УДК 004.457 Дата подачи статьи: 04.04.18
DOI: 10.15827/0236-235X.123.469-474 2018. Т. 31. № 3. С. 469–474

Выбор пакетного менеджера для многоверсионных приложений

В.А. Галатенко [1], *д.ф.-м.н., зав. сектором автоматизации программирования, galat@niisi.ras.ru*
М.Д. Дзабраев [1], *инженер, mdzabraev@niisi.ras.ru*
К.А. Костюхин [1], *к.ф.-м.н., старший научный сотрудник, kost@niisi.ras.ru*

[1] *Федеральный научный центр Научно-исследовательский институт системных исследований РАН, г. Москва, 117218, Россия*

Все разработчики ПО рано или поздно сталкиваются с проблемой создания и распространения дистрибутивов их программных продуктов. При этом необходимо учитывать возможности по поддержке уже существующих изделий, то есть замене старых дистрибутивов новыми. Используя качественное средство для создания дистрибутивов, разработчики получают возможность распространять свои изделия на более широкий спектр инструментальных платформ, а также осуществлять необходимую и своевременную поддержку этих изделий.

Авторы статьи ограничиваются рассмотрением UNIX-подобных систем, в большинстве которых в настоящий момент распространены такие средства создания дистрибутивов (пакетные менеджеры), как dpkg, yum. Эти пакетные менеджеры функционируют в соответствии со стандартной концепцией установки программ в UNIX. Стандартная концепция подразумевает, что программы будут устанавливаться в стандартные каталоги, такие как /usr/bin, /usr/local/bin и так далее. При обновлении программы (пакета) обычной практикой является замещение старых файлов новыми. Стратегия замещения такого рода может носить деструктивный характер. Имеется в виду ситуация, при которой после обновления ПО некоторые программы или библиотеки перестают работать. Не исключено, например, что после обновления может перестать работать и сам пакетный менеджер. Часто пользователь оказывается в ситуации, когда для поддержки совместимости ему необходимы старые версии ПО. В этом случае приходится прибегать к практике сборки программ и библиотек из исходных текстов и ручной установки, такой как make install. Установка такого рода является необратимой и очень опасной, поскольку в этом случае велика вероятность удаления или замещения файлов, находящихся под контролем пакетного менеджера.

В качестве решения описанных проблем предлагается пакетный менеджер NIX [1]. Основным достоинством этого менеджера является то, что полностью исключается деструктивное воздействие с его стороны. Это достигается путем установки каждого пакета в изолированную локацию, при этом всю ответственность за область видимости того, что было установлено, берет на себя пакетный менеджер.

Ключевые слова: *пакетный менеджер, NIX, установка программ, дистрибутив, версионность.*

В настоящее время мир программно-аппаратного обеспечения (как специального, так и общего назначения) стремительно изменяется. Несколько раз в год выходят обновления, зачастую критические, заставляющие разработчиков ПО своевременно реагировать на них. Кроме того, гибкая (agile) методика разработки ПО стала по сути стандартом de facto.

Все это привело к тому, что многие разработчики вынуждены создавать и поддерживать одновременно несколько версий своих продуктов как для соответствия новым требованиям пользователей, так и для обеспечения обратной совместимости со старыми версиями других приложений и/или библиотек, также используемых в проекте.

В ОС типа Unix традиционная схема развертывания новой версии приложения состоит в замене старых файлов приложения, расположенных по стандартным путям, на новые. При этом старые файлы можно сохранить в другом месте, чтобы восстановить предыдущую версию приложения в случае, если новая не будет работать. Добиться одновременной работы старой и новой версий приложения бывает очень сложно, необходимо вручную осуществлять множество административных действий по конфигурированию системы.

Соответственно, классические пакетные менеджеры, такие как yum, не предоставляют удобных путей решения проблемы одновременной доступности нескольких версий одного и того же приложения. В то же время сравнительно недавно появились пакетные менеджеры, ориентированные на решение указанной проблемы. Наиболее известным из них и стремительно набирающим популярность является пакетный менеджер NIX.

NIX предназначен для создания дистрибутивов, не конфликтующих друг с другом (и с другими версиями одного и того же дистрибутива) по зависимостям в инструментальной файловой системе. Это означает, что устраняется риск потери работоспособности старых версий приложения, а также других приложений, работающих в указанной инструментальной системе и зависящих от устанавливаемого приложения.

Далее пакетный менеджер NIX и технология создания в нем дистрибутивов рассматриваются более подробно.

