Concurrent Programming using Umple

Mahmoud Husseini Orabi, Ahmed Husseini Orabi and Timothy C. Lethbridge
School of Electrical Engineering and Computer Science, University of Ottawa,
800 King Edward Avenue, Ottawa, Canada

Keywords: Umple, Active Object, Composite Structure, UML.

Abstract: UML standards lack systematic solutions that can handle concurrency and time constructs in a single unified notation. In this paper, we integrate concurrency as a part of Umple, a combined modelling and programming language. Our extensions can help ease component-based development of real-time, distributed, and embedded applications. The work, which is based on an extended implementation of active object, enables better validation of systems, and improves usability from the developer's perspective. We describe the syntax and semantics of our Umple extensions, and also show how the constructs are transformed into C++. After that, we evaluate our work qualitatively, by comparing Umple with other specifications that focus on time as key; we show that Umple gives more flexibility than UML and MARTE. We also evaluate our work quantitatively using LOC and cyclomatic complexity metrics, showing that a developer would have to write many fewer lines of code when using Umple than when programming directly in C++.

1 INTRODUCTION

Many of the existing programming languages tend to be limited in how to handle concurrency easily, and hence require additional third-party libraries. Even though some languages such as C++, support concurrency, they tend to have challenges related to different compiler vendors, standards, and thread APIs for operating systems and embedded devices.

Concurrency results in abstract patterns of interactions, which seem best handled by integrating it with model objects, and also with the syntax of programming languages. The most elemental such patterns are synchronous and asynchronous method invocation.

Prior to the work reported in this paper, Umple (Orabi, Orabi, and Lethbridge, 2016) supported concurrency in three forms, basic active objects, do activities, and queued state machines (QSM) (Alghamdi, 2010). Active objects and do activities spawn threads, separate from their owning class's thread, to execute some action code. An active object is activated from its class constructor, while a do activity is activated while in a state machine state. A QSM has a separate threat to enqueue events from the thread that processes events.

The prior version of Umple omitted asynchronous method invocation and time constructs such as delay, polling, and timeout.

In this paper, we describe how we extend concurrency in Umple to support 1) data isolation, 2) thread communication through asynchronous messages, 3) processing each task one at a time to satisfy run-to-completion semantics, eliminating concurrency issues, and 4) a generic approach that covers operations, state machine events, and actions.

We refer to our extensions as the *active features*, as they are derived from the active object pattern (Lavender and Schmidt, 1996), which enforces concurrency best practices.

We distinguish between active and passive (Ober and Stan, 1999) in terms of their capabilities to execute in their own thread, and to initiate a control activity, such that each method is executed internally and sequentially. An active Umple class means that the class must have at least one active method.

In Umple, development can be performed at different levels, model or action code. At the model level, the users write their model in Umple syntax, while in the action code, the users write the code in a selected target language. Languages that Umple supports include Java, C++, and PHP. In this paper, our target language is C++.

We will highlight the possible levels at which active can be applied, such as methods and action code. After that, we will discuss the basic flow of active objects, which consists of several actors such as proxies, messages, and schedulers.

Orabi, M., Orabi, A. and Lethbridge, T.
Concurrent Programming using Umple .
DOI: 10.5220/0006713005750585
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 575-585
ISBN: 978-989-758-283-7

Our contributions related to concurrency can be summarized as follows:

- The implementation of the features related to concurrency in Umple, including code generation for real-time applications.
- Introducing an active object pattern that extends the one introduced by Lavender (Klein, Lu, and Netzer, 2003; Lavender and Schmidt, 1996). Our pattern aims to enhance communication among active objects as in the points below.
- Simplifying active features at the action code level such as future (Cplusplus.com, 2016), and other time constructs, using simple Umple keywords.
- Easy handling of complex time constraints related to asynchronous and synchronous method invocation, using simple Umple keywords.
- Introducing a new pattern, call/resolve/then to ease invocation strategies, callbacks, data resolution, and error handling.

There is an extensive literature about Umple (Badreddin, Forward, and Lethbridge, 2014; Badreddin, Lethbridge, and Forward, 2014; Lethbridge, Abdelzad, Husseini Orabi, Husseini Orabi, and Adesina, 2016; Orabi et al., 2016), but for readers not familiar with it, the following is a very brief summary: It is toolkit comprising a) A textual syntax based on UML that can be incorporated into target language code or vice-versa; b) a compiler that generates code for modelling constructs; c) a diagram-generator for UML and other diagrams; and d) a model analysis engine. It can be run in Eclipse, on the command line and on the web.

2 ACTIVE FEATURES IN UMPLE

We implement active features at three levels, the Umple construct, the concurrency model, and the code generated in the target language. An Umple construct is a semantic and syntactic extension of Umple that refers to behavior, introduces new keywords, and specifies how they can be used. We discuss each construct by way of scenarios with excerpts from the code generated in C++.

