ICIC Express Letters
Volume 11, Number 2, February 2017

ICIC International ©2017 ISSN 1881-803X
pp. 285–290

A NEW NOTATION TN FOR EXPRESSING CONCURRENCY IN PROGRAM DESIGN

Hongyan Yao, Xuebo Sun and Han Bao

School of Software
University of Science and Technology LiaoNing
No. 185, Qianshan Middle Road, Lishan District, Anshan 114051, P. R. China
abroat@163.com

Received July 2016; accepted October 2016

Abstract. *To leverage the computation power provided by multi-core computers, developers must resort to the concurrency programming. To the best of our knowledge, concurrency program design as a goal is often explored at a level of abstraction which is complicated and hard to understand. Few literature concerns about the intuitive concurrency design. Consequently, the concurrency program is usually thread-unsafe because developers cannot get sufficient data info from design diagram to guide the next coding, even the activity diagram, a UML-based diagram, merely presents the control flow, missing data info instead. This paper focuses on the intuitive concurrency design and proposes a new task notation (TN) to specify the task. A diagram that is composed of TNs called TND provides a better way than activity diagram in modelling the concurrency. A case study with TND employed indicates TND can effectively express data-collaboration or data-racing; moreover, TND eases to build the intuitive mapping between TN and its implementation.*

Keywords: Task, Concurrency design, Data racing, Thread, Multi-core

1. **Introduction.** Multi-processor or multi-core computer has become a commodity. Consequently, the trend for scalability of today's general-purpose programs can no longer be simply fulfilled by faster CPUs; rather, programs must now be designed to take advantage of the inherent concurrency in the underlying computational model. In other words, in order to leverage the power provided by the multi-processor machine within a single application, developers must resort to multi-thread. However, writing correct and efficient concurrent programs has remained a challenge. It is mainly because the non-determinism caused by thread scheduling makes finding errors through testing much less likely.

For pushing forward the application of concurrency programming and for improving the people's interest of using multi-thread techniques, some prepared works have been undergoing. For instance, [1] stressed the concurrency is a necessary manner to improve the application's performance; developers must be experienced in parallelizing object-oriented desktop application before they are willing to use it. For guiding how to code concurrent program, [2,3] presented two proposals separately. One suggested improving the GOF patterns for effectively coding the concurrency program; the other summarized some classic concurrent scenarios and then suggested which multi-thread design pattern is appropriate for. For the concern of quality of concurrency programming, [4] proposed an automatic verifier for concurrent object-oriented language (Java is supported only); [5] presented some tactics and rules for quality guarantee in concurrency coding.

Literature above mentioned relatively contributes to the concurrency coding, not the concurrency design. It has been known that the concurrency program in pure code will be hard to understand, modify, or extend if there are no accompanied design diagrams. So how to design the concurrency in diagrams is as important as improving concurrency coding. However, to date, only a few articles paid more attention to that. For instance, [6] focused on analyzing the design diagrams for concluding which UML-based diagrams are appropriate for representing the concurrency scenarios such as competitive or coordinated; furthermore, [7,8] shared almost the same thought and proposed separately the revised UML-based method for expressing concurrency requirements. Their proposals make the sequence diagram (a kind of UML diagram) a little mess. For example, expanding the sequence diagram to express the concurrency semantic [8] is inappropriate because that will bring too much arrow notations that decreased the readability of the diagram. In fact, Martin and Martin in their book [9] noted the sequence diagram should concisely serve for communication, rather not to express the details. If the details are needed to be expressed, use code.

Based on above concurrency coding and design introduction, we believe that concurrency design should be resolved prior to developer's adopting concurrency coding pattern; moreover, the concurrency design should concisely express the concurrency requirements without trivial details.

