Электронные системы

UDC 004.274

T.L. Zakharchenko
National Technical University of Ukraine “Kiev Polytechnic Institute”,
Peremogy ave., 37, Kyiv, 03056, Ukraine.

Adaptive hardware development

The paper is trying to attract attention to perspective hardware design methodology. The point of the methodology is to stress vital role of pragmatics in hardware design through direct influence of pragmatics on tools used in process of design. The author showed basics of pragmatics dependent approach and proposes solution for automated transition from semantic notation of solution of problem given to specific syntaxes, described its implementation.

The design approach introduced has such advantages as: investment saving, production of correct solutions and of course support of design process, which leads to optimal solution. The author applied proposed approach to hardware design, and such application allows increasing speed/chip area ratio in resulting designs. The implementation of adaptive design environment, created as the proof of concept proves ability to remove margin between software and hardware compiling the same semantic description of problem's solution into x86 assembly and Verilog code. But for now it is not recommended to use this piece of software in industry, it should be improved with different design tricks and optimizations to make designs produced with the adaptive design environment truly efficient. In the future the author is going try to use different algebraic structures in the adaptive design environment to analyze their efficiency. Reference 5, figures 2.

Keywords: *compositional approach, reconfigurable computing, adaptive hardware.*

Introduction

Nowadays the industries of hardware development, as well as of software development experiences crisis. The symptoms of the crisis are well-recognized. First, it is that developers experiencing problems with complexity management, as E. Dijkstra[1] claimed, that software has so deep hierarchical complexity that programmer should simultaneously deal with separate bits and hundreds of megabytes, the difference in sizes of approximately 1 to 10^9. Today’s software is even more complex. And sometimes developers are applying old methods of design even they are obviously irrelevant to task given. With size of projects number of defects grows too[2]. For example if project is less then 2 KLOC (thousands lines of code), it has 0-25 defects per KLOC. In case of big projects >512 KLOC it has 4-100 per KLOC. Errors on the early stages of design are leading to big expanses. For example, error in architecture found on the test stage costs 15 time higher then it would be fixed on architecture design stage[3]. The other problem is so called “investments saving problem”. One can not separate solution of task given from syntax of its implementation, due to this we have losses of the quality code when changing development platform.

The root of crisis described is simplified understanding of design process and developer’s orientation on the goal without taking process of design in account. Really, the design process involves three aspects: pragmatics, semantics, and syntax. It is descending from pragmatics to semantics and all of them supplement one another. Such process widely acknowledged among developers but very often violated by ignoring pragmatics.

In order to solve or mitigate mentioned above problems, we propose new design method which will take pragmatics in consideration and will support design process of developer. It is incomparable to other approach because it possesses new qualities which old ones do not have. The method applicable to both software and hardware design processes. Problems already faced in software design will soon become known in hardware design. So following method will applicable to both hardware and software.

Now consider representatives of existent design approaches. There are plenty reconfigurable patterns developed and described in DeHon et al [4] can be considered as step forward to reusability. An effort towards multiplatform code to mitigate investments saving problem was undertaken by M. Tarver, creator of LISP based Shen language, which produces program code in various other languages [5]. The problem getting more important with development of new computing platforms for which old designs can not be applied because of developer's concentration on result.

Open-closed and compositional approaches

First, let us consider open and closed systems. In the design process one divides the task on primary and secondary (important and not important) parts. Everything is secondary in open system, but this design system is the most robust system, where almost everything can be changed, but there are few tools to conduct these changes. It is up to developer to create and select tools to design within this system. Considerable flaw of open system is that developer's design process mostly performed in his or her mind. Contradictive to open system is closed system approach. Everything is secondary in such systems. It has a rigid structure and a few points to change. However it has rich set of specialized tools to do such changes. Both of them are not optimal extremes.

Normally, designer does not fallow any one of these approaches. He or she selects both primary and secondary parts. But this division is deeply subjective. And we try too bring a science to this process of division, because for now it is more art then science.

