STUDIA INFORMATICA 2002
Volume 23 Number 2A (48)

Przemysław SZMAL
Politechnika Śląska, Instytut Informatyki

SAMOREPLIKACJA PROGRAMÓW – CYKLE POKOLENIOWE I EWOLUCYJNE[1]

Streszczenie. Podobnie jak organizmy biologiczne, niektóre programy wykazują zdolność do samoreplikacji. Cykle przekształceń programów w ramach ich wytwarzania i samoreplikacji częściowo się pokrywają. W przebiegu procesów znaczącą rolę odgrywa interakcja ze środowiskiem, będąca jednym z czynników powodujących ewolucję postaci lub działania programu. W dyskusji nad samoreplikacją programów warto uwzględnić aplikacje wieloplikowe.

PROGRAM SELF-REPLICATION – GENERATION AND EVOLUTIONAL CYCLES

Summary. Like biological organisms, some programs exhibit self-replicating capacity. Transformation cycles applied during program production and self-replication partially coincide. During the processes, interactions with the environment play an essential role. They can – among other factors – cause evolutionary changes in program form and operation. It is worth while to extend discussion on program self-replication on multi-file applications.

1. Wstęp

W ostatnich latach wiele uwagi poświęca się zagadnieniom samoreplikacji programów. Samoreplikacja może być rozpatrywana z wielu różnych punktów widzenia, od filozoficzno-teoretycznego po ściśle użytkowy. W ramach pierwszego rozważa się możliwość tworzenia sztucznych struktur i układów wykazujących cechy charakterystyczne dla organizmów

[1] Opracowanie wykonano w ramach projektu badawczego 7 T11C 017 21 finansowanego przez Komitet Badań Naukowych w latach 2001 – 2004

żywych, w tym ich zdolność do wytwarzania nowych, w wysokim stopniu do nich podobnych egzemplarzy. Ostatni obejmuje m.in. problemy związane z patologią wirusów komputerowych i próbami jej opanowania. Zwięzłe wprowadzenie w te zagadnienia, jak też wykaz i omówienie ważniejszych pozycji literaturowych im poświęconych można znaleźć w [1]. Sporo informacji na temat wirusów zawiera [2]. W niniejszym artykule samoreplikacja jest rozpatrywana jako specyficzny sposób powstawania programów, jest ona zestawiona z klasycznym sposobem wprowadzania programów do systemu.

Zasady, zgodnie z którymi przebiega samoreplikacja programów w systemach informatycznych, przedstawiono w pracy [3]. Przyjmuje się, że w obrębie systemu istnieje pula, w której są przechowywane programy przygotowane do wykonania, w tym również programy samoreplikujące się, oraz środowisko, umożliwiające wykonywanie programów. Samoreplikacja wiąże się z następującym scenariuszem zdarzeń:

1. W punkcie wyjścia w puli istnieje pewna liczba programów.
2. Obecność programu w puli obliguje system do aktywacji programu na obowiązujących w systemie zasadach.
3. Po zaktywizowaniu programu wytwarza on swoją kopię i powoduje wprowadzenie jej do puli. Gdy to nastąpi, kopia staje się pełnoprawnym programem. W ten sposób następuje podtrzymanie łańcucha zdarzeń: w odpowiednim czasie kopia może być przez system zaktywizowana i wytworzyć kolejną kopię.

W zależności od rozpatrywanej kategorii programów ich pula może być zorganizowana w pamięci operacyjnej komputera lub w pamięci zewnętrznej, może też być skojarzona z węzłem sieci Internet lub z całą siecią.

W związku z przedstawionym scenariuszem nasuwa się kilka uwag.

Warto się zastanowić, jak należy rozumieć określenie *program*. Problem omawiamy w p. 2 i wracamy do niego ponownie w p. 5.

Należy zauważyć, że scenariusz odnosi się do zaostrzonego wariantu samoreplikacji, który określimy jako *samoreplikację aktywną*. W praktyce możemy mieć również do czynienia z *samoreplikacją pasywną*, w ramach której samoreplikujący się program w trzecim kroku scenariusza nie wykonuje w całości, a tylko **inicjuje** akcje, w których wyniku po jakimś czasie powstaje jego kopia i zostaje wprowadzona do puli. Akcje te składają się na pewien cykl, który określimy jako *cykl pokoleniowy*. Jego organizację dyskutujemy w p. 3.

