *.wsdl przy użyciu modeli EMF do modeli Java.

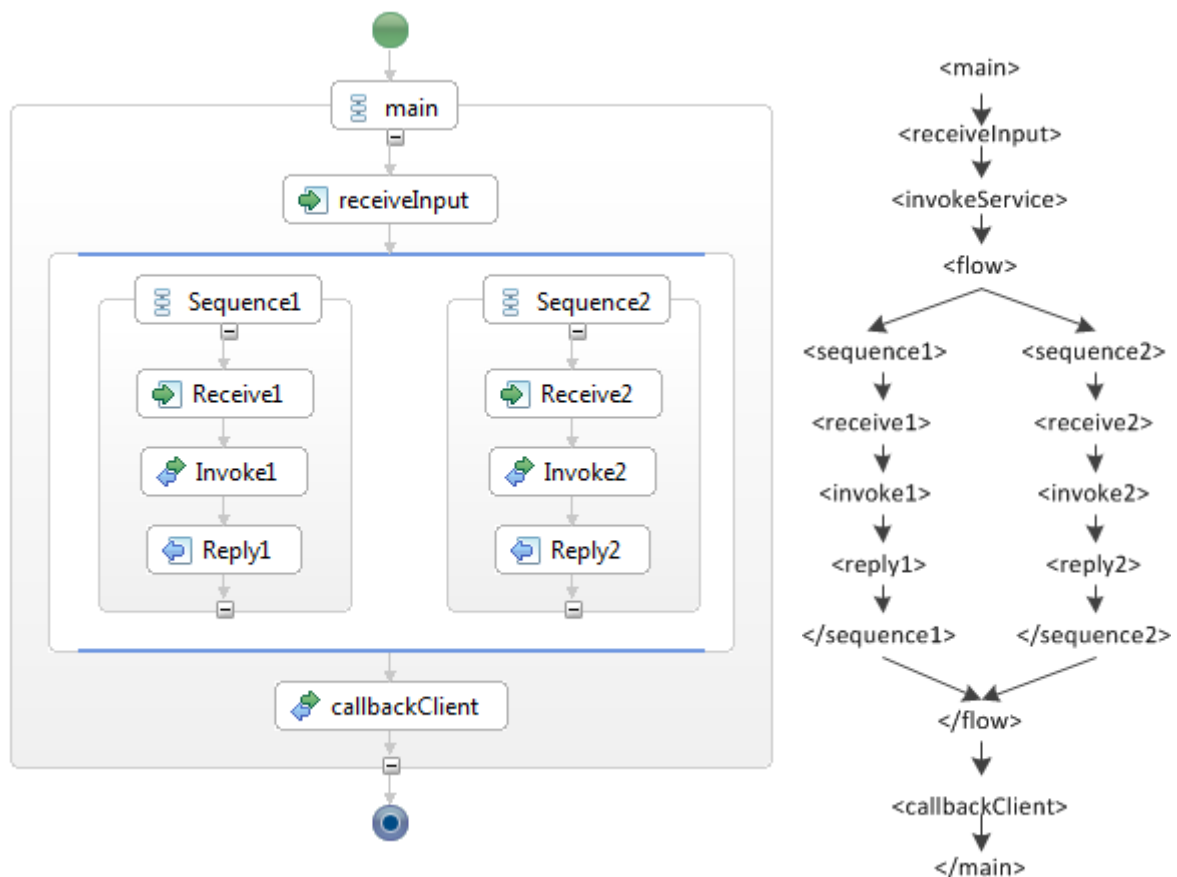


Rys. 6.2 Schemat importu przy użyciu Eclipse Modelling Framework (EMF) na podstawie [5]

Komunikacja z obiektem BPEL DAO przedstawiona na rysunku Rys. 6.1 odbywa się dwukierunkowo w przypadku Analizatora – wysłanie żądania wczytania procesu oraz w odpowiedzi odebranie wczytanego procesu w postaci obiektu EMF. W przypadku Aktualizatora komunikacja z obiektem BPEL DAO zachodzi jednokierunkowo – wysłanie żądania do zapisu procesu w postaci EMF do pliku. Komunikacja z obiektem WSDL DAO prezentowanym na rysunku Rys. 6.1 zachodzi jednokierunkowo z Analizatorem – wysłanie żądania do wczytania listy plików zawierających opis usług sieciowych. Dwukierunkowa komunikacja z WSDL DAO zachodzi natomiast z Komparatorem - jego zadaniem jest wyszukiwanie dopasowań pomiędzy zmiennymi procesu zarówno typów prostych jak i złożonych – polegając na wysłaniu żądania przesłania wczytanego do postaci EMF pliku WSDL oraz w odpowiedzi przekazanie obiektu.

