Część systemów zostanie tutaj pokrótce omówione, jednak uwaga zostanie zwrócona głównie na tych funkcjonalnościach, które są dostępne w systemie *DoForMe!* w celu porównania.

Kolejną zaletą jest także to, że lista akcji posiada swego rodzaju "obiekty" - każdy kolejny wiersz to kolejny obiekt. Dzięki temu bardziej intuicyjne jest dodawanie lub modyfikowanie jego parametrów. W przypadku zmiany parametrów w systemach wykorzystujących język programowania, zazwyczaj zostają zmienione argumenty przekazywane do funkcji, co może się nie spodobać użytkownikom nie mającym doświadczenia w programowaniu.

Precyzja czasu wywołania zadania zależy od pracy, którą użytkownik zamierza zautomatyzować. Dla przykładu gdy tester oprogramowania testuje kilka funkcjonalności za pomocą uprzednio nagranych zadań, gdzie każde zadanie jest zapisane w innym pliku, wtedy zbyt długie byłoby wywoływanie testów np. co minutę (pod warunkiem, że testy byłyby w miarę krótkie), zatem system typu *Macro Scheduler* nie zdałby egzaminu, ponieważ jego czas jest mierzony co do minuty[[1]](#footnote-2) (aczkolwiek istnieje obejście tego za pomocą wywołania zadania o pełnej minucie z użyciem komendy Wait, która usypia program na podaną ilość sekund[[2]](#footnote-3)). We wspomnianych wcześniej systemach włącznie z *DoForMe!* czas jest mierzony co do sekundy, aby objąć zakresem większą część użytkowników.

System *AutoClickExtreme*[[3]](#footnote-4) jest bardzo precyzyjny pod względem wykonywanych akcji. Udostępnia on szereg parametrów i funkcjonalności umożliwiających sprecyzowanie miejsca lub obiektu, na którym operujemy. Dla przykładu, do wykonania każdego kliknięcia możliwe jest pobranie zrzutu fragmentu ekranu, którego program będzie musiał poszukać zanim wykona akcję, jeśli taki fragment nie istnieje program czeka na interwencję użytkownika i wznawia pracę gdy pobrany fragment pojawi się na ekranie[[4]](#footnote-5).

Poza tym gdy użytkownika nie ma przy komputerze takie powiadomienie staje się z oczywistych względów bezużyteczne, jednak oprogramowanie oprócz tego oferuje także logi, z których możemy dowiedzieć się jak przebiegał proces wykonywania zadań[[5]](#footnote-6).

Program *AutoClickExtreme* automatyzuje pracę ale w nieco odmienny sposób - zamiast wykorzystywać język skryptowy ze zdefiniowanymi funkcjami, operuje on na liście, na której znajdują się poszczególne akcje użytkownika[[6]](#footnote-7), np. kliknięcie czy poruszenie myszką. Takie rozwiązanie jest prostsze gdy docelowa grupa użytkowników ma niewiele wspólnego z programowaniem, ponieważ użytkownik nie operuje stricte na języku programowania:

Jednak w przypadku bardziej doświadczonych użytkowników oznacza to brak szczegółowej modyfikacji wykonywania zadania. Na przykład w powyższym przykładzie z *AutoClickExtreme* nie jest możliwa modyfikacja sposobu ruchu kursora myszki (ani nawet podglądu po jakich współrzędnych kursor myszy się porusza). Co więcej, struktura pliku z zapisanymi akcjami jest bardziej skomplikowana niż w przypadku prostego pliku tekstowego ze skryptem, przez co trudniejsze jest odgadnięcie co robi dane zadanie nie posiadając odpowiedniego oprogramowania:

