Templet: Domain-specific Language for Concurrency Control*

Sergey V. Vostokin
Dept. of Information Systems and Technologies
Samara State Aerospace University named after S.P. Korolyov (national research university)
Samara, Russia
sergey.vostokin@gmail.com

***Abstract*—The article presents a new domain-specific language (DSL) for concurrency control. Runtime library is a common way to implement concurrency control. However, this method often leads to an increase in application complexity. The design of new concurrent language is natively difficult. We propose a compromise solution which uses DSL with C++ programming language. The article discusses DSL syntax, programming model, and implementation of concurrency control in the Templet language.**

Keywords—domain-specific language; concurrency control; actor model; language-oriented programming; skeleton programming

I. INTRODUCTION

The Templet language [1] is a domain-specific language (DSL). It is designed to be used together with a sequential procedural or an object-oriented programming language. The new property of the language is an explicit specification of the process-channel computation semantics with a marked serial code.

The article focuses on a design of the markup language. The design concepts of the language basically follow the concept of the language-oriented programming [2,3]. The algebraic-like notation similar to the CSP formalism was applied to describe processes and interactions [4]. The idea of a minimalistic design with emphasis on the basic abstractions is taken from the programming language Oberon [5].

The language design is based on the three concepts. The first one is so called active markup. Usually a markup is read form the source file and produces some effects in the target file (e.g. adding synchronization and/or communication commands). In our approach source and target is one file. The file is rebuild during preprocessing. The markup controls an file transformation to keep a desired code structure.

The second one is a programming model. We introduce a diffusive (with no locking) programming model that describes concurrent activity as a message exchange between sequential processes (agents, actors). They are activated by incoming messages. The channels defines message exchange protocols. The model avoids concurrent data access, hence it is easier to use when multithreading.

The third concept is based on description of concurrent activity with sequential code. This method is derived from a formal theories that consider parallel process as set of behaviors (sequences of system states and/or atomic actions). We simulate such a sequences with random number generator.

The following paragraphs illustrate these concepts with Templet syntax and code examples. The article ends with an overview of language benefits. The experimental preprocessor and code samples in marked C++ are available at http://templet.ssau.ru/templet.

II. ACTIVE MARKUP

To describe the syntax, an extended Backus-Naur Formalism called EBNF is used. The following EBNF rules describe the block structure of a module.

```
module = {base-language|user-block} module-scheme
         {base-language|user-block}.
user-block = user-prefix base-language
             user-postfix.
module-scheme =  scheme-prefix
           { channel | process } scheme-postfix.
```

The module code consists of a single module scheme section and multiple code sections in C++ language with highlighted user blocks. These sections are distinguished from the rest of the code by means of C++ comments. For example, the marked C++ code may look as follows. The blocks' names according to the markup language syntax are shown on the right side.

```
#include <runtime.h>          <-- base-language

/*templet$$include*/          <-- user-prefix
  #include <iostream>         <-- base-language
/*end*/                       <-- user-postfix

/*templet*                    <-- scheme-prefix
  *hello<function>.           <-- module-scheme
*end*/                        <-- scheme-postfix

void hello(){                 <-- base-language
/*templet$hello$*/            <-- user-prefix
  std::cout <<
''hello world!!!'';           <-- base-language
/*end*/                       <-- user-postfix
}                             <-- base-language
```

Lexical analyzer defines the boundaries of the blocks by signatures, recognizing specific sub-strings in a character

This work was supported by the Ministry of Education and Science of the Russian Federation within the framework of the Program designed to increase the competitiveness of SSAU among the world's leading scientific and educational centers over the period from 2013 till 2020; and it was partially supported by the RFBR grant 15-08-05934 A.

stream. For example, the module scheme may be preceded by a combination of characters `/*templet*`, and finish by `*end*/`. User block prefixes include identifiers for binding the blocks with module scheme: `/*templet$hello$*/` bound with `*hello<function>`. The module is a program skeleton, and user blocks are extension points. Module scheme defines the structure of program skeleton.

The markup language provides a *mapping algorithm*. Mapping is a module transformation carried out by rewriting the module code. Mapping is applied only to a module with syntactically correct scheme. As a result of this transformation the code and the scheme becomes isomorphic meaning that the scheme can be reproduced from the code and vice versa. New user blocks may appear. Existing user blocks may move to new positions or turn into comments.

III. PROGRAMMING MODEL

The module scheme includes definitions of the two DSL classes: channel and process. The channel describes communication, while the process describes data processing. Any DSL class inherits its behavior from `BaseChannel` or `BaseProcess` Templet runtime classes. The classes should be implemented in a way that the following behavior is possible.

```
class Channel: public BaseChannel{
 public:
  // test whether the channel it accessible
  bool access_client(){...} // at client side
  bool access_server(){...} // at server side
  // client sends entire channel to server
  void send_client(){...}
  // server sends entire channel to client
  void send_server(){...}
...
};

class Process: public BaseProcess{
 public:
  // receive data on the channel
  virtual void recv(BaseChannel*);
  // bind a channel to the process as client
  bool bind_client(BaseChannel*){...}
  // .. or server
  bool bind_server(BaseChannel*){...}
...
}
```

The `BaseChannel` has the following behavior. The access to the channel alternately belongs to pair of processes called client and server. The client process has access right to the channel in the beginning of computations. Methods `access_client()` and `access_server()` allow client or server to check for access. Methods `send_client()` and `send_server()` can be used to grant access from client to server or from server to client respectively.

