International Research Journal of Applied and Basic Sciences
© 2014 Available online at www.irjabs.com
ISSN 2251-838X / Vol, 8 (9): 1271-1275
Science Explorer Publications

IRJABS

A Textual Tool for Concurrent Programming

Elaheh Azadi Marand[1*], Elham Azadi Marand[1], Moharram Challenger[1]

1. Computer Engineering Department, Islamic Azad University-Shabestar Branch, Shabestar, Iran

Corresponding Author email: azadi.e4509@gmail.com

ABSTRACT: By rapid development in software industry, capacities in graphical tools for concurrent programs in which several calculating process occur simultaneously have also been developed. A Domain Specific Language (DSL) has made an abstract level in a way in which environments based on these languages have created efficient methods that can overcome the complexity of programs. Based on a meta-model presented for concurrent programs, in this paper, we have suggested a textual concrete syntax, and showed that how created syntax has been used in exact code generation. On the other hand, the syntax for DSL is being supported by textual modeling language. Application of this syntax has been presented in Dining philosophers' problem which has been selected as a study case.
Keywords: Concurrent Programming, Domain Specific Language, Meta-model, Model Driven Engineering

INTRODUCTION

A huge amount of programming world includes applications which are run simultaneously. By developing in multi-core processors and assigning cores for every processor, a necessary need arises to programmers to use modern techniques to increase performance. This need, for high speed and high computation, is important motivator for concurrent program users.

Since in real and critical systems time is a considerably important parameter, i.e. events should finish in an appropriate time or responding time should be short, doing chores in concurrency is important. In addition to multi-processor and multi-core computers, concurrent programs increases efficiency for single architectural computers when entrance and exit operations are slow or blocked. On the other hand, concurrent programs can increase the responding speed of heterogeneous events in immediate and routine systems (Taylor et al, 1992). By considering all advantages mentioned above, concurrent programming has disadvantages as well. The calculative overhead resulted from synchronization, difficulty of error detection and high complexity of programming and planning with attention to several performing order (parallel and serial) are instances of problems for these kinds of programs. In addition, processors controlling and utilizing shared sources add to the complexity of programming (Hartly, 1998). To produce concurrent programs which are error-prone and complex, we need to have approaches by which costs are reduced and time is shortened. These methods should basically be independent from target programming language. Promoting abstract level in producing concurrent programs is one of the possible ways in reducing complexity level. In this method by increasing the abstract level, challenges between programmer and designing details decrease, and separation happens between designing and code generation.

Model Driven Engineering (MDE) is a way by which abstract level in producing software increases. By focusing on a specific domain, and by making models in various levels of abstraction, automation increases, and finally access to the codes will happen (Schmidt, 2006). More importantly, with primarily controlling of the models, we can detect most of the errors and prevent several problems to occur in production process (Challenger et al., 2011). By developing Domain Specific Language (DSL), unlike General Purpose Languages (GPLs), a programmer design his/her program in a specific domain (based on contracts and structures of the domain) and generates codes (Oliveira et al, 2009). To this end, using domain concepts and collecting them into the language, programmers will be aid and will result in promoting programming understanding (Ceh et al., 2011).

In (Azadi et al., 2013), we have attended designing trends, performing and tools needed in production these kinds of languages, in (Azadi et al., 2013), we also mentioned a trend of a Domain Specific Language, or Domain Specific Modeling Language (DSML) (Kelly, 2008). In this paper, we try to introduce textual modeling language for concurrent programming which is based on meta-model (Azadi et al., 2014). We also showed that by changing concurrent programs to a model and extracting codes, we stepped in complexity reduction in concurrent programs.

The rest of the paper is organized as below: section 2 discusses the meta-model given for concurrent programs. In section 3, the textual concrete syntax is being discussed. In section 4, dining philosophers' problem is presented as a case study. Finally, in section 5, we conclude the paper.

ABSTRACT SYNTAX

The concrete syntax of a Domain Specific Language (DSL) introduces the conceptual key terms and their relations in that language, and declares how these concepts may combine to make models. Therefore, designing a language essentially needs a Meta-model creation (Voelter et al., 2013) (Mernik et al., 2005). In this section, we will discuss the meta-model given in (Azadi, 2014) which shows the concrete syntax of concurrent programs.