|  |
| --- |
| Wycinek pliku z zapisanym zadaniem z powyższego przykładu z *AutoClickExtreme* |
| num=0//ver=5.9  UN=000003, node=0, TypeAct=4  {  x962y501t0x961y501t265......  }  UN=000004, node=0, TypeAct=1  {  MainWindow:  exe="C:\Windows\Explorer.EXE", SbyExe=0  capt="Program Manager", class="Progman", typeCaptSearch=4 handle=10148  ChildWindow: capt="FolderView", class="SysListView32", typeCaptSearch=0 hierarchy="((0:0))", id=1, handle=1014c enable=1 Style=1442855744 exStyle=0  LimitSearch=5, PermitSetFocus=0, screen="C:\Users\jajcek\Desktop\dir\_aip\_Program Manager\ScrChild\_un=000004\_x495y427.bmp"  Borders(0, 1280, 0, 1024, 0, 1280, 0, 1024)  CLICK(xx=599, yy=501):  key=1, updown=513, dTime=171:  key=0, updown=514, dTime=94)  } |

W przypadku planowania zadań i układania harmonogramu sposób jego prezentacji w programie *AutoClickExtreme* jest także oparty o listę. Jest to rozwiązanie dosyć nieczytelne w przypadku dużej ilości zadań w harmonogramie, w szczególności gdy użytkownik dodaje je nie po kolei, ponieważ oprogramowanie nie oferuje żadnego sposobu sortowania takiej listy[[7]](#footnote-8). Bardzo podobne rozwiązanie posiada system *WinAutomation*, który jednak oferuje sortowanie takiej listy względem wybranego przez nas parametru za pomocą kliknięcia nagłówka odpowiedniej kolumny*[[8]](#footnote-9)*. W systemie *DoForMe!* został zaimplementowany całkowicie odmienny sposób zarządzania zadaniami. Do tego celu wykorzystuje się wbudowany kalendarz, na który "nakłada" się zadania na odpowiednie dni określając godzinę oraz powtórzenia. Co więcej, system oferuje powiadomienia m.in. w formie graficznej w przypadku sytuacji gdy w jednym dniu jest więcej niż jeden skrypt do wykonania o tej samej godzinie.

|  |
| --- |
| Sposób prezentacji zadań przez system *AutoClickExtreme* - brak sortowania |
| ace_task_unsorted.bmp |

|  |
| --- |
| Sposób prezentacji zadań przez system *WinAutomation* - sortowanie po ostatniej kolumnie |
| ace_task_unsorted.bmp |
| Sposób prezentacji zadań przez system *DoForMe!* - zadania przypisane do odpowiednich dni miesiąca; zaznaczony dzień 13 wrzesień – kolizja dwóch skryptów |
| ace_task_unsorted.bmp |

Różne systemy mają także różne sposoby powiadamiania użytkownika o zadaniach z harmonogramu, które będą np. wykonane za niedługi czas, są aktualnie wykonywane lub zakończono ich wykonywanie. Jednym ze sposobów jest wykorzystanie prostego powiadomienia wyświetlonego na ekranie, które oferuje np. *WinAutomation* i które nie wymaga interwencji użytkownika[[9]](#footnote-10). Takie powiadomienie posiada jednak wadę w przypadku gdy użytkownik aktualnie pracuje na komputerze, bowiem ten sposób nie oczekuje od użytkownika żadnej odpowiedzi, dlatego gdy nadejdzie czas na uruchomienie zadania, system po prostu przejmie kontrolę nad komputerem i wykona swoje zadanie jednocześnie przerywając pracę użytkownika. System umożliwia także wyświetlenie okna dialogowego, które wymaga akcji ze strony użytkownika, jednak wspomniane rozwiązanie nie jest dostępne z poziomu opcji programu a jedynie poprzez odpowiednie zdefiniowanie akcji w zadaniu do wykonania. Tego typu powiadomienie oferuje także m.in. *Macro Scheduler[[10]](#footnote-11)*. W przypadku systemu *DoForMe!* nie ma potrzeby odwoływania się do akcji/skryptu zadania, wystarczy zaznaczenie odpowiedniej opcji programu. Dodatkowo możliwe jest dodanie także dźwięku.

