SR–obiektowo zorientowany język programowania rozproszonego*

Aleksander Laurentowski, Krzysztof Zieliński
Instytut Informatyki AGH

28 stycznia 1993

Celem artykułu jest przybliżenie języka SR – jednego z rozwijanych obecnie narzędzi programowania równoległego i rozproszonego w lokalnych sieciach komputerowych. Jako język stosunkowo niskiego poziomu SR umożliwia programowanie efektywne a jego główne zalety to prostota składni i bogactwo dostępnych mechanizmów synchronizacji. Ponadto SR stanowi środowisko do tworzenia systemów w pełni obiektowo zorientowanych.

1 Wprowadzenie

Spośrod wielu istniejących języków programowania rozproszonego SR wyróżnia się przede wszystkim **prostotą** zdefiniowanej składni oraz **wielością** dostępnych mechanizmów synchronizacji i komunikacji. Jego autorzy postawili sobie za cel zgromadzenie w jednym przejrzystym języku konstrukcji, za którymi kryją się prawie wszystkie dobrze znane z teorii systemów operacyjnych i programowania równoległego metody komunikacji i synchronizacji, takie jak m.in. zdalne wywołanie procedury (RPC), dynamiczne kreowanie procesów, przekazywanie sygnałów, rendezvous czy semafory. Poszczególne z nich można odnaleźć w Adzie, CSP, Parallel C, Lindzie czy notacji rejonów krytycznych i monitorów, ale w SR są one dostępne wszystkie w jednym środowisku. Są to w większości mechanizmy stosunkowo niskiego poziomu, jednak dzięki dużym możliwościom ich łączenia programista może tworzyć nawet bardzo złożone konstrukcje, elastycznie dostosowując tworzone rozwiązania do potrzeb problemu. Dodatkowo starano się nie stracić zbyt wiele na efektywności i sile owych mechanizmów. Wydaje się, iż w dużej mierze zamysł ten się powiódł.

SR (w pełnym brzmieniu: *Synchronizing Resources*) jest projektem powstałym na początku lat 80-tych i rozwijanym do dziś na University of Arizona w Tucson, USA. Grupą badawczą SR kierują tam Gregory R. Andrews i Ronald A. Olsson [1]. Przy projektowaniu SR oparto się przede wszystkim na Moduli, Pascalu i DP *(Distributed Processes)*. Najwcześniejsza wersja języka, zwana dziś SR_0 przewidywała mechanizmy asynchronicznego przekazywania wiadomości *(asynchronous message passing)* oraz rendez-vous. Implementacja tego ostatniego była na owe czasy unikalna: proces obsługujący rendez-vous (czekający) mógł ustalać kolejność obsługi wywołań na podstawie wartości przekazywanych parametrów [2]. W miarę upływu czasu koncepcję i składnię języka udoskonalano i rozszerzano [3], a w roku 1988 udostępniono (jako *public domain*) bardzo bogatą implementację środowiska SR w wersji 1.1 [4]. Artykuł ten bazuje głównie na doświadczeniach z eksploatacji tej właśnie wersji. Obecnie osiągalna jest wersja 2.0 SR z marca

*Pracę wykonano w ramach grantu KBN nr:8.0077.91.01.

1992 roku, szczegółowo omówiona w [6][1]. W punkcie drugim przedstawiamy ogólną strukturę języka i charakter dostępnych w nim mechanizmów. Punkt trzeci omawia pokrótce technikę projektowania programów w SR i właściwości aktualnej implementacji języka.

2 Charakterystyka ogólna języka

2.1 Model obliczenia

Program napisany w SR może wykonywać się jednocześnie w wielu przestrzeniach adresowych przyporządkowanych do zbioru komputerów, stanowiących węzły sieciowe. Hierarchia środowiska programowego jest czterostopniowa, co jest uwidocznione na rys. 1. Najwyższy poziom to **maszyny fizyczne** – jedno- lub wieloprocesorowe komputery, połączone siecią lokalną. W przestrzeniach adresowych maszyn fizycznych rezydują **maszyny wirtualne**, będące dzielonymi obszarami pamięci dla umiejscowionych w ich wnętrzu **zasobów** *(resources)* i **globaliów** *(globals)*.

