COMPOSITION, EQUIVALENCE AND INTEROPERABILITY: AN EXAMPLE

Leszek Lechowicz (Department of Electrical and Computer Engineering, Northeastern University, Boston, MA; llechowi@ece.neu.edu); Mieczyslaw M. Kokar (Department of Electrical and Computer Engineering, Northeastern University, Boston, MA; mkokar@ece.neu.edu)

ABSTRACT

This paper describes an ongoing effort to use formal methods of software specification and refinement in order to achieve interoperability of Cognitive Radios. In particular, we are interested in the scenario in which two nodes negotiate the composition of software functionality from simpler components, including the ability to infer that the composed module has the same functionality as requested. We show examples of the use of the formal language (Metaslang) to express composition, refinement and abstraction. Moreover, we show that a formal reasoning system can infer the equivalence of two structurally different modules. We also discuss our current research directions towards solving the goal of representation and composition of behavioral (dynamical) models of radio components.

1. INTRODUCTION

We consider a scenario in which two cognitive radios negotiate the use of software components needed for (better) communication. When one radio does not have the component that the other radio asks for, it attempts to achieve the requested functionality by composing it using other components that it has in its software library. One of the questions that the radios need to answer in such a case is whether a specific type of composition results in a functionality that is equivalent to the request.

In order to implement such a scenario, one needs to select a language that is capable of expressing not only descriptions of components, but also their functionality and composition. In addition to this, the formal system built on this language must provide means for inferring equivalences between components. And on top of all this, the language must be able to capture behavioral characteristics of components, i.e., it needs to be able to describe dynamics of systems. It turns out that a relatively powerful language and logic are needed for such an application.

In our previous work [1] we proposed that an approach based on constructivism [5, 6] could be used for achieving dynamic interoperability between Cognitive Radio (CR) nodes. We based our proposal on several key assumptions, of which the most important one was the existence of a *base ontology*, which is a set of basic concepts that is shared among all cognitive radios. Since all more advanced concepts can be expressed in terms of the base ontology, Cognitive Radio nodes can always understand each other. A CR node can add facts to its knowledge databases based on its current status and also based on the data obtained from other CRs. It can also use an inference engine to reason about those facts and can generate composite software components based on data gathered from other nodes.

During our further investigation it became apparent that the previously proposed interoperability scenario has some limitations.

- Equivalent software modules can have different internal structure. For example a software module implementing the equation $f(a,b,c,d) = (a+b)(c+d)$ is functionally equivalent to the module implementing the equation $f(x,y,v,u) = (xv + xu + yv + yu)$ but their respective internal structures are different.
- The same functional modules operating on different data types are seen as two different structures as the structure-based approach does not allow for easy separation of the abstract functionality from the underlying data type.
- The lack of "understanding" of the functionality might lead to implementation inefficiencies. For example, if a CR node receives a description of the quadrature modulator expressed in terms of the base ontology elements (e.g., a composition of an adder, two multipliers and a phase shifter together with the appropriate connections among these sub-components), the CR node might not be able to realize that there might be an alternative, more efficient implementation. In that particular example, if the CR node has hardware support for the multiply-add operation (hardware MAC unit), using a single MAC unit in place of an adder and a multiplier would probably result in better overall efficiency (see Fig. 1).
- There is no obvious way to express dynamics - time dependencies and constraints. One of the possible methods is the use of delay components in the description of the composite software modules. That might however additionally complicate the structure of the module, especially when pipeline processing is considered.

To address all these issues we followed the following approach.

1. Use a more expressive language for describing components. We chose Metaslang for this exercise. Metaslang is a language that supports composition

using constructs of category theory, like morphism and colimit. We investigate how to express the fact that two (syntactically different) components are semantically the same. Moreover, we use Metaslang to capture common parts of different components.

2. Use a tool (Specware) that supports the use of Metaslang, to define abstract specifications of radio components and their further refinements through morphisms and colimit operations.
3. Use a theorem prover that is easily integrated with Specware. We used a theorem prover (Snark) to prove conjectures on functional equivalences of components.
4. Use temporal logic to capture behavioral aspects of components. We used temporal logic to capture such issues as delays and pipelining.
5. Investigate ways of representing behavioral components. We came to the conclusion that an extension to Metaslang, like the Accord framework recently developed at Kestrel institute, might be helpful here.

This paper is organized as follows. In Section 2 we give simple examples of using Metaslang and Specware and show how composition can be expressed in Metaslang. In Section 3 we show how to capture common parts of different components. In Section 4 we continue the same thread by showing how we were able to (automatically, using a theorem prover) infer that two seemingly different components have the same functionality. In Section 5 we discuss the issue of expressing dynamic behaviors. And finally in Section 6 we present our conclusions and directions for future research.