Язык программирования NIX

Для пакетного менеджера NIX [1] разработан одноименный функциональный язык программирования. Он не является языком общего назначения и предназначен именно для создания пакетов.

В рамках NIX каждый пакет – это функция. Тело функции описывает процедуру сборки и уста-

новки пакета. На вход функция получает свои зависимости. Вычисление функции равнозначно конфигурации, сборке и установке пакета.

Язык NIX обладает очень важным свойством – прозрачностью ссылок (referential transparency [2]). Оно означает, что выражение языка всегда вычисляется с одним и тем же результатом. Из этого следует, что вычисление функции с фиксированными аргументами всегда будет приводить к одному и тому же результату. Таким образом, если функция вызывалась с одними и теми же аргументами на разных платформах, результат установки будет одинаковым.

Иногда создателям пакетов приходится прикладывать большие усилия, чтобы сборка пакета при одинаковых зависимостях давала один и тот же результат. Для этого приходится снабжать пакет так называемыми заплатами (патчами, от англ. patch).

С подробным описанием языка NIX можно ознакомиться в [3, 4], эффективным с точки зрения изучения языка может оказаться интерактивный курс [5].

Как и во многих функциональных языках, в языке NIX отсутствуют декларации, а есть только выражения (expressions). Выражением называется единица языка, которая может быть упрощена до конкретного значения. Например, $x + 5$ можно упростить до одного числа, а $x = 5$ нет, конструкция

```
if (y) { // ветвление с одной ветвью
 // что-то вычисляется
}
```

тоже не выражение.

Функции в языке NIX являются лямбда-выражениями и могут принимать только один аргумент на свой вход. Функция состоит из названия единственного аргумента, тела и двоеточия, которое разделяет аргумент и тело, например, *x*: *x* * *x* + 1. Чтобы вычислить функцию на значении *y*, необходимо записать его справа от функции: (*x*: *x* * *x* + 1)*y*.

Чтобы ветвление на основе *if* являлось выражением, оно должно содержать как true-ветвь, так и false-ветвь. Например, функция, возвращающая абсолютное значение *x*: *if* (*x* > 0) *then x else –x*.

В языке NIX отсутствует оператор присваивания, однако существует конструкция языка, позволяющая назначить выражение на имя переменной, но сделать это можно только один раз. Например, выражение

```
let
 y = 5;
 f = x: x*x;
in
 f f y
```

В языке NIX есть следующие типы данных: строка, целое число, список, множество (пары ключ-значение/словарь/хэш-таблица), пути (например /home/user1), булевы значения, null, функция. Множества записываются в таком формате:

```
{
    x = 1;
    y = 2;
        g = u : v: u + v;
}
```

Множество состоит из записей. После каждой записи должна стоять точка с запятой. Каждая запись состоит из имени, значения и знака =, отделяющего имя от значения. Значением элемента списка может являться функция. Таким образом получается конструкция, похожая на объект. Для обращения к значению элемента множества используется точка:

```
let
    s = {x = 1; y = 2;};
in
    s.x + s.y
```

Если предполагается, что функция должна принимать множество, ее можно записать в следующем виде: {*x*, *y*}: *x* + *y*.

Таким образом, выражение ({*x*, *y*}: *x* + *y*) {*x* = 1; *y* = 2;} будет вычислено с целочисленным результатом, равным трем.

Множества могут быть обычными и рекурсивными. В рекурсивных множествах допускается, чтобы один элемент множества вычислялся на основе его другого элемента. Для создания рекурсивного множества необходимо поместить ключевое слово *rec* перед определением множества, например *rec* {*x* = 1; *y* = *x* + 1;}.

Определение списка состоит из открывающей квадратной скобки, элементов списка, разделенных пробелами, и закрывающей квадратной скобки, например [1 “123” true].

Строки заключаются либо в двойные кавычки “123”, либо в пару одинарных ‘’456’’.

Внутри строки можно использовать конструкцию “${expr}”, где expr является произвольным выражением языка, которое вычисляется к строковому значению, например:

```
rec {
    version = “8.1”;
    name = “gdb-${version}”;
}
```

Приведенных сведений о языке NIX достаточно для понимания того, что такое пакет в NIX и как его написать самому.

Пакет в NIX

Пакетом в NIX можно назвать произвольное выражение, которое вычисляется к деривации (derivation). Деривация, с точки зрения языка NIX, является множеством с некоторым набором атрибутов. Для пользователя важно понимать, как писать выражения, которые будут вычисляться к деривациям. В NIX пакеты принято оформлять в виде функций в формате {*dep*1, *dep*2, … }: *body*. body является выражением, которое на основе аргументов *dep*1, *dep*2, … формирует деривацию. Для формирования дериваций, как правило, используются вспомогательные инструменты (generic builders).