In this paper we propose an intuitive concurrency design notation called TN (Task Node), which can intuitively express a task, its needed shared data, and its collaboration with the other tasks. A diagram that is composed of TNs called TND (Task Node Diagram) is capable of providing sufficient data and relation info for the coder to practice his concurrency pattern. The TN in this paper is different from the 'Task Notation' mentioned in [10]; ours focuses on expressing the concurrent data in program model, while theirs, by constructing a ConcurTaskTrees, is mainly used for organizing user interfaces. Furthermore, ConcurTaskTrees method does not stress the tasks concurrency.

Our proposal provides a more effective way than activity diagram (a kind of UML diagram for expressing the control flow of related activities) in expressing the concurrency scenario. The analysis for the activity diagram and its deficiencies please refer to Section 2.

The rest of the paper is organized as the following. Section 2 analyzes the deficiencies of UML-based concurrency expression (activity diagram in particular); Section 3 proposes the notation of TN and TND. Section 4 presents a case study with the use of our proposal. Section 5 concludes the paper.

2. Analysis for the Deficiencies of Existing UML-Based Concurrency Expression.

With respect to the UML-based concurrency expression, we stress three deficiencies. 1) From the existing UML-based concurrency expression we can merely get the control flow, missing data info, and that will make the mapping from design diagram to the matching code difficult. In UML-based manner there are two diagrams that may be used for concurrency expression: activity diagram (AD) [11] and sequence diagram (SD) [12]. AD is coarse-grained while SD is fine-grained. The route from AD to coding is like this: design AD at first and then refine some activities in SD; finally, coding with referring to SD. Both AD and SD provide control flow but they all do not hold data info in terms of concurrency semantics. Some [4] tried to adapt the SD to support concurrency description but apparently Martin and Martin [9] claim their opposite attribute, because putting extra elements in SD (a lower-level diagram) to reflect the concurrency usually makes the SD look from concise to mess (many "combined fragment" and invoking lines needed to be appended; brings more confusions while communication). We approve his opinion and furthermore we believe concurrency is just a technique for coding and it should be embodied in the code file, rather not being designed in a diagram like SD. At present, AD is an alternative that supports concurrency control-flow description but it is weak to provide shared data info for supporting the relationship of activities collaboration; it just specifies what activity steps are going to do, through the notation of "Fork Node" or "Join Node". This directly results in that the coder in facing of AD will be puzzled in "what data I handle? One activity is restricted by the other one or not?" In short, AD provides no data info for guiding the following design or coding on how to handle the concurrency feature: data racing and data collaboration. 2) The other deficiency stressed is this: it is difficult to expand the concurrent activity in AD. For example, if you want to add an extra concurrent activity under the notation of "Fork Node", then that could bring a disaster to the coder because the coder is not aware of whether the data used for this added activity is needed to be synchronized or not; even worse, this new added activity may lead to a deadlock occurring among all concurrent activities (a possible reason is that applying racing-data is in disorder). 3) Another deficiency about AD concurrency expression is that we cannot get from the "Fork Node" to know which activity involved in data-racing and which activity serves as a data producer; that does rarely the benefits for coder to choose an appropriate concurrency pattern such as mutual-lock or producer/consumer pattern.

Based on the above analysis for UML-based concurrency expression we sum up as these: a new notation is needed for concurrency expression which should present concurrent activities as well as the data shared between them. From consulting the new notation the coder should be consistently transform it from an activity into a code fragment. Next, our proposal is presented for meeting these demands. The goal of our proposal, with respect to the concurrency requirements, is to link up AD and coding.

3. The Proposal of Notation TN and TND.

We named the TN as task notation because TN is the concept for logic, not for implementation; thread is the concrete conception for implementation. So the activities defined in AD are going to be mapped to task instead of thread. TN is defined as Definition 3.1. Semantics for its marks are listed in Table 1.

Definition 3.1. *A task notation, TN, is defined as* $TN := \{Id, Pre, Post, R, CF\}$.