Rozpatrując cykl pokoleniowy warto pamiętać, że działania programu są uwarunkowane jego treścią i bieżącym stanem. Stan programu od momentu jego aktywacji nieustannie zmienia się; na te zmiany istotny wpływ mają interakcje programu ze środowiskiem, w ramach którego jest on wykonywany. Zmianom może podlegać również treść programu.

Sam fakt wygenerowania kopii, jak i moment, w którym ono nastąpi, w ogólnym przypadku może zależeć od tych interakcji, bieżących lub skumulowanych.

Omówienia wymaga też kwestia kopii programu powstającej w wyniku cyklu pokoleniowego. Można rozumieć przez nią kopię dokładną (bliskie temu jest określenie *replika*). Wydaje się, że można rozszerzyć dyskusję na przypadek kopii przybliżonej, podobnej pod pewnymi względami do oryginału, lecz z nim nieidentycznej. Zagadnienia zmian postaci i zachowania się programu, stanowiące podstawę do wyróżnienia *cykli ewolucyjnych* w programach kolejnych pokoleń, omawiamy w p. 4.

2. Jednostki samoreplikacyjne

Tym, co faktycznie powiela się w procesach samoreplikacji, są wydzielone pod względem logicznym lub fizycznym jednostki programowe, które w tym artykule określamy jako *jednostki samoreplikacyjne* lub *SR-jednostki*. Każda SR-jednostka zawiera fragment odpowiedzialny za wytworzenie jej kopii, jak też fragment wprowadzający kopię do systemu. Zwykle zawiera też fragmenty wykonujące inne czynności. Obsługa tych ostatnich może być złożona, kod z nią związany może wypełniać znaczną część jednostki.

W określonych okolicznościach SR-jednostka może stanowić tylko fragment większego programu. Ma to miejsce np. w przypadku wirusów komputerowych. Kiedy indziej jednostka może obejmować cały program. Szczególny rodzaj stanowią programy należące do kategorii określanej jako *quine* [4]. Działanie takiego programu – zapisanego w określonym języku programowania – sprowadza się do wygenerowania na wyjściu jego własnego tekstu w postaci źródłowej bez korzystania z danych wejściowych. Programy typu *quine* opracowano już w wielu językach – zarówno bardzo popularnych, w tym *C/C++*, *Java*, *Pascal*, *Lisp* i in., jak i mniej znanych.

Dla ułatwienia dalszej dyskusji wprowadzamy dodatkowe pojęcia *efektywnej SR-jednostki* i *projednostki SR*. Pierwsze oznacza jednostkę w postaci, w której jest ona wykonywana (interpretowana). Drugie odnosi się do zapisu jednostki w określonym języku programowania, jak też do każdej formy pośredniej, z jaką można mieć do czynienia przy przechodzeniu od postaci źródłowej do postaci wykonywalnej. W wyjątkowych przypadkach języków bezpośrednio interpretowalnych, jak np. *Lisp*, projednostka może być tożsama z efektywną SR-jednostką.