2. COMPOSITION OF COMPONENTS

Specware is a framework created by Kestrel Institute that implements some results of their research into the application of category theory in formalized software development. Specware supports systematic construction of software from abstract specifications to executable code through a series of refinements [2]. In Specware, the software design process starts with an abstract specification, which through category theory operations of morphism and colimit can be refined up to the point where source code in Lisp, Java or C can be generated. An automated theorem prover (such as SRI's SNARK [7]) can be used in each of the refinement steps to prove their correctness. If this process is followed rigorously, the resulting code is correct (i.e. it strictly adheres to the axioms defined in the abstract specifications).

The basis of Specware is a category of specifications and specification morphisms [2]. Pavlovic and Smith [3] define specification as a finite presentation of a theory. The signature of a specification defines concepts for describing objects, operations and properties in some domain. The axioms included in the specification put constraints on the meaning of the symbols.

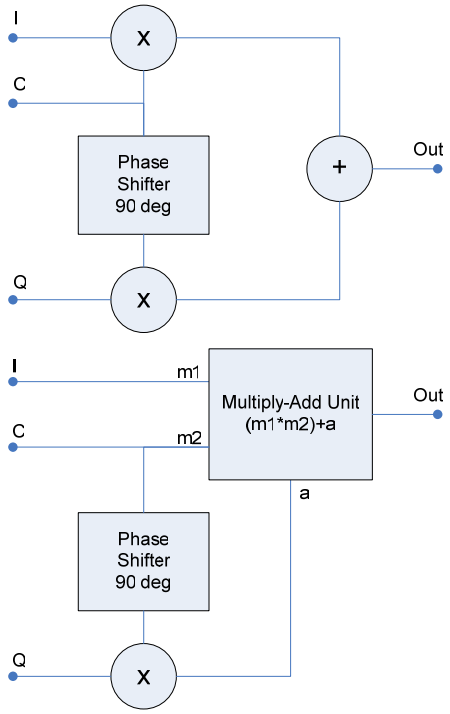

Figure 1 Functionally equivalent composite software modules.

The following code shows an example of two simple specifications encoded in Metaslang – the language of Specware.

```
BinRel = spec
  type E
  op le infixl 24 : E*E -> Boolean
endspec

PreOrder = spec
  import BinRel

  axiom reflexivity is
    fa(x) x le x = true

  axiom transitivity is
    fa(x,y,z)
      ( x le y ) && ( y le z ) => ( x le z )
endspec
```

The first spec *BinRel* defines an abstract specification with an undefined type E and a binary operation, *le*. The second spec (*PreOrder*) refines the first one by adding two axioms concerning the op *le*.

A specification morphism is a translation of the language of one specification into the language of another such that all theorems from the source specification are preserved in the target specification.

```
Antisymmetry = spec
  type X
  op binOp : X*X -> Boolean
  axiom antisymmetry is fa(x,y)
      binOp(x,y) && binOp(y,x) => x = y
endspec

m_BinRel_Antisymmetry =
  morphism BinRel-> Antisymmetry
    { E +-> X, le+->binOp }
```

In the example above, there exists a morphism (*m_BinRel_Antisymmetry*) from *BinRel* to *Antisymmetry* such that all types and ops of the source specification are mapped into appropriate types and ops of the target specification. In this particular case, *E* is mapped to *X* and *le* is mapped to *binOp*.

A specification diagram is a directed graph in which nodes represent specifications and edges represent specification morphisms. For the example specifications and morphism given above we can create the following diagram.

```
BinRelDiag = diagram {
  n1 +-> BinRel,
  n2 +-> PreOrder,
  n3 +-> Antisymmetry,
  e1: n1->n2 +-> morphism BinRel -> PreOrder {},
  e2: n1->n3 +-> m_BinRel_Antisymmetry
}
```

The first three lines in the definition of the diagram define its nodes (n1, n2, n3) and map them to appropriate specs. The definitions of e1 and e2 describe the edges in the graph and map them to morphisms. Note that since the *PreOrder* specification imports *BinRel* (in other words, it is an extension of *BinRel*) the morphism from *BinRel* to *PreOrder* is trivial, i.e. all types and ops of *BinRel* map into themselves.