Table 1. Semantic for each mark in TN

Mark	*Semantic*
TN	*to specify a task notation*
Id	*unique identifier for the task*
Pre	*to indicate tasks prior to TN*
Post	*to present tasks after TN*
R	*predicates to claim the relations of TNs; RR if data-racing, RC if data-collaboration*
CF	*the corresponding code fragment that supports the TN*

The notation of TN is straightforward. A TN is a set that is composed of elements that must be specified when designing the concurrent scenario. We take a scenario as an instance to demonstrate the usage of TN: in AD an activity, labeled A_0, is forked into four activities, labeled as A_{01}, A_{02}, A_{03} and A_{04}; A_{02} and A_{03} are independent from each other; A_{01} and A_{02} have data-racing in their executions; A_{03} and A_{04} have data-collaboration when moving forward; finally all sub-activities are merged with A_1. If this scenario is diagramed with AD, see Figure 2(a), some semantics are lost such as data-racing or data-collaboration between certain two activities; also Figure 2(a) does not provide more data-aspect info that is important for coding realization. However, if Figure 2(a) is replaced by the TNs specification of Figure 1, semantics are all kept. In Figure 1, each activity in Figure 2(a) is mapped to a TN under using our formal definition.

According to Figure 2(a), T_0 has no *Pre* but it has four $Post :< T_{01}, T_{02}, T_{03}, T_{04} >$. That indicates T_0 activates those tasks, which are going to run parallel with T_0. T_0 holds the data-collaboration with T_{01} on the shared data dph_0, expressed by $RC(T_0, T_{01}, dph_0)$ (at

$$TN := \{T_0, < T_{01}, T_{02}, T_{03}, T_{04} >, < RC(T_0, T_{01}, dph_0) >, cph_0\}$$
$$TN := \{T_{01}, T_0, T_1, < RC(T_0, T_{01}, dph_0), RR(T_{01}, T_{02}, dph_{12}) >, cph_{01}\}$$
$$TN := \{T_{02}, T_0, T_1, < RR(T_{02}, T_{01}, dph_{12}) >, cph_{02}\}$$
$$TN := \{T_{03}, T_0, T_1, < RR(T_{03}, T_{04}, dph_{34}) >, cph_{03}\}$$
$$TN := \{T_{04}, T_0, T_1, < RC(T_{04}, T_1, dph_1), RR(T_{04}, T_{03}, dph_{34}) >, cph_{04}\}$$
$$TN := \{T_1, < T_{01}, T_{02}, T_{03}, T_{04} >, < RC(T_{04}, T_1, dph_1) >, cph_1\}$$