A concurrency model refers to the model used to handle concurrency among operations in terms of communication and synchronization. Concurrency models can be contrasted based on their behavior patterns and mechanisms applied for inter-process communication such as shared-state or message passing. Examples of concurrency models include actor model (AM) (Sutter and Larus, 2005) and active object (AO). The differences between both models are indicated in (Rouvinez and Sobe, 2014).

We base our work on the active object model as it 1) decouples method execution from invocation, 2) enables invocation using a function call interface or delegate (Microsoft.com, 2015), 3) assures data isolation between the caller and receiver, and 4) employs various message-passing mechanisms.

The classic active object pattern typically involves the following elements, 1) *interface*: defines accessible methods; commonly known as active methods or public interface methods, 2) *client*: implements the interface, 3) *proxy*: another simple object internal to the client that the client invokes to access other methods of the system in a thread-safe manner, 4) *request*: invoked by a client to a proxy, 5) *scheduler*: organizes how requests execute, 6) *response*: has different forms such as callbacks, variables, and future objects.

Typically, the active object pattern employs a simple FIFO queue with serial execution of the pending requests in the queue. This means that complicated scenarios such as prioritized queues or quasi-concurrency models are not considered. We will show in this paper how we have managed to overcome such limitations. Our work hence extends the classic active object pattern in that a) there is a more sophisticated internal scheduler mechanism; b) there is a more sophisticated form of internal concurrency; c) there is a prioritized FIFO queue; and d) time constraints are allowed to permit deferred or periodic calls.

3 STRUCTURE

The *active* keyword is used to declare active methods (Snippet 1 - Line 2). An active method has the same constraints as regular methods such as having a unique name, return type, and signature. Once a class has at least one active method, it will be considered as an active class.

The main components of our active object pattern include active object, public interface, message, scheduler, multicast delegate, and future object.

We use template meta-programming (TMP) and the curiously recurring template pattern (CRTP) to enable reusability through traits and mixins (Smaragdakis and Batory, 2000). This approach allows to generically define active features, determine the traits needed, and attach traits to a method callback (delegate) within a class object. A delegate or alternatively a function pointer is a variable used to invoke a callback method (Rahman, 2013). It can be either unicast or multicast, based on the number of callbacks to be dispatched.

We use variadic template, a template with variable arguments, to make asynchronous function

call parameters matching the exact number of original delegate method parameters. Therefore, it will be in sense similar to invoking the original method, but it will instead defer the execution to the active object.

An active method is invoked in a similar manner to regular methods (Snippet 3 - Line 8).

In terms of code generation, we rely on the Active API (Table 1 and Table 2), which we implemented as a part of this paper. An Active instance receives two parameters, the class to which an active object belongs and the return type of that active object, in addition to the types of the parameters defined in the active method (Snippet 1 – Lines 2 and 3). The additional parameters are String and Integer in this case.

```
1  class ActiveMethodDeclaration {                    Umple
2    String active activeMethodExample (String
3             param1, Integer param2) {
4      return "This is an active method:"+ param1
5           <<","<< param2;
6    }
7  }
```

Snippet 1: A simple active method declaration.

```
1  //This portion is the public interface from the          C++
2  ActiveMethodDeclaration.h file
3  Active<ActiveMethodDeclaration, string, string, int>
4            activeMethodExample;
5  Scheduler _internalScheduler;
6
7  //This portion is a constructor from the
8  ActiveMethodDeclaration.cpp file
9  ActiveMethodDeclaration::ActiveMethodDeclaration()
10  : activeMethodExample(this, and_internalScheduler,
11  andActiveMethodDeclaration::_activeMethodExample)
12
13 }
14 .....
15 string _activeMethodExample(string param1){
16    return "This is an active method:"+ param1 << ","
17      << param2;
18 }
```

Snippet 2: Portions of the generated code for Snippet 1.

The name of the *Active* instance is the same as the active method defined in the Umple model (Snippet 1- Line 2). The content of the active method is placed in a private method in the generated code, and its name is prefixed with an underscore (Snippet 2 – Line 15). The Active instance refers to this private method (Snippet 2 - Line 11).

The *Scheduler* API is used to handle the execution queue of active methods based on their order of invocation and priorities. When an active object exists in a class, we generate an internal Scheduler (Snippet 2 - Line 5) that will be used by all *Active* instances (Line 10 for instance)

An active method without a return type will be assumed void. If an active method does not have parameters, a user will not need to worry about bureaucratically passing empty parentheses.

