STUDIA INFORMATICA 2007
Volume 28 Number 4 (74)

Piotr GAWRON, Jarosław MISZCZAK, Ryszard WINIARCZYK
Instytut Informatyki Teoretycznej i Stosowanej PAN
Piotr WYCISK
Politechnika Śląska, Instytut Informatyki

KOMPILATOR JĘZYKA QCL – QCL2QML[1]

Streszczenie. W artykule przedstawiony jest kompilator proceduralnego języka Quantum Computation Language do języka sekwencyjnego Quantum Markup Language. Wadą języka QML jest jego słaba skalowalność. Wykorzystanie wysokopoziomowego języka QCL rozwiązuje ten problem poprzez wprowadzenie elementów proceduralnego paradygmatu programowania. Umożliwia to wielokrotne wykorzystanie kodu do podobnych problemów o różnych rozmiarach.
Słowa kluczowe: kompilacja, informatyka kwantowa

QCL2QML – A QCL LANGUAGE COMPILER

Summary. Article presents compiler which takes code written in procedural Quantum Computation Language as the input and produces sequential Quantum Markup Language code. The disadvantage of QML language is its low scalability. Usage of high-level language such as QCL solves this problem by introducing elements of procedural programming paradigm. It allows code reusability to solve similar problems of different size.
Keywords: compilation, quantum information

1. Wstęp

Informatyka kwantowa jest nową dziedziną wiedzy, która ma swoje początki w latach osiemdziesiątych dwudziestego wieku. W roku 1982 Richard Feyman zauważył, że komputery klasyczne nie są w stanie efektywnie symulować dużych układów kwantowych. Zapro-

[1] Ta praca jest wspierana przez grant Ministerstwa Nauki i Szkolnictwa Wyższego nr N519 012 31/1957.

ponował on, by jedne układy kwantowe symulować przy użyciu innych oraz zbudować komputer bazujący na zasadach rządzących mikroświatem. Zasady te są doskonale opisywane przez mechanikę kwantową. Na początku lat dziewięćdziesiątych różni badacze pokazali, że komputer kwantowy jest w stanie efektywnie symulować układy kwantowo-mechaniczne.

W roku 1994 Peter Shor zaprezentował efektywne rozwiązanie, przy użyciu komputera kwantowego, problemu faktoryzacji liczb całkowitych. Wynik ten spowodował gwałtowny wzrost zainteresowania informatyką kwantową. Rok później Lov Grover pokazał algorytm kwantowy wyszukiwania w nieuporządkowanym zbiorze. Złożoność tego algorytmu nie przekracza rzędu $O(n^{1/2})$.

Od lat dziewięćdziesiątych ubiegłego wieku trwają prace nad projektowaniem i implementacją języków programowania przeznaczonych dla maszyn kwantowych. Jednym z takich języków jest Quantum Computation Language.

QCL może być uważany za metajęzyk programowania, ponieważ program nie jest przewidziany do tego, by był uruchamiany na komputerze kwantowym, tylko na komputerze klasycznym, który kontroluje maszynę kwantową. Z punktu widzenia użytkownika, cały układ może być rozumiany jako klasyczny komputer probabilistyczny. Wejście i wyjście układu są klasyczne. Co więcej, stan komputera, który kontroluję maszynę kwantową, jest klasyczny i można mówić, że każdy stan komputera odpowiada jakiemuś krokowi programu.

2. Budowa kompilatora

2.1. Program qcl2qml

Nazwa programu qcl2qml jest skrótem Quatnum Computer Language to Quantum Markup Language Converter. Program został napisany w języku C++ i skompilowany przy użyciu kompilatora GCC4.1. Projekt konwertera jest kompilowany przez powszechnie znane narzędzie make.

Dokumentacja kodu źródłowego projektu została sporządzona przy użyciu systemu dokumentacji doxygen i jest dołączona do projektu.

2.2. Przeznaczenie programu

Program dokonuje konwersji z języka QCL, który jest jednym ze sposobów zapisu algorytmu kwantowego do języka QML, który opisuje algorytm kwantowy w formie obwodu kwantowego.

Jedyny znany symulator to qcl[2], który jest symulatorem komputera kwantowego. Symulację można wykonać na dowolnej stacji roboczej bądź serwerze, które są w stanie dokonać symulacji systemów złożonych z kilkunastu kubitów.