As it can be seen in Figure 1, this meta-model is composed of several meta-elements and relationships between elements. Considering that to run any application (especially concurrent programs), there is an urgent need to determine what program is running. So meta-element “Program” and assign attributes Name for this meta-element, it supports the user who wants to run a program. As regards in concurrent programming, Thread has key roles, allowing users to perform functions simultaneously. Therefore, we consider a meta-element, named “Thread” that is connected to “Program” meta-element through a link and shows this fact that each program can have multiple threads. Threads need to have a start point to begin their tasks from. “StartResumePoint” meta-element intended in our proposed meta-model is start point.

In addition to the starting point Which is considered for threads, each thread should be the end point until eventually after carrying out the tasks assigned to it, be specified the duties of this thread is finished. Meta-element “StopPoint” is endpoint. Threads are very applicable when we need to wait for the response of another section of the program which increases overall speedup. Thus, the “WaitPoint” meta-element is necessary for threads' waiting procedure. In normal cases, execution stops in a point, thread goes to waiting mood and restarts later. “StartResumePoint” meta-element supports this state. Besides, if a program includes several threads, these threads can meet in a point, necessitating “MeetingPoint” element. Thus, threads link to this meta-element which is also connected to “MeetingTask” meta-element, supporting the concept that the results of each thread' s performance is gathered in this meta-element, attached to Task meta-element by a link.

Furthermore, we refer to the tasks assigned to the threads, so it is necessary to consider meta- element, such as “Task”. This meta-element represents tasks that will be granted to any of threads based on priority. So that each of the “StartResumePoint”, “StopPoint” meta-elements and “WaitPoint” meta-element has been within “Task” meta-element, namely the command that is starting point of a thread located within a task and the task is attached to the thread and the Task also is connected to thread. Above all, meta-element “MeetingTask” inherits from the meta-element “Task” and it is connected by an inheritance link. One can say, each thread should have at least one task which may contain subtasks. Apart from these, we defined another class, “CLASS” meta-element, including any class containing fields and methods collection that work on the data and provide services for service receivers. As can be seen this meta-element is connected to “Program” meta-element through a link. The important point is that both of meta-elements “Thread” and “CLASS” Have been placed inside the “Program” meta-element. In addition to the classes considered, we have considered the meta-element Name of “ThreadStore”. So that meta-element “ThreadStore” is thought as container that Threads can be inside it. Also; a link exists between “ThreadStore” and "Thread", since a thread can be placed inside a “ThreadStore” or “Thread”. An important point that should be regarded is that use of a thread should be realized by creating an object from its class. “Thread” and "CLASS" meta-elements presume a class and we should be able to create an instance of "CLASS" and "Thread" class. Meta-element “ThreadObject” Represents Object created from Class “Thread” and meta-element “GeneralObject” is Object created from “CLASS” meta-element.

After developing each instance thread, the type of the instance should be identified. “DataMember” meta-element represents data members and “FunctionMember” meta-element refers to method members. It must be mentioned that developed objects either in data member or method member forms can be transferred among threads. They identify the point that data or method member can be shared or locally used. The concept of inheritance can be used in these meta-elements in which an object inherits the features of other object. This is important since it supports hierarchical classification.

Accordingly, if the data is located on shared memory and multiple threads want have access simultaneously to that data; Then Sync issue arises that for this purpose, we have considered “Synchronization” meta-element. Considering these concepts, when be coordinating discussion, the problem of mutual exclusion and deadlock arises that our meta-model is intended for concurrent programs is no exception to this rule; Therefore, we have considered “MutualExclution” and “Deadlock” meta-elements. Tacitly, for managing access to exclusive resources, techniques such as Semaphore, Monitor, and Lock are prevalent in concurrent programs. We have

utilized these techniques in our suggested meta-model for managing exclusive access to have exact controls in coordinating threads. We have named this meta-elements “Semaphore”, “Monitor” and “Lock”.