Zasoby są zasadniczymi jednostkami języka i de facto obiektami w rozumieniu klasyfikacji systemów obiektowo zorientowanych. Zawierają bowiem struktury danych (określające stan obiektu) i operacje, dzięki którym możliwe są obliczenia i komunikacja. Językowe pojęcie zasobu umożliwia tworzenie obiektów zarówno w modelu pasywnym (udostępniającym zespół operacji, których wykonania żąda się głównie z zewnątrz) jak i aktywnym, w którym operacje należące do danego obiektu są wywoływane i wykonują się wewnątrz niego (mówimy wtedy o *procesach implementujących* obiekt); często stosowane modele mieszane są dostępne w sposób naturalny. Wszelkie jednostki *(entities)*, takie jak typy, stałe czy operacje, zdefiniowane wewnątrz zasobu mogą pozostać jego własnością lub być udostępnione na zewnątrz dla grupy zasobów "zainteresowanych" ich wykorzystaniem. Globalia slużą do specyfikacji jednostek globalnych – umożliwiają dzielenie typów i stałych pomiędzy zasobami, które zgłoszą takie życzenie.

Procesy wewnątrz zasobu mogą bez przeszkód komunikować się ze sobą w ramach wspólnej przestrzeni adresowej, jaką jest dla nich zasób oraz z procesami znajdującymi się w innych zasobach tej samej lub odrębnej maszyny wirtualnej, w szczególności umiejscowionej na oddalonym fizycznym węźle sieci. Mechanizmy komunikacyjne są niezależne od położenia maszyn wirtualnych i transparentne na poziomie języka. Na przykład asynchroniczne przekazanie wiadomości pomiędzy procesami rezydującymi w jednej maszynie wirtualnej ma tę samą składnię i znaczenie co mechanizm ten działający pomiędzy dwoma różnymi maszynami.

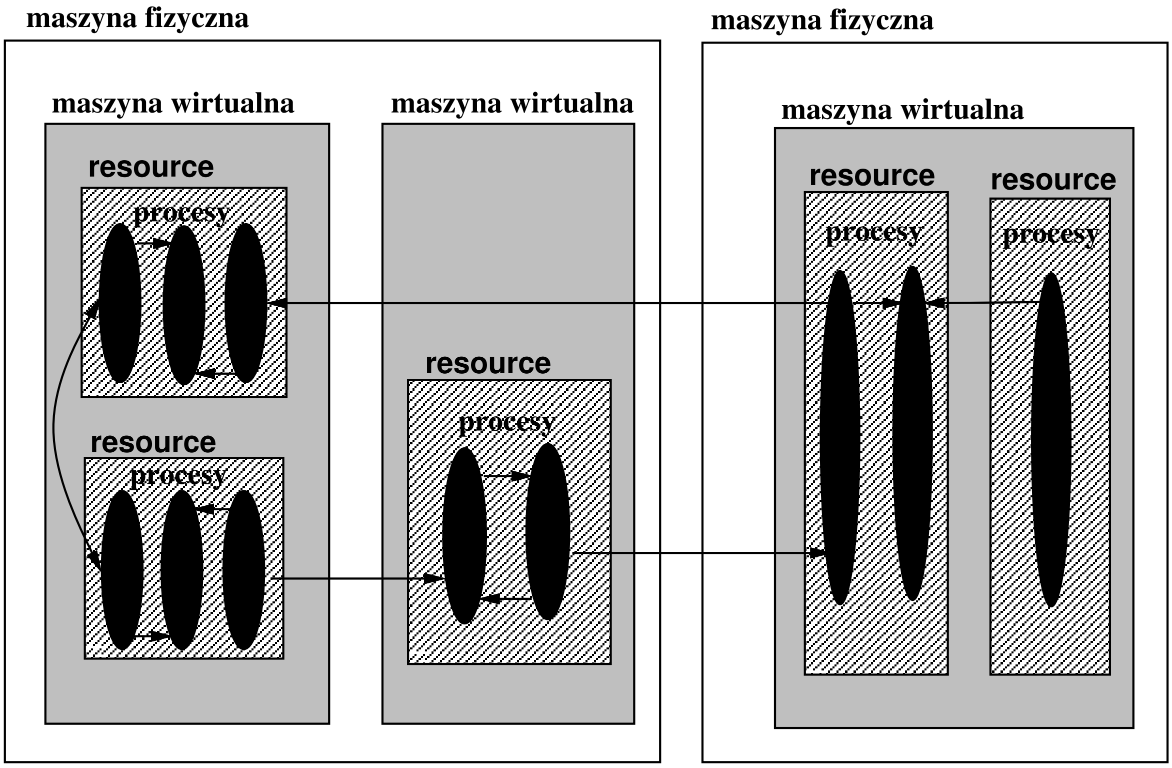
Zastosowania języka sięgają od najprostszych modeli obliczeń sekwencyjnych, realizowanych na pojedynczym procesorze, poprzez obliczenia z pamięcią dzieloną na architekturach wieloprocesorowych, aż po systemy rozproszone, działające na zbiorze maszyn wirtualnych rozlokowanych na sieci stacji roboczych.

