Umple as a Component-based Language for the Development of Real-time and Embedded Applications

Mahmoud Husseini Orabi, Ahmed Husseini Orabi and Timothy Lethbridge
University of Ottawa, 800 King Edward, Ottawa, Ontario, Canada

Keywords: Umple, Component Modelling, UML, Ports, Connectors, Composite Structure.

Abstract: Component-based development enforces separation of concern to improve reusability and maintainability. In this paper, we show how we extended Umple (http://try.umple.org) to support component-based development. The development of components, ports, and connectors is enabled using easy-to-comprehend keywords. Development is supported in both textual and visual representations. The design pattern followed in our implementation is the active object pattern. We show a comparison between Umple and other modelling tools. We show that Umple has a set of component features comparable to commercial modelling tools, but is the most complete, particularly with regard to code generation, among the open source tools.

1 INTRODUCTION

We describe in this paper how we have extended Umple to support component-based development.

The main motivation behind our research is to simplify component-based development, specifically of real time systems. By component-based development, we mean the implementation of systems from concurrent components with well-defined interfaces, such that components can communicate together via ports and connectors.

Among common programming languages, a few allow real-time development such as C and C++. Languages such as Java require additional efforts to support real-time development.

Model-driven approaches have for a long time been used to develop real-time applications. Compared with directly using a programming language, model-driven approaches give advantages such as enabling multiple target generation, fewer lines of code, and a high level of abstraction.

However, the existing open-source modelling tools have limitations such has having limited capabilities for real-time development.

Umple is an open source model-oriented programming language that allows developers to write their models either visually or textually (Badreddin et al., 2014).

Code generation in Umple has options for different target languages such as Java, C++, PHP, and Ruby. Users can insert their code bodies in code-enabled elements such as methods, states, transitions, and operations. Inserted code can be language-specific, so users can add a code snippet for each target language.

Major UML concepts are supported in Umple such as classes, associations, attributes, and state machines (Badreddin et al., 2014a; 2014b; 2014c). An Umple developer does not need to be a UML expert to write models.

In this paper, we will highlight two of our key contributions. First, we will show how we extended Umple to support component-based modelling with new keywords and corresponding semantics. The development of components will be available both textually and visually similarly to other Umple features. Second, we will show a comparison between Umple and other modelling tools. In this comparison, we will pinpoint the features that are crucial for the support of component modelling. We will show how we managed in our implementation to cover the pinpointed features.

We follow the active object pattern for the implementation of the component-based features (Lavender and Schmidt, 1996). An active object is an object that runs concurrently in a separate thread.

The focus in our discussion is to show how we can use Umple to develop component-based models. Thus, we will not talk in detail about the content of the generated code.

Orabi, M., Orabi, A. and Lethbridge, T.
Umple as a Component-based Language for the Development of Real-time and Embedded Applications.
DOI: 10.5220/0005741502820291
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 282-291
ISBN: 978-989-758-168-7

The Umple models that we will show in this paper assume that the selected target language is C++. These models can be generated and rendered using UmpleOnline (http://try.umple.org).

The support of real-time code generation is also a part of our research but we will not have it addressed in this paper.

In a similar manner to languages such as Java and C++, an Umple developer must define a main function. The content of a main function must be written in the syntax of the selected target language. We will not necessarily show the main functions for all of the Umple models that we will show in this paper.

This paper is organized as follows. In Section 2, we will discuss how component modelling is supported in Umple. In Section 3, we will give an extended example written in Umple. In Section 4, we will show a comparison between Umple and other component-based modelling tools.

2 COMPONENT MODELING USING Umple

In this section, we will discuss the newly introduced keywords to Umple used to support component modelling.

2.1 Umple Components

A component is a structured class that encapsulates a set of active methods and ports. An active method executes within its thread of control, initiates an activity concurrently, and ensures that data is sent and received between ports immediately while following some restrictions such as time constraints.

A class becomes a component if it has at least one active method, port, or connector.

An active method is defined as a regular method but a developer will additionally need to use the keyword "*active*". In Figure 1, there are two active methods defined in Lines 2 and 5. SimpleComponent in Figure will be a component, since it owns active methods

```
1  class SimpleComponent {                          Umple
2    active method1 {
3        cout << "Method1" << endl;
4    }
5    active methodWithParameters(int i){
6        cout << "Method2" << i <<endl;
7    }
8  }
```

Figure 1: An example of component definition.