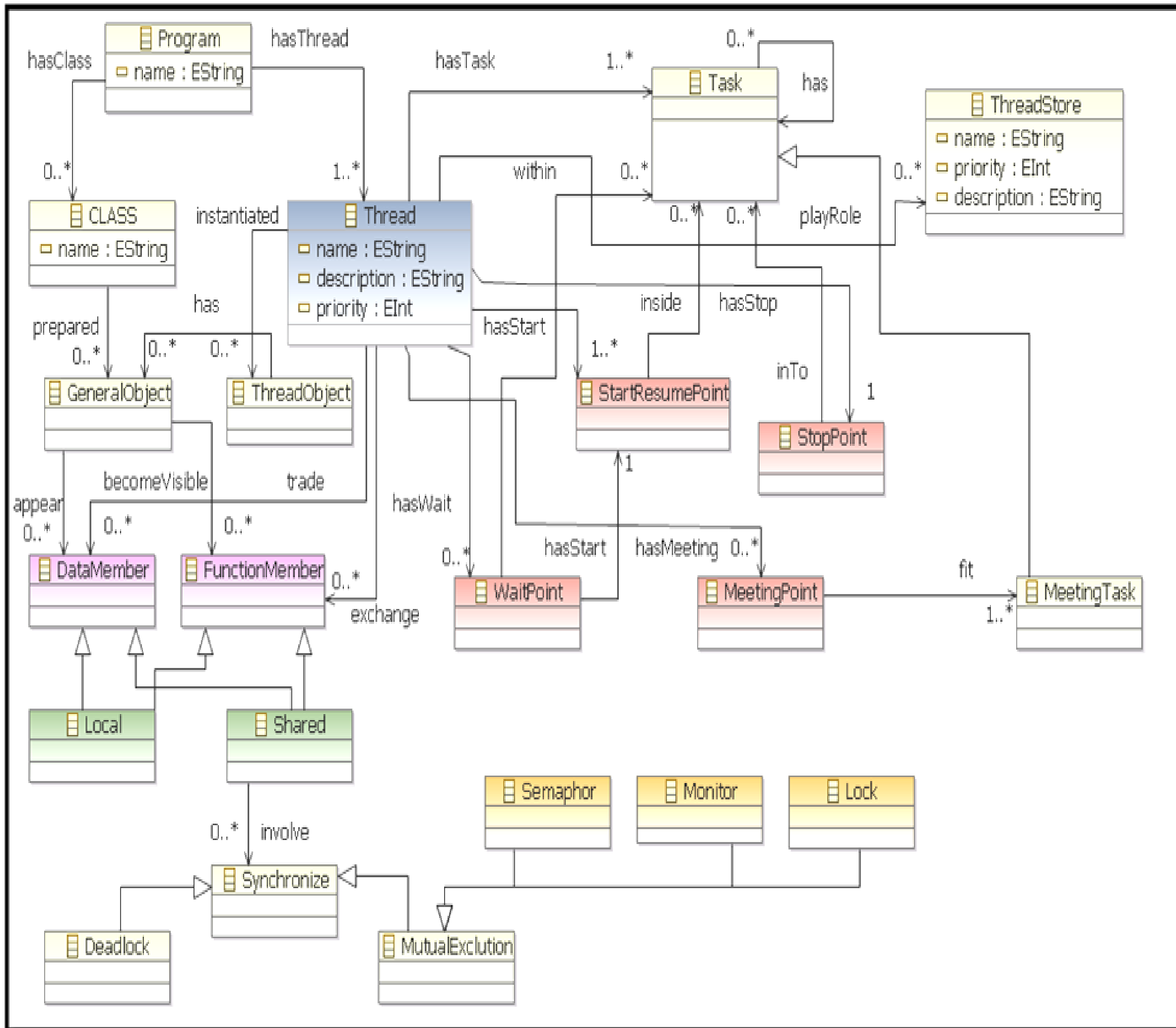

Figure1. Meta-model for concurrent programs

Textual Concrete Syntax

Concrete syntax of a Domain Specific Language describes the appearance of that language. For a context language, a provided concrete syntax comes to the sight as structural context (Demirkol et al., 2013). However, this syntax can be graphically appeared for a modeling language (Voelter et al., 2013). In (Azadi et al., 2014) we showed a graphical concrete syntax, and design Readers/Writers problem to understand and apply it clearly.