3.1 Public Interfaces

We use an *Active* template-based class to define public interface methods to decouple method invocation from execution (Klein et al., 2003; Lavender and Schmidt, 1996). We employ *delegate* and TMP to preprocess a public interface function to have the same signature of a delegated method, in addition to defaulted additional arguments such as priority and delay. Snippet 3 is a basic example to define an active method.

```
1  class Test {                                          Umple
2     int active call (int val1, int val2) {
3        return val1*val2;
4     }
5     public static void main(int argc, char * argv[]) {
6        Test test;
7        FutureResult<int> mul = test.call(2,2);
8     }
9  }
10
```

Snippet 3: Basic active objects in Umple.

We have two internal types of public interface methods, *active* and *async*, for each of which we generate API to handle certain time constructs (Table 1).

They differ based on their ability to spawn a thread; in particular, *async* has its own concurrent thread, similar to the behaviour of a do activity. *Async* is mainly used with repetition or periodic constructs.

Table 1: Basic APIs of time constructs.

p1, p2… pn refers to a dynamic parameters; i.e 0 to *

API	Example
Active	*methodCall(p1, p2, ..., pn, priority, delay, timeout)*
AsyncMethod	*methodCall (p1, p2, ..., pn, priority, period, delay, timeout)*

Active and *Async* extend the original methods, such that additional parameters are added to handle time constructs (Table 2); the default parameter values are zero. The methods that require asynchronous execution rely on the async Method, while those that do not require it rely on the Active (Table 2). Both Active and Async rely on the Scheduler API.

It is important to mention that in Umple, for language usability purposes, we allow developers to define their main functions in a way similar to Java (Snippet 3 - Line 6).

```
1  class Test {                                                    C++
2      ....
3  public:
4     Test ():
5       call(this,and_internalScheduler, andTest::
6  callImpl){}
7        Active< Test , int, int, int> call;
8
9  protected:
10         int_call(int val1, int val2) {
11                 returnval1 * val2;
12         }
13 private:
14         Scheduler _internalScheduler;
15             ....
16         };
17     ....
18 int main(int argc, char * argv[]) {
19         Test test;
20         FutureResult<int> mul = test.call(2,2);
21     }
   }
```

Snippet 4: An example of generated code of an active object.

Table 2: Time-based constructs.

The possible scopes are action code and model.

Constructs	Description	API	Scope
Priority	Sets a priority to determine the order of invocation in a queue	Both	Action code
Timeout	Sets the maximum waiting time for a task to be completed.	Both	Both
Delay	Causes intentional delay	Both	Both
Period	Determines the polling time for rechecking a method	Async	Both

In Snippet 4, the generated code for Snippet 3, Line 9 defines a delegate to a function that has two parameters of the type int, and its return type is also int. The initialization of an active method takes place in the constructor Line 5, which specifies a callback method and an active object scheduler. There will be no difference in method execution, except that it returns a *future* response of class FutureResult. FutureResult can be considered as a proxy that communicates with the active object, and holds the response containing the result, status, and errors. We discuss more about this in the next section.

In generated Umple code, we follow the same structure shown in Snippet 4. In Line 6, a public method is created for the active method call. The implementation of the call is defined in an internal method _call as in Line 9. The visibility of call is protected instead of private in order to give the ability for subclasses to use or inherit it. Therefore, call acts as a public interface or client, while _call acts as a servant. There is no need to use history variables or other mechanism to handle inheritance anomalies.

3.2 Future

Future is a shared-object proxy that provides a channel between clients and active objects. It provides wait-set functionality (such as wait, notify and wait for a specific time) and a set of functions to inquire about an asynchronous response of an active method containing availability, content and/or errors. *Future* in Umple is an instance of FutureResult.

```
1  class Test {                                                   Umple
2      int active call (int intValue) {
3          return intValue*2;
4      }
5      public int main(int argc,  char* argv[]) {
6          Test test;
7          FutureResult<int> result =
8                test.call(1, 10);
9          result.wait(9000);
10         assert(result.ready());
11         cout<<result.data();
12     }
13 }
```

Snippet 5: Wait-set example.

Table 3: Status types of active object execution.

Status	Description
Pending	Not yet processed or activated due to queue requirements such as its order and priority in the queue.
Waiting	Processed and ready for execution in a queue.
Deferred	Postponed from being executed for not satisfying guard constraints, and added to deferred list to be recalled when constraints satisfied.
Done	Executed and completed without errors.
Error	Completed with errors.

FutureResult shows a simple use of the wait-set functions (Snippet 5 - Lines 9-11). There is one active method with a single int parameter (Line 2), and it is invoked once (Line 7). There are other optional

parameters (with values 1 and 10 in this example) referring to priority and delay (Line 8). The wait function has an optional argument to specify the expected waiting time (Line 9), otherwise it will throw a timeout exception. Each method returns a response describing a status (Line 11).

Table 3 shows the possible statuses, each of which is wrapped in an instance of FutureResult.

