The Use of Temporal Logic to Represent the Templet Language Runtime Behavior*

Sergey V. Vostokin

Samara State Aerospace University, 34, Moskovskoye shosse, Samara, 443086, Russia

Abstract. The article presents the concurrent programming language named Templet from the point of its formal specification. The language is based on the actor formalism. We use Lamport's Temporal Logic of Actions (TLA) to define the language runtime library behavior. A review of the programming tool functionality including domain-specific language Templet, the runtime library, and some practical applications is given.

1 Introduction

The Templet is a domain-specific language (DSL) for concurrent programming. The language programming tools include a preprocessor and several runtime libraries that can be found at `https://github.com/Templet-language`. The DSL demonstrates a novel approach to extend C++ programming language with actor semantics of execution. The approach is a variant of the language-oriented programming, where automatic code synthesis is guided by DSL.

Our design goal was to make a simple and practical tool for concurrent programming based on the actor formalism. The actor model gives advantages when applications are highly-parallel by nature: multi- and many- core systems, high performance computing, industrial control systems, Internet of Things, business process management, and others.

We propose a formal specification of the actor programming model for the Templet language. The goal of the specification is to explain how the Templet language runtime works independently from a particular implementation. The specification helps application programmers to understand Templet DSL semantics. Also the DSL serves as a guideline for runtime library implementers.

2 Related Works

Well-known parallel programming tools are implemented as a library (Intel TBB, Akka), entirely new language (Go, Scala), or extension for existing language

* 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.

(OpenMP, Intel Cilk Plus). We propose a DSL-based approach [1] which is natively compatible with existing compilers, libraries, integrated development environments. We use the actor model [2] that is a good basis for effective execution in shared and distributed memory architectures (e.g. Erlang). Our laguage is a kind of skeleton programming tools [3]. The DSL algebraic-like syntax is similar to Hoare's CSP notation [4]. The idea of simple design for complex problems was implied by Wirth's Oberon [5] The language supports model-driven development [6]. However, the use of a temporal logic for actor-like runtime library specification was not studied earlier.

The article has the following structure. First, we introduce the formal specification of the variant of actor programming model for the language. We use the Temporal Logic of Actions introduced by Leslie Lamport [7] for the specification. Then, we describe a runtime library that implements given specificafion. After that the binding between runtime library and application code is discussed. Finally, we make a brief overview of some practical examples.

3 Specification of Programming Model

The tool was designed for a synthesis of programs with actor execution semantics. Our method of actor model implementation focuses on passing the right of access to objects and on passing the activity between actors rather then on classical message passing. This approach enables us to build precise specification and a compact implementation of the runtime.

Let Var be a set of all program variables. We suppose that there is a function $\mathbb{IF}$ that defines to which actor (process) or channel (message) a variable belongs[1].

$$\mathbb{IF} : \mathrm{Var} \to \{actor, message\} \times \mathbb{N} \ , \tag{1}$$

where a number from set $\mathbb{N}$ denotes an actor or message identifier. In a specific implementation (namely in C++) this binding may be expressed with classes, structures, or simply implied by a programmer.

Let the set

$$\{p[1], p[2], \ldots, p[i], \ldots, cp[1], cp[2], \ldots, cp[j], \ldots, c[1], c[2], \ldots, c[j], \ldots\} \tag{2}$$

denotes all runtime variables with the following components: (a) the process activity for the actor i is expressed as $p[i] \in \{0, 1\}$, where Boolean 0 means *not active*, and Boolean 1 means *the actor is active, it handels a message*; (b) the channel activity is expressed as $c[j] \in \{0, 1\}$, where Boolean 0 means that *the message already arrived* to an actor, and Boolean 1 means that *the message is transmitted*; (c) the array $cp[j] \in \mathbb{N}$ of runtime variables stores an identifier of an actor to which message j belongs. A message belongs to an actor, if this

[1] We use terms *actor* and *process* interchangeably. The actor is an object that controls the sequential *process* of *message* handling. Also we assume that the *channel* is an object that controls *message* exchange.

message is being transferred to this actor now, or had been transferred to this actor already.

By using these designations we can express the runtime atomic actions that change the system state.

$$A_1 \equiv \exists! j : \neg p[i] \wedge c[j] \wedge cp[j] = i \wedge p'[i] \wedge \neg c'[j] \wedge cp'[j] = i \ . \tag{3}$$

The formula (3) means that (a) when message arrives, it activates a handling procedure; and (b) only one message can be handled at the same time[2].

$$A_2 \equiv p[i] \wedge \neg p'[i] \ . \tag{4}$$

The action denoted by the formula (4) is executed at the end of message handling.

$$A_3 \equiv \exists i : p[i] \wedge cp[j] = i \wedge \neg c[j] \wedge c'[j] \ . \tag{5}$$

The formula (5) means that any message which is accessible during a handling procedure can be sent to any actor.