FIGURE 1. Formal definitions for a set of related TNs

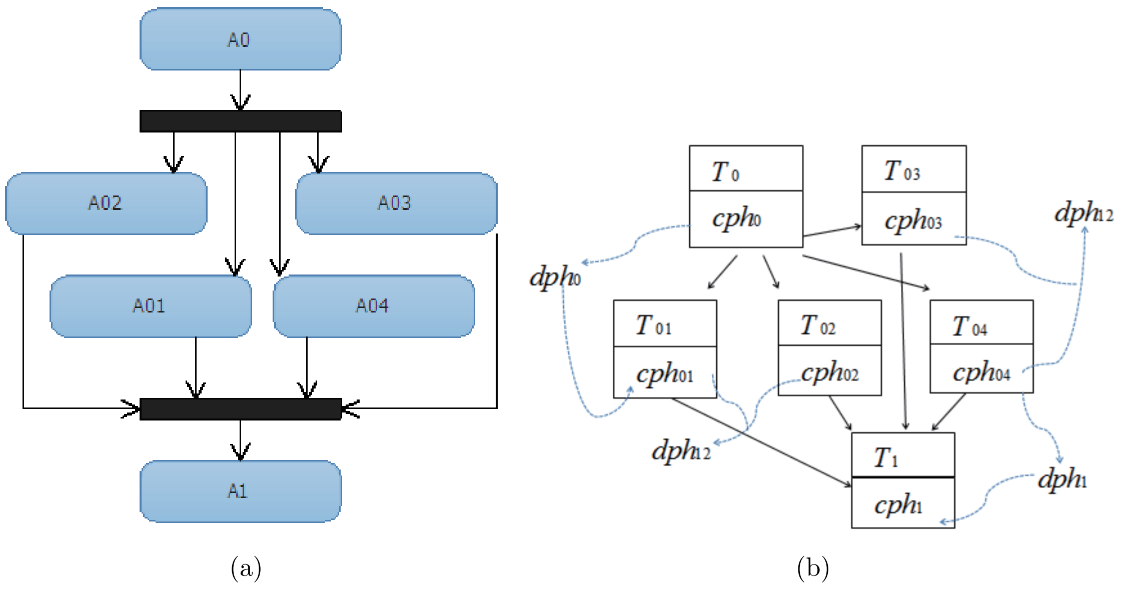


FIGURE 2. An AD and TND for modelling the concurrent tasks, respectively

the time dph_0 denotes just a data placeholder, which will be replaced once the corresponding contend for dph_0 is settled down); in alike manner, cph_0 is a code placeholder, too. It denotes the code stream that supports T_0 as well as $RC(T_0, T_{01}, dph_0)$. The other five TNs are more or less the same in semantics. For instance, T_{01}, which has a *Pre* T_0 and a *Post* T_1, holds the data-collaboration of $RC(T_0, T_{01}, dph_0)$ and data-racing of $RR(T_{01}, T_{02}, dph_{12})$; at the same time cph_{01} denotes a supposed code fragment that supports T_{01}. The last TN T_1 is stressed because it has no *Post* but has four *Pre* $< T_{01}, T_{02}, T_{03}, T_{04} >$; that means T_1 has to wait $< T_{01}, T_{02}, T_{03}, T_{04} >$ are all finished before it goes on, for T_1 must ensure the shared dph_1 between T_1 and T_{04} is ready. cph_1 is supposed to support T_1.

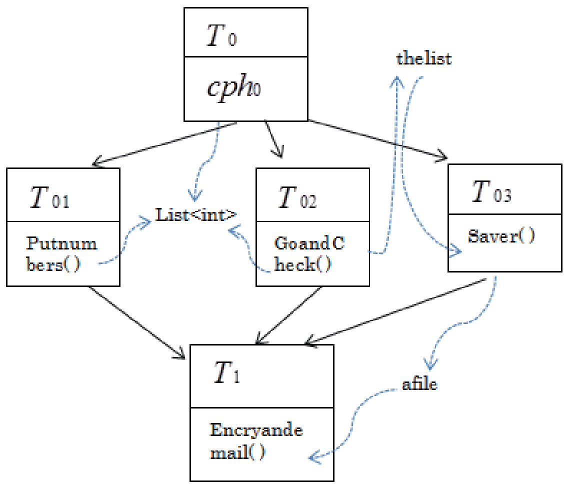
It is evident that Figure 1 defines tasks without semantics lost. Even the categories of data-collaboration or data-racing are specified, which provide sufficient valuable data info for easing the following concurrency coding. Transforming Figure 1 into graphics of TND will make the notation more expressive. See Figure 2(b), solid arrows represent the tasks time sequence; dotted arrows indicate what data is shared in data-racing or what data is shared in data-collaboration. If dotted line starts from the edge of a rectangle, like the arrow that targets dph_0 from T_0, that means dph_0 is the data produced by T_0 and it will be consumed only in cph_{01}. This reflects a simple data-collaboration; if dph_0 is shared between T_{01} and T_{02}, then the dotted line from dph_0 to cph_{01} will be omitted, meaning the data produced will be shared and mutually used. If tasks want to data-racing, dotted line will start from CF. For example, T_{01} and T_{02} both want data-racing dph_{12}, and then two dotted arrows started from cph_{01} and cph_{02} are merged and target the same data: dph_{12}.