2.2 Podstawowe konstrukcje języka

Program w SR składa się z trzech (oddzielnie kompilowalnych) części:

- globaliów *(globals)*
- specyfikacji zasobów *(resource specifications)*
- ciał zasobów *(resource bodies)*

[1] Pakiet SR w wersji 2.0 jest dostępny via anonymous ftp z *cs.arizona.edu*



Rysunek 1: Model obliczenia w SR

Globalia zawierają deklaracje stałych i typów dzielonych pomię-dzy zasobami. Ich deklaracja ma formę:

global identyfikator
 deklaracje stałych i typów
end identyfikator

Zasoby są sparametryzowanymi modułami (obiektami), których instancje są tworzone dynamicznie, za pomocą instrukcji **create**. Zdefiniowane w ich wnętrzu jednostki, tj. typy, stałe, zmienne i operacje są obiektami wzajemnie dzielonymi w obrębie *(scope)* danego zasobu. W sensie języka zasoby określone powyżej jako aktywne są realizowane przez jeden lub więcej procesów wykonywanych w tej samej dzielonej przestrzeni adresowej. Z punktu widzenia systemu operacyjnego UNIX procesy wykonują się jako wątki *(threads)* a współbieżność jest zapewniona poprzez przełączanie kontekstu procesora *(context switching)*. Procesami systemu operacyjnego są natomiast maszyny wirtualne, w których wnętrzu rezydują zasoby.

Deklaracja zasobu składa się z dwóch części: specyfikacji i ciała (podział ten może być pominięty, jeśli nie zachodzi potrzeba odrębnej kompilacji).
Specyfikacja zasobu ma składnię:

resource identyfikator
 import *lista identyfikatorów zasobów*
 deklaracje operacji, stałych i typów eksportowanych na zewnątrz
end identyfikator

Specyfikacja wyszczególnia wszystkie elementy, które zasób udostępnia na zewnątrz (eksportuje) – mogą być nimi operacje, stałe i typy. Mechanizm importu umożliwia korzystanie z tych elementów. Udostępnia się je poprzez wyliczenie po dyrektywie **import** odpowiednich zasobów. Każdą z operacji deklaruje się podając po słowie kluczowym **op** jej nazwę (identyfikator) oraz typy argumentów i zwracanej wartości. Definicje operacji, opisujące sposób ich realizacji znajdują się w ciele (*body*) zasobu.

Ciało zasobu zawiera: procesy implementujące zasób, deklaracje elementów dzielonych pomiędzy tymi procesami oraz kod początkowy i końcowy (opcjonalnie):

```
body identyfikator(lista parametrów formalnych)
    import lista identyfikatorów
    deklaracje lokalnych obiektów dzielonych
    procesy
    initial blok kodu początkowego end
    final blok kodu końcowego end
end identyfikator
```

Ciało zasobu dziedziczy wszystkie obiekty importowane lub zadeklarowane w uprzednio skompilowanej specyfikacji. Składnia języka dopuszcza opcjonalność większości wyżej opisanych deklaracji, a ponadto mogą one pojawiać się w dowolnej kolejności. Obowiązuje jednak zasada, że obiekt (element) musi zostać zadeklarowany przed pierwszym odwołaniem do niego.