The formula (6)

$$A_4 \equiv c[j] \wedge \neg c'[j] \tag{6}$$

denotes a message coming to an actor.

The *liveness* properties of the runtime assume that, if actions A_2 (4) and A_4 (6) are enabled long enough they will eventually be executed. There are so called *weak fairness* conditions (WF). Combining formulas (3)-(6) with the temporal operators and assuming that the initial system state is $I \equiv \exists i : p[i] \vee \exists j : c[j]$, actor i state is $f_1 \equiv p[i]$, and channel j state is $f_2 \equiv (c[j], cp[j])$ we finally have formula

$$S \equiv I \wedge \Box[A_1 \vee A_2]_{f_1} \wedge \Box[A_3 \vee A_4]_{f_2} \wedge WF_{f_1}(A_2) \wedge WF_{f_2}(A_4) \tag{7}$$

that completes the runtime behavior specification[3].

4 Runtime Library Implementations

The specification (7) is used to define the primitive operations of the language runtime library. The runtime has three primitive operations. The first, `recv()` is a callback procedure. It is activated in the context of some actor i to process the received message j:

$$recv_{(call)}(i, j) \equiv \neg p[i] \wedge c[j] \wedge cp[j] = i \wedge p'[i] \wedge \neg c'[j] \wedge cp'[j] = i \ , \tag{8}$$

[2] To express a change of the system state we use primed variables (var') in logical formulas. They denote the values of the variables in the next state ($t+1$), while the non-primed variables (var) express a values in the current system state (t). Note that variables i and j are temporal constants. Their values are unknown, but constant in time.

[3] The notation $\Box[A]_f$ defines that for every pair of system states expression $A \vee (f = f')$ is true. This mean that at any time the system state is changed by A action or remains unchanged.

$$recv_{(return)}(i) \equiv p[i] \wedge \neg p'[i] \ . \tag{9}$$

The action (8) is performed when the `recv()` procedure is called from the runtime. The action (9) is performed on `recv()` procedure return.

The second, `access()` is a logical function. It is called by a programmer from the `recv()` procedure. The function tests whether variables associated with the message j are accessible from the handling procedure of the actor i:

$$access(i, j) \equiv cp[j] = i \wedge \neg c[j] \ . \tag{10}$$

Finally, `send()` is a procedure for the message sending. This procedure is also called by a programmer from `recv()` procedure. The procedure sends a message j to the actor i:

$$send(i, j) \equiv cp'[j] = i \wedge c'[j] \ . \tag{11}$$

The following C++ code fragment is a part of the language runtime library.

```
struct engine{ std::vector<chan*> ready;};// engine
struct proc{ void(*recv)(chan*, proc*);};// actor/process objects
struct chan{ proc*p; bool sending;};// message/channel objects

inline void send(engine*e, chan*c, proc*p){
  if (c->sending) return;
  c->sending = true;
  c->p = p;
  e->ready.push_back(c);
 }
 inline bool access(chan*c, proc*p){
     return c->p == p && !c->sending;
 }
 inline void run(engine*e){
  size_t rsize;
  while (rsize = e->ready.size()){
    int n = rand() % rsize; auto it = e->ready.begin() + n;
    chan*c = *it; e->ready.erase(it); c->sending = false;
    c->p->recv(c, c->p);
  }
}
```

This runtime library is used to simulate concurrent execution on a single processor. The call to C++ library function `rand()` is used to randomize a program behaviors. The library is a debug implementation of the specification (7). The library code is located in the file `cpp11runtime/lib/dbg/tet.h`. We also have a runtime support for effective sequential execution (`.../seq/tet.h`), parallel multithreaded execution (`.../par/tet.h`), and speedup prediction using discrete event simulation (`.../sim/tet.h`) in the `cpp11runtime` repository. These runtimes are also implementations of the model (7).

5 Binding Between Runtime and Application Code

The direct use of the runtime library may be difficult for the following reasons: (a) the relation of formula (1) between a variable and its control object (actor or message) should be syntactically supported; (b) every read/write operation with message variable should be preceded by `access()` call (10); (c) the code should present entities in the application domain. The Templet preprocessor was implemented to solve these problems. It performs the synthesis of the required additional code.