6.1. Transformacja procesu do postaci grafu.

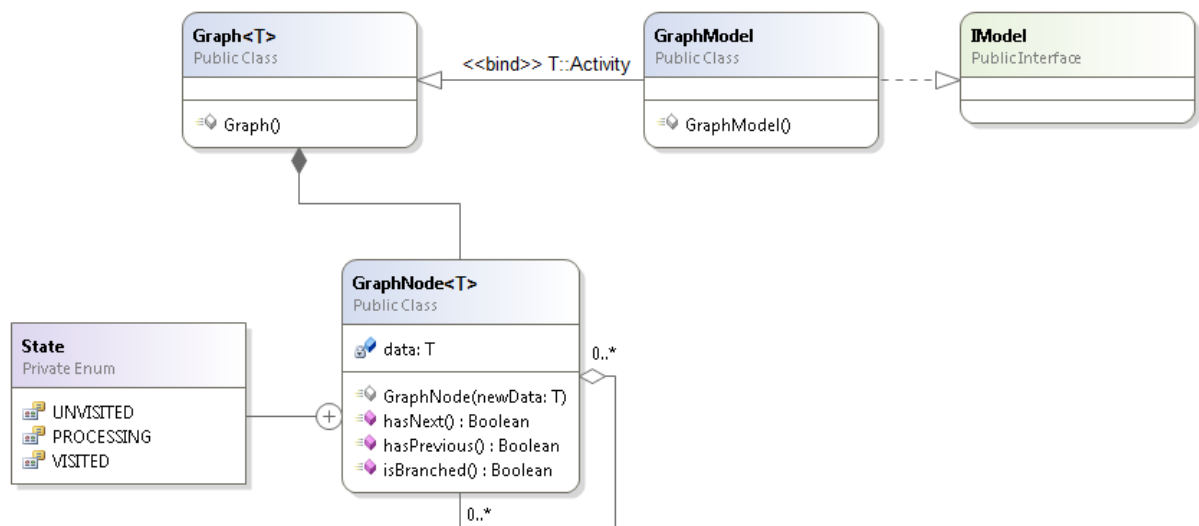
W ramach projektu wtyczki został zaprojektowany oraz zaimplementowany Transformator tworzący z procesu BPEL z modelu BPEL Designer graf aktywności głównego przebiegu procesu. Proces BPEL w notacji Eclipse BPEL Designer oraz jego odpowiednik w postaci grafu przedstawiono na rysunku Rys. 6.3.



Rys. 6.3 Proces BPEL z odpowiednikiem w postaci grafu na podstawie [6].

Model grafu reprezentuje strukturę procesu macierzystego, którego aktywności proste są reprezentowane jako pojedyncze węzły grafu. Aktywności złożone zostały zamodelowane w grafie jako pary węzłów – węzeł otwierający aktywność złożoną oraz węzeł zamykający. Model grafu jest tworzony na zasadzie iteracji po wszystkich elementach aktywności procesu BPEL i w zależności od złożoności elementu produkowany jest pojedynczy węzeł grafu lub ich para. Transformator trafiając na aktywność złożoną generuje węzły – otwierający oraz zamykający – po czym rekurencyjnie wywołuje metodę transformującą dla wszystkich aktywności znajdujących wewnątrz aktywności złożonej. W ten sposób powstaje model procesu w postaci grafu.

Zaprojektowany model grafu jest klasą implementującą interfejs modelu oraz dziedziczącą po klasie implementującej graf jak na diagramie klas przedstawionym na rysunku Rys. 6.4.



Rys. 6.4 Diagram klas modelu grafu.

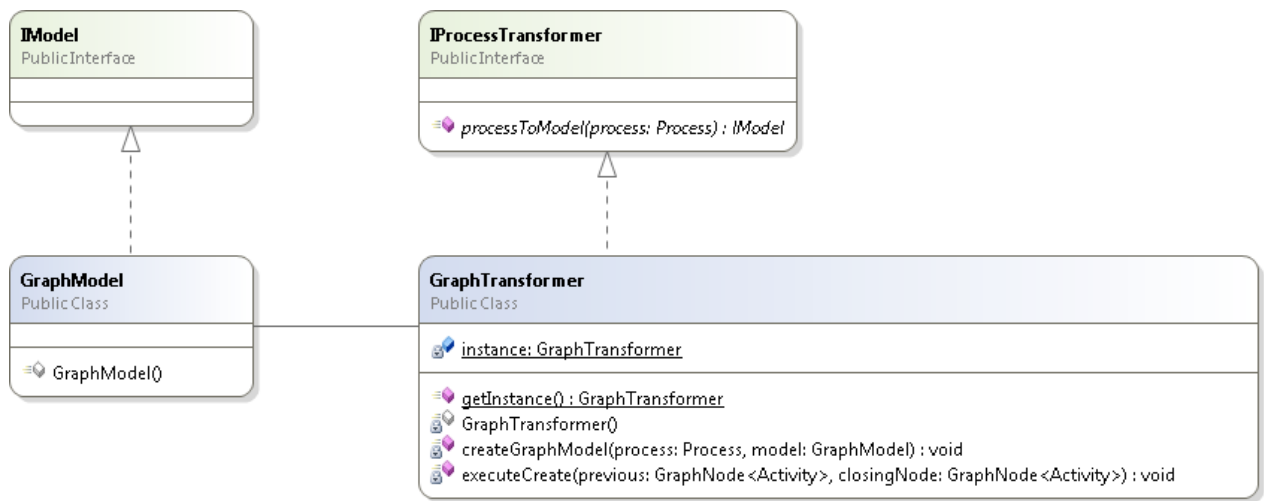
Graf zaimplementowany na potrzeby projektu ze względu na specyfikę problemu – nie mają znaczenia odległości między wierzchołkami, a jedynie istnienie połączeń między nimi – posiada jedynie informację o połączeniach z innymi wierzchołkami. Graf reprezentowany jest przez klasę generyczną, czyli także reużywalną na potrzeby innych modeli bazujących na strukturze takiego grafu. Graf reprezentowany jest przez klasę zawierającą jedynie referencję węzła głównego grafu (*ang. root node*). Pojedynczy węzeł grafu zawiera obiekt modelowanego typu oraz stan węzła będący wartością enumeracyjną. Pojedynczy wierzchołek grafu może znajdować się w stanie nieodwiedzony (*UNVISITED*), procesowany (*PROCESSING*) oraz odwiedzony (*VISITED*). Klasa węzła grafu zawiera także dwie listy z referencjami do obiektów omawianej klasy. Pierwsza zawiera listę węzłów grafu bezpośrednio poprzedzających, natomiast druga, listę węzłów grafu będących bezpośrednimi następnikami dla danego wierzchołka.