Fraunhofer Quantum Computing Simulator[3] jest aplikacją web, która dodatkowo zawiera graficzny edytor obwodów kwantowych. Edytor jest w formie appletu[4] uruchamianego w oknie dowolnej przeglądarki internetowej obsługującej applety. Obwody kwantowe tworzone w graficznym edytorze nazywane są zadaniami i mogą zostać skierowane do wykonania w symulatorze. Symulator to klaster 32-węzłowy z 56 GB pamięci RAM, który pozwala na wykonywanie symulacji systemów złożonych z 31 kubitów.

Możliwość pisania algorytmów kwantowych w języku QCL, a wykonywanie ich na klastrze jest jednym z powodów powstania projektu qcl2qml.

Przedstawiono dwa różne podejścia do opisu symulacji komputera kwantowego. W przypadku QCL opisem jest program klasyczny, w którym występują instrukcje modyfikujące kubity, będące składowymi rejestrów kwantowych. Zapis algorytmu jest w postaci znanej każdemu programiście. Natomiast język QML jest bliższy opisowi symulacji w postaci obwodu kwantowego. Tu algorytm kwantowy zapisuje się albo przez wstawianie bramek kwantowych do obwodu, albo przez zapis kodu źródłowego w charakterystycznej (dla języków pochodnych XML) formie. Jednak pisanie kodu w QML jest karkołomnym zadaniem, bowiem język służy do opisu obwodu kwantowego, a nie zapisu algorytmu. W przypadku QML dopiero obwód kwantowy jest sposobem zapisu algorytmu kwantowego.

2.3. Opis konwersji

Konwersja zapisu algorytmu z języka QCL do QML odbywa się według zasady: wykonaj kod QCL, a sekwencję operacji na typach kwantowych zapisz w formie obwodu kwantowego w QML. Poniżej zostały przedstawione zasady konwersji:

Konwersja bramek jednokubitowych

QCL	Macierz	QML	gate type
`Pauli(1,q);`	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	X	`<Gate Type="PAULI_X"/>`
`Pauli(2,q);`	$\begin{bmatrix} 0 & i \\ -i & 0 \end{bmatrix}$	Y	`<Gate Type="PAULI_Y"/>`
`Pauli(3,q);`	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	Z	`<Gate Type="PAULI_Z"/>`

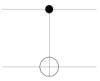
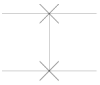
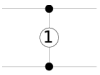
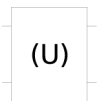
[2] http://tph.tuwien.ac.at/~oemer/

[3] Dostępny pod adresem http://qc.fraunhofer.de

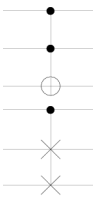
[4] Java Applet

`H(q)`	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$	H	`<Gate Type="HADAMARD"/>`
`RotX(a,q);`	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & i\\ i & 1\end{bmatrix}$	RX	`<Gate Type="RX"/>`
`RotY(a,q);`	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ -1 & 1\end{bmatrix}$	RY	`<Gate Type="RY"/>`
`RotX(-a,q);`	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & -i\\ -i & 1\end{bmatrix}$	-RX	`<Gate Type="-RX"/>`
`RotY(-a,q);`	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & -i\\ -i & 1\end{bmatrix}$	RY	`<Gate Type="-RY"/>`
`T(q)`	$\begin{bmatrix}1 & 0\\ 0 & e^{i\frac{\pi}{4}}\end{bmatrix}$	T	`<Gate Type="T_GATE"/>`
`S(q)`	$\begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix}$	S	`<Gate Type="S_GATE"/>`
`V(ϕ,q)`	$\begin{bmatrix}1 & 0\\ 0 & e^{\frac{2\pi}{2^k}}\end{bmatrix}$	1	`<Gate Type="PHASE" Divisions="f"/>`
`Matrix2x2(.,q)`	$\begin{bmatrix}a & b\\ c & d\end{bmatrix}$	(U)	`<Gate Type="UNITARY_2" Matrix="..." />`