Oto jeden z najprostszych programów, jakie da się napisać w SR :

```
resource hello()
  process writing
    write("Hello world!")
  end
end
```

2.3 Mechanizmy programowania obiektowo zorientowanego

Spójrzmy na zasób SR jako obiekt: jego *metody* to operacje zawarte w specyfikacjach – udostępnione na zewnątrz umożliwiają dokonywanie działań na obiekcie; jego *stan* opisują struktury danych zadeklarowane wewnątrz (w ciele) zasobu. Środowiska oparte o obiekty i posiadające przestrzeń klas, tj. obiektowo zorientowane *(object-oriented environments)* posiadają szereg specyficznych własności. Jedną z nich jest dziedziczność w hierarchii klas. Rodziny obiektów w SR mogą być tworzone za pomocą mechanizmu rozszerzania *(extension)* zasobów bardziej abstrakcyjnych, rozumianych jako klasy. Mechanizm ten uzyskuje się poprzez dodanie do specyfikacji aktualnie definiowanego zasobu dyrektywy **extend**, po której następować powinny nazwy zasobów, których wszystkie zadeklarowane w odpowiednich specyfikacjach elementy mają być dziedziczone, np.

```
resource Figure
  op calc_area() returns area : real
end Figure
```

jest abstrakcyjnym typem – modelem figury geometrycznej, pomyślanym pod kątem obliczania jej pola. Model ten możemy ukonkretnić dla własnych potrzeb z użyciem dyrektywy **extend**:

```
resource Circle
   extend Figure
body Circle(radius : int)
   proc calc_area() returns area
      area := 3.1416*radius*radius
   end
end Circle
```

Poszerzmy jeszcze naszą rodzinę obiektów:

```
resource Square
   extend Figure
body Square(side : int)
   proc calc_area() returns area
      area := side*side
   end
end Square

resource Complement
   import Circle, Square
   op calc_area_complement() returns real

body Complement( size : int )
   var cc : cap Circle
   var cs : cap Square

   initial
      cc := create Circle(size)
      cs := create Square(2*size)
   end

   proc calc_area_complement() returns diff
      diff := cs.calc_area() – cc.calc_area()
      destroy(cs)
      destroy(cc)
   end

end Complement
```

Funkcja (metoda) *calc_area_complement()* służy do obliczania różnicy pól: kwadratu o boku a i koła o średnicy a. Kwalifikator **cap** oznacza *capability*, czyli wskaźnik na obiekt – będący tu zasobem. Można tworzyć i posługiwać się tego typu wskaźnikami także w odniesieniu do maszyn wirtualnych a nawet samych operacji. Zasoby można tworzyć (i niszczyć) dynamicznie za pomocą instrukcji **create** (i **destroy**) w lokalnej przestrzeni adresowej lub na zdalnych maszynach wirtualnych. Umożliwia to kwalifikator **on**, dodawany do linii polecenia **create**. I tak

var *my_machine* : **cap vm**
var *my_resource* : **cap** *Complement*
var *difference* : **real**

my_machine := **create vm**() **on** "tulip"
my_resource := **create** *Complement*(888) **on** *my_machine*
difference := *my_resource.calc_area_complement*()

oblicza różnicę pól pomiędzy kwadratem o boku równym 888 i kołem o takiej samej średnicy poprzez utworzenie zasobu *Complement* z odpowiednim parametrem aktualnym i wykonanie jego jedynej operacji (metody). Zasób ten jest tworzony na maszynie wirtualnej *my_machine*, uprzednio umiejscowionej na komputerze o nazwie "tulip", będącym węzłem sieci lokalnej. Kwalifikator **cap** oznacza tu wskaźnik (*capability*) na maszynę wirtualną, o czym informuje predefiniowana stała **vm**. Warto zwrócić uwagę, iż ten sam mechanizm tworzenia obiektów – w sensie składniowym i pojęciowym – stosuje się symetrycznie do maszyn wirtualnych i zasobów, choć jak już wspomniano ich reprezentacja w systemie operacyjnym jest różna. Jest to efekt realizacji założenia projektowego SR: najprostsza możliwa składnia przy mnogości dostępnych mechanizmów.