Given the specification diagram, Specware can produce the colimit of the specs in the diagram. The result of the colimit operation is a spec that contains all the types, ops and axioms of the specs in the diagram in which all types and ops that are linked through morphisms are identified as the same. The colimit of the above diagram can be produced by adding the following statement to the MetaSlang file.

```
PartialOrder = colimit BinRelDiag
```

The result of that operation can be examined in the Specware control shell:

```
    showx test1#PartialOrder
```

As a result, the following listing will be displayed:

```
spec
type {X, E}
op  {binOp, le} : X * X -> Boolean
import translate (BinRel) by { type E +-> {X, E,
E},  op le +-> le}
axiom reflexivity is fa(x : E) x le x = true
axiom transitivity is fa(x : E, y : E, z : E) x le
y && y le z => x le z
axiom antisymmetry is fa(x : X, y : X) binOp(x, y)
&& binOp(y, x) => x = y
endspec
```

3. ABSTRACTION AND COMMONALITY OF COMPONENTS

In our current experiments we looked into how the category theory ideas developed in Specware could help with solving the interoperability problem. One of the problems with the structure-based interoperability scenario was the difficulty of separating the abstract functionality of the composite software module from the underlying data type.

For example a multiply-add unit processing real samples represented by floating point numbers will be composed quite differently than a unit processing pairs of integers representing complex samples. Nevertheless, both will have some commonality. If we simply treat such two components as totally different, we loose the ability to capture the commonality. This will lead to some inefficiency since the part that is common to both will be represented twice, while in fact both components are "the same" at some level of abstraction. The notion of abstract specification and its refinement through morphism and colimit operation seem to be very well suited for addressing this problem.

```
Samples = spec
  type Sample
  type NonZeroSample = (Sample | nonzero?)

  op Sample.zero: Sample
  op Sample.one: Sample

  op Sample.nonzero?: Sample->Boolean
  def Sample.nonzero?(x) = x ~= Sample.zero

  op Sample.multiply: Sample*Sample->Sample
  op Sample.add: Sample*Sample->Sample
  op Sample.minus: Sample->Sample

  op Sample.subtract: Sample*Sample->Sample
  def Sample.subtract(x,y) = add( x, minus(y))

  op Sample.next: Sample->Sample
  op Sample.prev: Sample->Sample

  axiom Sample_mul_ax is
    fa(a:Sample, b:Sample)
      Sample.multiply(a,b) = Sample.multiply(b,a)
  axiom Sample.add_ax is
    fa(a:Sample, b:Sample)
      Sample.add(a,b) = Sample.add(b,a)
  axiom Sample.mul_add_ax is
    fa(a:Sample, b:Sample, c:Sample)
      Sample.multiply(a, Sample.add(b,c)) =
        Sample.add( Sample.multiply(a,b),
                    Sample.multiply(a,c) )
```

```
endspec

IntSamples = spec
  import Samples
  type Sample = Integer

  def Sample.zero = 0
  def Sample.one = 1
  def Sample.multiply(x,y) = x*y
  def Sample.add(x,y) = x+y
  def Sample.minus(x) = -x
endspec

CplxIntSamples = spec
  import Samples
  type Sample = { re:Integer, im:Integer}
  def Sample.zero = { re=0, im=0 }
  def Sample.one = { re=1, im=0 }
  def Sample.multiply(x,y) =
      { re = ( x.re*y.re - x.im*y.im ),
        im = ( x.re*y.im + x.im*y.re ) }
  def Sample.add(x,y) =
      { re = (x.re+y.re), im = (x.im+y.im) }
  def Sample.minus(x) = { re = -x.re, im = -x.im }

  op  Sample.conj: Sample -> Sample
  def Sample.conj(x) = { re = x.re, im = -x.im }
endspec
```

In the MetaSlang code shown above an abstract specification *Samples* is defined. Two concrete refinements *IntSamples* and *CplxIntSamples* (for real and complex samples, respectively) are also defined. If a software module using *Samples* is defined, it can easily be refined into a concrete software module specification using real samples or a module using complex samples. It is generally considered a good practice to introduce real data types at the very end of the refinement process, just before the source code is generated.

```
MorphInt =
  morphism Samples -> IntSamples { }

MorphCplxInt =
  morphism Samples -> CplxIntSamples { }

Adder= spec
  import SampleSpec#Samples
  op Adder.Func: Sample*Sample -> Sample
  def Adder.Func(x,y) = Sample.add(x,y)
endspec

Adder_Int = Adder[MorphInt]
Adder_CplxInt = Adder[MorphCplxInt]
```

The example above shows the underlying data type refinement. Specifications *Adder_Int* and *Adder_CplxInt* are obtained by specification substitution, which is a simplified form of colimit operation.