2.1.1 Parts (Subcomponents)

A component can own multiple parts (subcomponents), which are instances of this component or other components. In such a case, a component is referred to as a composite component. Subcomponents can also be composite.

A subcomponent defines the hierarchical composition and internal structure of its owning components. For example, in Figure 2, part "a" shows an instance of type "A", and part "b" shows an instance of its type "B", while both instances exist in the context of their owner instance "c" that is of type "C". The Umple code of Figure 2 is in Figure 3.

Figure 2: Multiple instances of different components.

```
1  class A { // A component                         Umple
2    active method1 {/* Empty */ }
3  }
4  class B {
5    active method2 {/* Empty */ }
6  }
7
8  class C {
9    A a;
10   B b;
11 }
```

Figure 3: An example of subcomponents.

When a part is added to component, the class of this component will have a composition relationship with the owning class of this part.

Composite relationships between a container and its parts can be defined by simply declaring an attribute of the part's type within the composite. More complex cases may require relationships to be specified explicitly using associations and generalizations.

Concepts such as generalization and multiplicity appear in both composite structure models and class diagrams.

2.1.2 Method Invocation

The invocation of an active method is asynchronous by default. Let us consider the simple example shown in Figure 4, in which a component, SimpleComponent has two active methods, method1, and method2. In Line 3, there is invocation

to the active method method2. The content of method2 will be executed asynchronously.

Invoking method2 will cause the text in Line 8 to be printed indefinitely, and the client will not be blocked.

```
1   class SimpleComponent {                                    Umple
2     active method1 {
3       method2();
4     }
5
6     active method2 {
7       while(true){
8         cout << "Keep outputting" << endl;
9       }
10    }
11  }
```

Figure 4: An example of asynchronous execution of active methods.

The keyword "*synchronous*" is used to allow a developer to force synchronous behaviour on an active method. The keyword "synchronous" precedes the "active" keyword as shown in Figure 5.

```
1   synchronous active method1 {                               Umple
2       //Some synchronous content
3     }
```

Figure 5: An example of synchronous execution of active methods.

If we used the keyword synchronous with method2 defined in Figure 4, this would have caused clients to be blocked indefinitely unless method2 is interrupted. The process of interrupting an active method will be discussed in the next section.

2.1.3 Atomic versus Interruptible Active Methods

Simply, a reentrant active object can be interrupted, while an atomic active object cannot.

In Umple, by default, an active method is interruptible. For simplicity, we do not have a keyword to define interruptible behaviour for an active method. On the other hand, if a developer wants make a method non-interruptible, they can use the keyword "atomic".

Similarly to the keyword "synchronous", the keyword "*atomic*" precedes the "active" keyword. The declaration will be the same as shown in Figure 5, but the keyword "atomic" will be used instead, as in Figure 6.

```
1   atomic active  method2{/*Empty body */ }                   Umple
```

Figure 6: An example of atomic execution of active methods.

A developer can still manually interrupt an active method; this can be done at the level of the target language. For that, we provide an API, FutureObject. To limit the scope of this paper, we will only give a very brief example of the use of FutureObject in Figure 7.

In Figure 7 in Line 8, there is a main function. In Line 9, an instance of a component named Test is declared; a call to a method of this instance is made in Line 10. When making a call to an active method, we get a FutureObject variable, which gives us more options to manage the execution process. In Line 11, there is a call to an API named "stop"; this API is used to interrupt the execution of an active method.

```
1   class Test {                                               Umple
2       active method2 {
3           while(true){
4               cout << "Keep outputting" << endl;
5           }
6       }
7
8       public int main(int argc, _TCHAR* argv[]) {
9           Test test1;
10          FutureObject<void> proxy1 = test1.method1();
11          proxy1.stop();
12      }
13  }
```

Figure 7: An example of interrupting active methods.

There are other API methods in FutureObject such as wait data, subscribe, and unsubscribe.

2.2 Ports

A port is a special attribute owned by a component; it is used as an interface for communication among components. A port is an interaction point used as the origin and/or destination of data to be transferred among components.