Среда NIX предоставляет такие инструменты для GNU Autotools, для пакетов языка Python, пакетов NodeJS и многих других.

Приведем пример NIX-пакета для программы Midnight Commander:

```
{stdenv, fetchurl, pkgconfig, glib, gpm, file, e2fsprogs,
 libX11, libICE, perl, zip, unzip, gettext, slang,
 libssh2, openssl}:

stdenv.mkDerivation rec {
 name = "mc-${version}";
 version = "4.8.19";

 src = fetchurl {
    url = "http://www.midnight-commander.org/downloads/${name}.tar.xz";
    sha256 = "1pzjq4nfxl2aakxipdjs5hq9n14374ly1l00s40kd2djnnxmd7pb";
 };

 nativeBuildInputs = [ pkgconfig ];

 buildInputs = [ perl glib slang zip unzip file gettext libX11 libICE
    libssh2 openssl ] ++
 stdenv.lib.optionals (!stdenv.isDarwin) [ e2fsprogs gpm ];

 configureFlags = [ "--enable-vfs-smb" ];

 postFixup = ''
    # remove unwanted build-dependency references
    sed -i -e "s!PKG_CONFIG_PATH=''${PKG_CONFIG_PATH}!PKG_CONFIG_PATH=$(echo "$PKG_CONFIG_PATH" | sed -e 's/./0/g')!" $out/bin/mc
 '';
}
```

Приведенное выражение является функцией. Она получает на вход свои зависимости. Тело функции – выражение, которое формируется из вызова функции stdenv.mkDerivation, в качестве аргумента вызываемой функции выступает рекурсивное множество *rec* { … }.

Внутри этого рекурсивного множества содержится вся необходимая для сборки информация. Атрибут *name* является обязательным при работе с функцией stdenv.mkDerivation.

Далее идет атрибут *version*, который используется для формирования атрибута *name*.

Затем обязательный аргумент src. Значение данного атрибута вычисляется при помощи функции fetchurl. Значение атрибута src – это путь до архива с исходными текстами для Midnight Commander.

Функция fetchurl получает на вход множество с двумя атрибутами. Первый атрибут определяет адрес, откуда будет скачан архив. Атрибут *sha*256 представляет контрольную сумму, которая должна соответствовать скачанному архиву.

Атрибут *nativeBuildInputs* является списком зависимостей пакета, которые будут использоваться при сборке пакета и не будут использоваться во время выполнения.

buildInputs определяет зависимости, которые будут использоваться во время выполнения. Операцией ++ обозначается конкатенация списков. Выражение stdenv.lib.optionals (!stdenv.isDarwin) [e2fsprogs gpm] вернет список [e2fsprogs gpm], если инструментальная платформа [6] будет принадлежать семейству Darwin, в противном случае будет возвращен пустой список.

configureFlags представляет список строк, каждая из которых будет передана как аргумент скрипту ./configure.

Перед описанием атрибута postFixup опишем фазы установки пакета. Существует стандартный набор фаз, через которые проходит пакет перед установкой. Имена стандартных фаз [7] следующие: unpack, patch, configure, build, check, install, fixup, installCheck, distribution.

На стадии unpack осуществляется распаковка архива с исходными текстами, на стадии patch к исходным кодам применяются патчи, если они были перечислены в атрибуте patches множества-аргумента функции stdenv.mkDerivation. Каждую из фаз по необходимости можно переопределить, также имеется возможность выполнения произвольного кода на языке shellscript перед/после каждой фазы. Для этого применяются специальные обработчики. В приведенном пакете выполняется обработчик postFixup, некоторым образом корректирующий бинарный файл.

Репозиторий nixpkgs

Все выражения, представляющие собой пакеты, хранятся в репозитории nixpkgs [8] на github. Все пакеты хранятся в виде одного большого выражения. nixpkgs не представляется одним файлом: части большого выражения разнесены по файлам и каталогам. Точкой входа в выражение является файл default.nix, который находится в корневом каталоге репозитория.