4. FUNCTIONAL EQUIVALENCE

The use of theorem provers makes it possible to prove that two software modules are equivalent. For example the following spec defines two ops *Func1* and *Func2*.

```
Funcs = spec
  import Adder
  import Multiplier

  op Funcs.Func1: Sample*Sample*Sample -> Sample
  def Funcs.Func1(a,b,c) =
    Multiplier.Func( a, Adder.Func(b,c) )

  op Funcs.Func2: Sample*Sample*Sample -> Sample
  def Funcs.Func2(a,b,c) =
    Adder.Func( Multiplier.Func(a,b),
                Multiplier.Func(a,c) )

  conjecture Funcs_eq_conj is
    fa(a:Sample, b:Sample, c:Sample)
      Func1(a,b,c) = Func2(a,b,c)
endspec
```

The evaluation of the conjecture *Funcs_eq_conj* is done by the theorem prover invocation:

```
p0 = prove Funcs_eq_conj in Funcs options
     "(use-resolution t) (use-paramodulation t)"
```

In the above example, SNARK [7] is able to prove that Func1 and Func2 are the same, even though they are defined in a different way. It has to be emphasized though that SNARK is not able to deal with any knowledge representations that require higher order logic.

Although Specware itself cannot optimize the implementation of a specific software module, at least it can recognize that an implementation is equivalent to the given specification even though it might be constructed using composite software modules rather than simple components from the base ontology. In the following example specification *QuadratureMod* is expressed with elements from base ontology. The receiving CR node composes specification *Quad2*, which uses a composite module MAC not present in the base ontology. The reasoner is able to prove that the specification used by the receiving CR node is equivalent to the original specification.

```
QuadratureMod = spec
  import CplxIntSamples

  op QuadratureMod.Func: Sample*Sample*Sample ->
Sample
  def QuadratureMod.Func(I,Q,C) =
    Sample.add( Sample.multiply( I, C ),
                Sample.multiply( Sample.conj(C), Q
) )
endspec

MAC = spec
  import CplxIntSamples
  import Adder_CplxInt
  import Multiplier_CplxInt

  op MAC.Func: Sample*Sample*Sample -> Sample
  def MAC.Func(m1,m2,a) =
    Adder.Func( Multiplier.Func(m1,m2),a )
endspec
```

```
Quad2 = spec
  import CplxIntSamples
  import Adder_CplxInt
  import Multiplier_CplxInt
  import PhShifter90Deg
  import MAC
  import QuadratureMod

  op Quad2.Func: Sample*Sample*Sample -> Sample
  def Quad2.Func(I,Q,C) =
    MAC.Func(I, C, Multiplier.Func(
                    PhShifter90Deg.Func(C), Q ) )

  conjecture Quad2_conj is
    fa(I:Sample, Q:Sample, C:Sample)
     Quad2.Func(I,Q,C) = QuadratureMod.Func(I,Q,C)
endspec

Quad2_p0 = prove Quad2_conj in Quad2
     options "(use-resolution t) (use-
paramodulation t)"
```

5. TEMPORAL LOGIC ELEMENTS IN COMPOSITE SOFTWARE MODULE SPECIFICATIONS

Practical implementations of Cognitive Radio algorithms require that timing constraints are considered when composite modules are constructed from simpler entities. It is especially important when hardware components are used as the algorithms have to adjust to delays introduced by pipelining, which is a technique regularly used to improve the speed of hardware implementations.

There are many *temporal logics* aimed at tackling different aspects of time in complex systems without introducing time explicitly. In our limited experimentation we borrowed a simple concept from Linear Temporal Logic – the operator X (next).

In discrete time systems (as all sample-based systems are) the operator X is a delay element. The X operation is not a function as understood by functional languages because the result of it doesn't depend only on the current value of the parameter. Instead, it actually returns the previous value of the parameter. In other words, the X operator uses a side effect – it returns the value remembered from the last time the function was called in the particular context.