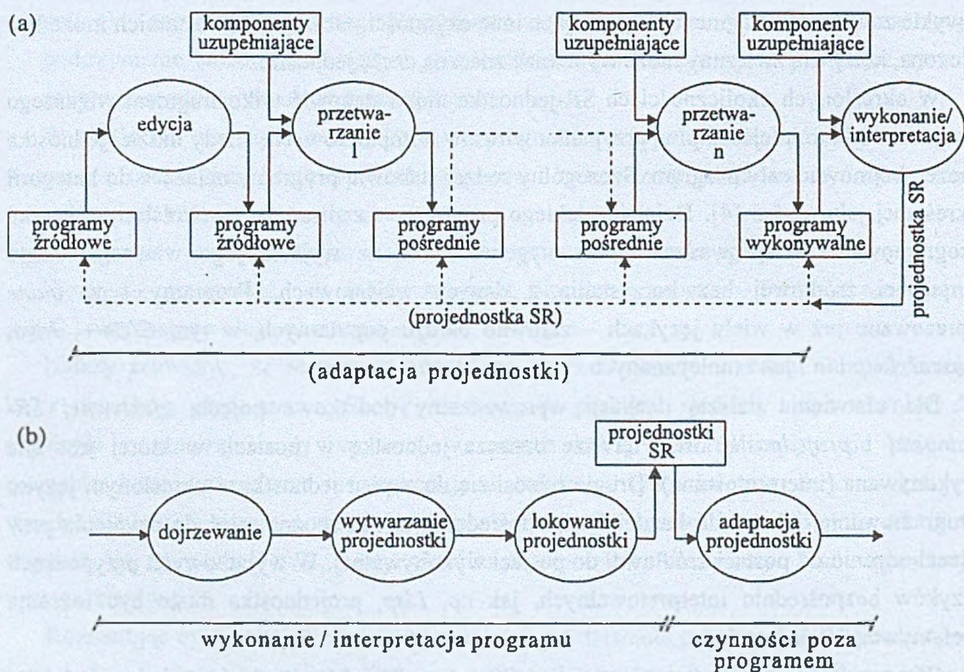
W uzupełnieniu umówimy się określać *SR-jednostkę* (efektywną lub projednostkę) jako *samodzielną*, gdy jest lub może być zapisana w oddzielnym pliku, który nie zawiera elementów niezależnych od jednostki. W przeciwnym przypadku *jednostka* jest *niesamodzielna*. Gdy treść jednostki zawiera się w całości w jednym pliku, mamy do

czynienia z *jednostką prostą,* gdy w większej liczbie plików – z *jednostką zagregowaną.* Jednostki zagregowane są dodatkowo dyskutowane w p. 5.

3. Samoreplikacja oglądana z perspektywy pojedynczego pokolenia: cykl pokoleniowy

Efektywna SR-jednostka może się pojawić w systemie na dwa sposoby: w ramach zwykłego cyklu rozwojowego lub w ramach cyklu samoreplikacyjnego, tu określanego jako cykl pokoleniowy. W obrębie obu cykli pewne etapy są lub mogą być wspólne. Diagramy obrazujące przepływ danych w ramach obu cykli pokazano na rys. 1.

Zwykły cykl rozwojowy prowadzi od postaci źródłowej, gdzie projednostka SR wraz z ewentualnymi innymi fragmentami programowymi jest przedstawiana w wybranym języku programowania, przez postacie pośrednie pojawiające się w procesie tłumaczenia (kompilacji) i scalania statycznego, do postaci wykonywalnej, a następnie do postaci efektywnej. Postać efektywna w trakcie wykonania może podlegać pewnym zmianom. Są one związane



Rys. 1. Zwykły cykl rozwojowy (a) i cykl pokoleniowy (b) programu samoreplikującego się
Fig. 1. Normal developmental cycle (a) and generation cycle (b) of a self-replicating program

z dynamicznym dołączaniem współpracujących fragmentów programowych, jak też z samomodyfikacją kodu programu.

Co do **cyklu pokoleniowego**, z uwagi na wygodę opisu przyjmiemy, że zaczyna się on z chwilą zaktywizowania programu z SR-jednostką macierzystą i kończy w chwili bezpośrednio poprzedzającej zaktywizowanie programu z SR-jednostką potomną. Cykl obejmuje trzy fazy: dojrzewania, generacji projednostki SR i transformacji projednostki.

W fazie dojrzewania program zawierający SR-jednostkę macierzystą wykonuje czynności poprzedzające wytworzenie projednostki potomnej. Mogą one m.in. obejmować scharakteryzowane wyżej zmiany postaci efektywnej programu, a w nim także SR-jednostki macierzystej.

W fazie generacji projednostki SR-jednostka macierzysta wytwarza projednostkę potomną, zapisuje ją w przyjętym języku i formacie i lokuje w systemie.

Wytworzenie projednostki w prostym przypadku polega na mechanicznym skopiowaniu treści SR-jednostki macierzystej. W bardziej zaawansowanych przypadkach zapis SR-jednostki potomnej powstaje w wyniku interpretacji treści jednostki macierzystej. W przypadku wirusów komputerowych elementem towarzyszącym interpretacji jest szyfrowanie zapisu. Czasami – jak w przypadku niektórych programów typu *quine* (por. wyżej p. 2) – generator jest po prostu zaszyty w tekście SR-jednostki. Należy zaznaczyć, że możliwe jest zorganizowanie wytwarzania jednostki potomnej w taki sposób, by uwzględniała ona bieżącą postać jednostki macierzystej (wariant progresywny) lub też postać wyjściową, istniejącą na początku fazy dojrzewania (wariant konserwatywny).