Когда пользователь инициирует установку пакета, запускается процесс первичного вычисления nixpkgs, после чего nixpkgs преобразуется в иерархическую структуру, состоящую из множеств. На первом уровне оказываются обычные пакеты типа nixpkgs.gcc и nixpkgs.mc. На второй уровень попадают пакеты, имеющие некоторую общность, например, nixpkgs.python27Packages является множеством, которое содержит пакеты для версии Python 2.7. В частности, это множество содержит пакет nixpkgs.python27Packages.django.

После установки пакетного менеджера NIX на машину каждого пользователя сохраняется выражение nixpkgs, по сути являющееся не более чем набором рецептов того, как собрать то или иное ПО. Никто не мешает сделать копию репозитория nixpkgs, внести в нее свои правки и работать с ней. Более того, можно написать полностью свое выражение со своим набором рецептов сборки и работать с этим выражением. Единственное условие – на первом уровне выражения должно оказаться множество.

Репозиторий nixpkgs представляет нестабильные выражения. Нестабильность означает, что какой-либо пакет может не собраться или для него может не оказаться заранее откомпилированных бинарных файлов. После тестирования нестабильного репозитория nixpkgs его текущее состояние перемещают в nixpkgs-channels.

Один из самых важных вопросов заключается в том, как много пакетов сейчас в nixpkgs и сопоставимо ли их количество с репозиториями для fedora и ubuntu. Ответ на этот вопрос дает таблица (значения взяты на 7.02.2017 г.):

Пакет	**Репозиторий**		
	nixpkgs	**ubuntu(16.04)**	**fedora(27)**
total	13 960	45 688	46 697
python2	1 730	2 716	1 529
python3	1 569	1 373	1 873

В строке с пометкой total отображается общее количество пакетов. Из этой строки видно, что количество пакетов в nixpkgs отстает от своих конкурентов примерно в 3,5 раза, тем не менее стоит отметить, что все наиболее важные и часто используемые пакеты уже попали в nixpkgs. По мнению авторов, на сегодняшний день имеется достаточное количество пакетов в nixpkgs, чтобы этот пакетный менеджер можно было использовать и не сталкиваться с проблемой отсутствия в репозитории какого-либо пакета.

Динамика активности репозитория nixpkgs относительно количества измененных строк кода за день представлена на рисунке.

Из приведенного рисунка можно заключить, что репозиторий набирает популярность. Вокруг него существует сообщество людей, пишущих NIX-выражения. Каждое предложение с изменением (merge request) проходит строгое рецензирование кода и процедуру автоматической сборки.

Структура хранилища /nix/store

Пакетный менеджер NIX производит установку программ не в корневой каталог (например /usr/bin), а в каталог /nix/store. Каждый пакет устанавливается в индивидуальный каталог с именем /nix/store/<hash>-<name>, где <hash> – значение хэш-функции [9, 10], вычисленное от входных параметров деривации (пакета). Входные параметры – это все атрибуты множества, которые передаются сборщику (derivation, stdenv.mkDerivation и им подобные). Параметр <name> – атрибут деривации name. Этот подкаталог будет иметь стандартную структуру с каталогами /nix/store/<hash>-<name>/{bin, usr/bin, …}.

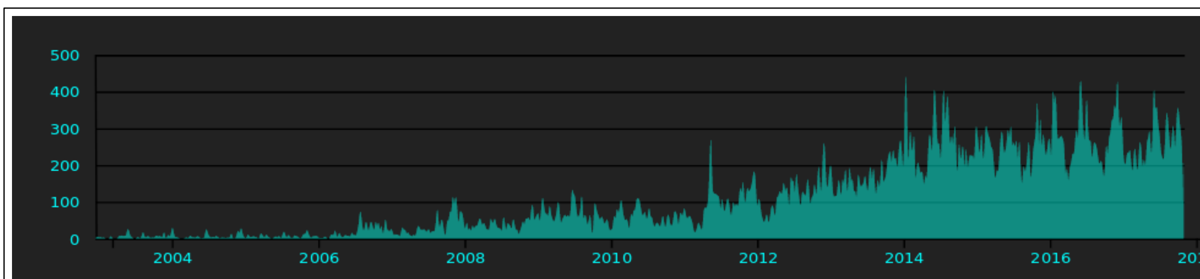
Пакетный менеджер NIX работает с файлами из /nix/store в следующем режиме. Все, что попало в /nix/store, больше никогда не будет изменено пакетным менеджером. Если та или иная деривация больше не используется, ее можно удалить при помощи сборщика мусора.