Separately, in (Azadi et al., 2014), we paved a set of semantic controls on Readers/Writers problem to achieve the ultimate models of the code. In this part of the paper, we suggest a concrete syntax for concurrent programs. So, after transforming concurrent program into an Ecore file, Figure 2, we produce EMF[1] files with the help of Xtext tool (Clark, 2008) which is developed by Itemis Company.

[1] Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf/ (last access: May. 2014)

```
01concurrent returns concurrent:
02{concurrent}
03 'concurrent'
04 '{'
05 ('programs' '{' programs+=Program ( "," programs+=Program)* '}' )?
06 ('threads' '{' threads+=Thread ( "," threads+=Thread)* '}' )?
07 ('homes' '{' homes+=CLASS ( "," homes+=CLASS)* '}' )?
08 ('dmembers' '{' dmembers+=DataMember ( "," dmembers+=DataMember)* '}' )?
09 ('fmembers' '{' fmembers+=FunctionMember ( "," fmembers+=FunctionMember)* '})?
10 ('tasks' '{' tasks+=Task ( "," tasks+=Task)* '}')?;
11   ...
12  DataMember returns DataMember:
13   DataMember_Impl | Local | Shared;
14   FunctionMember returns FunctionMember:
15   FunctionMember_Impl | Local | Shared;
16   Task returns Task:
17   Task_Impl | MeetingTask;
18   MutualExclution returns MutualExclution:
19   MutualExclution_Impl | Semaphor | Monitor | Lock;
20   ...
21  Shared returns Shared:
22 {Shared}
23 'Shared'
24 name=EString
25 '{'
26 ('name1' name1=EString)?
27 ('size' size=EInt)?
28 ('involve''('involve+=[Synchronization|EString]("," involve+=[Synchronization|EString])* ')' )?
29 '}';
```

Figure2. Grammar of the language in Xtext tool

The resulted structure is a textual editor that is developed as the concrete syntax of concurrent programs, and enables the operator to perform variety of concurrent programs in the format of model on this structure. Note that, graphical models in created tools, as the next step, will be transformed into executable code of concurrent programs. This is the future work of this paper.

Case Study

The problems of concurrency are well-known in computer science e.g. in producer/consumer problem, philosophers' dining problem, readers/writers problem, and fair hairstyle problem (Hoare, 1974) (Stallings, 2011). In this part of the article, we have selected Dining philosophers' problem as a study case and design it in the textual editor.

In this problem several, say five, silent philosophers sit at around of a table with bowls of spaghetti in front of each. Forks are placed between each pair of adjacent philosophers. (An alternative problem formulation uses rice and chopsticks instead of spaghetti and forks). Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it's not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to the others. A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them. Eating is not limited by the amount of spaghetti left: assume an infinite supply. The problem is how to design a discipline of behavior (a concurrent algorithm) such that each philosopher won't starve; i.e., can forever continue to alternate between eating and thinking assuming that any philosopher cannot know when others may want to eat or think (Stallings, 2011).

Modeling dining philosophers' problem in the textual editor

As it can be seen in Figure 3, we have supposed every philosopher as a class of thread which includes: start point, waiting and stop points. They start their duty in start point, and then wait for each other in waiting point and finish their duty in stop point.

Needless to say, all these points are inside a task. A duty given to a philosopher is to eat spaghetti with forks, therefore, to utilize thread classes; we have specified a separate object for each thread (a philosopher).

In addition, we put forks in a shared memory names dining table for philosopher to use them. An important point is that when philosophers want to use forks simultaneously, synchronizing issue will occur. To avoid this problem we use semaphore technique to manage unique sources access.