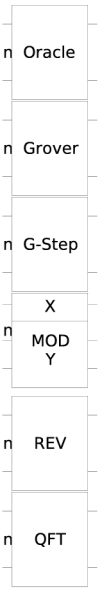
Konwersja bramek dwukubitowych

QCL	Macierz	QML	qml tag
`CNot()`	$\|00\rangle\rightarrow\|00\rangle$ $\|01\rangle\rightarrow\|01\rangle$ $\|10\rangle\rightarrow\|11\rangle$ $\|11\rangle\rightarrow\|10\rangle$		`<Gate Type="CNOT"/>`
`Swap(q1,q2)`	$\|00\rangle\rightarrow\|00\rangle$ $\|01\rangle\rightarrow\|10\rangle$ $\|10\rangle\rightarrow\|01\rangle$ $\|11\rangle\rightarrow\|11\rangle$		`<Gate Type="SWAP"/>`
`CPhase(ϕ,q)`	$\|11\rangle\rightarrow\exp(\frac{2i\pi}{2^k})\|11\rangle$		`<Gate Type="PHASE" Divisions="k"/>`
`Matri4x4(..,q)`	$\begin{bmatrix}a_{11} & \cdots & a_{14}\\ \vdots & \ddots & \vdots\\ a_{41} & \cdots & a_{44}\end{bmatrix}$		`<Gate Matrix="" Type="UNITARY_2"/>`

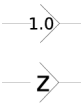
Konwersja bramek trzykubitowych

QCL	Macierz	QML	qml tag
brak	$\|111\rangle \rightarrow \|110\rangle$		<Gate Type="TOFFOLI"/>
brak	$\|101\rangle \rightarrow \|110\rangle$		<Gate Type="FREDKIN"/>

Konwersja bramek n-qubitowych

QCL	Macierz	QML	qml tag
query(...)			<Gate Size="n" Type="ORACLE" BasisState="N"/>
search(q,N)			<Gate Size="n" Type="GROVER" Steps="1" BasisState="N"/>
diffuse(q)			<Gate Size="n" Type="GROVER_STEP" BasisState="0x255"/>
ModExp(...)			<Gate Modulus="1" Size="3" Type="MODULO" Base="1" Index="1"/>
q1<->q2	$\|123..n\rangle \rightarrow \|n..321\rangle$		<Gate Size="n" Type="REVERSE"/>
dft(q)			<Gate Size"n" Type="QFT"/>

Konwersja pozostałych bramek

QCL	opis	QML	qml tag
set(n,q)	ustawianie stanu		<Gate Type="PREPARATION" Probability="1.0"/>
measure	pomiar		<Gate Type="MEASUREMENT_Z"/>
operator(...)	podobwód		<Gate Size="2" Type="CIRCUIT"/>
randomize(q)	wartość losowa		<Gate P1="0.5" Size="2" P0="0.5" Type="RANDOM" CaseSize="1"/>

2.4. Analiza i projekt aplikacji

Elementem wspólnym dla obydwu sposobów opisu symulacji komputera kwantowego jest sekwencja operacji dokonanych na kubitach.

Projekt aplikacji dokonującej konwersji z QCL do QML w najbardziej ogólnej formie powinien uwzględniać etapy konwersji, które określane są jako przód konwertera, reprezentacja pośrednia i tył (nazywany również generatorem kodu). Na rysunku 1. można wyróżnić trzy etapy obecne w konwerterze:

$$QCL \xrightarrow{interpreter} qusim \xrightarrow{generator\ kodu} QML$$

Rys.1. Fazy konwersji

Fig. 1. Conversion phases

1. QCL – źródło w języku QCL przetwarzane jest przez interpreter, sekwencja wszystkich operacji kwantowych zapisywana jest do wewnętrznej reprezentacji.
2. Qusim – wewnętrzna reprezentacja sekwencji operacji kwantowych, jako odpowiednik kodu pośredniego.
3. QML – generator kodu do języka QML.