FutureResult has error- and data-resolving functions. They can be used to throw exceptions such as timeout.

3.3 Scheduler

Scheduler handles mutual exclusion among queues and message requests. The process of message queuing may delay some tasks, even if trivial, such as read or status check tasks. For example, when there is a method request to update a value, it will put a mutex on some variables.

There are three common mechanism to implement the internal concurrency of an AO. These are *serial*, *quasi-concurrent*, and *full-concurrent* (Fuks, Ostroff, and Paige, 2004; Meyer, 1993; Wegner, 1990). *Serial* or *sequential* creates only one thread, which uses a first-in-first-out (FIFO) queue to process messages and executes only one message at a time while other messages wait in a queue. Q*uasi-concurrent* extends *serial* to have an auxiliary queue to enable simultaneous message processing, but only one message can be in execution state at a time.

Full-concurrent takes a different direction by creating multiple threads and enabling simultaneous message executions. However, there will be a need to control message execution, by guarding the shared-state and using wait-set features to pause and resume threads. Also, the messages need to be separated into different independent containers to guarantee message order and run-to-complete semantics.

We extend *quasi-concurrent* to have three priority-based double-ended queues: requests, pending, and deferred executions. A scheduler can be linked with a spawned thread such as an *async* method to control concurrency of the owning objects.

3.4 Messages

Message refers to invocation information, which contains the method delegate and arguments passed. We extend the message to include optional information, such as priority, delay, and guards.

A guard is an anonymous function with a Boolean operator that checks satisfiability. It is mainly used by the scheduler to make a decision to filter, execute, or defer. A defer decision adds a message to a deferred list of messages, such that deferred messages of higher priorities are executed first.

3.5 Time Constraints

Table 2 shows the set of time constraints we introduced into Umple. In an active method, a time constraint can be set at the operation level, action code level, or both of them. The operation level refers to the active method definition, while the action code level refers to the user code written in that active method body.

Typically, action code of an active method is executed sequentially within its owning active object's thread. Nevertheless, some specialized Umple time constructs can be used to enable asynchronous execution, such that it will need to spawn a thread to run concurrently; refer to *period* in the API column, in Table 2.

4 METHOD INVOCATION

In this section, we discuss the features we implemented to improve writing the action code of an active method.

Writing action code in the target language may have limitations. For instance, the C++ 03 standard does not provide an easy way to define anonymous functions. At the model level, we need to have a way to regulate the process of handling error exceptions or *then* calls.

Although the content of an active method runs in a separate thread, this may still have some limitations. For instance, within the action code of the same active method, we may find it important to invoke other methods, which could be regular methods.

In terms of the code generation, an *Active* instance will be created to wrap the regular method within active execution.

The *trigger* operator, "/" is used to invoke a method, either active or nonactive, or an anonymous body. It is used to enforce active behaviour on a method even if it is not defined active. For instance, the call in (Snippet 6 - Line 8) will have its own thread, while the call in (Line 9) will not.

Trigger can be used to embed and call anonymous functions (Snippet 6 - Lines 13-15). In the generated code, we create a new method that has the content of the anonymous function. We make sure that this anonymous method has a unique name derived from the active method name and nonactive method being invoked, augmented with an integer to distinguish each case.

The *call/then* pattern is similar to the try/finally pattern that exists in common programming

languages such as C++ and Java, but it differs in that it works asynchronously. Simply, we wrap method invocation within an anonymous body (Snippet 6 - Lines 19-31). We can directly write code (Lines 19 and 23), or invoke other methods (Line 26). The code generated will make a call to the then body (Lines 22 and 28) after the call body (Lines 20 and 26).

The *call/then/resolve* pattern is used to handle exceptions, such that the resolve body is only invoked upon exceptions (Snippet 6 – Lines 30 and 40).

```
1  class MethodInvocation{                                   Umple
2    void regularMethod(){
3      cout << "A regular method but can
4           be invoked actively";
5    }
6
7    void active regularInvocation(){
8      /regularMethod();
9      regularMethod();
10   }
11
12   void active annomousMethod(){
13     /{
14       cout << "Anonymous body" ;
15     }
16   }
17
18   void active thenMethod(){
19     /{
20       cout <<"Call body" ;
21     }.then({
22       cout <<"Then body";
23     })
24
25     /{
26       this-> regularInvocation();
27     }.then({
28       cout <<"Then body";
29     }).resolve({
30       cout << "Handle Exception";
31     })
32   }
33
34   void active deferredTest(){
35     [/{cout << " Anonymous";},
36      /{this->regularInvocation ();}
37      ].then({
38        cout << " Then body";
39      }).resolve({
40        cout << "Handle Exception";
41      })
42   }
43 }
```

Snippet 6: Method invocations.