A *port* is defined as a regular attribute preceded by any of the keywords "in", "out" or "port" (dual). In Figure 8, three ports are defined in Lines 2-4.

```
1   class SimpleComponent {                                    Umple
2     public in Integer inputPort;  // An in port
3     public out Integer outputPort1;  // An out port
4     public port Integer dualPort1;  // A dual port
```

Figure 8: An example of port definitions.

External and internal ports can be specified through access modifiers such as "public" or "private". Private ports can only access the parts of their owning component

A port is considered a behaviour port if it is used to trigger an event in a state machine.

There are no restrictions on the port type. A port for instance can even be a component.

A port value is transmitted to other ports or components via connectors. When a port type is complex, we apply an appropriate serialization/deserialization technique. A port value is serialized into an intermediary object that can be transmitted in the form of messages. When messages of a transmitted object are received, they are deserialized back to the original object form.

In terms of data transmission and communication, we support both messages and signals, which both are received as 'events' by their recipient.

A message-based event is synchronous and used in point-to-point communication, while a signal-based event is asynchronous. When invoking a synchronous event, the sender will wait until receiving a response.

For simplicity, we will refer to both messages and signals as messages. For distinction, we will describe a message as asynchronous when referring to a signal.

Message transmission is handled by the generated code; developers do not need to worry about the underlying structure of data transmission among components.

A port can have multiplicity, which refers to the lower and upper number of subscribers or clients initiating a communication. It also refers to the process of replicating messages to multiple clients.

If a port has multiplicity of a value more than one, port messages will be replicated to multiple clients. When the upper bound of port multiplicity is unbound, this means that there is no restriction on the number of messages to be replicated.

A port with optional multiplicity means that it is not mandatory to broadcast or replicate messages from this port.

From the above, we can say that port multiplicity is about connection configuration. For instance, it can be used to manage the minimum and maximum number of clients communicating

In Umple, multiplicity is defined within a port declaration or through a port binding definition. In a port definition, multiplicity value can be defined within square brackets after a port attribute name. An example is shown in Figure 9; the port defined in Line 1 has fixed multiplicity of 4, while the port defined in Line 2 is unbounded.

```
1  public in Integer somePort[4];                    Umple
2  public out Integer anotherPort[*];
```

Figure 9: An example of defining port multiplicity.

For simplicity, we do not provide specific keywords to define the type of a port such as service, relay, or end ports. Instead, based on the semantic, type, and connection between ports, a port type is inferred.

For instance, if the modifier of a port attribute is private, this port is considered a non-service port. An "in" port can be either relay or end (Selic, 1998). A relay port means that it propagates messages to other ports as opposed to end ports. As in example, the port "pn1" defined in Line 2 in Figure is a relay port.

2.3 Connectors

A connector or binding is used to specify how a communication channel is initiated, and how data is transferred.

In other words, a connector is used to route requests from a "provided" interface of a component to a "required" interface of the same component or another component. In this context, a connector is used to specify the lower and upper multiplicity bounds of subscribers and clients.

The operator "->" is used to define a connector. This is the same notation used for UML associations in Umple, since a connector acts like an association between active objects. The port on the left hand side is the source, and the port on the right hand side is the target.

If a class interconnects between components via a connector, it will be considered a component even if does not own active methods or ports.

A connector can be defined between two ports in the same component (Line 4 in Figure 10) or different components (Lines 49 and 50 in Figure 14).

In the code generated, we create a method for each port, and it has the same name of this port. This method is used to send signals in the case of "out" ports, or receive signals in the case of "in" ports.

```
1  class Test {                                      Umple
2    public in Integer pIn1;  // An in port
3    public out Integer pOut1;  // An out port
4    0..1 pIn1 -> * pOut1;  // A connector
5
6    after constructor {
7      // Send a value to the other port
8      pOut1(1);
9    }
10
11   after pIn1(int data) {
12       cout << data << endl;
13   }
14 }
```

Figure 10: An example of port binding.

In Figure 10, we use the "after" keyword in two places, constructor (Line 6) and pIn11 (Line 11); the "after" keyword is one of the existing Umple features that enable aspect-orientation.

In Line 8, a signal will be sent via pOut1 upon constructing the class. This signal will be received by pIn1. The value received by pIn1 will be printed out (Line 12).