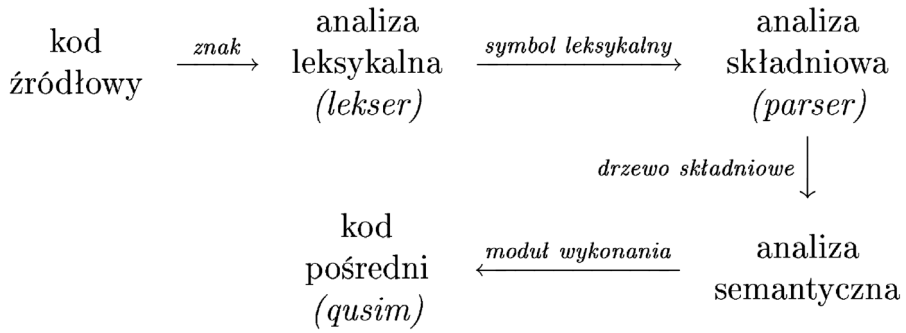
2.4.1. Interpreter QCL

Interpreter języka QCL składa się z czterech części określanych jako:

1) analizator leksykalny,
2) analizator składniowy,
3) analizator semantyczny,
4) moduł wykonania.

2.4.1.1. Analizator leksykalny

Analizator leksykalny jest pierwszą fazą interpretera, jego zadaniem jest czytanie tekstu źródłowego napisanego w języku QCL oraz generowanie symboli leksykalnych przekazywanych do analizatora składniowego [1]. Rysunek 2 przedstawia umiejscowienie analizatora leksykalnego w aplikacji.



Rys. 2. Fazy interpretera QCL
Fig. 2. QCL interpreter phases

Analizator leksykalny powstał przy użyciu generatora analizatorów leksykalnych Flex. Flex jest narzędziem generującym programy, które służą do wyszukiwania wzorców w tekście, opierając się na wyrażeniach regularnych.

Format pliku źródłowego dla Flexa.

```
definicje
%%
lista reguł
%%
kod użytkownika
```

Zasada działania tak zbudowanego leksera jest następująca: z ustalonego źródła (pliku lub standardowego wejścia) pobierane są kolejne znaki, które porównywane są z wcześniej

zdefiniowanymi wzorcami. Wzorce zapisywane są za pomocą wyrażeń regularnych. Dla każdego wzorca może zostać zdefiniowana akcja.

Przykład definicji w Flexie.

```
ALPHA [a-zA-Z]
DIGIT [0-9]
ALNUM {ALPHA}|{DIGIT}
```

Napisem w języku QCL jest sekwencja znaków ujęta w cudzysłów "...". Napis nie może zawierać cudzysłowiu.

Flex – przykład reguły z warunkiem początkowym strng rozpoznającej napisy w kodzie źródłowym.

```
 \"   { BEGIN(strng); strbuff.erase();}
<strng>\"  { BEGIN 0 ;
     yylval.STRING = &strbuff;
     return tokSTRINGCONST;
   }
```

Analizator uruchamia się przez wywołanie funkcji o nazwie qcllex() (standardowo funkcja ma nazwę yylex()). Po wywołaniu funkcji analizatora leksykalnego pobierane są znaki do momentu dopasowania do wzorca, następnie wywoływana jest akcja dla dopasowanego wzorca i następuje wyjście z funkcji qcllex(). W celu pobrania kolejnego symbolu leksykalnego należy ponownie wywołać funkcję qcllex().

Istnieje możliwość deklarowania warunków początkowych, gdzie każdy warunek ma swój identyfikator. Pewne wyrażenia mogą być rozpatrywane tylko w kontekście tych identyfikatorów. W ten sposób rozwiązano eliminacje komentarzy oraz tworzenie napisów pomiędzy znakami " " na poziomie analizatora leksykalnego.

Generowanie leksera następuje przez przetworzenie pliku źródłowego przez narzędzie Flex flex plik_źródłowy.l, po czym utworzony zostanie plik źródłowy w języku C o nazwie lex.yy.c, w którym zawarty będzie kod źródłowy leksera wraz z funkcją yylex(). Standardowy przedrostek yy w nazwie funkcji leksera może być zdefiniowany przez programistę. W takim przypadku wszystkie obiekty (zmienne i funkcje) towarzyszące lekserowi występują ze zmienionym przedrostkiem. Umożliwia to załączenie kilku lekserów w obrębie jednej aplikacji.

2.4.1.2. Analizator składniowy

Bison jest narzędziem ogólnego zastosowania do generowania analizatorów składniowych, który konwertuje plik opisujący bezkontekstową gramatykę LALR(1) w kod źródłowy w języku C parsujący tę gramatykę [2]. Narzędzie to zostało wykorzystane w projekcie do utworzenia parsera.