Deferred list is a way to combine multiple resolve bodies in a single call (Snippet 6 – Lines 35-36). After the execution of a deferred list, the then or resolve bodies are invoked one time (Lines 38 and 40).

5 CONSTRAINTS

Constraints, either logical or time, are used to guard active method from being invoked unless they are satisfied. These constraints can be applied on scheduler (Table 2), method (Snippet 7 - Line 10), or action code (Line 5) level.

Table 2 shows the list of time constructs that can be applied upon method or action code invocation. A simple example of using the *period* keyword is shown in Snippet 7- Lines 10-13.

```
1  class LogicalConstraintsTest{                             Umple
2    Boolean flag= false;
3
4    void active activeMethod{
5      [flag==true]/{
6        cout <<"Execute only if Flag is true" ;
7      }
8    }
9
10   [period(1000)]
11   void active periodMethod(){
12       cout <<"Called each one second";
13   }
14
```

Snippet 7: Constraints.

6 EVALUATION

We evaluate our work, qualitatively and quantitatively. In our qualitative evaluation, we compare our work against relevant specifications.

In our quantitative evaluation, we evaluate a set of snippets for different variation of concurrency using Umple.

6.1 Qualitative Evaluation

In this section, we show a comparison (Table 4) of our approach against two common specifications, UML (OMG, 2011a) and Modelling and Analysis of Real Time and Embedded systems (MARTE) (OMG, 2011b), which both are used to manage time. We aim in this comparison to show how our implementation of concurrency and active objects can cover the core requirements specified in those specifications.

MARTE is an OMG standard for embedded applications and real-time modelling (OMG, 2011b); it is aligned with UML specifications.

MARTE extends UML in order to have better handling for embedded systems.

Table 4: A comparison of time management specification.

		UML	**MARTE**	**Umple**
Timing model		SimpleTime	Time Package	Language constructs
Time constraints		OCL	Logical and physical constraints	Logical and physical OCL constraints Time units and other accessible variables
Time expressions		Limited (timing diagrams)	Conditional assertions and Jitters	Supported on end-to-end flows and action code
Synchronization		Limited (sequence and activity diagrams)	Supported (TimedConstraint)	Language constructs
Event management	Repetition	Limited	TimedConstraint	Poll
	Reaction			call/then/resolve
	Delay			Delay
	Periodic			Timeout, period, and priority
	Other		Burst, aperiodic, sporadic, time intervals, and workload generator	Delay, guards, and constraints
Scheduling		Sequence and activity diagrams	End-to-end flows	End-to-end flows Support other behavior Umple models. Active blocks
Composition		Interfaces (provides and requires) Runnable entities, takes, operation	Interfaces (provides and requires)	Interfaces (Provides and Requires) Support composition rules through active invoke blocks.
Synchronization semantics		RTC	Depends on synchronization (read/write) events	Depends on synchronization (read/write) events and the precedence of their logical relationship Supports RTC Supports the four types of communication
Trigger		Classes and state machines	Trigger	Trigger Blocks Call/'then/resolve patterns

In Table 4, there are two levels of comparison, operating systems and modelling levels. The comparison criteria at the operating system level are timing extensions, activation events, and scheduling. The comparison criteria at the modelling level are *composition*, *synchronization semantics*, and *trigger*.

The *timing extension* criterion consists of subcriteria: *timing models*, *time constraints*, *time expressions*, and *synchronization*. By a timing model, we refer to how timing is handled in the given technology. In UML, the root package used to handle time is SimpleTime (OMG, 2011a).

In terms of the timing model for MARTE, temporal properties are handled using the "Time" package (Mallet, 2008); this package is often used with a non-normative annex of MARTE, Clock Constraint Specification Language (CCSL). On the other hand, timing in Umple is handled directly using Umple constructs.

In terms of *time constraints*, UML uses Object Constraint Language (OCL) (Gherbi and Khendek, 2006).

In MARTE, there are physical and logical constraints. Both are handled using a clock model, which is handled at the model level, mainly using the CCSL mentioned above. In Umple, we follow OCL semantics when defining physical or logical constraints (Section 5).

By *time expressions*, we refer to constructs provided by a language or specification to create time expressions; e.g. to define time constraints. Time expressions are limited in UML, but a specialized type of sequence diagrams, a *timing diagram* is used to handle time constraints and expressions (Gherbi and Khendek, 2006).

MARTE provides a direct way to define several time expressions such as conditional assertions and jitter.

We showed before that we have three levels to handle time expressions, task, queue, and scheduler levels; constructs used to define time expressions were summarized in Section 5.

In Umple, the constructs provided to handle constraints and time are used to support end-to-end flow.