Model właściwy grafu wykorzystywanego w dalszej części analizy rozszerza klasę generyczną *Graph<T>* z parametrem typu *org.eclipse.bpel.model.Activity*. W efekcie model złożony jest z węzłów zawierających jako element *data* aktywności składające się na transformowany proces BPEL.

Zdefiniowany model jest w niniejszym projekcie wynikiem transformacji modelu procesu BPEL dostarczonego przez wtyczkę BPEL Designer.

Transformacja odbywa się przy użyciu obiektu klasy *GraphTransformer* implementującej zachowanie interfejsu *IProcessTransformer*, który narzuca na klasy implementujące go konieczność stworzenia definicji metody *processToModel*. Parametrem wywołania wymienionej metody jest instancja obiektu klasy implementującej interfejs *org.eclipse.bpel.model.Process*, czyli przechowywany w pamięci proces BPEL. Elementem zwracanym przez metodę *processToModel* jest obiekt klasy implementującej interfejs *IModel*. W ten sposób zaprojektowany transformator procesu do wykorzystywanego w niniejszym

projekcie modelu w postaci grafu korzysta z wzorca projektowego *Fabryka*. Na rysunku Rys. 6.5 przedstawiono diagram klas wchodzących w skład funkcji transformowania.



Rys. 6.5 Transformator – diagram klas.

Na przedstawionym diagrami klas można zauważyć prywatny konstruktor klasy transformatora. W projekcie zablokowano możliwość tworzenia kilku instancji obiektu tej klasy. Podjęta została taka decyzja ponieważ obiekt klasy *GraphTransformer* jest obiektem bezstanowym i oferuje jedynie operacje na obiekcie dostarczonym mu z zewnątrz, dlatego wykorzystano wzorzec *Singletona* – do instancji obiektu możemy się odwołać jedynie poprzez statyczną metodę *getInstance()*, która zapewnia nam istnienie jednej instancji obiektu w obrębie maszyny wirtualnej *Java*.

Korzystając z wiedzy dotyczącej procesów BPEL wiadomo, że przebieg procesu jest zamknięty w sekwencji głównej – główny przebieg procesu – będącej instancją klasy implementującej interfejs *org.eclipse.bpel.model.Activity* – tak jak wszystkie elementy procesu składające się na przebieg przebieg. W celu rozpoczęcia transformacji konieczne jest znalezienie aktywności procesu rozpoczynającej ten przebieg, który jednocześnie zostanie użyty jako węzeł główny powstające grafu procesu BPEL. Znalezienie aktywności głównej polega na iteracji z wykorzystaniem iteratora wszystkich elementów składających się na proces:

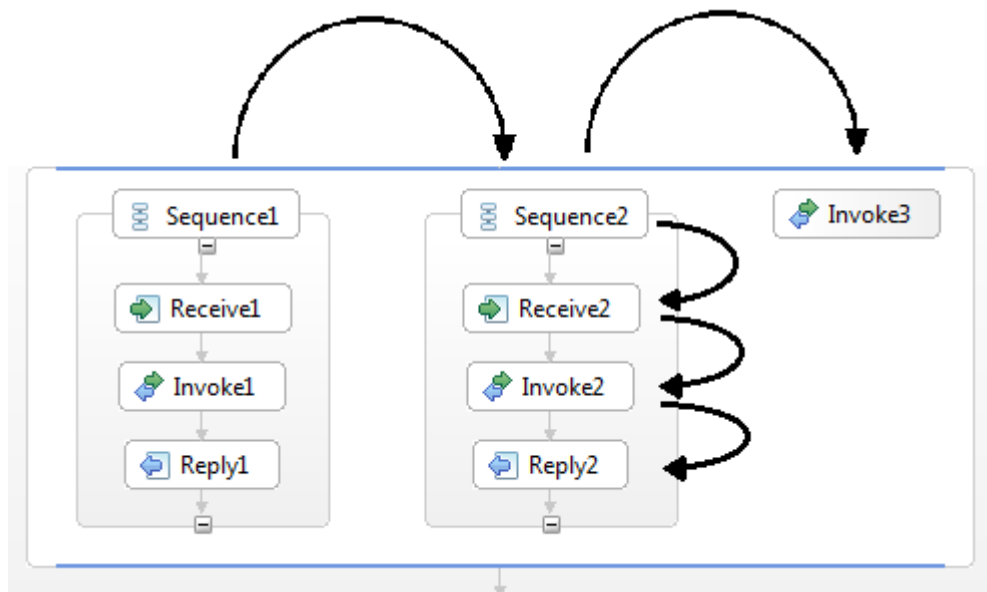
```
TreeIterator<EObject> processIterator = process.eAllContents().
```

Przeszukiwanie trwa do momentu znalezienia pierwszej aktywności – iteracja zostaje przerwana. Stworzony zostaje element *root* generowanego grafu, ponieważ sekwencja jest aktywnością złożoną zostaje utworzony dodatkowy węzeł grafu, który stanowi wierzchołek dodany jako ostatni do struktury.