Ulokowanie jednostki w systemie oznacza – w przypadku jednostek samodzielnych – utworzenie i wprowadzenie do systemu pliku zawierającego treść jednostki. W przypadku jednostek niesamodzielnych (m.in. wirusów) treść jednostki jest w drodze edycji dołączana do istniejącego pliku wymaganego typu.

W wyniku fazy generacji powstaje projednostka w postaci charakterystycznej dla jednego z etapów zwykłego cyklu rozwojowego. **W fazie adaptacji projednostki** jednostka przechodzi pozostałe etapy cyklu rozwojowego, przyjmuje postać wykonywalną, po czym system bezpośrednio lub pośrednio jest informowany o tym, że nowa jednostka jest gotowa do uruchomienia. Tym samym może się rozpocząć kolejny cykl pokoleniowy.

Przejścia między etapami cyklu mogą być stymulowane przez SR-jednostkę macierzystą w czasie, gdy jest ona aktywna. W przypadku wirusów komputerowych (lecz nie tylko) jednostka ta jedynie inicjuje cykl. Impulsy do rozpoczęcia kolejnych etapów pochodzą od użytkownika lub programów systemowych wykonujących określone rutynowe operacje. Zdarza się, że impulsy są wysyłane nieświadomie, wbrew interesom i intencjom użytkownika. Sam cykl jest nieraz bardzo rozciągnięty w czasie.

Najprostszy cykl pokoleniowy zamyka się na poziomie postaci efektywnej – nie wymaga tworzenia plików. W jego ramach w innym obszarze pamięci operacyjnej komputera jest tworzony nowy (skopiowany) ciąg instrukcji programowych, oddawany następnie w gestię mechanizmu zarządzającego przydziałem czasu procesora. Jedna ze skrajnych odmian samoreplikacji polega na utworzeniu nowego procesu na bazie istniejącego już w pamięci operacyjnej i niepowielonego fragmentu programowego. Przypadki samoreplikacji programów realizowanej w pamięci operacyjnej są – oprócz innych – dyskutowane w [5].

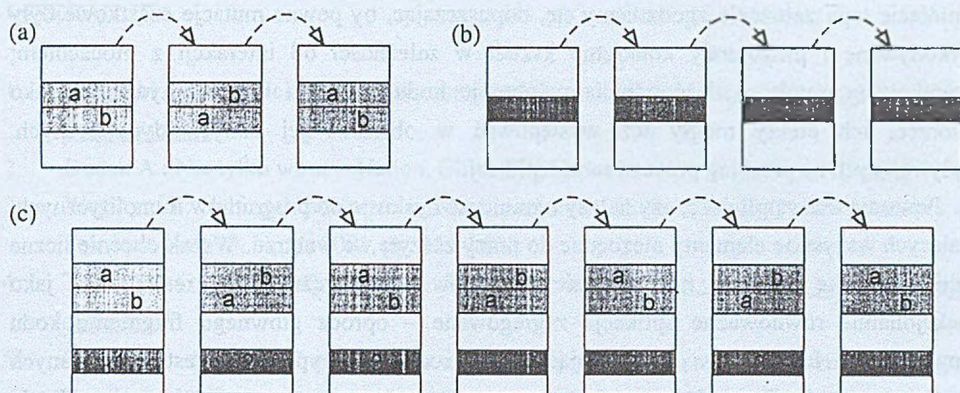
4. Samoreplikacja oglądana w perspektywie łańcucha pokoleń: cykl ewolucyjny

Jak wspomniano wyżej, kopiowanie SR-jednostki dokonywane w ramach samoreplikacji nie musi sprowadzać się do mechanicznego powielania kodu programu. W typowym przypadku wytwarzanie kopii jest związane z interpretacją kodu, traktowanego jako specyficzny zestaw danych. Takie podejście pozwala na kształtowanie własności jednostki potomnej w szerokim zakresie.