Synchronization refers to a way used to enforce timing requirements and data flow among channels and events. Examples of synchronization include defining a maximum data rate between input and output events, maximum and minimum jitter, time interval, and absolute and relative duration. An absolute duration refers to hard real-time requirements, which do not accept any sort of delay. A relative duration refers to soft requirements, which accept delays using concepts such as jitter and latency. Latency refers to amount of time taken for

transmission between source and target; e.g. response. Jitter varies over time, since it refers to the variation of latency over time, such as in milliseconds. Stable connections have less jitter (S. Rappaport, 2001).

Synchronization can be either enforced on input or output events. Output synchronization is supported by all items in our comparison. In UML, synchronization is handled via activity and sequence diagrams.

In MARTE, the package TimedConstraint handles both input and output synchronization. In Umple, we handle synchronization using time constructs and the call/then pattern (Sections 4 and 5).

Event management is the way method invocation is handled. Method invocation is temporal and event-oriented so we prefer to refer to the whole process as event management. The subcriteria of our comparison include *repetition*, *reaction*, *delay*, *periodic*, and *other*.

Repetition means making the same calls or invocation several times over a period. However, a repetition rate does not necessarily refer to a repeated sequence of events; it also refers to receiving events from different places such as ports, at the same time. In such a case, the appropriate guards and logical conditions must be applied in order to ensure data acceptance.

A clock port is a good example to describe handling repetition rates. For example, every two seconds a port can receive multiple signals at the same time. In such a case, the port must provide a way to recognize these signals incoming from different places, and properly process them in the right sequence based on the logical and physical constraints.

Reaction is self-explanatory as it refers to the reaction to events or method invocation; this reaction can also involve sending new signals or making new method invocations. Predefined constraints or guards are important to manage reactions.

Delay refers to how to handle delays that are either unintentional or intentional. By intentional delay, we mean that a developer intentionally wants a delay to occur. Unintentional delay refers to delays that occur because of unexpected circumstances such as networking; examples of handling related to this context include jitter, latency, and timeout. *Periodic* refers to the appropriate ways used to handle delays and repetitions such as jitter and latency. By *other*, we refer to any other general terms or additional keywords provided by specifications.

UML does not provide a direct way to handle the abovementioned concepts of event management. A developer will need to implement their event management mechanisms manually. For example, they will need to implement a clock port manually. Event signals are done using diagrams such as state machine, sequence, and composite structure. Concepts such as repetition will be done manually such as using for loops.

In MARTE, a base class, TimedConstraint is used to handle event management. MARTE provides several options to manage events such as burst, aperiodic, sporadic, time intervals, and workload generator.

In Umple, we support event management using time and logical constraints at different levels: model, scheduler, and action code (Section 5). In terms of reactions, we rely on the call/resolve/then patterns (Section 4). Generally, we rely on OCL constructs to build guard conditions.

Scheduling refers to the way that events are scheduled for a period. Scheduling is important to handle timing constraints. In UML, sequence and activity diagrams can be used to handle the sequence of events. Concepts such as join and fork can be used to enable creation or merging of multiple paths of execution.

On the other hand, MARTE has end-to-end flows to handle scheduling. An end-to-end flow enables method invocation from different places or diagrams according to a sequence. A sequence in this context means what method(s) to be invoked next upon method execution. As well, MARTE uses the fork and join constructs.

Similarly, in Umple, we support end-to-end flows. For instance, we can invoke a state machine method from an active method. Joining and forking are also supported in Umple. For example, we can define multiple code blocks in an active method (Section 4), or multiple regions in a state machine.

At the modelling level, we focus in our comparison on the context of platform, analysis, resources, and workflow behaviour. This is summed up to three comparison items, *composition*, *synchronization semantics*, and *trigger*. Generally, *composition* is handled using composite structure diagrams. All items in our comparison rely on interfaces, mainly as *provide* or *require ports*.

Umple supports communication via R-Ports and P-Ports (Orabi et al., 2016) as chains of events. The common ports used in MARTE include FlowPort and MessagePort (Espinoza, Gérard, Lönn, and Kolagari, 2009). Additionally in Umple, developers are able define composition rules such as constraints and guards.

Synchronization semantics refers to the semantics followed for synchronization processes; we mentioned earlier in this section what we mean by synchronization. In UML, the default semantics is run-to-completion (RTC), since active objects are not a part of the UML constructs. For example, direct calls for state machines will have RTC behaviour.

In MARTE, synchronization depends on notifications occurring from read and/or write operations. Such operations require using an appropriate locking mechanism.

We support all types of semantics including RTC, since we support the four types of communication. For example, we can invoke state machine methods via asynchronous methods. This is mainly because in Umple, semantics can be written directly at the code level. As well, synchronization can be applied on events, which can be aligned in a prioritized queue.