--- o lua

Jeśli zapytamy programistę czym jest język skryptowy, zapewne w większości przypadków otrzymamy parę przykładów jaki język należy do tej kategorii. Jednak większość osób tak naprawdę nie wie dlaczego. Znaczna liczba źródeł definiuje go jako język umożliwiający sterowanie konkretnymi zadaniami w systemie (ang. Job Control)[[11]](#footnote-12). Wywodzi się on głównie z czasów superkomputera (ang. mainframe) firmy IBM®, który używał języka JCL (Job Control Language) do uruchamiania określonych programów w systemie bez interwencji manualnej (ang. batch processing[[12]](#footnote-13)). Języki te początkowo miały bardzo dużo ograniczeń, np. język JCL umożliwiał m.in. deklarację zmiennych/parametrów, warunkowe uruchamianie zadań, część możliwości modularności oraz parametryzowane procedury, które bardziej przypominały znane z C makra[[13]](#footnote-14).

Większość tego typu języków jest także określanych jako „Glue Languages”. Pojęcie to oznacza, że takie języki służą także do łączenia komponentów (narzędzi, modułów) niekoniecznie napisanych w języku skryptowym. Dzięki temu możliwe jest połączenie programów napisanych w różnych językach, które rozwiązują pojedyncze problemy, aby wspólnie rozwiązać większy problem.

Z powodu szybkiej i specyficznej ewolucji języków skryptowych (np. Perl, który posiada cechy języka skryptowego, systemowego jak i idei maszyny wirtualnej[[14]](#footnote-15) podobnej do Java™), konkretna definicja określająca czym naprawdę jest język skryptowy jest trudna do ustalenia. Cechy charakterystyczne takich języków, które są często podawane[[15]](#footnote-16) to m.in.:

- dynamiczne typowanie, np. w Lua:

x = "string"

x = 5

oba wyrażenia są poprawne i nie wymagają ręcznego określenia typu przez programistę)

- zazwyczaj są interpretowane zamiast kompilowane

- głównie używane do pisania krótszych i prostszych programów

- czym jest lua i jakie ma możliwości

Lua jest jednym z przenośnych języków skryptowych najczęściej wykorzystywanym w branży gier komputerowych[[16]](#footnote-17), który zbliżony jest najbardziej do języków proceduralnych. Działa na każdej maszynie, która udostępnia kompilator ANSI C, począwszy od systemów wbudowanych aż do mainframe’ów. Umożliwia między innymi rozszerzalność (dodawanie) funkcji, których implementacje egzystują w zewnętrznym kodzie napisanym np. w języku C[[17]](#footnote-18), Java[[18]](#footnote-19) lub Delphi[[19]](#footnote-20). Ponadto od roku 2003, gdy zadebiutowała wersja 5.0, działa na wirtualnej maszynie Parrot, która w odróżnieniu od większości wirtualnych maszyn (które są bazowane na stosie) bazuje na rejestrach[[20]](#footnote-21), dzięki temu zwiększona została, w niektórych przypadkach ponad dwukrotnie, wydajność w dostępie do zmiennych lokalnych. Co więcej posiada automatyczne zarządzanie pamięcią wraz ze swoim własnym garbage collector’em, który został dodany w wersji 5.1[[21]](#footnote-22) oraz który jest niezwykle istotny w grach komputerowych.

- jak dziala lua

Działanie interpretera Lua oparte jest o emitowanie instrukcji dla maszyny wirtualnej „w locie”, czyli w trakcie procesu parsowania, aczkolwiek zapewniając także pewne optymalizacje, np. utworzenie kodu (wyemitowanie instrukcji) dla wyrażeń związanych ze stałymi i zmiennymi jest opóźnione. Innymi słowy interpreter najpierw generuje prostą strukturę, w której umieszcza potrzebne mu dane na ich temat i na jej podstawie optymalizuje kod[[22]](#footnote-23).