2.4 Incoming and Outgoing Message Handling

A watch constraint is, like any other Umple constraint, a Boolean condition in square brackets. In the simplest case, it is just the name of a port, and essentially means 'true when data is present on that port'. Examples are shown in Lines 9, 15, 26, and 33 in Figure 14.

A watch constraint is defined before the declaration of an active method. It specifies that the method to handle a port message. An active method can be alternatively considered an incoming or outgoing method based on the port direction.

A watch constraint can encompass other logical and time constraints, as well as multiple ports. The active methods that use watch constraints are used to monitor messages incoming from other ports, or to propagate messages to other ports.

For example, the watch in Line 9 in Figure 14means that the active method "increment" listens to the incoming signals of the port pIn1.

A watch constraint can also define a guard to filter out unnecessary messages. For example, in Line 26 in Figure 14, the active method "stop" will continue as long as the value of a port attribute is less than 10; otherwise, this active method will do nothing.

Generated port attributes are thread-safe. The assumption is that composite components interact in a distributed environment. A port attribute can be accessed from several places, or even accessed externally. To enable this, we make sure that a port value can be accessed in two forms, read-only and read-write.

In read-only form, all clients can access a *history* value of a port attribute. On the other hand, read-write access is given to a single client at a time. This will prevent concurrent update, which can lead to other serious issues such as access violation or data loss. If other clients try to have read-write access at the same time, they will be put into a priority queue if another read-write operation is still in progress.

A priority queue is a FIFO queue, in which requests have priority values set to zero by default. A request that has the highest priority is executed first even if it has been received after other requests in the queue.

2.4.1 Triggers

In the context of an active method, a trigger is defined using the operator "/" appearing at the start of a statement; it is used to invoke a code block or method without blocking (i.e. asynchronously). A code block used in this way is an anonymously defined active block. An example is shown in Lines 8-12 in Figure 11.

```
1   synchronous active method1 {                                  Umple
2       while(true){ /*some asynchronous content*/};
3   }
4   active method2 {
5       method1();
6   }
7   active method3 {
8       /{
9           while(true){
10              cout << "This is anonymous content" << endl;
11          }
12      }
13      /method1();
14      method2();
15  }
```

Figure 11: An example of different triggers and invocation.

Figure 11 shows different types of execution. The anonymous active block defined in Lines 8-12 is executed in an independent thread, since it is preceded by the operator "/". The same is true for the call to method 1. Both of these are immediately dispatched when method3 starts. Without the operator "/", the active block would keep the rest of method3 waiting forever, since there is an infinite loop defined within it.

There is a synchronous active method, method1 defined in Line 1; this method is invoked asynchronously in Line 13. We refer to this kind of behaviour as half-synchronous.

In Line 14 in Figure 11, a synchronous call to asynchronous method, method2 is made. In method2, there is a call to the synchronous method method1; we refer to this type of execution as half-asynchronous.

To wrap up, in Umple, there are four types of execution, full-asynchronous, full-synchronous, half-asynchronous, and half-synchronous.

Full-asynchronous means that active method execution starts asynchronously and remains as such until the end of execution. In a similar manner, full-synchronous means that an active method execution

starts synchronously and remains synchronous until the end of execution. On the other hand, half-synchronous execution means that execution starts synchronously but at some point, has some asynchronous behaviour. Similarly, half-asynchronous execution means that execution starts asynchronously but some synchronous behaviour is enforced during the execution.

2.4.2 Call/then/Resolve Invocation Pattern

Umple has an invocation pattern that we refer to as the call/then/resolve pattern. This pattern provides a flexible representation of the commonly known try/catch/finally.

There is no specific keyword for the "call" part. It is simply a trigger action or method invocation.

The patterns "then" and "resolve" have keywords after their name. Both patterns are optional. We can have different variations such as call/resolve, call/then, and call/then/resolve. The part "then" refers to a callback. Thus, once an active method is done executing, this "then" part will be triggered. The part "resolve" is invoked in case of failures or errors; an example will be shown in Figure 13.

Figure 12 shows the different variations of the call/resolvee/then pattern.