A *trigger* is a well-known concept that refers to a method or procedure to be invoked upon a condition or event. In UML, triggers are defined at the level of classes and state machines (Kaneiwa and Satoh, 2010).

In MARTE, triggers are defined as "Trigger" objects. Triggers in Umple can be defined using call/'then/resolve patterns or state machines. A trigger in Umple is indicated using the '/' operator as well as the call/then/resolve pattern.

6.2 Quantitative Evaluation

In this section, we evaluate aspects of our work based on measuring software complexity.

We calculate McCabe Cyclomatic Complexity at the model level based on the Boolean constraints defined, such that each constraint has a weight of two; i.e. two constraints defined will be valued 4. We calculate the complexity ratio as $100 - (Umple\ Mcabe/McCabe) \times 100$. This corresponds to the percent reduction of complexity when writing in Umple, as opposed to the generated C++ code. We computed Cyclomatic Complexity using the LocMetrics tool (http://www.locmetrics.com/).

The code generated by Umple provides additional lightweight libraries to support multi-threading, distributing, and serialization. We exclude this code to avoid bias: In other words, code that would be written by developers in Umple is compared against code that would need to be written by developers in C++ if Umple was not available.

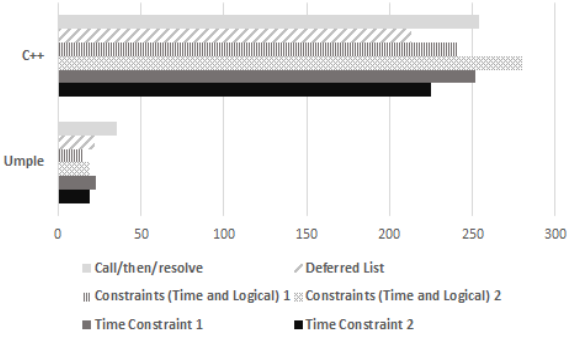

Figure 1: LOC comparison.

In Figure 1, there is a high statistical significant ($p < 0.0001$ and $t = 22.098$) reduction in lines of code between Umple models and their generated C++ code. This reduction averages 222 LOC and 90.9% difference and is roughly constant, hence independent of model size.

Figure 2 shows the reduction in percentage for various Umple models that exercise the Umple constructs. More details about the Umple models used in our evaluation are in (Orabi, 2017).

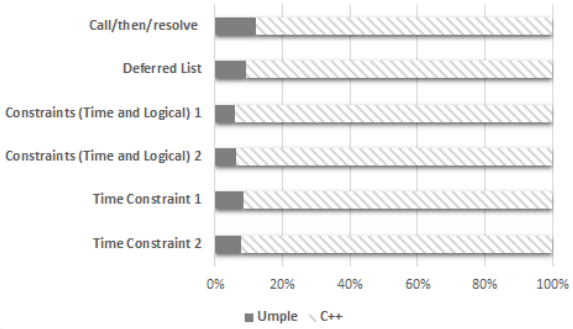

Figure 2: LOC comparison by percentage.

The cyclomatic complexity reduction averages about 67.45%. Figure 3 shows doughnut graph for cyclomatic difference between C++ and Umple.

Figure 3: Cyclomatic Complexity doughnut.

A threat to validity of this analysis is that the C++ code written by a developer might be rather different from that generated by Umple. It may be possible for a developer to leave out some parts, or find other ways to make the C++ more compact. However, we suggest that writing such compact C++ might in fact make it more obfuscated, and hence add even more to complexity.

7 CONCLUSIONS

In this paper, we showed how Umple provides major features required for concurrent programming. The focus was on showing how active object development in Umple will be simplified such that a user will not need to worry about all of its challenges and implementation details.

Concurrent programming in Umple is supported at the model level meaning that concurrency will be consequently enforced at the generated code level. We showed how concurrency definition and implementation in Umple could easily help a developer to optimize performance of their applications

Using simple Umple constructs, a user is able to define their time constraints. We evaluated our work on two bases, qualitative and quantitative. In the qualitative evaluation, we showed a comparison between standards (UML and MARTE) used for time management, and Umple. The essence behind our comparison was to show how Umple can meet time requirements specified in these common standards. For quantitative evaluation, we showed a comparison, based on LOC and cyclomatic complexity, between Umple models and their generated code in C++, based on which we showed significant statistical difference.

For future work, we will highlight the communication among active objects in a distributed environment. This requires the implementation of concepts such as ports and composite structure.

REFERENCES

Alghamdi, A. (2010). *Queued and Pooled Semantics for State Machines in the Umple Model-Oriented Programming Language (Master's thesis)*. University of Ottawa.