Zasady tworzenia generatora są podobne do tych, które są obecne w Flexie. Plik źródłowy ma podobną budowę:

Ogólny format pliku źródłowego dla Bisona.

```
deklaracje
%%
reguły translacji
%%
procedury pomocnicze w C
```

„Program w języku QCL to sekwencja instrukcji oraz definicji.” Lista produkcji dla symbolu top_symbol w Bisonie ma postać przedstawioną poniżej. Produkcje w postaci $A \rightarrow error\alpha$ używane są do odzyskiwania kontroli po napotkaniu błędu.

Przykładowa lista produkcji dla analizatora składniowego.

```
top_symbol: stmt   { $$ = $1; }
 | def    { $$ = $1; }
 | stmt top_symbol  { $1->addNext($2); $$ = $1; }
 | def top_symbol  { $1->addNext($2); $$ = $1; }
 | error '\n'   { $$ = new qcl::Error(); }
 | error tokEOF   { $$ = new qcl::Error(); }
;
```

Generowanie parsera odbywa się w ten sam sposób co leksera. Należy użyć narzędzia bison z nazwą pliku z opisem gramatyki jako parametrem wywołania. Zostanie wygenerowany plik w języku C ze źródłami parsera. Jeżeli wcześniej do pliku parsera zostanie dołączony plik leksera, to wygenerowana funkcja parsująca yyparse() będzie wywoływała funkcję leksera yylex() w celu pobierania symboli leksykalnych.

W ten sposób do projektu dołączone zostały źródła leksera i parsera.

Struktura programu QCL zapisana w formie notacji BNF została przedstawiona w pracy [17]. Język QCL zaprojektowany przez Ömera [7] został również zaimplementowany w symulatorze qcl. Źródła projektu qcl dostępne pod adresem http://tph.tuwien.ac.at/~oemer/ stały się pomocne w tworzeniu przodu konwertera.

2.4.1.3. Moduł wykonania instrukcji

Produktem, jaki otrzymujemy po wykonaniu funkcji prasera, jest drzewo wyprowadzenia. Rolę kodu trójadresowego [1] pełnią obiekty klas rozmieszczone według analizy składniowej. Wszystkie obiekty języka QCL są klasami dziedziczącymi po klasie qcl::Symbol. Funkcja wirtualna qcl::Symbol:pass(), która jest wspólna dla wszystkich obiektów języka QCL, pełni rolę wykonania przez interpreter akcji semantycznej związanej z obiektem. Na przykład qcl::ForStmt::pass() spowoduje wykonanie pętli for.

Hierarchia klas przedstawiona jest w pracy [17].

2.4.1.4. Kod pośredni

Rolę kodu pośredniego pełni sekwencja operacji wykonywanych na qubitach, z której w łatwy sposób można wygenerować. Przestrzeń nazw qusim zawiera klasy, w których są zawarte szczegóły, dotyczące każdej wykonanej operacji kwantowej oraz definicje podobwodów.

2.4.1.5. Generator QML

Klasy przedstawione w pracy [17] pełnią rolę specjalizowanego do QML generatora XML. Sprawdzanie poprawności plików QML nie zostało uwzględnione w obecnej formie konwertera, jednak utworzenie pliku z definicją typu dokumentu (DTD) jest krokiem w kierunku, by było to możliwe.

2.5. Specyfikacja zewnętrzna

Aplikację uruchamia się z linii poleceń qcl2qml plik.qcl. Po poprawnym wykonaniu programu zawartego w pliku plik.qml na standardowe wyjście wypisany zostanie plik z opisem obwodu kwantowego w języku QML. Dokonując przekierowania, można zapisać program QML w pliku.

W obecnej chwili nie ma możliwości wysyłania plików QML bezpośrednio do symulatora. Jak podaje [11], można zrobić to w następujący sposób. Po utworzeniu nowego zadania w symulatorze i otwarciu się edytora graficznego w pasku adresu przeglądarki należy dodać /file_edit_form do URL pliku qml. Spowoduje to pojawienie się formularza, który umożliwi wysłanie pliku qml do symulatora. Po wysłaniu pliku w oknie edytora obwodów kwantowych pojawi się przekonwertowany do obwodu kwantowego algorytm z plik.qcl.