The current preprocessor implementation uses special comments to separate the three parts: DSL commands, manually written code, and automatically generated code. The DSL commands are used to define properties of actors and messages. Actors and message are entities of corresponding C++ classes.

For example, DSL command `*actor.` inside the comments

```
/*$TET$templet$!templet!*/
/* *actor. */
/*$TET$*/
```

means, that the preprocessor should generate a C++ class `actor` with actor behavior. The fragment of this class is shown below.

```
class actor:public TEMPLET_DBG::Process{
/*$TET$actor$!userdata!*/
double some_user_defined_variable;
/*$TET$*/
};
```

The generated fragment has an *extension point*, where the user can add his/her own manually written code. The extension point in the example above is between the marks `/*$TET$actor$!userdata!*/` and `/*$TET$*/`. The automatically generated code also includes calls to runtime library functions `access()` and `send()` inside overloaded `recv()` method. This code is omitted here for the reason of its complexity. A programmer can change both DSL commands and code inside the extension points. The preprocessor re-synthesizes code, keeping isomorphism between DSL specification and the generated C++ code.

6 Templet DSL Syntax Example

A more realistic example of actor and channel definitions in Templet DSL from `cpp11runtime/samples/templet` is shown below.

```
~Link=
    + Begin ? argSin2 -> Calc | argCos2 -> Calc;
      Calc ! result -> End;
      End.
```

```
*Parent=
      p1 : Link ! result -> join;
      p2 : Link ! result -> join;
    + fork(p1!argCos2, p2!argSin2);
      join(p1?result, p2?result).
*Child=
      p : Link ? argCos2 -> calcCos2 | argSin2 -> calcSin2;
      calcCos2(p?argCos2,p!result);
      calcSin2(p?argSin2,p!result).
```

This DSL code defines a program that tests for the correctness of the $\sin^2 x + \cos^2 x = 1$ identity. The generated program calculates $sin^2 x$ and $\cos^2 x$ concurrently in two `Child` actors, while the `Parent` actor coordinates them. The channel `Link` controls an order of message (`argSin2`, `argCos2`, `result`) exchange.

After the DSL difinition a programmer has to code *extension points* in C++ language. The DSL fragment above implies the following extension points: (a) definifions of `argSin2`, `argCos2`, `result` message structures; (b) definitions of `fork`, `join`, `calcCos2` and `calcSin2` C++ class methods; (c) possibly definitions of `Parent` and `Child` C++ class data members. Finally, a programmer creates actors as the `Parent` class and the `Child` class entities and connects the actors with the `Link` class entities. This C++ code is trivial and does not require a knowledge in the field of concurrent programming. The DSL syntax explanation can be found in [8].

7 Applications Overview

The `cpp11runtime` package provides another three samples to illustrate the practical use of the Templet language. The Gauss-Seidel method for solving the Laplace equation is located in the `samples/pipeline` directory. This example illustrates the use of the language 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 real parallel execution.

An example from the field of linear algebra is located in the `samples/ringmult` directory. This is an illustration of distributed matrix multiplication algorithm. Our implementation of the actor model is well suited for specifying computations both for shared and distributed mamory architectures.

The business process model example is located in the `samples/order` directory. The Templet language 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.

8 Conclusion

Our research shows a practical interest of the DSL-based approach for concurrent programming. This approach is based on specification of an actor-oriented runtime with Lamport's TLA formalism. We got a fully working but relatively simple implementation of the Template language for programming in shared memory architectures. In the future, we plan to extend the tool for distributed programming, multiple languages support, and semantic analysis of the DSL. The language has been deployed online at *http://templet.ssau.ru/templet* as a part of scientific computing web service.

References

1. Ward, M.P.: Language-oriented programming. Software-Concepts and toolkits 15(4), 147–161 (1994
2. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. Proceedings of the 3rd international joint conference on Artificial intelligence, pp. 235–245 (1973)
)
3. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience 40(12), 1135–1160 (2010)
4. Hoare, C.: Communicating sequential processes. In: The origin of concurrent programming, pp. 413–443. Springer (2002)
5. Wirth, N.: The programming language Oberon. Software: Practice and Experience 18(7), 671–690 (1988)
6. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. Software, IEEE 20(5), 36–41 (2003)
7. Lamport, L.: The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems 16 (3), 872–923 (1994)
8. Vostokin, S.: Templet: a markup language for concurrent programming. arXiv preprint arXiv:1412.0981 (2014)