Badreddin, O., Forward, A., and Lethbridge, T. C. (2014). *Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity*. *SERA (selected papers)* (Vol. 430). https://doi.org/10.1007/978-3-642-30460-6

Badreddin, O., Lethbridge, T. C., and Forward, A. (2014). A Test-Driven Approach for Developing Software Languages. In *MODELSWARD 2014, International Conference on Model-Driven Engineering and Software Development* (pp. 225–234). SCITEPRESS - Science and and Technology Publications. https://doi.org/10.5220/0004699502250234

Cplusplus.com. (2016). Future. Retrieved June 20, 2001, from http://www.cplusplus.com/reference/future/

Espinoza, H., Gérard, S., Lönn, H., and Kolagari, R. T. (2009). Harmonizing MARTE, EAST-ADL2, and AUTOSAR to Improve the Modelling of Automotive Systems. In *The workshop standard, AUTOSAR*.

Fuks, O., Ostroff, J. S., and Paige, R. F. (2004). SECG: The SCOOP-to-Eiffel code generator. *Journal of Object Technology*, *3*, 143–160. https://doi.org/10.5381/jot.2004.3.10.a3

Gherbi, A., and Khendek, F. (2006). UML Profiles for Real-Time Systems and their Applications. *Journal of Object Technology*, *5*(4), 149–169.

Kaneiwa, K., and Satoh, K. (2010). On the complexities of consistency checking for restricted UML class diagrams. *Theoretical Computer Science*, *411*(2), 301–323. https://doi.org/10.1016/j.tcs.2009.04.030

Klein, P. N., Lu, H. I., and Netzer, R. H. B. (2003). Detecting race conditions in parallel programs that use semaphores. *Algorithmica (New York)*, *35*(4), 321–345. https://doi.org/10.1007/s00453-002-1004-3

Lavender, R. G., and Schmidt, D. C. (1996). Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2* (pp. 483–499). Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Lethbridge, T. C., Abdelzad, V., Husseini Orabi, M., Husseini Orabi, A., and Adesina, O. (2016). Merging Modeling and Programming Using Umple. In *International Symposium on Leveraging Applications of Formal Methods, ISoLA 2016* (pp. 187–197). https://doi.org/10.1007/978-3-319-47169-3_14

Mallet, F. (2008). Clock constraint specification language: Specifying clock constraints with UML/MARTE. In *Innovations in Systems and Software Engineering* (Vol. 4, pp. 309–314). https://doi.org/10.1007/s11334-008-0055-2

Meyer, B. (1993). Systematic concurrent object-oriented programming. *Communications of the ACM*. https://doi.org/10.1145/162685.162705

Microsoft.com. (2015). Delegates (C# Programming Guide). Retrieved January 1, 2017, from https://msdn.microsoft.com/en-CA/library/ms173171.aspx

Ober, I., and Stan, I. (1999). On the Concurrent Object Model of UML*. In *5th International Euro-Par Conference Toulouse, France, August 31 – September 3, 1999 Proceedings* (pp. 1377–1384). https://doi.org/10.1007/3-540-48311-X_193

OMG. (2011a). UML 2.4.1. Retrieved January 1, 2015, from http://www.omg.org/spec/UML/2.4.1/

OMG. (2011b). *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Retrieved from http://www.omg.org/spec/MARTE/1.1/PDF

Orabi, M. H. (2017). *Facilitating the Representation of Composite Structure, Active objects, Code Generation, and Software Component Descriptions for AUTOSAR in the Umple Model-Oriented Programming Language (PhD Thesis)*. University of Ottawa. https://doi.org/10.20381/ruor-20732

Orabi, M. H., Orabi, A. H., and Lethbridge, T. (2016). Umple as a Component-based Language for the Development of Real-time and Embedded Applications. In *Proceedings of the 4th International Conference on Model-Driven Engineering and*

Software Development (pp. 282–291). SCITEPRESS - Science and and Technology Publications. https://doi.org/10.5220/0005741502820291

Rahman, M. (2013). Delegate. In *Expert C# 5.0* (pp. 187–211). Berkeley, CA: Apress. https://doi.org/10.1007/978-1-4302-4861-3_7

Rouvinez, T., and Sobe, A. (2014). Comparison of Active Objects and the Actor Model, Universite De Neuchatel, Institut D'informatique, Rapport De Recherche, RR-I-AS-14-06.1.

S. Rappaport, T. (2001). *Wireless Communications: Principles and Practice*. Prentice Hall; 2 edition.

Smaragdakis, Y., and Batory, D. S. (2000). Mixin-Based Programming in C++. In *Proceeding GCSE '00 Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers* (pp. 163–177).

Sutter, H., and Larus, J. (2005). Software and the concurrency revolution. *Queue*, *3*(7), 54. https://doi.org/10.1145/1095408.1095421.

Wegner, P. (1990). Concepts and paradigms of object-oriented programming. *ACM SIGPLAN OOPS Messenger*. https://doi.org/10.1145/382192.383004.