3 Operacje i mechanizmy komunikacji

Tak jak zasoby są modelami obiektów, operacje są modelami akcji wykonywanych na obiektach. Deklaracje operacji mogą pojawiać się w specyfikacjach zasobów, samych zasobach oraz wewnątrz procesów. Zadeklarowane operacje mogą być wywoływane za pomocą instrukcji **call** lub **send**. W zależności od sposobu wywołania zmienia się mechanizm synchronizacji i komunikacji między procesami:

sposób wywołania	operacja	efekt
call	**proc**	wywołanie procedury (w szczególności zdalne lub rekursywne)
call	**in**	rendez-vous
send	**proc**	tworzenie procesu
send	**in**	przesłanie sygnału

3.1 Instrukcje wywołania

W SR wyróżnia się dwie instrukcje wywołania:

- **call** operacja(*lista argumentów*),
- **send** operacja(*lista argumentów*),

gdzie "operacja" jest jedną z operacji zasobu aktualnie definiowanego, reprezentującym operację polem wskaźnika *(capability)* na jakiś zasób lub wskaźnikiem na samą operację. Argumenty mogą być przekazywane przez wartość (kwalifikator **val** – domyślny), zmienną (kwalifikator **var**) oraz (zwracane) jako rezultaty (**res**).

Instrukcja **call** zakończy się, gdy wywoływana operacja wykona się (lub zostanie obsłużona) i rezultaty zostaną zwrócone albo zostanie wykryty błąd lub awaria (patrz punkt 3.5). Wykonanie **send** kończy się bezwarunkowo. Stąd **call** jest instrukcją synchroniczną a **send** asynchroniczną. Domyślnie nie ma ograniczenia na sposób wywołania operacji, można jednak takowe nałożyć, dodając do deklaracji operacji (w specyfikacji zasobu) odpowiednią restrykcję: {**call**} lub {**send**}.

3.2 Typy operacji

Operacja może być realizowana jako

- **proc** albo
- jedna lub więcej instrukcji **in**

Operacja typu **proc** jest uogólnieniem procedury. Ma ogólną składnię:

proc identyfikator_operacji(*lista parametrów*) **returns** identyfikator_rezultatu
blok
end identyfikator_operacji

gdzie "identyfikator_operacji" jest identyczny z użytym w deklaracji, znajdującej się w specyfikacji zasobu. W momencie wywołania instrukcji **proc** powstaje jej instancja, której do wykonania przydzielany jest wątek *(thread)*. W zależności od sposobu wywołania operacja wykonuje się jak proces lub procedura (w szczególności zdalna): **call** zawiesza proces-ojca do momentu obsłużenia lub przerwania instrukcji, **send** natomiast jest rodzajem rozwidlenia wątku (*lightweight fork*), jako że proces macierzysty i instancja instrukcji **proc** wykonują się współbieżnie (jako wątki).

Innym typem operacji są instrukcje **in**, mające ogólną strukturę:

in *gałąź* [] ... [] *gałąź* **ni**

Każda z *gałęzi* ma składnię i sposób wywołania podobny do **proc**, może być jednak uzupełniona o wyrażenia synchronizujące i kolejkujące:

nazwa_operacji (*lista_parametrów*) **returns** rezultat
and wyrażenie_synchronizujące **by** wyrażenie_kolejkujące $\longrightarrow$ *blok*

Instrukcja **in** zawiesza proces do momentu, gdy jedna z operacji, stanowiących gałęzie zostanie wywołana i odpowiadające jej logiczne wyrażenie synchronizujące, przyjmie wartość *TRUE*. Wówczas wykonywany jest kod zawarty w *bloku*. Gdy kilka gałęzi spełnia powyższe wymagania, jedna z nich jest wybierana niedeterministycznie. Poszczególne wywołania danej operacji są obsługiwane w kolejności, w jakiej się pojawiały, co może być zmienione za pomocą **by** – najwyższy priorytet uzyskają wtedy wywołania o najmniejszej wartości arytmetycznego wyrażenia kolejkującego. Wyrażenie to może być pominięte. Wyrażenie synchronizujące również jest opcjonalne i implicite prawdziwe w przypadku pominięcia. Co istotne oba te wyrażenia mogą odwoływać się do wartości parametrów operacji. Zależnie od sposobu wywołania (**call** lub **send**) instrukcja **in** zapewnia mechanizm synchronizacji typu rendez–vous lub przesyłania komunikatu (*message send/receipt*). Łączy więc w sobie cechy zarówno instrukcji **select** języka Ada [8] jak i strzeżonej instrukcji wejścia *(guarded input statement)* CSP [7]. Możliwość wpływania na mechanizm synchronizacji za pomocą argumentów wywołań znacznie wzmacnia siłę języka i może bardzo uprościć rozwiązania wielu problemów programistycznych (przynajmniej na poziomie logicznym).