Utworzony węzeł główny oraz węzeł „zamykający” stają się parametrami wywołania metody:

executeCreate(GraphNode<Activity> previous, GraphNode<Activity> closingNode).

Rozpoczyna się iteracja po elementach stanowiących bezpośrednie potomstwo aktywności z węzła *previous*. Pierwszym krokiem jest sprawdzenie typu obiektu – procesowane są jedynie aktywności, czyli instancje klas implementujących interfejs *org.eclipse.bpel.Activity*. Następnie każda iterowana aktywność jest badana pod kątem złożoności, niezłożone elementy procesu natychmiast trafiają do grafu jako *data* nowo utworzonego węzła i jeśli parametr wywołania *previous* nie zawierał aktywności przepływu (*Flow*) następuje przepisanie referencji – *previous* zaczyna wskazywać na nowo dodany węzeł grafu. W sytuacji gdy iterowany element jest aktywnością złożoną następuje wywołanie rekurencyjne dla węzłów otwierającego – uprzednio dodanego do grafu – i zamykającego tę aktywność. Po zakończeniu wywołania rekurencyjnego przepisywana jest refencja *previous* dla tego samego warunku, co w przypadku aktywności prostych. Ostatnim krokiem pętli – w przypadku gdy parametr wywołania metody zawierał aktywność przepływu – dodanie węzła zamykającego (parametr wywołania) jako następnika dodanego nowego węzła. Dla aktywności złożonych innych niż przepływ (*Flow*) dokładnie takie samo przepisanie następuje po zakończeniu ostatniej iteracji. Takie zachowanie algorytmu wynika z różnicy sposobu iteracji po bezpośrednich potomkach zachodzącej pomiędzy przepływem (*Flow*), a innymi aktywnościami złożonymi. Na rysunku 6.6 przedstawiono różnice pomiędzy iteracją *Flow* – każdy iterowany element powinien wskazywać na węzeł zamykający przepływ jako na element następny - a iteracją po innych złożonych aktywnościach procesu, w tym przypadku sekwencji – tylko ostatni iterowany element powinien wskazywać na węzeł zamykający jako na element następny.