Программы, установленные пакетным менеджером NIX, имеют зависимости только из /nix/store. Стоит отметить, что к числу таких зависимостей относятся динамический загрузчик и bash. Для достижения изоляции такого рода создатели деривации иногда модифицируют исходный код программы. Для использования динамического загрузчика из /nix/store все .elf-файлы в автоматическом режиме модифицируются утилитой patchelf, которая корректным образом изменяет путь до динамического загрузчика. Помимо изменения пути до динамического загрузчика, изменяются специальные комментарии, содержащие строки запуска интерпретаторов, вместо #! /bin/bash подставляется #! /nix/store/<hash>-bash-<version>.

Управление областью видимости

Пакетный менеджер NIX устанавливает ПО в нестандартные локации. При запуске программы исполняемый файл ищется в путях, перечисленных в переменной окружения PATH. Поэтому для нахождения установленной программы необходимо, чтобы она оказалась в одном из каталогов, перечисленных в PATH. За то, какие исполняемые файлы будут найдены в /nix/store, отвечает пакетный менеджер NIX.

Пакетный менеджер NIX позволяет делать разные профили для каждого пользователя. Под профилем имеется в виду набор программ, видных пользователю. Наличие разных профилей для раз-



Активность репозитория: по оси X изображен временной отрезок 2004–2018 г., по оси Y – количество измененных строк кода за день

Repository Activity

ных пользователей означает, что если из двух существующих пользователей A и B пользователь A поставит в свое окружение программу P, а пользователь B не установит программу P, то пользователю A программа будет видна, а пользователю B нет. Отметим, что традиционно в UNIX-подобных операционных системах установка производится глобально и установленная программа видна всем.

Вдобавок к множественным профилям NIX позволяет вести версионный контроль профилей. После каждой операции установки/удаления/обновления создается новая генерация профиля. Если по каким-либо причинам установленная программа не удовлетворяет пользователя, имеется возможность переключения к предыдущей генерации одной командой.

Каждый из каталогов /nix/store/<hash>-user-environment имеет структуру, схожую со структурой корневого каталога, и подкаталоги bin, etc и им подобные. Файлы, находящиеся в подкаталогах, являются символическими ссылками на реальные файлы.

Бинарный кэш

Пакеты в NIX описывают процесс сборки программы из исходных текстов. Если каждый пользователь будет собирать каждую программу из исходных текстов, это отнимет у него очень много времени на ожидание сборки, а иногда и вовсе окажется невозможным из-за того, что сборка некоторых пакетов требует большого количества оперативной памяти.

Для решения этой проблемы существуют хранилища, содержащие заранее скомпилированные программы. Поэтому алгоритм установки следующий. Сначала пакетный менеджер проверит существование заранее откомпилированных бинарных файлов в бинарном кэше. Если такие файлы существуют, они будут получены от удаленного сервера. Если удаленный сервер не располагает откомпилированными файлами, будет запущена сборка из исходных текстов.

Репозиторий NIX содержит стабильную и нестабильную версии бинарного кэша. Нестабильная версия представляет прекомпилированные файлы для репозитория github.com/NixOS/nixpkgs. После того, как репозиторий nixpkgs пройдет тщательное тестирование, он перемещается в репозиторий github.com/NixOS/nixpkgs-channels. Стабильная версия репозитория гарантирует, что для каждого пакета, для которого необходима сборка, в бинарном кэше будут содержаться прекомпилированные бинарные файлы.

Заключение

В статье был рассмотрен пакетный менеджер, который принципиальным образом отличается от стандартных пакетных менеджеров. В архитектуре пакетного менеджера NIX отвергается стандартная концепция глобальной установки программ, при этом новая концепция учитывает особенности стандартных систем сборки, что практически всегда позволяет без модификации системы сборки программы использовать ее в пакетном менеджере NIX. Пакетный менеджер NIX управляет файлами таким образом, что всякая операция установки/удаления/обновления гарантированно не будет деструктивно воздействовать на систему.

Литература

1. Getting NIX. URL: https://nixos.org/nix/download.html (дата обращения: 03.04.2018).
2. Referential transparency. URL: https://wiki.haskell.org/Referential_transparency (дата обращения: 03.04.2018).
3. NIX by example. URL: https://medium.com/@MrJamesFisher/nix-by-example-a0063a1a4c55 (дата обращения: 03.04.2018).
4. Learn NIX. URL: https://learnxinyminutes.com/docs/nix/ (дата обращения: 03.04.2018).
5. A tour of NIX. URL: https://nixcloud.io/tour/?id=1 (дата обращения: 03.04.2018).
6. Configure terms and history. URL: https://gcc.gnu.org/onlinedocs/gccint/Configure-Terms.html (дата обращения: 03.04.2018).
7. NIX phases. URL: https://nixos.org/nixpkgs/manual/#sec-stdenv-phases (дата обращения: 03.04.2018).
8. NIX repository. URL: https://github.com/NixOS/nixpkgs (дата обращения: 03.04.2018).
9. NIX pills. Derivation. URL: https://nixos.org/nixos/nix-pills/our-first-derivation.html (дата обращения: 03.04.2018).
10. NIX pills. Store paths. URL: https://nixos.org/nixos/nix-pills/nix-store-paths.html (дата обращения: 03.04.2018).