Na początek przyjrzymy się temu w jaki sposób reprezentowane są dane w języku Lua, opiszemy strukturę przechowującą je oraz jakie korzyści i wady z niej wynikają.

Następnie przejdziemy do opisu tabel, które de facto są jedynym mechanizmem do tworzenia struktur danych wykorzystywanych w Lua[[23]](#footnote-24). Wyjaśnimy tutaj także metodę Brent'a, która została w nich wykorzystana.

W kolejnym kroku opiszemy funkcje oraz ich domknięcia. Przyjrzymy się sposobowi w jaki są one reprezentowane po kompilacji oraz jak przechowywane są potrzebne jej dane.

Na koniec zostawimy opis działania maszyny wirtualnej.

- dane w lua

Danymi w Lua są: nil, boolean, liczby (domyślnie Double, ale prosta jest kompilacja Lua używając Float lub Long, z tego względu, że niektóry urządzenia nie mają wsparcia sprzętowego dla podwójnej precyzji), string, tabele, funkcje, wątki oraz własne zdefiniowane typy przez użytkownika. Dane w Lua reprezentowane są jako pary (t,v) w oznakowanych uniach w następujący sposób[[24]](#footnote-25):

|  |  |
| --- | --- |
| typedef struct {  int t;  Value v;  } TObject; | typedef unin {  GCObject \*gc;  void \*p;  lua\_Number n;  int b;  } Value; |

gdzie wartość t oznacza jakim typem jest wartość v. Wartości Boolean (pole b) oraz liczbowe (pole n) są reprezentowane bezpośrednio w unii, za to wartości typu string, tabele, funkcje, wątki (pole gc) oraz dane zdefiniowane przez użytkownika (pole p) - jako wskaźniki. Wartość nil nie jest reprezentowana za pomocą struktury Value, ponieważ do jej identyfikacji wystarczy sama wartość t.

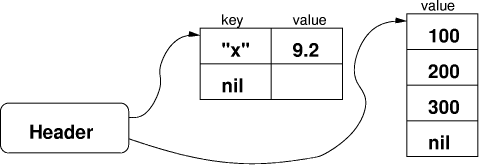
Rozwiązanie takie ma oczywiście swoje konsekwencje. Na przykład kopiowanie wartości może być w niektórych przypadkach dosyć kosztowne. Weźmy dla przykładu kopiowanie liczby 64-bitowej double na maszynie 32-bitowej:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INT | | | | DOUBLE | | | | | | | | PADDING | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| WORD | | | | WORD | | | | WORD | | | | WORD | | | |

Rozmiar struktury TObject jest równy 12 bajtów (16 bajtów jeśli uwzględnimy tzw. padding). Maszyna 32-bitowa, kopiuje dane w porcjach po 4 bajty, tzw. słowo (ang. word), zatem aby skopiować liczbę typu double potrzebne jest 3 (lub 4) kopiowań słów.

- tabele w lua

Język ten nie udostępnia typu tablicowego, dlatego do symulowania tablic wykorzystywane są właśnie tabele. Do wersji 4.0 tabele były zaimplementowane jako zwykłe tzw. hashmapy, jednak od wersji 5.0 tabela jest połączeniem tablicy (która nie przechowuje wartości klucza) z hashmapą:



Możliwymi konfiguracjami par tutaj są x->9.2, 1->100, 2->200, 3->300. Ponadto jej rozmiar jest dynamiczny - rozszerza się gdy dodajemy do niej jakiś element oraz kurczy gdy usuwamy (poprzez przypisanie nil'a). Gdy tabela potrzebuje więcej miejsca, rozmiary tablicy liczone są na nowo:

Rozmiarem tablicy jest największe takie n, dla którego przynajmniej połowa elementów w tablicy pomiędzy 1 a n (aby uniknąć marnowania miejsca przy tablicach rzadkich) oraz przynajmniej 1 element pomiędzy a (aby uniknąć rozmiaru gdy jest wystarczające) są w użyciu. Po przeniesieniu tego na język matematyczny:

Oznaczmy tablicę jako:

Oznaczmy element tablicy, który jest w użyciu jako 1, a element, który nie jest w użyciu jako 0.