The `BaseProcess` has the following behavior. Methods `bind_client()` and `bind_server()` establish connection between a process (as a client or as a server) and a channel. Method `recv()` is called at the moment getting access to the channel. The channel is passed as `recv()` argument.

The implementation also carries out the rules below. If the process gets access to multiple channels, it takes several consecutive calls to `recv()` in random order. If some process sends the access to another process, the other process will sooner or later get the access to a channel.

IV. CONCURRENT EXECUTION SEMANTICS

The program implementation in C++ language should provide the opportunity for nondeterministic performance. Nondeterminism of program execution is simulated by means of pseudo-random numbers.

```
void TempletProgram::run()
{
  size_t rsize;
  // while message queue is not empty
  while(rsize=ready.size()){
    //select random channel which
    //is currently sending message
    //then exclude this channel
    //from the message queue
    //and move it to not sending state

    int n=rand()%rsize;
    auto it=ready.begin()+n;
    BaseChannel*c=*it;ready.erase(it);
    c->sending=false;

    //extract the process to which the message
    //was sent from the channel
    //run message handling method recv()
    //for the channel and
    //pass the channel as
    //the argument to this method

    c->p->recv(c);
  }
}
```

For truly parallel execution of code appropriate libraries are necessary. Some modifications to mapping algorithm may also be required.

V. MODULE SCHEME SYNTAX

This is a complete EBNF description of module scheme in the Templet language.

```
channel = '~' ident [params]
          ['=' state {';' state}] '.'.
state = ['+'] ident [ ('?'|'!')  [rules] ].
rules = rule { '|' rule }.
rule = ident { ',' ident } '->' ident.
process = '*' ident [params]
          ['=' ((ports [';' actions])
            | actions) ] '.'.
ports = port {';' port}.
port  = ident ':' ident
        ('?'|'!')[(rules ['|' '->' ident])
        |( '->' ident)].
actions = action {';' action}.
action  = ['+'] [ident ':'] disjunction ['->'
           ([ident] '|' ident) | ident].
disjunction = conjunction { '|' conjunction}.
conjunction = call {'&' call}.
call = ident '(' [args] ')'.
args = ident ('?'|'!') ident
     {',' ident ('?'|'!') ident }.
params = '<' ident {',' ident} '>'.
```

For example, there is a program that checks the trigonometric identity $\sin^2 x+\cos^2 x=1$. When process of `Master` class sends x values to working processes of `Worker` class via channels of `Link` class. The master gets the squares of trigonometric functions and calculates their sum in return. Channel protocol to verify the trigonometric identity may be coded in Templet DSL like below.

```
~Link = +BEGIN ? ArgCos -> CALCCOS |
                 ArgSin -> CALCSIN;
```

```
        CALCCOS ! Cos2 -> END;
        CALCSIN ! Sin2 -> END.
```

Processes for checking the trigonometric identity can be defined as follows.

```
   *Master =
          p1:Link ! Sin2 -> join;
          p2:Link ! Cos2 -> join;

         +fork(p1!ArgSin,p2!ArgCos);
          join(p1?Sin2,p2?Cos2).

   *Worker =
          p : Link ? ArgSin -> sin2 | ArgCos ->
cos2;
          sin2(p?ArgSin,p!Sin2);
          cos2(p?ArgCos,p!Cos2).
```

A program is a network of objects. Objects are instances of classes in C++ programming language. These classes are in turn derived from channels or processes in the Templet language. The network of objects is defined in the C++ programming language.

VI. Applications Overview

The current implementation includes another three samples to illustrate the practical use of the Templet domain language.

The Gauss-Seidel method for solving the Laplace equation is th first one. This example illustrates the use of the toolkit in the field of scientific computing. It also shows how the simulation runtime can help to predict program performance without an explicit mathematical model or parallel execution.

An example from the field of linear algebra is the second one. This is an illustration of distributed matrix multiplication algorithm. Our implementation of the actor model is well suited both for shared and distributed parallel architectures.

The business process model example is the third one. The Templet DSL can be used to model and analyze concurrency in non-technical systems, for example, in the area of business process modeling. We studied a business scenario written in a human language and composed a formal specification for the scenario in the Templet language. The static type analysis, debugging, and testing of the program were used to verify the correctness of the specification. In particular, we compared programmatically generated event sequences with expected sequences for the studied business process. This example illustrates that in our approach much of model verification is done by C++ compiler and Templet runtime.

VII. Results

The implementation of the domain-specific language toolkit showed the following benefits of our approach.