2.6. Przykład konwersji

Przedstawiony niżej przykład konwersji nie jest algorytmem rozwiązującym problem, a jedynie demonstracją działania konwertera. Poniższy plik źródłowy

Przykład testowanego pliku:

```
qureg a[2];
qureg b[1];
qureg c[1];
qureg d[2];
int i;

set(3,a);
H(a);
Swap(b,c);
CNot(a,d);
for i = 1 to 3 {
    Swap(d,a);
}
```

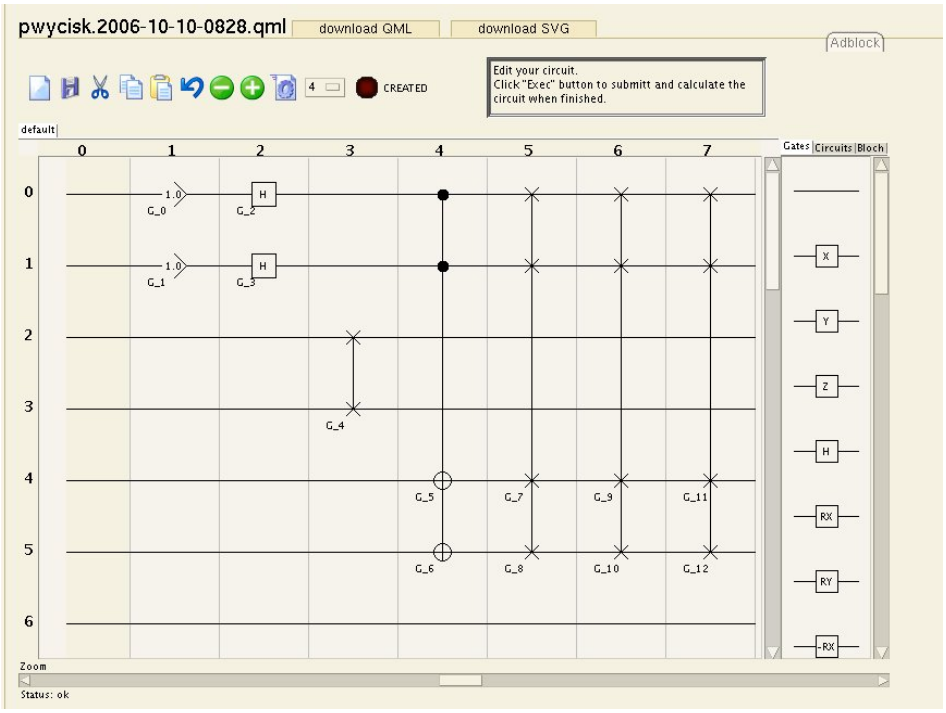
Fragment wygenerowanego pliku w QML:

```
<QML>
  <Job Id="">
    <Date Performed="0" Requested="1157483296394"/>
    <Status Info="1157483296394CREATED" Current="CREATED" Error="NONE"/>
    <Method Threshold="0.0050" Performed="NONE" Requested="AUTO"/>
    <Computation RunTime="0" QueueTime="NONE" Cpus="0" Accuracy="1.0" MBytes="0"
EstimatedTime="0"/>
  </Job>
```

```
<Circuit Name="default" Size="28" Id="default.qml" Description="">
  <Operation Step="0">
  </Operation>
  <Operation Step="1">
    <Application Name="G" Id="0" Bits="0">
      <Gate Type="PREPARATION" Probability="1"/>
    </Application>
    <Application Name="G" Id="1" Bits="1">
      <Gate Type="PREPARATION" Probability="1"/>
    </Application>
  </Operation>
  <Operation Step="2">
    <Application Name="G" Id="2" Bits="0">
      <Gate Type="HADAMARD"/>
    </Application>
    <Application Name="G" Id="3" Bits="1">
      <Gate Type="HADAMARD"/>
    </Application>
  </Operation>
  <Operation Step="3">
    <Application Name="G" Id="4" Bits="2,3">
      <Gate Type="SWAP"/>
    </Application>
  </Operation>
  <Operation Step="4">
    <Application Name="G" Id="5" Bits="0,4">
      <Gate Type="CNOT"/>
    </Application>
    <Application Name="G" Id="6" Bits="1,5">
      <Gate Type="CNOT"/>
    </Application>
  </Operation>
  <Operation Step="5">
    <Application Name="G" Id="7" Bits="4,0">
      <Gate Type="SWAP"/>
    </Application>
    <Application Name="G" Id="8" Bits="5,1">
      <Gate Type="SWAP"/>
    </Application>
  </Operation>
  <Operation Step="6">