W języku dostępne są pewne uproszczenia (skróty składniowe) umożliwiające zwięzły zapis mechanizmów komunikacji, np. instrukcja

receive operacja(*zmienne*)

odpowiada instrukcji **in**, która oczekuje na wywołanie "operacji" a następnie nadaje "zmiennym" wartości parametrów aktualnych. Zgodnie z oczekiwaniami operacja ta, wywoływana przez **send**, zapewnia asynchroniczny sposób przesyłania wiadomości, natomiast w połączeniu z wywołaniem **call** - przesyłanie synchroniczne.

Przy użyciu instrukcji **send** i **receive** można w prosty sposób zaimplementować operacje semaforowe, umożliwiające synchronizację dostępu do zmiennych dzielonych wewnątrz zasobu. Przykładowo jeśli s jest operacją zadeklarowaną:

op *s()*

instrukcje:

receive *s()*
send *s()*

symulują odpowiednio operacje semaforowe P i V. W istocie operacje zadeklarowane i wywoływane w ten właśnie sposób są wewnętrznie implementowane tak, jak gdyby były semaforami.

3.3 Inne mechanizmy komunikacji

SR dostarcza kilku instrukcji znacząco rozszerzających możliwości współpracy procesów i ich wzajemnego oddziaływania:

- **co** – umożliwia współbieżne wywoływanie i wykonywanie operacji:

  co instrukcja $\longrightarrow$ *blok instrukcji końcowych*
  // ...
  // instrukcja $\longrightarrow$ *blok instrukcji końcowych*
  oc

  Poszczególne "instrukcje" są wywołaniami pojedynczych operacji przez **call** lub **send**. Po zakończeniu obsługi każdego wywołania przewidziana jest opcjonalnie możliwość wykonania dodatkowego *bloku instrukcji końcowych* (post-processing block). Wykonywanie instrukcji **co** kończy się, gdy zostaną obsłużone wszystkie wywołania i bloki końcowe lub gdy w jednym z tych bloków wykona się instrukcja **exit**. W tym drugim wypadku niedokończone wywołania nie zostaną przerwane, ale będą się wykonywać "w tle", równolegle z wątkiem, który wywołał **co**. Jest to interesująca właściwość, zwłaszcza jeśli wyobrazimy sobie np. zreplikowaną rozproszoną bazę danych, której wszystkie pliki muszą zostać uaktualnione pomimo iż proces uaktualniający zakończył działanie po dokonaniu zmian w zaledwie kilku z nich. W każdej z wykonywanych równolegle instrukcji może być użyta zmienna sterująca, określająca ilość powtórzeń tejże instrukcji, np.

co (i:=1 **to** N) **call** *file[i].update*(values) **oc**

gdzie *file* jest tablicą wskaźników do zasobów plikowych.

- **return** – powoduje zakończenie wykonania najbardziej zagłębionej na danym poziomie operacji **in** lub **proc** (efekt po stronie obsługującego). W przypadku wywołania operacji przez **call** instrukcja ta również kończy działanie (efekt po stronie wywołującego).
- **reply** – działa podobnie jak **return**, ale pozwala operacji obsługującej istnieć dalej i wykonywać się, pomimo iż parametry **proc** zostały zwrócone do wywołującego (umożliwia to implementację tzw. mechanizmów *konwersacji* [5]).
- instrukcje sekwencyjne i iteracyjne: **if**, pętle **do** i **fa** (*for all*), które oparte są na strzeżonych instrukcjach Dijkstry [9].