Since MetaSlang is a functional language it cannot implement the X operator. That limitation however does not prevent us from defining the X operator as an abstract operation with some axioms. We might not be able to refine specifications to the actual source code, but we could reason about the operator itself in an abstract way.

```
 LTL = spec
   import Samples
   op LTL.X: Sample -> Sample

   axiom X_ax1 is
     fa(a:Sample, b:Sample)
       LTL.X( Sample.add( a,b ) ) =
         Sample.add( LTL.X(a), LTL.X(b) )
   axiom X_ax2 is
     fa(a:Sample, b:Sample)
       LTL.X( Sample.multiply( a,b ) ) =
         Sample.multiply( LTL.X(a), LTL.X(b) )
 endspec

 Delay = spec
   import LTL

   op Delay.Func: Sample -> Sample
   def Delay.Func(y) = LTL.X(y)
 endspec

 Adder_1Delay = spec
   import Samples
   import LTL

   op Adder_1Delay.Func: Sample*Sample -> Sample
   def Adder_1Delay.Func(x,y) =
         LTL.X(Sample.add(x,y))
 endspec

 Multiplier_1Delay = spec
   import Samples
   import LTL

   op Multiplier_1Delay.Func: Sample*Sample->
Sample

   def Multiplier_1Delay.Func(x,y) =
         LTL.X(Sample.multiply(x,y))
 endspec

 MAC_2Delay = spec
   import Samples
   import LTL

   op MAC_2Delay.Func: Sample*Sample*Sample ->
Sample

   def MAC_2Delay.Func(x,y,z) = LTL.X(
     Sample.add( LTL.X( Sample.multiply(x,y) ),
                 LTL.X( z ) ) )
 endspec

 MAC_2X = spec
   import Delay
   import MAC_2Delay
   import Multiplier_1Delay
   import Adder_1Delay
   import MAC_2Delay

   op MAC_2X.Func: Sample*Sample*Sample -> Sample
   def MAC_2X.Func(x,y,z) =
     Adder_1Delay.Func(
       Multiplier_1Delay.Func(x,y),
       Delay.Func(z) )

   conjecture MAC_2X_conj is
     fa(x,y,z)
       MAC_2X.Func(x,y,z) =
       MAC_2Delay.Func(x,y,z)
 endspec

 MAC_2X_p0 =  prove MAC_2X_conj in MAC_2X
     options "(use-resolution t)
       (use-paramodulation t)"
```

In the above example *MAC_2X* – a specification of multiply-add unit that introduces a sample delay of two steps - is created from the available components *Multiplier_1Delay*, *Adder_1Delay* and *Delay*. This specification has been proven to be equivalent to *MAC_2Delay* expressed in terms of basic *Sample* operations and the X operator.

6. CONCLUSIONS AND FUTURE WORK

In this paper we described the results of some experiments in which we tried to apply certain elements of category theory to the problem of Cognitive Radio interoperability. We successfully demonstrated the feasibility of this approach as we were able to overcome some shortcomings of the structure-based approach we proposed previously. The practicality of this approach is still however debatable as it requires the use of a theorem prover, The available theorem prover SNARK is limited to first-order logic, which prevents the software specifications from using more expressive language constructs. The use of Isabelle – a higher order logic prover, experimentally introduced to the latest version of Specware - has not been evaluated at this point, but the experimental character of both Isabelle and its support in Specware lead us to believe that this option would probably also encounter significant problems.

The fact that MetaSlang is a functional language limited our success with the application of temporal logic elements to software component specification. We were unable to produce source code for modules using the X operator. We were able however to use its abstract definition and axioms in proving the functional equivalence of module specifications.

In our future work we plan to evaluate Accord, another formal software development framework from Kestrel Institute. Accord is an extension of Specware and it introduces a concept of evolving specifications (e-specs). Evolving specifications allow for the introduction of behavioral elements (like states and their transitions) into specifications of software modules.

7. REFERENCES

[1] L. Lechowicz, M. Kokar. *Achieving Dynamic Interoperability of Communication: Transfer of Ontology and Rules Between Nodes.* In Proceedings of the Software Defined Radio Technical Conference SDR'06, 2006.

[2] Y. V. Srinivas, R. Jullig. *SPECWARE: Formal Support for Composing Software.* Tech. Rep. KES.U.95.5, The Kestrel Institute, Palo Alto, CA, 1995.

[3] D. Pavlovic, D. R. Smith. *Software Development by Refinement.* In UNU/IIST 10th Anniversary Colloqium, Formal Methods at the Crossroads: From Panaea to Foundational Support. Springer-Verlag, 2003.

[4] *Specware 4.2 Language Manual*, Kestrel Development Corporation, 2007

[5] Ference Marton and Shirley Booth. *Learning & Awareness.* Hillsdale, N.J.: Lawrence Erlbaum Associates, 1997.

[6] CSCL (Computer Supported Collaborative Learning), Pedagogical Information Science at the University of Bergen, Norway. Avaialble at: http://www.uib.no/People/sinia/CSCL/web_struktur-4.htm

[7] M.E.Stickel, R.J.Waldinger and V.K.Chaudhri. A Guide to SNARK. www.ai.sri.com/snark/tutorial/tutorial.htm.