    <Application Name="G" Id="9" Bits="4,0">
      <Gate Type="SWAP"/>
    </Application>
    <Application Name="G" Id="10" Bits="5,1">
      <Gate Type="SWAP"/>
    </Application>
  </Operation>
  <Operation Step="7">
    <Application Name="G" Id="11" Bits="4,0">
      <Gate Type="SWAP"/>
    </Application>
    <Application Name="G" Id="12" Bits="5,1">
      <Gate Type="SWAP"/>
    </Application>
  </Operation>
</Circuit>
<CircuitLib>
</CircuitLib>
<GateLib>
  <Gate Type="IDENT"/>
  <Gate Type="PAULI_X"/>
  <Gate Type="PAULI_Y"/>
  <Gate Type="PAULI_Z"/>
  <Gate Type="HADAMARD"/>
  <Gate Type="RX"/>
  <Gate Type="RY"/>
  <Gate Type="_RX"/>
  <Gate Type="_RY"/>
  <Gate Type="T_GATE"/>
```

```
    <Gate Type="S_GATE"/>
    <Gate Type="PHASE" Divisions="1"/>
    <Gate Matrix="1.0,i0.0,0.0,i0.0,0.0,i0.0,1.0,i0.0" Type="UNITARY_1"/>
    <Gate Type="CNOT"/>
    <Gate Type="SWAP"/>
    <Gate Type="CPHASE" Divisions="1"/>
    <Gate
Matrix="1.0,i0.0,0.0,i0.0,0.0,i0.0,0.0,i0.0,0.0,i0.0,1.0,i0.0,0.0,i0.0,0.0,i0.0,
0.0,i0.0,0.0,i0.0,1.0,i0.0,0.0,i0.0,0.0,i0.0,0.0,i0.0,0.0,i0.0,1.0,i0.0"
Type="UNITARY_2"/>
    <Gate Type="TOFFOLI"/>
    <Gate Type="FREDKIN"/>
    <Gate Size="2" Type="ORACLE" BasisState="0x255"/>
    <Gate Size="2" Type="GROVER" Steps="1" BasisState="0x1"/>
    <Gate Size="2" Type="GROVER_STEP" BasisState="0x255"/>
    <Gate Modulus="1" Size="3" Type="MODULO" Base="1" Index="1"/>
    <Gate Size="2" Type="EXP" Duration="0.25">
      <HTerm Matrix="0.0,0.0,1.0" Index="0"/>
      <HTerm Matrix="0.0,0.0,1.0" Index="1"/>
      <HTerm Matrix="0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0" Index="0,1"/>
    </Gate>
    <Gate Size="2" Type="REVERSE"/>
    <Gate Size="2" Type="QFT"/>
    <Gate Type="PREPARATION" Probability="1.0"/>
    <Gate Type="MEASUREMENT_Z"/>
    <Gate Size="2" Type="CIRCUIT"/>
    <Gate P1="0.5" Size="2" P0="0.5" Type="RANDOM" CaseSize="1"/>
  </GateLib>
</QML>
```