3.4 Przykład

Omówione mechanizmy staną się zapewne jaśniejsze po przedstawieniu prostego przykładu. Proponujemy rozwiązanie klasycznego problemu czytelników i pisarzy: wielu czytelników może jednocześnie mieć dostęp do zasobu (czytać książkę), natomiast pisarze mają wyłączność na pracę z książką, tzn. gdy jeden z pisarzy pisze, nikt inny nie ma prawa wykonywać jakichkolwiek operacji na dzielonym zasobie. Proponowane rozwiązanie to jeden zasób SR, w którym procesy czytelników i pisarzy rywalizują ze sobą o dostęp do dzielonej zmiennej, przy czym synchronizacja tego dostępu jest zapewniona przez specjalny proces koordynujący:

```
resource book
body book()

# tu deklaracja  zmiennej dzielonej pomi'edzy procesami

  const NREADERS := 5
  const NWRITERS := 3
  var num_of_readers := 0
  var writer_active := false
  op go(i : int) {send}
  op start_reading(strd : int)         # parametry okre'slaj'a
  op end_reading(enrd : int)           # nr procesu wykonuj'acego
  op start_writing(stwr : int)
  op end_writing(enwr : int)


  process reader(r := 1 to NREADERS)
    var i : int
    receive go(i)  # oczekiwanie na sygna'l do startu
    send start_reading(r)
    # czytanie
    send end_reading(r)
```

```
    end

    process writer(w := 1 to NWRITERS)
       var i : int
       receive go(i)  # j.w. – a wi'ec wszyscy maj'a na starcie r'owne szanse
       send start_writing(w)
       # pisanie
       send end_writing(w)
    end

    process coordinator
       var who : int

       do true →
            in start_reading(who) and not writer_active →
                 write("czytelnik", who, "czyta")
                 num_of_readers++

            [] end_reading(who) →
                 write("czytelnik", who, "konczy czytanie")
                 num_of_readers--

            [] start_writing(who) and not writer_active & num_of_readers= 0 →
                 write("pisarz", who, "pisze")
                 writer_active := true

            [] end_writing(who) →
                 write("pisarz", who, "konczy pisanie")
                 writer_active := false
            ni
       od
    end


    process startup
       co ( i := 1 to NREADERS + NWRITERS )  send go(i)  oc
    end
end
```

Proces *startup* pełni funkcję startera dla "czekających w blokach" procesów–zawodników; sprawiedliwie umożliwia wszystkim równy start. Mimo iż trzy procesy pisarzy sygnalizują jednocześnie chęć dostępu do książki, proces *coordinator* realizuje zawsze tylko jedno z tych żądań.

3.5 Obsługa błędów wykonania

Systemy rozproszone, przez swą złożoność bardziej niż inne podatne są na wszelkiego rodzaju błędy i awarie swoich elementów. Ponadto ze swej definicji powinny cechować się wysokim stopniem niezawodności dostarczanych usług. Stąd potrzeba istnienia nie tylko niezawodnych narzędzi i technik tworzenia takich systemów, ale również mechanizmów wykrywania i zapobiegania skutkom nieuchronnych awarii.

W aktualnej implementacji języka SR nie są jeszcze dostępne procedury obsługi awarii sprzętu, sieci, przepełnienia pamięci czy zniszczenia zasobów, niemniej autorzy zaplanowali dwa mechanizmy rozwiązywania tego typu problemów:

- *handlers* - procedury obsługi błędów, dowiązane do instrukcji i wywoływane w momencie wystąpienia błędu realizacji jednej z nich. Po wykryciu błędu instrukcja jest wykonywana wg przepisu zadanego procedurą obsługi błędu, a po jej zakończeniu sterowanie przekazywane jest do procesu–ojca;

- instrukcja **when** umożliwia monitorowanie błędów powstających podczas komunikacji między procesami. Zapobiega np. zablokowaniu serwera podczas oczekiwania na wywołanie, które nigdy nie nastąpi na skutek awarii procesu, zasobu, maszyny wirtualnej lub procesora klienta, np.

  when failed(*source*) **send** operation(*arguments*)

4 Technika tworzenia oprogramowania w SR

4.1 Konsekwencje założeń i właściwości języka

Jak już wspomniano, SR jest językiem stosunkowo niskiego poziomu. Fakt ten, jak również założenia o możliwości dynamicznej alokacji pamięci dzielonej na różnych poziomach hierarchii (czyli tworzenia maszyn wirtualnych i zasobów) nadają środowisku SR następujące cechy:

- Każdy program w SR jest oddzielną i samodzielną jednostką, będącą zbiorem maszyn wirtualnych wraz z hierarchiczną strukturą istniejących w ich wnętrzach zasobów. Nie istnieje (jak to ma miejsce np. w Emeraldzie [10]) jądro systemu, świadczące usługi na rzecz pojawiających się wciąż na nowo aplikacji – są one w pełni samodzielne i nie mają możliwości komunikacji pomiędzy sobą.