Every open-closed system may be considered as open-closed environment, which can produce new open-closed system, which can be considered as open-closed environment too. This iterative process starts with open-closed system with consists of initial set of genetic structures, which can be combined to produce derivatives. Those derivatives may be used as genetic structures of iterative process on the next step (fig. 1). Every step and selection of initial set of genetic structures is defined by pragmatics.

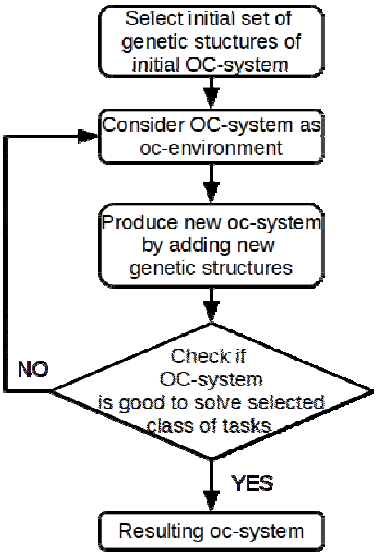

Fig. 1. OC-system design process

Due to usage of algebraic structures, it is possible to design pragmatically and mathematically correct hardware and programs, because they designed with algebraic approach with taking pragmatics in account and relevant design tools are provided. So there will be no side effects in resulting solution, therefore no bugs. By explicit definition pragmatics through genetic structures, many mistakes on earlier design stages will be eliminated. Strict appliance of "rule and divide" principle in design process will help to manage complexity of task. Investment saving achieved by possibility of translate such semantic solution to any syntax needed and by well defined design process string with pragmatics. The resulting OC-system may be treated as new "programming language" oriented on specific class of problems. Costs for new language development are very high. C programming language had been developed from 1969 to 1973 by team in Bell Labs. Java programming language had been developed from 1979 to 1983 by "Green Team" in Sun. With proposed approach new language design time will be significantly reduced due to automation of design process. Moreover, development time of Java would be cut if C programming language would used as basis for the new language. But there was no means to use it as basis. Development of both Java and C continues to nowadays, but there is no means to support the process of development of those languages.

Genetic structures form algebra. And every computational task can be decomposed with Turing-complete algebra, because, basically, computational task is function. There are two types of genetic structures: compositions and functions.

Compositions are operations over set of operations and other compositions.

The expression that describes composition looks like following:

$$G = f(f_1, f_2, ...f_n),$$

where f is composition itself, and f1, f2,…fn function used as arguments of composition. And functions look like:

$$F = f(a_1, f_2, ..., a_n),$$

where f is function and a1, a2, an are elements of carrier set (e.g integers).

The result of composition application is function. Result of application looks as follows:

$$F = f(f_1(x_1, x_2, ..., x_i), f_2(x_1, x_2, ..., x_k), ...,$$
$$f_n(x_1, x_2, ..., x_k), x_1, x_2, ..., x_m)$$
$$i, j, k < m$$

This approach is applicable to both hardware and software. Explicit usage of compositions increases correctness and may clarify point of departure on the start of design process.

Implementation

Proposed methodic can be partly automated in part where semantic solution translated to syntax. Software for transition from semantic representation to syntax was successfully implemented as a proof of concept. Church's algebra was chosen as algebraic structure the system. Basic operations of the algebra are: increment, assignment to zero, and selection argument by specific index. Compositions are: application, primitive recursion, and minimization.

Primitive recursion is composition that takes two functions as arguments R(g,h). Firstly, it computes initial argument update with function *g*. Then iteratively apply function h until stop:

$$\begin{aligned} &f(X,0) = g(X) \\ &f(X,1) = h(X,0,g(X)) \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &f(X,m+1) = h(X,m,f(X,m)) \end{aligned},$$

where X is set of arguments.

Minimization $M(f(x_1,x_2,...,x_n))$ is to find the smallest root of equation

$$f(x_1,x_2,...y) = x_n,$$

where y increments by one starting with 0 until expression become truth.