Additional language constructions are not required to explain the meaning of an algorithm with concurrent control. This is similar to approach based on object-oriented libraries STL [6], TBB [7], CCR [8], Boost [9], and others. However, the markup and preprocessing technique reduces the amount of manual coding.

More reliable protection against programming errors is provided. This feature is compatible with concurrent programming languages Go [10], Occam [11], Limbo [12], Erlang [13]. Static type checking in the C++ language helps to prevent incorrect connection of message source and message recipient. Semantic checking can also be implemented at the preprocessor level. For example, one can check the attainability of a state in the communication protocol for channels and the possibility to call a method for processes. The check can also be carried out during the program execution. If pair of processes does not perform a prescribed messaging protocol, calculations will stop.

Behavior of the Templet program can be investigated in more detail by means of problemoriented debugger. The mapping algorithm can add code to provide information to the debugger. The performance prediction of a parallel program is also possible. Discrete event simulation library can easily replace standard execution mechanism.

The markup language is a mean of skeleton programming and code reuse [14, 15]. One can design a universal skeleton for programs with similar control flow and adapt it to specific applications. The adaptation is made by the changing of message variables and handlers. This technique can be used for programming multi-core and many-core systems [16, 17].

The markup language defines concurrent execution with sequential code. This technique is used in incremental lateralization. A number of well-known [18,19] and experimental [20,21] tools for defining iterative or recursive parallelism are based on markup. We adopted the same method for a process-channel model of concurrent control.

The DSL language can be applied to different general-purpose programming languages. It is compatible with the modern technologies [22,23] used in industrial process control software development.

VIII. Conclusion

Our research shows a practical interest of the DSL-based approach for concurrency control. We got a fully working but relatively simple implementation of the Templet domain-specific language. This implementation had been deployed online as a part of the web service *TempletWeb* (template.ssau.ru/templet).

References

[1] S. Vostokin, "Templet: a markup language for concurrent programming," arXiv preprint arXiv:1412.0981, 2014.

[2] M.P. Ward, "Language-oriented programming," Software-Concepts and Tools, vol. 15, no. 4, pp. 147–161, 1994.

[3] S. Dmitriev, "Language oriented programming: The next programming paradigm," JetBrains onBoard, vol. 1, no. 2, pp. 1–13, 2004.

[4] C. Hoare, "Communicating sequential processes," in The origin of concurrent programming. Springer, 2002, pp. 413–443.

[5] N. Wirth, "The programming language Oberon," Software: Practice and Experience, vol. 18, no. 7, pp. 671–690, 1988.

[6] B. Stroustrup, The C++ programming language. Pearson Education, 2013.

[7] J. Reinders, Intel threading building blocks: outfitting C++ for multi-core processor parallelism. "O'Reilly Media, Inc.2007.

[8] J. Richter, "Concurrent affairs-concurrency and coordination runtime." MSDN MagazineLouisville, pp. 117–128, 2006.

[9] B. Schäling, The boost C++ libraries. Boris Schäling, 2011.

[10] The Go programming language specification. Google Inc., 2009, http://golang.org/doc/go_spec.html.

[11] Occam programming manual. INMOS Limited. Prentice Hall Direct, 1984.

[12] D.M. Ritchie, "The Limbo programming language," Inferno Programmer(TM) Manual, vol. 2, 1997.

[13] J. Larson, "Erlang for concurrent programming," Communications of the ACM, vol. 52, no. 3, pp. 48–56, 2009.

[14] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," Parallel computing, vol. 30, no. 3, pp. 389–406, 2004.

[15] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," Software: Practice and Experience, vol. 40, no. 12, pp. 1135–1160, 2010.

[16] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Skeletons for multi/many-core systems." in PARCO, 2009, pp. 265–272.

[17] Y. Karasawa and H. Iwasaki, "A parallel skeleton library for multi-core clusters," in Parallel Processing, 2009. ICPP'09. International Conference on. IEEE, 2009, pp. 84–91.

[18] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," Computational Science & Engineering, IEEE, vol. 5, no. 1, pp. 46–55, 1998.

[19] R. Blumore, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," Journal of parallel and distributed computing, vol. 37, no. 1, pp. 55–69, 1996.

[20] N. Konovalov, V. Krukov, and Y. L. Sazanov, "C-dvm-a language for the development of portable parallel programs," Programming and Computer Software, vol. 25, no. 1, pp. 46–55, 1999.

[21] S. Abramov, A. Adamovich, A. Inyukhin, A. Moskovsky, V. Roganov, E. Shevchuk, Y. Shevchuk, and A. Vodomerov, "Opents: an outline of dynamic parallelization approach," in Parallel Computing Technologies. Springer, 2005, pp. 303–312.

[22] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation," Software, IEEE, vol. 20, no. 5, pp. 36–41, 2003.

[23] B. Selic, "The pragmatics of model-driven development," IEEE software, vol. 20, no. 5, pp. 19–25, 2003.