Wtedy szukamy takiego największego , że:

oraz

Dzięki temu, że tabela wykorzystuje także zwykłą tablicę, to w niektórych przypadkach możliwe jest zmniejszenie zużycia pamięci nawet dwukrotnie, ponieważ tablica nie wymaga dodatkowej pamięci na klucze. Ponadto dostęp do danych za pomocą klucza liczbowego jest szybszy, ponieważ nie wymaga wykonania procedury hashującej w połączeniu z metodą Brent'a.

- metoda brent'a

Metoda ta polega na rozwiązywaniu kolizji przy hashowaniu oraz optymalizacji dostępu do danych. Bazuje ona na koncepcji, która zakłada żeby więcej czasu poświęcić na wstawianie elementów niż ich znalezienie. Dlatego też jest ona głównie wykorzystywana przy kompilatorach lub interpreterach.

Załóżmy, że:

rozmiar\_tablicy - najbliższa liczba pierwsza, większa od liczby elementów

H1(klucz) - funkcja hashująca = klucz % rozmiar\_tablicy

H2(klucz) - przesunięcie (offset) = (klucz / rozmiar\_tablicy) % rozmiar\_tablicy

oraz

i - liczba miejsc, które musimy odwiedzić aby znaleźć element (A), który wstawiamy

j - liczba dodatkowych miejsc, które musimy odwiedzić aby znaleźć element (B), który przesuwamy

Chcemy znaleźć minimalną sumę i+j (i > 0 oraz j > 0).

Rozwiązywanie kolizji przebiega następująco:

1. Na początku musimy spróbować przesunąć element, z którym wstawiany element koliduje (oznaczmy to jako punkt ), do punktu . Gdy zmienia się w górę oznacza to, że musimy spróbować przesunąć B (aktualnie istniejący element) pod indeks przesunięty o .

2. Jeśli punkt jest zajęty to sprawdzamy czy jest możliwe zapisanie sumy w innej kombinacji (której jeszcze nie było), tzn. sumę można zapisać także jako , czyli punkt . Zmiana oznacza, że teraz naszym elementem B będzie element, który jest pod indeksem przesuniętym od B o offset . Następnie próbujemy przesunąć B o , jeśli się nie uda to próbujemy dopasować kolejne oraz . Itd.

Innymi słowy szukamy po kolei odpowiedniej minimalnej kombinacji , czyli: itd.

Jeśli jest kilka możliwych kombinacji dla przesunięć to to czy wybierzemy czy zależy od naszej implementacji, aczkolwiek głównie stosuje się tą pierwszą konwencję.

- funkcje w lua

Gdy dochodzi do momentu kompilacji funkcji, Lua generuje jej prototyp, który zawiera:

* instrukcje dla wirtualnej maszyny
* tablicę z stałymi użytymi w funkcji
* dane dla debuggera

Podczas działania programu gdy dochodzi do wywołania funkcji, silnik Lua generuje domknięcie funkcji (ang. *closure*). Każde domknięcie posiada wskaźnik do:

* prototypu od którego zostało utworzone,
* tabeli ze zmiennymi globalnymi (*environment*),
* tabeli ze zmiennymi lokalnymi z poza funkcji (*upvalues*).