Rezultatem będzie obwód kwantowy przedstawiony w graficznym edytorze QML.



Rys.3. Obwód kwantowy wygenerowany przez qcl2qml – obwód kwantowy na rysunku składa się z zerowania, bramki hadamarda, bramki swap, kontrolowanej negacji oraz ciągu bramek swap

Fig.3. Quantum circuit generated by qcl2qml – quantum cirquit is composed of following gates: reset, hadamard gate, swap gate, controlled negation and sequence of swap gates

3. Podsumowanie

Przedstawiona praca ma na celu dostarczenie wygodnego narzędzia programistycznego wspierającego badania symulacyjne w dziedzinie informatyki kwantowej, poprzez połączenie dwóch różnych systemów programowania i symulacji komputerów kwantowych. Największą zaletą przedstawionego rozwiązania jest rozszerzenie możliwości systemów, opartych na języku QML, na metodach programowania proceduralnego. Język QML jest słabo skalowalny w takim znaczeniu, że kod złożonej bramki kwantowej jest definiowalny tylko dla danej liczby kubitów. QCL pozwala na obejście tego problemu, gdyż wprowadza on złożone struktury programistyczne.

LITERATURA

1. Aho A. V., Sethi R., Ullman J.D.: Kompilatory. Reguły, metody i narzędzia. WNT, Warszawa 2002.

2. Donnelly C., Stallman R.: The Bison. Yacc-compatible Parser Generator. Free Software Foundation, 2006. http://www.gnu.org/software/bison/manual/pdf/bison.pdf.
3. Gawron P.: Symulacja komputerów kwantowych. Praca magisterska. Politechnika Śląska w Gliwicach, Gliwice 2003.
4. Knill E.: Conventions for Quantum Pesudocode. Los Alamos National Laboratory, Los Alamos 1996.
5. Mauerer W.: Semantics and Simulation of Communication in Quantum Programming University Erlangen-Nurenmberg, 2005.
6. Nielsen M. A., Chuang I. L.: Quantum Computation and Quantum Information. Cambridge University Press, 2000.
7. Ömer B.: A Procedural Formalism for Quantum Computing. Department of Theoretical Physics Technical University of Vienna, 1998. http://tph.tuwien.ac.at/~oemer.
8. Ömer B.: Quantum Programming in QCL. Institute of Information Systems Technical University of Vienna. http://tph.tuwien.ac.at/~oemer
9. Preskill J.: Lecture Notes for Physics 229: Quantum Information and Computation California Institute of Technology, 1998.
10. Rieffel E., Polak W.: An Introduction to Quantum Computing for Non-Physicists. quant-ph/9809016.
11. Fraunhofer Quantum Computing Simulator – Manual. http://www.qc.fraunhofer.de.
12. Paxson V.: Flex, version 2.5. A fast scanner generator. Free Software Foundation, 1995 http://www.gnu.org/software/flex/manual/html_node/flex_toc.html.
13. Quantiki – the free-content WWW resource in quantum information science that anyone can edit. http://www.quantiki.org.
14. Wikipedia – The Free Encyclopedia http://www.wikipedia.org, 2006.
15. Schiff L. I.: Mechanika kwantowa. Państwowe Wydawnictwo Naukowe, Warszawa 1977.
16. Winiarczyk R., Gawron P.: Symulacja komputerów kwantowych. ZN Pol. Śl. Studia Informatica Vol. 22, No 3 (45), Gliwice 2001.
17. Wycisk P.: Programowanie komputerów kwantowych. Praca magisterska. Politachnika Śląska, Gliwice 2006.

Recenzent: Prof. dr hab. inż. Jerzy Klamka

Wpłynęło do Redakcji 1 października 2007 r.

Abstract

In the paper we present compiler of procedural Quantum Computation Language which produces as its outcome string of commands written in sequential Quantum Markup Language. The disadvantage of QML language is its low scalability. Usage of high-level language such as QCL solves this problem by introducing elements of procedural programming paradigm. It allows code reusability to solve similar problems of different size.

The compiler is divided to four parts:

1) lexical analyzer,
2) syntactic analyzer,
3) semantic analyzer,
4) execution module.

Lexical analyzer generates from string of characters string of tokens. Syntactic analyzer generates syntactic tree. Semantic analyzer uses execution module to generate internal representation called *qusim*. This internal representation is passed to QML code generator which may seen as specialized XML generator.

Adresy

Piotr GAWRON: Instytut Informatyki Teoretycznej i Stosowanej PAN,
ul. Bałtycka 5, 44-100 Gliwice, Polska

Jarosław MISZCZAK: Instytut Informatyki Teoretycznej i Stosowanej PAN,
ul. Bałtycka 5, 44-100 Gliwice, Polska

Ryszard WINIARCZYK: Instytut Informatyki Teoretycznej i Stosowanej PAN,
ul. Bałtycka 5, 44-100 Gliwice, Polska, ryswin@iitis.gliwice.pl

Piotr Wycisk: Politechnika Śląska, Instytut Informatyki,
ul. Akademicka 16, 44-100 Gliwice, Polska