Figure3. Modeling Dining philosophers' problem in the textual editor

```
Concurrent.mydsl5
concurrent {
    programs { Program "Dining philosophers" { hasThread (Philosopher0, Philosopher1, Philosopher2, Philosopher3, Philosopher4)
        hasClass (Other)}}
    threads {Thread Philosopher0 {description "Eating Spaghetti" priority 0 hasTask (EatingSpaghetti0) hasStart (StartPhilosopher0)
          hasStop StopPhilosopher0 hasWait (WaitPhilosopher0) instantiated (Philosopher0Obj) trade (Fork0)},
          Thread  Philosopher1 {description "Eating Spaghetti" priority 1 hasTask (EatingSpaghetti1) hasStart (StartPhilosopher1)
          hasStop StopPhilosopher1 hasWait (WaitPhilosopher1) instantiated (Philosopher1Obj) trade (Fork1)},
          Thread Philosopher2 {description "Eating Spaghetti" priority 2 hasTask (EatingSpaghetti2) hasStart (StartPhilosopher2 )
          hasStop StopPhilosopher2 hasWait (WaitPhilosopher2) instantiated (Philosopher2Obj) trade (Fork2)},
          Thread Philosopher3 {description "Eating Spaghetti" priority 3 hasTask (EatingSpaghetti3) hasStart (StartPhilosopher3)
          hasStop StopPhilosopher3 hasWait (WaitPhilosopher3) instantiated (Philosopher3Obj) trade (Fork3)},
          Thread Philosopher4 {description "Eating Spaghetti" priority 4 hasTask (EatingSpaghetti4) hasStart (StartPhilosopher4)
          hasStop StopPhilosopher4 hasWait (WaitPhilosopher4) instantiated (Philosopher4Obj) trade (Fork4)}}
          homes {CLASS Other {prepared  (otherObj)}}
dmembers {DataMember Fork0, DataMember Fork1, DataMember Fork2, DataMember Fork3, DataMember Fork4, Shared Table {involve ("semaphore" )}}
    tasks {Task EatingSpaghetti0 {has (EatingSpaghetti0)}, Task EatingSpaghetti1 { has (EatingSpaghetti1)},
    Task EatingSpaghetti2 { has(EatingSpaghetti2)},Task EatingSpaghetti3 {has(EatingSpaghetti3)},Task EatingSpaghetti4 {has( EatingSpaghetti4)
    srpoints {StartResumePoint StartPhilosopher0 {inside (EatingSpaghetti0)}, StartResumePoint StartPhilosopher1 {inside (EatingSpaghetti1)},
            StartResumePoint StartPhilosopher2 {inside(EatingSpaghetti2)},StartResumePoint StartPhilosopher3 {inside(EatingSpaghetti3)},
            StartResumePoint StartPhilosopher4 {inside(EatingSpaghetti4)}}
            spoints { StopPoint StopPhilosopher0 {inTo (EatingSpaghetti0)},StopPoint StopPhilosopher1 {inTo (EatingSpaghetti1)},
            StopPoint  StopPhilosopher2 {inTo (EatingSpaghetti2 )},StopPoint StopPhilosopher3 {inTo (EatingSpaghetti3)},
            StopPoint  StopPhilosopher4 {inTo (EatingSpaghetti4 )}}
           wpoints { WaitPoint WaitPhilosopher0 {hasStart StartPhilosopher0 within (EatingSpaghetti0 )},
           WaitPoint WaitPhilosopher1 { hasStart StartPhilosopher1 within (EatingSpaghetti1 )},
           WaitPoint WaitPhilosopher2 {hasStart StartPhilosopher2 within ( EatingSpaghetti2 )},
           WaitPoint WaitPhilosopher3 { hasStart StartPhilosopher3 within (EatingSpaghetti3 )},
           WaitPoint WaitPhilosopher4 {hasStart StartPhilosopher4 within (EatingSpaghetti4 )}}
    gobject { GeneralObject otherObj {appear (Fork0,Fork1,Fork2,Fork3,Fork4)}}
    tobject {ThreadObject Philosopher0Obj {has(otherObj)},ThreadObject Philosopher1Obj {has(otherObj)} ,
    ThreadObject Philosopher2Obj {has( otherObj )},ThreadObject Philosopher3Obj {has(otherObj)},ThreadObject Philosopher4Obj {has(otherObj)}}
    sync {Semaphor "semaphore"}
}
```

CONCLUSION

In this paper, a textual language for concurrent programs based on a meta-model is discussed. We used Dining philosophers' problem as a case study. With designing models on provided textual tool, operators can quickly detect errors with considering language semantic controls and performing them over models.