Jeżeli nowo utworzona jednostka programowa jest identyczna z jednostką macierzystą, można mówić o *samoreplikacji czystej*. W przeciwnym razie mamy do czynienia z *samoreplikacją z mutacjami*. Można mówić o *mutacjach cząstkowych*, związanych z pojedynczymi zmianami w tekście (bez przesądzania repertuaru tych zmian), oraz o *mutacji wypadkowej*, stanowiącej sumaryczny efekt mutacji cząstkowych. Mutacje cząstkowe mogą polegać np. na usunięciu lub wstawieniu pewnego fragmentu programu, zastąpieniu jednego fragmentu innym lub na transpozycji dwóch fragmentów.

Można wyróżnić kilka charakterystycznych rodzajów mutacji.

Mutację określimy jako *zamkniętą*, jeżeli w skończonej liczbie pokoleń potomnych przynajmniej raz wystąpi jednostka identyczna z wyjściową jednostką macierzystą. Nawroty mogą następować cyklicznie (z cyklem regularnym lub nieregularnym) lub acyklicznie. *Nawrót cykliczny* następuje, gdy poszczególne mutacje cząstkowe są sterowane przez pewien algorytm deterministyczny (w szczególnym przypadku – pseudolosowy). *Nawroty acykliczne* mogą mieć miejsce wtedy, gdy wybór niektórych mutacji cząstkowych jest dokonywany losowo. Warto zauważyć, że – z formalnego punktu widzenia – z mutacją zamkniętą mamy do czynienia m.in. w przypadku samoreplikacji czystej. Przeciwieństwem mutacji zamkniętej jest *mutacja otwarta*, w której niezależnie od długości łańcucha pokoleń nie ma powrotu jednostek do postaci wyjściowej. Przykłady kilku mutacji zamkniętych przedstawiono schematycznie na rys. 2.



Rys. 2. Przykładowe zamknięte cykle ewolucyjne: (a) cykl bazujący na transpozycji 2 fragmentów, (b) cykl bazujący na 3-krotnej wymianie fragmentu, (c) złożenie mutacji cząstkowych (a) i (b)

Fig. 2. Sample recurring evolutional cycles: (a) cycle based on transposition of 2 fragments, (b) cycle based on triple fragment exchange, (c) composition of partial mutations (a) and (b)

Z mutacjami postaci jednostek może się wiązać zmiana (mutacja) sposobu ich działania. Gdy mutacja powoduje taką zmianę, *mutację* uważa się za *semantycznie istotną* (znaczącą). Szczególny rodzaj mutacji stanowią *mutacje semantycznie nieistotne* (nieznaczące), przy których – mimo zmiany treści jednostki – zachowanie jednostki potomnej nie różni się od macierzystej.

Rozważmy dla przykładu dwa fragmenty programowe zapisane w języku C:

if (b) a = a + 1; else a = a – 1; (1)

if (!b) a = a – 1; else a = a + 1; (2)

Mutacja polegająca na zastąpieniu fragmentu (1) przez (2) lub na odwrót jest semantycznie nieistotna.

Mutacje semantycznie nieistotne stanowią charakterystyczną cechę grupy wirusów określanych jako polimorficzne [2, s. 19]. Istotność bądź nieistotność są cechą, którą można rozpatrywać w odniesieniu do wszystkich omówionych rodzajów i zakresów mutacji postaciowych: cząstkowych i wypadkowych, otwartych i zamkniętych.

5. Aplikacje samoreplikujące się

Jeden z silniejszych warunków nakładanych przez niektórych autorów (np. [4]) na programy podlegające samoreplikacji głosił, że SR-jednostka przy wytwarzaniu swojej kopii wszystkie elementy do tego potrzebne powinna znajdować w swoim wnętrzu. Na częściowe

ominięcie tego założenia zgodziliśmy się, dopuszczając, by pewne mutacje cząstkowe były wykonywane i przybierały konkretny kształt w zależności od interakcji z otoczeniem; interakcje te mogły znaleźć odbicie w obrębie kodu jednostki wykorzystywanego jako wzorzec, ich efekty mogły też występować w obszarze jej danych dynamicznych, wpływających na przebieg procesu samoreplikacji.