As the other graphical elements in Figure 2(b) have the similar semantics, unnecessary detailed explanations are not given further.

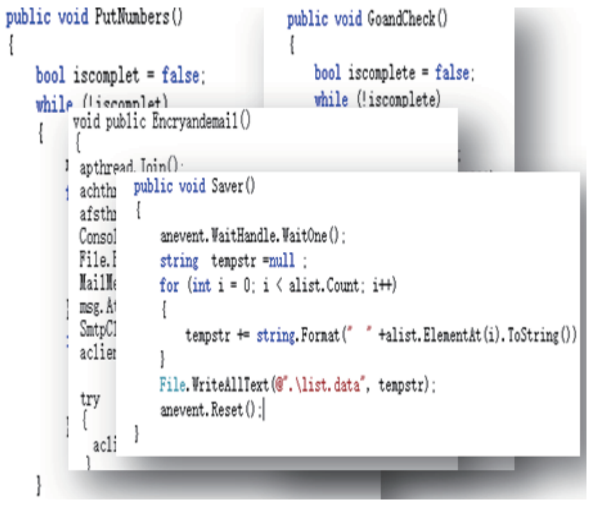
4. A Case Study Using TN and TND to Model the Concurrency Scenario.

In this section a case study is presented as illustration for the usage of TN and TND. The case scenario is supposed as this: at first, we prepare a data structure for handling; secondly, three concurrent jobs are activated. One job adds five numbers randomly each time into the data structure; the other job devotes to checking the data structure including sorting numbers and deleting the specified numbers that meet the rule: "number mod 3 is 0". Another job saves data to a file when the data structure is ready; finally, one job is going to encrypt the file and email it.

With regard to the case description we do not draw the activity diagram as a start because it contains merely the control flow for the jobs. We directly present the TND, instead of TN's formal specifications, to demonstrate TND can help trace the concurrency implementation intuitively. A TND corresponding to this case is shown in Figure 3(a). Roughly, each CF in TNs is no longer the code placeholder but the concrete CF, which is extracted from Figure 3(b). Figure 3(b) presents the current implementations for T_{01}, T_{02}, T_{03}, and T_1.



(a)



```
public void PutNumbers()
{
    bool iscomplet = false;
    while (!iscomplet)
    {
```

```
public void GoandCheck()
{
    bool iscomplete = false;
    while (!iscomplete)
```

```
void public Encryandemail()
{
 apthread.Join();
 achthı
 afsthı
 Consol
 File.I
 MailMe
 msg.At
 SmtpCl
 aclien

 try
 {
   acli
 }
```

```
public void Saver()
{
     anevent.WaitHandle.WaitOne();
     string  tempstr =null ;
     for (int i = 0; i < alist.Count; i++)
     {
         tempstr += string.Format("  " +alist.ElementAt(i).ToString())
     }
     File.WriteAllText(@".\list.data", tempstr);
     anevent.Reset();
}
```

(b)

FIGURE 3. TND together with tasks implementation to maintain the CF mapping

At the time, if the implementation for T_0 is not settled down, then cph_0 still takes its place. You can replace it once its corresponding content is finished.

Five tasks are designed in Figure 3(a): T_0 is, on one hand, to prepare the data typed in List<int>; on the other hand, to generate T_{01}, T_{02} and T_{03}. T_{01} and T_{02} are going to use the List<int> data mutually. T_{01} performs its CF Putnumbers() to add numbers into List<int> and T_{02} runs GoandCheck() to check and sort List<int>. Furthermore, T_{02} will inform T_{03} when the data of "thelist" is ready. In alike manner, T_{03} runs with the "thelist" and produces the data of "afile". When T_1 gets the knowledge that three concurrent tasks are done, T_1 will perform its CF, Encryandemail(), to do its job: encrypt "afile" and email it.