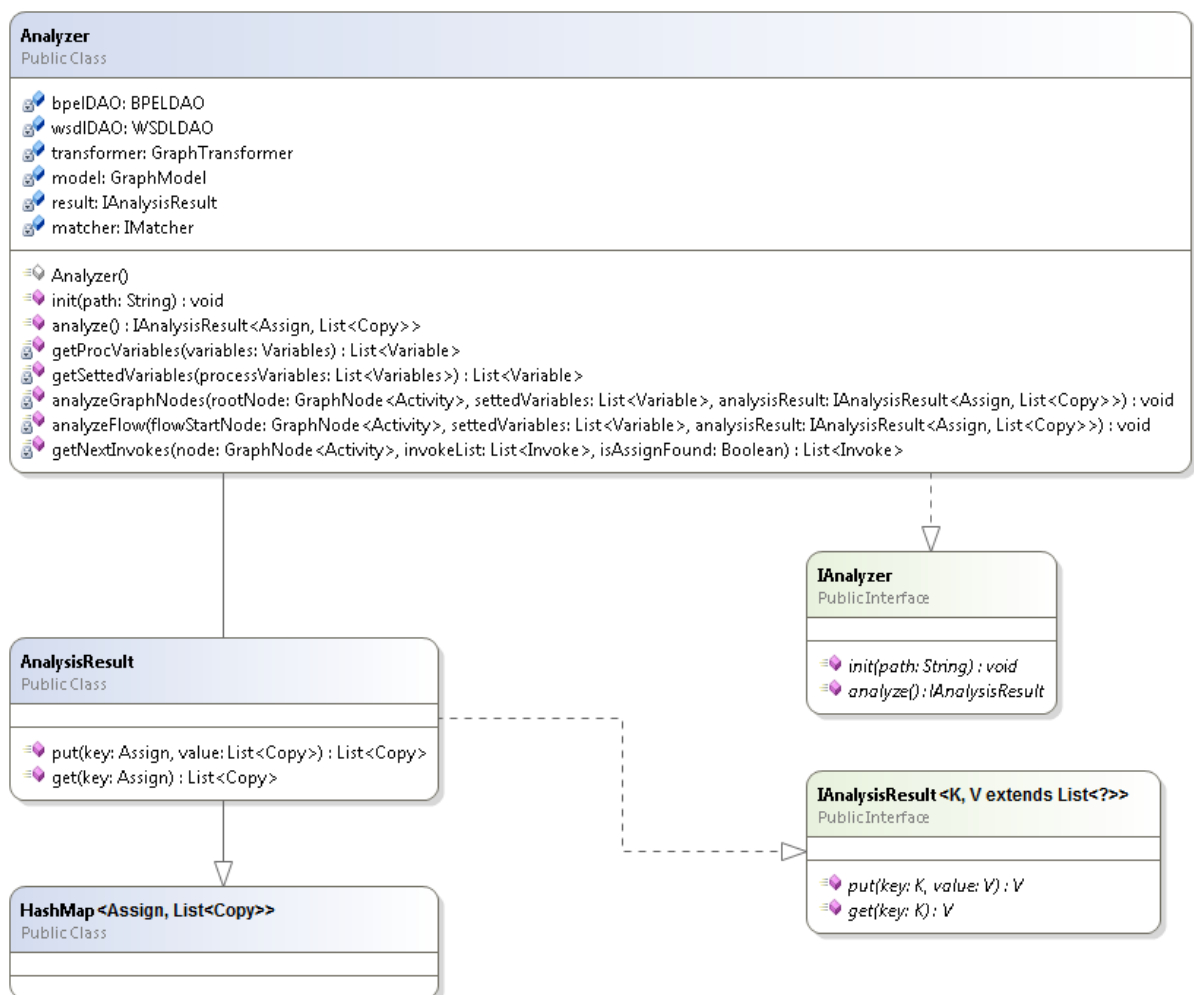


Rys. 6.6 Różnica w sposobie iteracji po potomkach aktywności złożonych.

Model grafu utworzony w wyniku opisanej transformacji jest gotowy do użycia przez analizator, w celu odnalezienia możliwych dopasowań wśród zmiennych procesu.

6.2. Analiza grafu procesu.

Kluczowym etapem działania zaprojektowanej wtyczki jest analiza modelu grafu procesu w celu znalezienia dopasowań pomiędzy istniejącymi zmiennymi procesu. Wynikiem działania analizatora jest lista aktywności przepisania wartości (*Assign*) zmapowanych na listy instrukcji kopiujących wartości pomiędzy odnalezionymi dopasowaniami. Na rysunku Rys. 7.7 zaprezentowano diagram klas, na którym przedstawiona została para interfejsów *IAnalysisResult* oraz *IAnalyzer* zaimplementowanych odpowiednio przez klasy *AnalysisResult* oraz *Analyzer*. Zwracany wynik analizy jest mapą zawierającą listę instrukcji *Assign* jako kluczy, które wskazują listy instrukcji *Copy* utworzonych w procesie analizy.



Rys. 7.7 *Analyzer* – diagram klas.

Inicjowanie procesu analizy grafu rozpoczyna się od utworzenia dwóch list. Pierwsza z nich zawiera wszystkie zmienne (*Variable*) występujące w analizowanym procesie –

zarówno zmienne typu złożonego jak i prostego – poprzez wyszukanie w wejściowym procesie wszystkich obiektów stanowiących instancję typu *org.eclipse.bpel.Variable*. Druga z inicjowanych list zawiera zestaw zmiennych procesu z ustawioną wartością oraz dodatkowo zmienna stanowiąca parametr wywołania procesu – wartości dostarczane do procesu przez konsumenta.

Rozpoczyna się iteracja po elementach grafu startując z węzła będącego węzłem głównym. Iterowane są tylko elementy nieodwiedzone. W każdej iteracji sprawdzane są trzy podstawowe warunki weryfikujące rodzaj aktywności zawartej w iterowanym węźle grafu – w przypadkach gdy wierzchołek zawiera aktywność *Flow*, *Assign* oraz *Invoke* wykonywane są dodatkowe funkcje wchodzące w skład algorytmu analizującego. Natrafienie na węzeł aktywności przepływu algorytm – jako, że jest to aktywność złożona – rozpoznawany jest jego rodzaj:

- otwierający,
- zamykający.

W przypadku węzła otwierającego przepływ omawiany algorytm zostaje wykonany jednocześnie dla każdego z równoległych przebiegów składających się na instrukcję *Flow* traktując je jak podprocesy - podgrafy. Dla węzłów zamykających iteracja jest przerywana – następuje zakończenie rekurencyjnego wywołania algorytmu. W sytuacji, w której iterowany węzeł grafu reprezentuje aktywność *Invoke* – sygnalizująca wywołanie zewnętrznej usługi w procesie – do listy zmiennych posiadających wartość dodana zostaje odpowiedź pochodząca z wywołanej usługi (*ang. Response*).

Kolejnym specyficznym przypadkiem jest węzeł reprezentujący aktywność przepisania wartości zmiennej/zmiennych (*Assign*). Trafiając na taki węzeł algorytm wyszukuje wszystkie instrukcje typu *Invoke*, których wywołanie w procesie następuje po wystąpieniu instrukcji *Assign* z aktualnie iterowanego wierzchołka grafu procesu – wyszukiwanie wierzchołka *Invoke* rozpoczyna się z aktualnie procesowanego węzła grafu. Ważnym elementem wyszukiwania jest fakt przerywania w momencie natrafienia na inną instrukcję *Assign* – by uniknąć sytuacji, w której jedna instrukcja przepisania wartości posiada instrukcję kopiującą wartości zmiennych do wywołań wszystkich usług zewnętrznych następujących po niej, aż do zakończenia procesu. W tej sytuacji mógłby powstać problem dotyczący wydajności procesu gdyż instrukcje kopiujące te same wartości byłyby powielone w innych instrukcjach *Assign* procesu. Wyszukiwanie instrukcji wywołania usług zewnętrznych jest także specyficzne dla aktywności przepływu, w tym wypadku każdy z przepływów podlega równoległemu wyszukiwaniu. Tak powstaje lista instrukcji *Invoke* zawierająca rezultat wyszukiwania wywołań usług zewnętrznych.