Powstaje też wątpliwość, czy należy ograniczać dyskusję do programów monolitycznych, mających wszystkie elementy niezbędne do pracy zaszyte we wnętrzu. Wszak obecnie liczne aplikacje, które mogłyby mieć postać programów monolitycznych, są realizowane jako funkcjonalnie równoważne aplikacje zagregowane – oprócz głównego fragmentu kodu korzystają z fragmentów uzupełniających w rodzaju skryptów i zestawów danych konfiguracyjnych. Te dodatkowe fragmenty są przechowywane w oddzielnych plikach. Zdolność do samoreplikacji może wykazywać cała aplikacja zagregowana, lub kilka jej fragmentów występujących wspólnie. Na bazie tej obserwacji wprowadziliśmy w p. 2 pojęcie SR-jednostek zagregowanych – wieloplikowych. W ramach dalszej dyskusji, która wykracza poza ramy niniejszego opracowania, można by wprowadzić klasyfikację aplikacji biorąc pod uwagę np. sposób rozlokowania składowych aplikacji w systemie czy też to, czy kopiowaniu podlegają wszystkie elementy, czy tylko niektóre; w tym ostatnim przypadku część plików niewymagająca kopiowania mogłaby być uznana za stały, niezbędny w procesie samoreplikacji element środowiska.

6. Uwagi końcowe

W artykule zostały przedstawione problemy związane z samoreplikacją programu. Zostało wprowadzone pojęcie jednostki samoreplikacyjnej, ułatwiające traktowanie w sposób ujednolicony powielanie się programów monolitycznych, wydzielonych fragmentów programowych i aplikacji zagregowanych. Rozpatrzono cykl pokoleniowy, w ramach którego następuje wytworzenie nowej jednostki z inicjatywy jednostki macierzystej i ewentualnie pod jej kontrolą. Wyeksponowano związek cyklu pokoleniowego ze zwykłym cyklem rozwojowym programów. Przedstawiono możliwe cykle ewolucyjne, widoczne w jednostkach należących do kolejnych pokoleń, gmatwające się w wyniku nakładania się stosunkowo prostych mutacji cząstkowych. Jako jeden z możliwych kierunków dalszych rozważań wskazano samoreplikację aplikacji zagregowanych.

LITERATURA

1. Sipper M.: The artificial self-replication page. Strona internetowa http://www.cs.bgu.ac.il/~sipper/selfrep/
2. Dudek A.: Nie tylko wirusy. Helion, Gliwice 1998.
3. Węgrzyn S.: Self-replications of informatic systems. Bulletin of the Polish Academy of Sciences, Vol. 49, No. 1, 2001, pp. 161-165.
4. Thompson G. P.: The Quine page. Strona internetowa http://www.nyx.net/~gthompso/quine.htm
5. Dobosz K.: A practical approach to self-replication of programs. Archiwum Informatyki Teoretycznej i Stosowanej (w druku).

Recenzent: Prof. dr inż. Stefan Węgrzyn

Wpłynęło do Redakcji 7 maja 2002 r.

Abstract

In the paper some problems related to self-replication of programs are presented. The paper refers to the self-replication scenario outlined in [3].

In Section 2 we introduce a notion of self-replication unit (SR-unit). SR-unit is a distinguished program entity, which is actually subject to duplication. Thanks to the notion it is possible to consider in a uniform manner the duplication of monolithic programs, distinct program fragments (a typical case for computer viruses!) and aggregated applications as well. A relevant SR-unit classification has been introduced.

In Section 3 we consider the generation cycle (meant as the full transition from a mother unit to a daughter one), in the framework of which a new unit is produced on the mother unit's initiative and may be under its control. After each cycle phase the unit takes some characteristic form. The end form is an effective one, executed or interpreted. We expose connections between the generation cycle and the normal cycle of program building. The connections are schematically shown in Fig. 1.

In Section 4 we consider changes in the framework of units of successive generations. Changes may be considered as mutations – partial, located in a specific point of program, or resultant, which are their combination. Mutations can form an evolutional cycle – open or recurring, the latter being periodic or aperiodic. In Fig. 2 we show some sample periodic

recurring cycles. We also consider mutations, which are possibly inessential from the point of view of program's semantics. Sample mutation of this type we show in formulas (1) and (2).

In Section 5, as one of possible topics for further considerations, we point out the problems of self-replication of aggregated applications, composed of multiple files of different types with programs and associated data.