Umple

```
1   /{
2       //call
3   }.resolve ({
4       //catch
5   }).then ({
6       //finally
7   })
8
9   /{
10      //call
11  }. then ({
12      //finally
13  })
14
15  /{
16      //call
17  }. resolve ({
18      //catch
19  })
```

Figure 12: Examples of call/resolve/then invocation.

2.5 Time and Message Handling Constructs

Time is key in real time development. A developer must be given the ability to define soft or hard real time behaviour. In Umple, we have a list of time-based constructs that give options to developers to define their time requirements. Table 1 is a summary of the basic time constructs.

Some constructs are defined at the task level such as poll and delay. A task level construct means that a new trigger starts based on this time-construct.

Other constructs can be defined at the action code level. Constructs at the action code level are defined as watch constraints in square brackets, which we referred to in Section 2.4.

A time construct expects a value in milliseconds. It was important to follow the same metrics used in known languages such as Java. Other constructs expects a numeric value such as "priority".

Table 1: The time and message constructs in Umple.

Constructs	Description
Poll	Invokes active methods in a regular manner.
Delay	Enforces some delay before method invocation.
Priority	Give a priority to an item in a priority queue.
Timeout	Sets a timeout maximum value before a task completion.
Period	Set a time interval to recheck if a condition is satisfied.
Latency	Sets the acceptable delay for a task.

Call/then/resolve and watch statements usually use time and message constructs. Figure 13 shows an example, in which a method, someMethod makes an asynchronous trigger call in Lines 4-5. There is a task time construct, "delay" used to enforce one-second delay before the active block is called. In Line 4, there are three watch constraints. The first is a logical condition used to ensure the active block only works as long as an attribute, "iteration" is less than 2500. The second watch constraint sets a priority of 1 on the defined active block. The third watch statement is a timeout construct of 5 seconds.

Umple

```
1   class Test {
2     Integer iteration= 0;
3     void someMethod(){
4       delay(1000)[ iteration <2500, priority(1),
5         timeout(5000)] / {
6           while(true){
7               cout << iteration << endl;
8               iteration++;
9           }
10      }.then ({
11        cout << " 2500 iterations reached within 5
12         << "seconds"<< endl;
13      }.resolve ({
14        cout << " 2500 iterations not reached within 5
15         << seconds"<< endl;
16      })
17    }
18  }
```

Figure 13: An example of time constructs.

During the execution of someMethod, the value of "iteration" will be incremented by one for each iteration step (Line 8). There are two possible

scenarios to happen. If "iteration" reaches 2500 in less than 5 seconds (the time specified in the "timeout" statement in Line 5), someMethod will exit and then the body in Line 10 will be called. In the second scenario, 5 seconds are exceeded but the value of "iteration" is still less than 2500; in such a case, the resolve body in Line 13 will be invoked.

In a well-designed Umple model, the interruption of an active method should be handled at the model-level using composite structure or timing constructs.

3 AN UMPLE COMPONENT-MODELING EXAMPLE

In this section, we will discuss a simple ping-pong example written in Figure 14 and visualized in Figure 15. We already talked about different segments of Figure 14 during our discussion in Section 2.

In Figure 14, there are two peers denoted as Component1 and Component2; they are contained in PingPongComponent component (Line 40), which will send the initial message (a valid integer value) to port pIn1 of the instance of Component1 to get the communication started (Line 47).

Upon receiving a message at port pIn1, the receiver will increment the received integer value by 1 (Line 11), and reply back to the other component with the new incremented value via port pOut1 (Line 15). The message propagation will continue between the two peers until the value of the sending integer becomes 10 (watch constraint on line 26), so the receiver will have to terminate the communication.

```
1   class Component1 { // A component                    Umple
2     public in Integer pIn1;  // An in port
3     public out Integer pOut1;  // An out port
4     pIn1 -> pOut1;  // A connector
5
6   // A watch constraint defining the in port on which a
7   //signal triggers the following.  An active method
8   invoked
9   //when a signal is received
10    [pIn1]
11    active pingIncrement  {
12      pOut1(pIn1 + 1);
13    }
14  // A watch constraint; the following is triggered when a
15  //signal is sent through this out port
16    [pOut1]
17    active logOutPort1 {
18      cout << "CMP 1 : Ping Out data = " << pOut1 <<
19  endl;
20    }
21   }
22
```

```
23  class Component2 {
24    public in Integer pIn2;
25    public out Integer pOut2;
26    pIn2 -> pOut2;
27    // A watch constraint, and a value constraint
28    [pIn2, pIn2 < 10]
29    active pongIncrementOrStop {
30        // Get a read-only copy of the current cached
31        //value at port Pin2
32        pOut2(pIn2 + 1);
33    }
34
35    [pOut2]
36    active logOutPort2 {
37      cout << "CMP 2 : Pong Out data = " << pOut2 <<
38  endl;
39    }
40  }
41
42  class PingPongComponent {
43    Component1 cmp1;
44    Component2 cmp2;
45    Integer startValue;
46
47    after constructor {
48      // Initiates communication in the constructor
49      cmp1-> pIn1(startValue);
50    }
51    cmp1.pOut1 -> cmp2.pIn2;
      cmp2.pOut2 -> cmp1.pIn1;
    }
```