W celu zbadania możliwości dopasowań - i na ich podstawie stworzenia instrukcji kopiujących - pomiędzy listą zmiennych zainicjowanych w procesie oraz listą zmiennych będących parametrami wywołania usług zewnętrznych został użyty dodatkowy algorytm w projekcie oddzielony od implementacji analizatora jako zewnętrzny mechanizm. *Matcher*, służący do badania dopasowań pomiędzy zestawami zmiennych został oddzielony od

analizatora. Algorytm porównujący bazując na dostarczonych listach zmiennych zainicjowanych oraz zmiennych użytych jako parametry wywołań usług zewnętrznych porównuje po dwa elementy pochodzące z obu list rozpatrując sytuacje przedstawione w tabeli Tab. 7.1. W każdym przypadku, gdy choć jedna ze zmiennych badanych jest typu złożonego *Matcher* odpytuje obiekt *WSDL DAO* w celu uzyskania definicji typu złożonego badanej zmiennej.

Tab. 7.1 Rodzaje typów par zmiennych porównywanych.

Rodzaj typu.	
Zmienna zainicjalizowana.	Zmienna – parametr.
Złożony.	Złożony.
Złożony.	Prosty.
Prosty.	Złożony.
Prosty.	Prosty.

W przypadku, gdy typy obu badanych zmiennych są złożone i jednocześnie to te same typy następuje bezpośrednie utworzenie instrukcji kopiującej wartość zmiennej zainicjalizowanej do zmiennej wywołania. W sytuacji gdy typy różnią się między sobą do warstwy dostępu do danych zostaje wysłana prośba o podanie definicji obu typów. Otrzymane w odpowiedzi definicje każdego z typów zostają przetłumaczone przez obiekt *Resolver* na listę ciągów znakowych w formacie:

MessagePartName.XSDElementTypeName.XSDElementName.

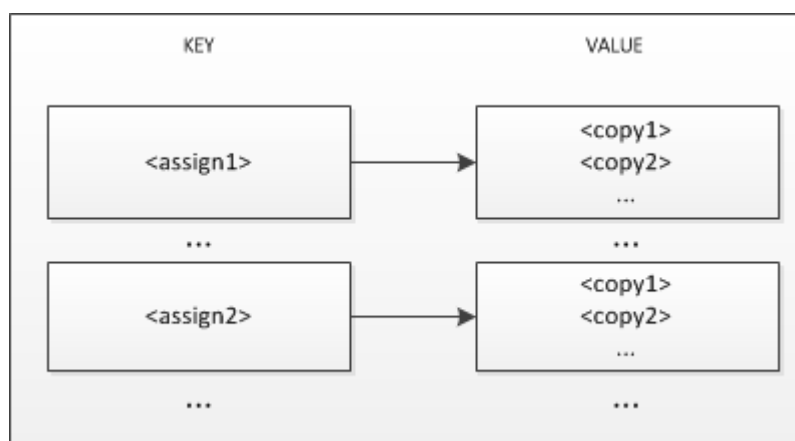
Obie listy „rozwiązanych” typów zostają porównywane pod względem zgodności nazw i typów elementów składowych, które tworzą typ złożony zmiennej biorącej udział w wyszukiwaniu dopasowań. Rezultatem procesu porównywania jest lista par – element typu złożonego z którego nastąpi przepisanie oraz element typu złożonego do którego nastąpi przepisanie wartości - ciągów znakowych w przedstawionym formacie. Tak utworzone dopasowania podlegają już tylko procesowi generacji instrukcji kopiujących, które są dodawane do listy wyjściowej mechanizmu wyszukiwania dopasowań.

Pary badanych zmiennych mogą również być zmiennymi o różnej złożoności typów (patrz tabela Tab.7.1), czyli pary *złożony-prosty* oraz *prosty-złożony*. W takim przypadku algorytm działa przygotowuje zmienną typu złożonego na tej samej zasadzie co w sytuacji porównywania dwóch zmiennych typu złożonego, czyli pobiera definicję typu złożonego z warstwy dostępu do danych (*WSDL*) i tłumaczy elementy typu na ciągi

w zaprezentowanym formacie. Następnie dokonuje porównań elementów typu złożonego ze zmienną typu prostego, jeżeli szereg porównań znalazł jakiegokolwiek dopasowania – typu i nazwy – następuje utworzenie instrukcji kopiującej i dodanie do listy instrukcji *Copy* będącej rezultatem działania mechanizmu poszukiwania dopasowań.

Mając do czynienia z dwoma typami prostymi algorytm dokonuje prostego porównania typu oraz nazwy i na podstawie jego wyniku generuje lub nie instrukcje kopiującą. Wygenerowaną instrukcję *Copy* podobnie dodaje do wyniku działania mechanizmu.

Utworzona w ten sposób lista instrukcji kopiujących trafia do obiektu *AnalysisResult* mającego strukturę mapy przedstawionej na rysunku Rys. 6.8.