Moreover, with relying on designed models and after ensuring about their validation, we generate code which resulted in reduction of complexity and increasing the speed of concurrent programs development.

REFERENCES

Azadi E, Challenger M, Khalilpour V. 2013. "Assessment and Comparison of designing methods, Implementation and required tools for Domain Specific Modeling Languages (DSMLs) development", National Conference on Electrical and Computer Engineering Islamic Azad University, Sarvestan Branch, Iran, pp. 1-6, (in Persian).

Azadi E, Challenger M. 2013. "Assessment and Comparison of designing methods, Implementation and required tools for Domain Specific Languages (DSLs) development", First National Conference on Advances in computer science and information retrieval approaches (bpj01), Islamic Azad University, Rodsar Branch, Iran, pp. 1-8, (in Persian).

Azadi E, PourhajiKazem AA. 2014. "Providing A Concrete Syntax of Graphical for Concurrent Programs", 11th National Conference on Computer and Intelligent System, Iran-Kish, pp. 1-5, (in Persian).

Azadi E, PourhajiKazem AA. 2014. "Providing Semantic Controls based on Concrete Syntax of Graphical for Concurrent Programs", National Conference on Computer Engineering and Information Technology, Shushtar-Branch, Iran, pp. 1-6, (in Persian).

Ceh I, Crepinsek M, Kosar T, Mernik M. 2011. "Ontology Driven Development of Domain-Specific Languages", Computer Sciebce and Information System (ComSIS), vol. 8, No. 2, pp. 317-342.

Challenger M, Getir S, Demirkol S, Kardas G. 2011. "A Domain Specific Metamodel for Semantic Web enabled Multi-agent Systems", Lecture Notes in Business Information Processing (LNBIP), Vol. 83, pp. 177-186.

Clark T, Sammut P, Willans J. 2008. "Applied Meta Modeling A Foundation For Language Driven Development", available online at: http//:www.ceteva.com, pp .1-224.

Demirkol S, Challenger M, Getir S, Kosar T, Kardas G, Mernik M. 2013. “A DSL for the Development of Software Agents Working in the Semantic Web Environment”, International Journal of Computer Science and Information Systems (ComSIS), pp. 1-6.
Hartly SJ. 1998. “Concurrent Programming: The Java Programming Language”, Published by Oxford Univercity Press, Inc., New York, pp.1-35.
Hoare CA. 1974. “Monitors: An Operating System Structuring Concept”, CACM, Volume 17:10, pp. 549–557.
Kelly S, Tolvanen JP. 2008. “DOMAIN-SPECIFIC MODELING Enabling Full Code Generation”, IEEE Computer Society Publications, ISBN 978-0-470-03666-2, pp. 1-446.
Mernik M, Heering J, Sloane A. 2005. “When and how to develop domain-specific languages”, ACM Computing Surveys, vol. 37(4), pp. 316-344.
Oliveira N, Joao M, Pereira V, Henriques PR, Cruz DD. 2009. “Domail-Specific Languages: A Theoretical Survey”, in Proceedings of the 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA2009), Lisboa, Portugal, pp.35-46.
Schmidt DC. 2006. “Model-Driven Engineering”, Published by the IEEE Computer Society, pp. 25-31.
Stallings W. 2011. **“**Operating Systems Internals And Design Principles (7th Edition)**”,** published by Pearson Education, pp. 1-816.
Taylor RN, Levine DL, Kelly ChD. 1992. “Structural Testing of Concurrent Programs”, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 18, NO. 3, pp. 206- 215.
Voelter M, Benz S, Dietrich C, Engelman B, Helander M, Kats L, Visser E, Wachsmuth G. 2013 “DSL Engineering”, pp. 1-558, Available online at: http://dslbook.org.