Software & Systems Received 04.04.18
DOI: 10.15827/0236-235X.123.469-474 2018, vol. 31, no. 3, pp. 469–474

A package manager for multiversion applications

V.A. Galatenko [1], *Dr.Sc. (Physics and Mathematics), Head of Programming Automation Department, galat@niisi.ras.ru*
M.D. Dzabraev [1], *Engineer, mdzabraev@niisi.ras.ru*
K.A. Kostyukhin [1], *Ph.D. (Physics and Mathematics), Senior Researcher, kost@niisi.ras.ru*

[1] *Federal State Institution "Scientific Research Institute for System Analysis of the Russian Academy of Sciences" (SRISA RAS), Moscow, 117218, Russian Federation*

Abstract. All software developers eventually face the problem of creating and distributing their software products. At the same time, it is necessary to take into account possibilities of supporting existing products, i.e. replacing old distributions with

new ones. When using a quality distribution tool, developers are able to distribute their products to a wider range of platforms, as well as provide the necessary and timely support for these products.

The authors of the article consider only UNIX-like systems, most of which include package managers as dpkg, yum. These package managers operate according to a standard concept of software installation in UNIX. The standard concept implies that programs are installed in standard directories such as /usr/bin, /usr/local/bin, and so on. When updating a program (package), it is common practice to replace old files with new ones. Such substitution strategy can be destructive. This means that after software update, some programs or libraries stop working. It is possible, for example, that a package manager itself may stop working after updating. A user is often in a situation when old versions of software are required to support compatibility. In this case, it is necessary to use the practice of building programs and libraries from source code and manual installation, such as "make install". This kind of installation is irreversible and very dangerous, since in this case the files under control of a package manager may be deleted or replaced.

The authors propose a package manager NIX [1] as a solution for the described problems. The most important advantage of this manager is that it completely excludes destructive impact on its part. This is achieved by installing each package in an isolated location controlled by a package manager.

Keywords: package manager, NIX, program installation, distribution, versioning.

References

1. *Getting NIX*. Available at: https://nixos.org/nix/download.html (accessed April 3, 2018).
2. *Referential transparency*. Available at: https://wiki.haskell.org/Referential_transparency (accessed April 3, 2018).
3. *NIX by example*. Available at: https://medium.com/@MrJamesFisher/nix-by-example-a0063a1a4c55 (accessed April 3, 2018).
4. *Learn NIX*. Available at: https://learnxinyminutes.com/docs/nix/ (accessed April 3, 2018).
5. *A tour of NIX*. Available at: https://nixcloud.io/tour/?id=1 (accessed April 3, 2018).
6. *Configure terms and history*. Available at: https://gcc.gnu.org/onlinedocs/gccint/Configure-Terms.html (accessed April 3, 2018).
7. *NIX phases*. Available at: https://nixos.org/nixpkgs/manual/#sec-stdenv-phases (accessed April 3, 2018).
8. *NIX repository*. Available at: https://github.com/NixOS/nixpkgs (accessed April 3, 2018).
9. *NIX pills. Derivation*. Available at: https://nixos.org/nixos/nix-pills/our-first-derivation.html (accessed April 3, 2018).
10. *NIX pills. Store paths*. Available at: https://nixos.org/nixos/nix-pills/nix-store-paths.html (accessed April 3, 2018).

Примеры библиографического описания статьи

1. Галатенко В.А., Дзабраев М.Д., Костюхин К.А. Выбор пакетного менеджера для многоверсионных приложений // Программные продукты и системы. 2018. Т. 31. № 3. С. 469–474. DOI: 10.15827/0236-235X.123.469-474.

2. Galatenko V.A., Dzabraev M.D., Kostyukhin K.A. A package manager for multiversion applications. *Software & Systems*. 2018, vol. 31, no. 3, pp. 469–474 (in Russ.). DOI: 10.15827/0236-235X.123.469-474.