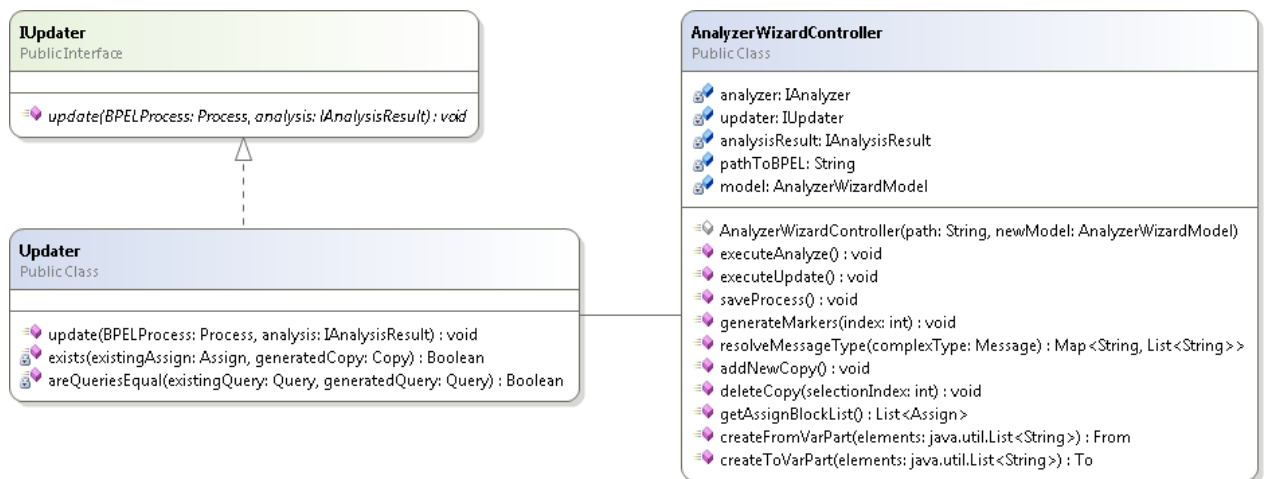


Rys. 6.8 Schemat struktury rezultatu analizy dopasowań.

Każdemu przeprocesowanemu węzłowi grafu procesu ustawiany jest na „odwiedzony” (*VISITED*), do dalszej analizy pobrany zostaje następny element z grafu, któremu zostaje ustawiony stan „procesowany” (*PROCESSING*). Po zakończeniu działania algorytmu produkt analizy jest gotowy do użycia go w celu zaktualizowania procesu BPEL poprzez dodanie nieistniejących do tej pory instrukcji kopiujących odpowiednim blokom przepisywania wartości zmiennych (*Assign*).

6.3. Aktualizacja instrukcji kopiujących

Wynik analizy procesu zawierający zbiór instrukcji kopiujących, które będą dodane do odpowiednich elementów *Assign* przetwarzanego procesu BPEL. Na rysunku Rys. 7.9 przedstawiono diagram klas opisujący mechanizm aktualizujący proces.



Rys. 6.9 *Updater* procesu BPEL – diagram klas.

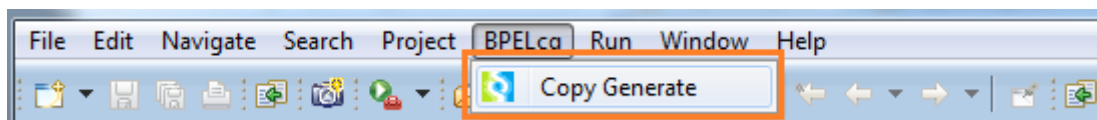
Aktualizacja procesu BPEL polega na dodaniu instrukcji kopiujących do wybranych bloków *Assign*. Iterując po elementach procesu sprawdzany jest ich typ, dla obiektów będących instancjami klasy implementującej interfejs *org.eclipse.bpel.mode.Assign* w pętli dodawane są kolejno elementy pochodzące z listy instrukcji kopiujących, na które wskazuje referencja procesowanego obiektu *Assign* – korzystając z mapy będącej wynikiem uprzedniej analizy procesu. Dodane zostają te instrukcje kopiujące, których aktualizowany blok nie zawiera.

Zaktualizowany proces jest gotowy do zaprezentowania użytkownikowi w postaci panelu konfiguracyjnego instrukcji *Assign*.

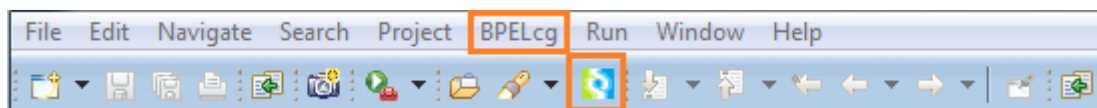
6.4. Graficzny interfejs użytkownika.