Figure 14: An Umple component-modelling example.

Each component in Figure 14 has its own incoming and outgoing ports, which are "pIn1" and "pOut1" defined in Lines 2 and 3 in Component1, and "pIn2" and "pOut2" in Component2 defined in Lines 22 and 23. As well, in each component, the connection bindings between this component and the other component are defined.

The port "pIn1" in Line 2 is a relay port, since it propagates signals to other ports in Component2 via "pOut1". In other words, the connector defined between "pIn1" and "pOut1" in Line 4 implicitly declares "pIn1" as a relay port.

The basic event methods in this example are pingIncrement (Line 10), pongIncrementOrStop (Line 27), logOutPort1 (Line 16) and logOutPort2 (Line 34).

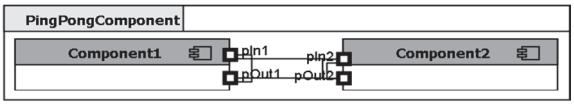

Figure 15: A simple ping-pong example using Umple.

The increment and log methods are defined as active methods designed to watch pIn1. Both methods do not have explicit parameters, because the port value is an implicit parameter. In addition, the name of the port is an implicit method for

Table 2: Comparison of modelling tools for component development (+ = supported; * = partial, - = not supported).

	Functional Features										Code Generation		
	Behaviour				Structural								
Tool	Decomposition	Composite states	Concurrent states	State class	Decomposition	Concept	Data Sharing	Communication	Type Definition	Simulation	Java	C++	C
starUML	+	+	+	-	*	*	*	*	*	-	-	-	-
eTrice	+	-	-	-	+	+ (Actors)	-	X (Port/ protocol)	+	+	+	-	+
ArgoUML	+	-	-	-	-	-	-	-	-	-	*	*	-
RSA-RT	+	+	+	-	+	+(Aggregation/ Composition)	+	X (Port/ protocol)	+	+	+	+	+
IBM Rhapsody	+	+	+	-	+	+(Aggregation/ Composition)	+	X (Port/ protocol)	+	+	+	+	+
PragmaDev	+	+	+	+	+	+(Agents)	+	X (Gate/Interface)	+	+	-	+	+
Umple	+	+	+	-	+	+(Aggregation/ Composition)	+	X (Port/ Active method)	+	+	+	+	-

propagating the data.

Note that in an alternative model, the methods could be defined as state machine events.

In method "pongIncrementOrStop", there is a guard on the data parameter to check if a value received is less than 10; otherwise, communication will end. The guard will filter out other messages.

4 COMPARISON WITH OTHER TOOLS

As stated earlier in this paper, the essence of our research is to introduce composite structure development into Umple, and hence enable component modelling among those who wish to model textually using an open source tool. We have compared Umple to other modelling tools in order to make sure that we fulfilled the core requirements of component-based modelling; the comparison is shown in Table 2.

The selected items in our comparison include industrial and research tools. The tools that we conducted our comparisons against are starUML, eTrice, ArgoUML, Rational Software Architect Real-Time (RSARTE), IBM Rhapsody, and PragmaDev.

Our comparison focuses on the composite structural features. The main such features that we found most widely supported during our research include decomposition, data sharing, communication, and type management. These features depend on the concepts of components, protocols, ports, and connectors. A protocol is used to define information flow between ports.