On the input semantics solution is passed. The software parses semantic representation of solution into semantic tree according to rules chosen for semantics formalization. Leaves of the tree are input variables and constants and non-terminal vertices are compositions and metacompositions (compositions of compositions and operations).

It is selected JSON representation of the tree. Every vertex represented in fllowing form:

```
{
        "name":character string,
        "id":numeric identifier,
        "static":[parameter list],
        "arguments":[{"no":1,
"value":argument value},...]
}
```

The name field is mnemonic designator of function, performed by specific vertex, id field is numeric vertex identifier for non-ambiguous interpretation of vertices. static field is list of vertex parameters. They can be terms and/or values. Here term means semantic representation of other tree. Term may be an argument of compositions which are to manipulate them. The arguments field is a list of function(non-terminal vertex) arguments, in other words, list of vertices, which are connected to this one. Every element of list characterized by number no and value value which is vertex. For example, a tree, which represents expression add(IN0, mul(IN1, IN2)) looks like this:

```
{ "arguments": [
            {"value": "IN0",
            "no": 0},
            {"value": {
                  "arguments": [
                  {"value": "IN1",
                  "no": 0},
                  {"value": "IN2",
                  "no": 1}],
                  "static": [],
                  "name": "mul",
                  "id": "384009"},
            "no": 1
            }],
      "static": [],
      "name": "add",
      "id": "415422"}
```

Add and mul are operations of addition and multiplication respectively. IN0, IN1, IN2 are input arguments for the expression.

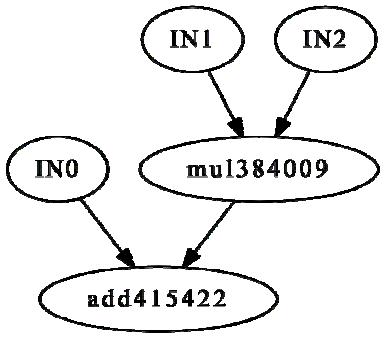

Fig. 2. add(IN0, mul(IN1,IN2))

For convenience of perception, the system produces tree representation of expression (fig. 2). It has some non-terminal vertices computer not familiar with, but algorithm of the system can replace them substituting instead non-familiar non-terminal vertices compositions of familiar ones: zero generators, operation of increment, operation of selection one of arguments to send it to output with static parameter, which denotes index of selected

argument, and recursion composition. Such compositions represented in JSON format too. Vertices with names starting with "R" represent primitive recursion composition.

The system produces HDL representation from such tree. To prove universality of the system, Moreover, generation of x86_64 assembly was added as well. The system developed with Python programming language. It is agile enough to modify the system quickly.

Conclusions

The new design approach introduced has such advantages as: investment saving, production of correct solutions and of course support of design process, which leads to solution of correctness problem, both mathematical and pragmatical. This allows to produce solution without side-effects. The investment saving problem is solved as well. The implementation of adaptive design environment, created as the proof of concept proves ability to remove margin between software and hardware compiling the same semantic description of problem's solution into x86 assembly and Verilog code. But for now it is not recommended to use this piece of software in industry, it should be improved with different design tricks and optimizations to make designs produced with the adaptive design environment truly efficient. In the future author is going try to use different algebraic structures in the adaptive design environment to analyze their efficiency.

References

1. *A. DeHon, J. Adams, M. deLorimier et al.* (2004) «Design patterns for reconfigurable computing», Field-Programmable Custom Computing Machines. 12th Annual IEEE Symposium on, Pp. 13-23.
2. 2. *M. Tarver* (2013), «The Book of Shen», Upfront Publishing Limited, P. 404.
3. 3. *David S. Alberts* (1976), «The economics of software quality assurance», AFIPS '76 Proceedings of, Pp. 433-442.
4. 4. *R. Camposano, D. Gope, S. Grivet-Talocia, V. Jandhyala* (2012), «Moore meets Maxwell», Design, Automation Test in Europe Conference Exhibition (DATE), Pp. 1275–1276.
5. 5. *A. Maltsev* (1972), «Algorithms and recursive functions», Wolters-Noordhoff Pub. Co., P. 368.
6. 6. *V. Redko* (1998), «Compositional programming basics», Programmirovanie, vol. 4, Pp. 3–13. (Rus)

Поступила в редакцию 20 сентября 2014 г.