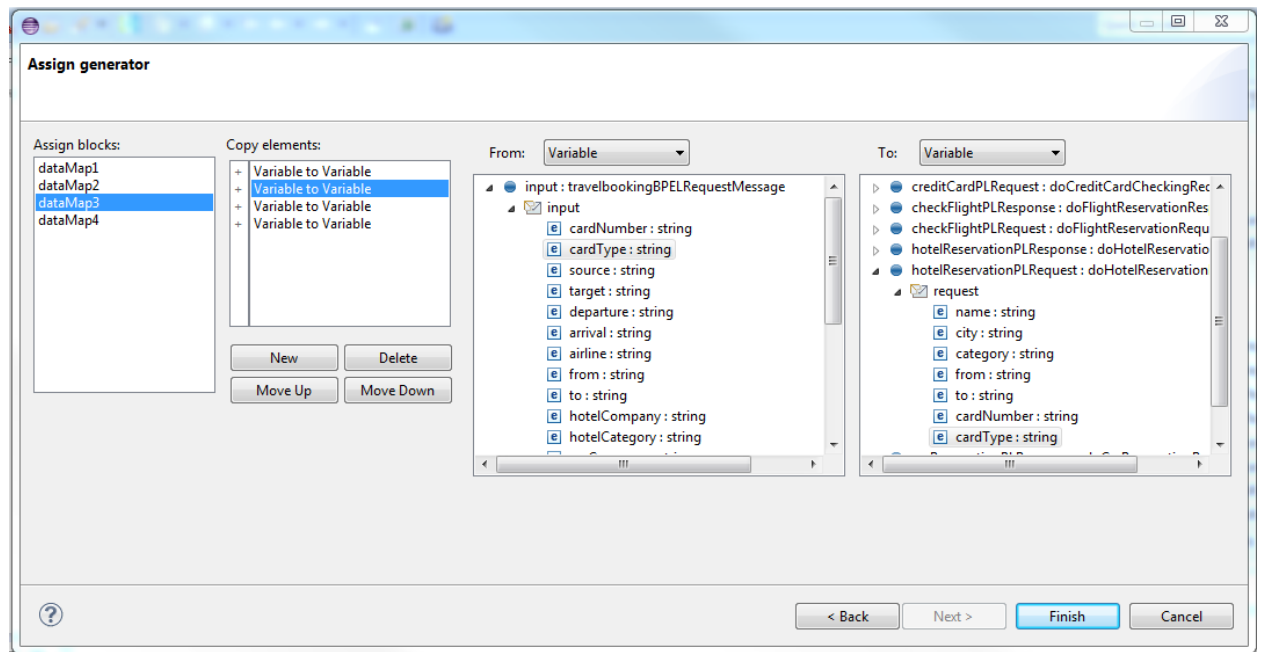
Wtyczka generująca automatycznie instrukcje kopiujące dostarcza interfejs użytkownika na który składa się:

- Menu (*BPELcg*) znajdujące się w Menu Bar – głównym menu środowiska Eclipse – umożliwiające uruchomienie generatora instrukcji kopiujących.



- Przycisk – podobnie jak menu *BPELcg* – aktywujący akcję uruchomienia generatora instrukcji kopiujących w procesie BPEL.





6.5. Konfiguracja wtyczki (PDE).

Opis konfiguracji wtyczki pod plugin Eclipse BPEL Designer – extension points.

7. Testy funkcjonalne.

=====				
Opis	przeprowadzonych	testów:	przebieg,	wyniki.
=====				

8. Podsumowanie.

=====

Ogólne	podsumowanie	projektu.
--------	--------------	-----------

=====

8.1. Napotkane problemy.

=====

Problemy,	z	którymi	stykano	się	podczas	realizacji	projektu.
-----------	---	---------	---------	-----	---------	------------	-----------

=====

8.2. Możliwości rozwoju.

=====

Dalsze perspektywy rozwoju projektu, jak można go rozwinąć, co można ulepszyć.
--

=====

9. Bibliografia.

- [1] <http://www.eclipse.org/bpel/>
- [2] Andrzej Ratkowski. Projektowanie transformacyjne procesów w architekturze usługowej, 2011.
- [3] <http://pic.dhe.ibm.com/infocenter/adiehelp/v5r1m1/index.jsp?topic=%2Fcom.ibm.etools.ctc.bpel.doc%2Fsamples%2Ftravelbooking%2FtravelBooking.html>
- [4] <http://www.slideshare.net/milliger/eclipse-bpel-designer-presentation-765528>
- [5] Nick Boldt and Dave Steinber. Introduction to the Eclipse Modeling Framework, eclipseCON 2006.
- [6] Siran Chen. Extraction of BPEL Process Fragments in Eclipse BPEL Designer. Stuttgart, 2009.
- [7] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. Head First. Design Patterns. 2004 O'Reilly Media, Inc.

10.Załączniki.

Bla bla bla.

10.1.Płyta CD.

Bla bla bla.

Dołączona płyta CD zawiera:

- Zestaw testowy przeznaczony dla koder/dekoder
 - *.v – pliki poszczególnych modułów koder/dekoder (Verilog)
 - *.mif – pliki inicjacyjne pamięci ROM modułów koder/dekoder
 - Idcelp_encoder_tester.qar – archiwum projektu Quartus II zawierające zestaw testowy koder
 - Idcelp_decoder_tester.qar – archiwum projektu Quartus II zawierające zestaw testowy dekoder
 - EncoderUSBReader.java – moduł programowy testera koder odczytujący dane z portu USB i zapisujący je do pliku (Java)
 - DecoderUSBReader.java – moduł programowy testera dekoder odczytujący dane z portu USB i zapisujący je do pliku (Java)
 - jd2xx.jar – biblioteka procedur komunikacji z układem FT245BM (Java)
- Zestaw wektorów testowych wraz ze stanami wewnętrznymi
- Elektroniczną wersję pracy dyplomowej magisterskiej

10.2.Instrukcja instalacji wtyczki BPELag (BPEL assign generator).

Instrukcja