During our research, we found that other modelling features could affect the development process of component-based applications even if they are not structural. The major feature categories that we found to have a direct impact on structural features include behaviour and simulation. For instance, behavioural features (primarily state machines) are required for behaviour ports. Simulation features are required for many customers that rely on component-based development. For example, in automotive development, simulation is crucial for a complete end-to-end testing process among software and hardware components.

In Table 2, the symbol "+" is used to indicate a moderate level of support; "*" is used to indicate a weak level of support, and "-" means no support.

The structural and C++ code generation features shown in Table 2 are our newly introduced features to Umple; other features listed that Umple supports, already existed before our research.

In our comparison, we do not go into deep detail. For example, we mention whether a tool supports code generation rather than mentioning the quality of the generated code.

In terms of communication, tools can differ based on the standards they support. For example, tools that follow UML standards use ports and

protocols for communication; those tools include eTrice, RSARTE, and IBM Rhapsody. On the other hand, tools that support SDL (Olsen et al., 1994) use gates and interfaces for communication; PragmaDev is an example.

In Umple, we follow UML terminology, which means that communication should rely on ports and protocols. However, we found that asking users to define protocols explicitly adds unnecessary overhead. Thus, we provide protocol-free approach in Umple, which means Umple uses inference to extract the required protocols based on the semantics of ports and connectors. Thus, in Table 2, we mentioned that our communication depends on ports and the pattern of active object.

For state machines, Umple support the major features listed in Table 2 except for the "State Class".

In terms of code generation, the most important target languages supported include Java, C++, and C. Umple supports real time code generation in C++ for all modelling features. Code generation for C and possibly Java will be supported in the future (Umple supports Java already for non-real-time features).

Most commercial tools support behavioural and structural decompositions, while the only open-source tool other than Umple that supports them is eTrice, and it does so only partially. Two other open-source tools (starUML and ArgoUML) have little or no code generation support for component-based modelling.

5 CONCLUSIONS

Umple is an open-source modelling tool. Prior to this work, it supported modelling features such as attributes, state machines, and associations. In this paper, we described new capabilities for composite structure and component-based development such as components, ports, and connectors. As well, we showed how simple time constructs can help users to meet their real time requirements easily.

In component-based modelling, a port definition depends on a protocol. Typically, a protocol defines the flow of signals among ports. In our effort to reduce complexity, we implemented a protocol-free approach, in which protocol information is automatically inferred based on the semantic information of ports and connectors.

We showed a short ping-pong example written as an Umple model; this example is available in the UmpleOnline editor (try.umple.org). The model looks simple, since it consists of only 50 lines of Umple. However, the generated code in C++ as a target language is far more complicated, and exceeds 2000 lines. Complex capabilities such as thread management, mutual exclusion and access queues are taken care of in the generated code. Trying to handle such concepts directly at the programming language level is not an easy task.

We showed how Umple supports both textual and visual development for real-time component modelling.

Part of our effort is that the generated code must not rely on any third-party library, to ensure that it can be deployed easily in real-time operating systems. Library-free code will also relieve developers of integration and configuration management effort.

We showed a comparison between Umple, and other industrial and open-source modelling tools, highlighting the modelling features in each tool. Umple supports most of the major features present in the compared tools.

ACKNOWLEDGEMENTS

OGS, NSERC, and ORF supported this work.

REFERENCES

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

Badreddin, O., Lethbridge, T. C., & Forward, A. (2014a). A Novel Approach to Versioning and Merging Model and Code Uniformly. In MODELSWARD 2014, International Conference on Model-Driven Engineering and Software Development, pp. 254-263. SCITEPRESS. doi:10.5220/0004699802540263

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

Badreddin, O., Lethbridge, T. C., & Forward, A. (2014c). Investigation and Evaluation of UML Action Languages. In MODELSWARD 2014, International Conference on Model-Driven Engineering and Software Development. 2014, pp. 264-273. SCITEPRESS. doi:10.5220/0004699902640273

Lavender, R. G., & 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. Boston, MA, USA.
Olsen, A., Færgemand, O., Møller-Pedersen, B., Smith, J. R. W., & Reed, R. (1994). Systems Engineering Using SDL-92. North Holland.
Selic, B. (1998). Using UML for Modeling Complex Real-Time Systems. ObjecTime Limited/Rational Software Whitepaper, 250–260.