1. MJT NET LTD, *Macro Scheduler - Scheduling Scripts*, <http://www.scribd.com/doc/36237955/Macro-Scheduler-Help#outer_page_16>, 2010 [↑](#footnote-ref-2)
2. BOB HANSEN, *Run script every 90 seconds*, <http://www.mjtnet.com/usergroup/viewtopic.php?p=26161>, 2010 [↑](#footnote-ref-3)
3. DENIS SAFONOV, *Smart Autoclicker and Software Testing Tool*, <http://www.autoclickextreme.com/>, 2011 [↑](#footnote-ref-4)
4. DENIS SAFONOV , *Pixel Control*, <http://www.autoclickextreme.com/articles/articles_pixel_control.html>, 2011 [↑](#footnote-ref-5)
5. WINAUTOMATION, *WinAutomation Tutorials - 15. Logs*, <http://www.youtube.com/watch?v=zk5IQaL8YH8>, 2012 [↑](#footnote-ref-6)
6. DENIS SAFONOV, *Task Manager help*, <http://www.autoclickextreme.com/help_en/tasker.htm>, 2012 [↑](#footnote-ref-7)
7. DENIS SAFONOV, *Task Manager help*, <http://www.autoclickextreme.com/help_en/tasker.htm>, 2012 [↑](#footnote-ref-8)
8. wersja 4.0.2 [↑](#footnote-ref-9)
9. ADMIN OF WINAUTOMATION'S BLOG, *Display Message VS Display Notification*, <http://www.winautomation.com/blog/display-message-vs-display-notification>, 2010 [↑](#footnote-ref-10)
10. MJT NET LTD, *Macro Scheduler - Command Reference*, <http://www.scribd.com/doc/36237955/Macro-Scheduler-Help#outer_page_99>, 2010 [↑](#footnote-ref-11)
11. RICH MORIN, VICKI BROWN, *Scripting languages*, <http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html>, 1999 [↑](#footnote-ref-12)
12. IBM® CORPORATION, *z/OS concepts,* [http://publib.boulder.ibm.com/infocenter/ zos/basics/topic/com.ibm.zos.zconcepts/zconcepts\_book.pdf](http://publib.boulder.ibm.com/infocenter/%20zos/basics/topic/com.ibm.zos.zconcepts/zconcepts_book.pdf) str 79, 2010 [↑](#footnote-ref-13)
13. IBM® CORPORATION, *IBM System/360 Operating System Job Control Language (C28-6529-4)*, <http://www.bitsavers.org/pdf/ibm/360/os/R01-08/C28-6539-4_OS_JCL_Mar67.pdf>, 1967 [↑](#footnote-ref-14)
14. PATRICK MICHAUD, *Announce: Rakudo Star — a useful, usable, "early adopter" distribution of Perl 6*, <http://rakudo.org/2010/07/29/rakudo-star-2010-07/>, 2010 [↑](#footnote-ref-15)
15. CUNNINGHAM & CUNNINGHAM INCORPORATION, *Glue Languages,* <http://c2.com/cgi/wiki?GlueLanguages>, 2011 [↑](#footnote-ref-16)
16. GDMAG STAFF, *Front Line Award Winners – programming tools*, <http://gdmag.com/blog/2012/01/front-line-award-winners.php>, 2011 [↑](#footnote-ref-17)
17. ROBERTO IERUSALIMSCHY, *Programming in Lua - Second Edition s. 215*, 2006 [↑](#footnote-ref-18)
18. JAMES ROSEBOROUGH, IAN FARMER, *LuaJ*, <http://sourceforge.net/projects/luaj/>, 2011 [↑](#footnote-ref-19)
19. DENNIS D. SPREEN, *Lua 5.1 for Delphi* 2010 , <http://blog.spreendigital.de/2009/09/28/lua-5-1-for-delphi-2010/>, 2009 [↑](#footnote-ref-20)
20. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 - The Virtual Machine*, <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 7, 2005 [↑](#footnote-ref-21)
21. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 - Introduction*, <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 1, 2005 [↑](#footnote-ref-22)
22. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 - An Overview of Lua's Design and Implementation*, <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 2, 2005 [↑](#footnote-ref-23)
23. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 – Tables,* <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 4, 2005 [↑](#footnote-ref-24)
24. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 - The Representation of Values*, <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 2, 2005 [↑](#footnote-ref-25)