The simulation becomes intuitively via consulting Figure 3(a), and the implementation for each task, see Figure 3(b), is easier to be defined because the data for collaboration or for racing has been specified explicitly ahead of time; moreover, the CF that handles the shared data builds a direct map between task and its implementation. As a comparison, we could assume that if there are no arrows and data info illustrated in Figure 3(a), just like Figure 2(a) we cannot intuitively get the following knowledge for the next detailed design or coding: 1) take what data as a basis to build the relationships between tasks; 2) tasks are in the relationship of coordination, data-racing, or both; 3) what data in each task should be considered for concurrency; 4) in a coded task, there are code fragments that have operations on some concurrent data.

Without the demonstration of Figure 3(a), we will miss much important concurrency info, which will lead the coding hard to comprehend. So in short, TND suits for concurrency design as well as for the communication between designers and coders.

5. Conclusion.

To design the concurrency scenario with activity diagram, on one hand, it will lose the data info that should be specified for guiding the following coding phase; on the other hand, the activity diagram is incapable of builtding an intuitively connection between concurrency design and its implementation. The paper proposes the TN notation aiming to provide a better way that can get the essence of concurrency and can express it in formal or in diagram TND. TN's formal definition and its usage stresses the difference between data-collaboration and data-racing; furthermore, TND demonstrates how to transform formal definition into an intuitive figure presentation, and a case study with TND employed illustrates its effectiveness: TND can express concurrency control flow as well as shared data info, which cannot be expressed in AD. As TND explicitly points out the mapping between tasks and their implementations, it is better than activity diagram in expressing concurrency scenario. To develop a tool for auto-transforming the TN formal definitions to TND is our future work.

REFERENCES

[1] N. Giacaman and O. Sinnen, Parallel task for parallelizing object-oriented desktop applications, *IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*, Atlanta, GA, pp.1-8, 2010.

[2] S. L. Mooney, H. Rajan, S. M. Kautz and W. Rowcliffe, Almost free concurrency! (Using GOF patterns), *Proc. of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA'10)*, NY, USA, pp.249-250, 2010.

[3] J. Zheng and K. E. Harper, Concurrency design patterns, software quality attributes and their tactics, *Proc. of the 3rd International Workshop on Multicore Software Engineering (IWMSE '10)*, NY, USA, pp.40-47, 2010.

[4] J. Smans, B. Jacobs and F. Piessens, VeriCool: An automatic verifier for a concurrent object-oriented language, *Formal Methods for Open Object-Based Distributed Systems, Volume 5051 of the Series Lecture Notes in Computer Science*, pp.220-239, 2008.

[5] H. Gomaa, Designing concurrent, distributed, and real-time applications with UML, *Proc. of the 28th International Conference on Software Engineering (ICSE'06)*, NY, USA, pp.1059-1060, 2006.

[6] W. M. Gentleman, Concurrency paradigms: Competitive, coordinated, and collaborative: Which control mechanisms are appropriate? *International Journal of Parallel Programming*, vol.44, no.2, pp.325-336, 2016.

[7] D. Kroening, Automated verification of concurrent software, *Reachability Problems, Volume 8169 of the series Lecture Notes in Computer Science*, pp.19-20, 2013.

[8] B. Morandi, S. West, S. Nanz and H. Gomaa, Concurrent object-oriented development with behavioral design patterns, *Software Architecture, Volume 7957 of the series Lecture Notes in Computer Science*, pp.25-32, 2013.

[9] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C#*, Prentice Hall, 2006.

[10] P. J. Molina, J. Belenguer and Ó. Pastor, Describing just-UI concepts using a task notation, *Lecture Notes in Computer Science*, pp.218-230, 2003.

[11] F. Gu, X. Zhang, M. Chen, D. Große and R. Drechsler, Quantitative timing analysis of UML activity diagrams using statistical model checking, *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, pp.780-785, 2016.

[12] S. Dahiya, R. K. Bhatia and D. Rattan, Regression test selection using class, sequence and activity diagrams, *IET Software*, vol.10, no.3, pp.72-80, 2016.