- Struktura maszyn wirtualnych oraz moment ich utworzenia i ewentualnego usunięcia zasobów musi być zaplanowana przez projektanta aplikacji na poziomie języka. Po kompilacji i uruchomieniu programu programista ani użytownik nie mogą dodać do niego żadnego obiektu o ile taka możliwość nie została wcześniej przewidziana (np. możliwe jest utworzenie zasobu na podstawie analizy parametrów odczytanych z zewnątrz, jednak odpowiednia instrukcja warunkowa musi być "zaszyta" w kodzie programu).

- Programista musi zatroszczyć się o właściwy "harmonogram" komunikacji między procesami i wybrać spośród mnogości oferowanych przez język mechanizmów odpowiednią reprezentację dla poszczególnych elementów tego dialogu.

4.2 Właściwości aktualnej implementacji języka

Aktualnie dostępna implementacja języka działa na platformie systemu operacyjnego UNIX. Prowadzone są prace nad zupełnie samodzielnym środowiskiem, pracującym wyłącznie w oparciu o sprzęt i niskiego poziomu usługi sieciowe. Póki co, program w SR działa jako zbiór procesów UNIXa (każdy reprezentujący jedną maszynę wirtualną), wewnątrz których równoległość jest symulowana przez specjalny zbiór procedur niskiego poziomu, zwany RTS (*run-time support*). Maszyny wirtualne wymieniają dane i komunikaty przez interfejs soketowy (*socket interface*). Zasadniczo w skład otoczenia programowego środowiska SR wchodzą: kompilator, linker i RTS. Kompilator produkuje kod pośredni w C a następnie używa standardowego kompilatora C do produkcji kodu wykonywalnego. Wynika stąd cenna możliwość dołączania do programu skompilowanych procedur napisanych w języku C lub też standardowych bibliotek tego języka.

Dostępne są wynikające z istoty UNIXa mechanizmy do współpracy z tym systemem:

- Ponieważ obsługa wejścia/wyjścia związana jest w systemie UNIX z pojęciem pliku, możliwe są różnorodne operacje na plikach, takie jak tworzenie, otwieranie, zamykanie, zapis, odczyt i usuwanie.
- Zdefiniowane są literały odpowiadające standardowym unixowym plikom: stdin, stdout, stderr.
- Możliwe jest dostęp do parametrów, z jakimi uruchomiony został program w SR (*command line parameters)*

5 Podsumowanie

W powyższym artykule przedstawiono pokrótce zarys właściwości SR – nowoczesnego języka programowania równoległego i rozproszonego. Starano się pokazać, iż jest on środowiskiem umożliwiającym programowanie obiektowo zorientowane, cechującym się wieloma zaletami, a przede wszystkim wielością dostępnych mechanizmów, ich elastycznością, prostotą, transparentnością i uniwersalnością. Cechy te sprawiają, iż język ten naszym zdaniem nadaje się znakomicie zarówno do nauki programowania równoległego i rozproszonego[2] jak i tworzenia zaawansowanych oraz wymagających subtelnych i zarazem efektywnych metod synchronizacji i komunikacji programów tej klasy.

6 Literatura

[1] G.R. Andrews. Synchronizing Resources. *ACM Trans. on Prog. Languages and Systems 3*, 4 (Oct. 1981), 405-430.

[2] G.R. Andrews. The distributed programming language SR – mechanisms, design and implementation. *Software–Practice and Experience 12*, 8 (Aug. 1982),719-754.

[3] G.R. Andrews and R.A. Olsson. The evolution of the SR language. *Distributed Computing 1*, 3 (July 1986), 133-149.

[4] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin and G. Townsend. An overview of the SR language and implementation. *ACM Trans. on Prog. Languages and Systems 10*, 1 (Jan. 1988), 51-86.

[5] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings Publishing Company, 1991.

[2] Język SR, wraz z systemami Emerald i PVM jest używany przez studentów informatyki AGH podczas kursu systemów i języków równoległych i rozproszonych

[6] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in practice.* Benjamin/Cummings Publishing Company, 1992.

[7] C.A.R. Hoare,*Communicating Sequential Processes.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[8] U.S. Department of Defence, *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A, 1983.

[9] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[10] R.K. Raj, E.D. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson and E.Jul. Emerald: A General-Purpose Programming Language. *Software – Practice and Experience 21*, 1 (Jan. 1991), 91-118