УДК 004.274

Т.Л. Захарченко

Національний технічний університет України “Київський політехнічний інститут”,

вул.Політехнічна, 16, Київ, 03056, Україна.

Розробка адаптивного апаратного забезпечення

У статті автор автор привертає увагу до перспективної методики розробки апаратного забезпечення. В її основі лежить чільне місце прагматики у розробці апаратного забезпечення через її вплив на інструментарій, що використовується у процесі розробки. Автор пояснює основи прагматикозалежного підходу та пропонує рішення для автоматичного переходу від семантичного запису вирішення задачі до конкретного синтаксису та описує реалізацію.

Запропонована методика розробки має такі переваги як: збереження інвестицій, продукування коректного рішення та, звичайно, підтримка процесу розробки, що приводить до оптимальних рішень. Запропонований підхід автор застосував до конструювання апаратного забезпечення, що дозволило збільшити відношення швидкість роботи/площа кристалу. Реалізація адаптивного середовища розробки, створена як доказ концепції, доводить можливість нівелювати різницю між апаратним та програмним забезпеченням через компіляцію семантичного опису рішення задачі в мову асемблера або Verilog. Але безпосередньо це середовище адаптивної розробки ще не достатньо якісне для використання на виробництві, його варто значно покращити. Бібл. 5, рис.2.

Ключові слова: *композиційний підхід, реконфігуровані обчислення, адаптивне апаратне забезпечення.*

УДК 004.274

Т.Л. Захарченко
Национальный технический университет Украины “Киевский политехнический институт”
вул.Политехническая, 16, Киев, 03056, Украина.

Разработка адаптивного аппаратного обеспечения

В статье автор пытается привлечь внимания к перспективной методике разработки аппаратного обеспечения. В ее основе лежит основополагающая роль прагматики задачи в разработке аппаратного обеспечения посредством влияния оной на инструментарий, используемый в процессе разработки. Автор объясняет основы прагматикозависимого процесса разработки и предлагает решение для автоматического перехода от семантической записи решения задачи до конкретного синтаксиса, описывает реализацию.

Предложенная методика разработки имеет такие преимущества как: сохранение инвестиций, продуцирование корректного решения и, конечно, поддержка процесса разработки, что приводит к оптимальному решению. Предложенный подход автор применил к конструированию аппаратного обеспечения, что позволило увеличить отношение скорость работы/площадь кристалла. Реализация адаптивной среды разработки, созданная как доказательство концепции, доказывает возможность нивелировать разницу между программным и аппаратным обеспечением посредством компиляции семантического описания решения задачи в код языка ассемблера либо в Verilog. Но на сегодня конкретно это решение не рекомендуется использовать на производстве, его следует значительно улучшить. Библ. 5, рис.2.

***Ключевые слова**: композиционный подход, реконфигурируемые вычисления, адаптивное аппаратное обеспечение.*

Список использованных источников

1. *E.W. Dijkstra* On the Cruelty of Really Teaching Computing Science (EWD-1036) . — center for American History, University of Texas at Austin, 2004.
2. *Jones, T.* Capers Program Quality and Programmer Productivity .— IBM technical report TR 02.764. — 2004. — pp. 42-78.
3. Steve McConnell Code Complete. — Microsoft Press, 2009. — 960p.
4. *A. DeHon, J. Adams, M. deLorimier et al*. Design patterns for reconfigurable computing. // Field-Programmable Custom Computing Machines. 12th Annual IEEE Symposium on. — 2004. — pp. 13-23.
5. *M. Tarver*, The Book of Shen .— Upfront Publishing Limited, 2013 .— 404 p.