Mechanization of Invasive Software Composition in F-Logic

Ade Azurat
University of Indonesia
Faculty of Computer Science
Depok, 16424
Indonesia

Abstract: F-Logic is originally developed to bridge between computational formalism and data specification language while providing clear object-oriented semantic. F-Logic provides partiality on attributes and methods. This partiality is also required by software composition in practice. Invasive Software Composition composes software component by applying invasive composition operator (composer) to adapt the component. This adaptation may involve partiality. The combination of object-oriented and partiality characteristic of F-Logic provides a promising framework for the mechanization of Invasive Software Composition including aspect separation.

Key–Words: Reusability, Reliability, Software Composition, Software methodology.

1 Introduction

As in electrical engineering component software is usually assumed as a *black box* unit. Based on this assumption, several formalisms are shown in [4, 5, 7, 8].

Latest development shows that it is possible to relax the *black box* constraint of component unit which will provide possibility for efficient composition. The *invasive software composition* is introduced in [2]. It considers the component unit as a *grey box* which allows some parts of the component to be changed. Therefore, the component interface should not be hard coded or fixed. The composition is conducted by transforming the component. This idea allows an advance degree of composition which required a specific language for composition such as Compost[2], Gloo[11], or Piccola[1].

Rather different to Compost, which provides implementation of composition and its component in Java, the Gloo Language is meant to be programming language independent. The bridge to concrete programming language such as Java is defined by providing a built-in gateway mechanism based on the λ-F concept[11]. Gloo refines the reliability difficulty in Piccola due to a conceptual gap between the mechanisms offered by Piccola as mentioned in[11]. The reliability of software composition is derived from the well defined λ-F calculus which provides a more integrated mechanism. The λ-F calculus is basically an abstraction of λ calculus for component software. It is in line with the development of λ-N[6] which has provided a formal foundation for software reusability.

1.1 F-Logic

The design of λ-F and λ-N calculus are important especially in invasive software composition which may require the modification of interface. Those calculus are trying to solve the problem that arises from the parameter permutation or incomplete parameter in method calls (partial calls). In invasive software component, the composition is not necessary on method call bases. It may happen on type (class boxes) or package bases (package boxes) where the permutation and possible partiality can also occur. However, the representation is basically in the same syntax, which required additional effort to differentiate each hierarchy. In type λ-N, the introduction of type could the be hierarchy to be differentiated but not really show the structure since the type is not related.

F-Logic[10] is originally developed to bridge between computational formalism and data specification language while providing clear logical semantic for the object oriented paradigm. The later issue is more popular, therefore F-Logic is more known as An Object-Oriented Knowledge Base Language.

The F-Logic is basically a set of F-Molecule's and rules. Rules are as in the descriptive, rule-based programming. F-Molecule's defines object description.

Derived from the need to be able to represent so called "null value" in database application, F-Logic accepts partiality on attributes and methods. It allows us to define some objects in a class but undefined on another. It also allows the incomplete parameter in method calls. The rich object-oriented notion in F-Logic, allows us to have hierarchy on object, including to define method as an object where the param-

eters are attributes. Within this partiality and object oriented characteristic of F-Logic, the need of permutation and partiality of invasive software composition as mentioned before can be defined without the problem of differentiating the hierarchy as mentioned before.

Moreover, F-Logic has a model-theoretic semantic and a sound and complete resolution based proof theory. It provides a natural framework for reliable composition reasoning. The resolution mechanism can be used as constraint check on incomplete or mismatched composition. In the case of incompleteness, the unification will find suitable value binding which acceptable by the system. Those mentioned characteristic of F-Logic provide a higher abstraction to do reasoning compare to other similar works which are based on Prolog [3, 9].

1.2 Invasive Software Composition

Invasive software composition composes software component by applying invasive composition operator (composer) to transform the component to adapt the composition. In [2] the Model of Invasive Software Composition is defined as follow:

A fragment box is a set of program elements. A fragment box has composition interface that consists of a set of hooks.

A hook is a point of variability of a fragment box, a set of fragments, or position that is subject to change.

A composer is a program transformer that transforms one or more hooks for reuse context.

1.3 Aspect Separation

A system may required update or feature addition. It required modification to the system. However, some modifications may not be compositional[12] in the sense it can just be added as a separate module or component. Some of the modification may required the whole component to be modified. For example, if we would like to have a debugging feature. All modules may have to put its own debug information, therefore all modules are infected. This situation is called *aspect weaving*.

Invasive Software Composition supports aspect separation. A *point cut* can be modeled as a specific type of homogeneous hook in all composed components. For example an aspect of speed customization on production cell case study can be modeled as a hook of type `speedHook` in all components. The composition rules should guarantee that all `speedHook` will be instantiated and will be checked for compatibility. This compatibility check is well modeled in resolution based such as F-logic (based on Prolog). The *horn-clause* style reasoning is automatically applied on the hook and fragment box definition for consistency.

2 Mechanization of ISC in F-Logic

The Object oriented and partiality characteristic of F-Logic allows quite straight forward software composition description in F-Logic. Some of the design decision that we made are merely concern with the execution of F-Logic program in the system we use, which is the Flora-2[14]. Therefore, we prefer the word mechanization because we believe we are able to mechanize things which are previously has to be done manually. Such as **hook identification**, **partial function substitution**, and **integrity constraint reasoning**. It is possible due to the reasoning capability of F-logic which allow us to automate some of the manual reasoning a human does on doing software composition.

Here is the basic schema to mechanize the invasive software composition in F-Logic:

A fragment box is defined as regular f-molecule, it defines its own class. The methods and attributes of the fragment box are defined straight forwardly as the attribute of the F-molecule.

A hook There is no specific declaration of hook. The user will choose and write it manually which element of the fragment boxes is a hook. It can be put as the parameter to the object instant. That parameter can be of any element of type or any attribute of the class.

A composer is defined as rule which deduce the value of a hook. The user composes component by writing rules. These rules will adapt the fragment box with the given parameter and available component and create a complete system (can be also said a composition)

We will explain more about the alternative using the production cell example as mentioned in [2].

2.1 Case Study: Production Cell

The production cell system consists of several machines. Those machines are modeled as components. They are `feed belt`, `rotary table`, `robot arm`, `press`, `deposit belt`. Those machines have metal blanks as their input and produce

metal plate tin boxes. For example, as hook we define the type on input and output to be the hook of type class. Notice that the feed belt and the deposit belt are actually the same machine, they only have different item. In practice they may really be the same type of machine. Therefore, we also should put one fragment to represent both of them.

Below are some descriptions of production cell fragment boxes and hooks in F-Logic.

```
conveyorFragment(T)[inputHook=>T,
                    outputHook=>T].

rotaryFragment(T)[piece1inHook=>T,
                  piece2inHook=>T,
                  piece1outHook=>T,
                  piece2outHook=>T].
```

We define here two fragments boxes. In F-logic, the codes above are called F-Molecules. An F-Molecule could be a description of a class, or a representation of objects or facts. The above codes are description of the class fragment boxes. The symbol `=>` represent type declaration. The first fragment box is the `conveyorFragment`. It has two attributes. Both have type T. The class is parameterized by type T, which represent the binding template for the hook. Notice that this `inputHook` and `outputHook` are attributes of class `conveyorFragment`, they can be seen as the $required$ and $provided$ port of the `conveyorFragment` fragment box.

2.1.1 Reusable component generation

The Invasive Software Composition composes software components by program transformation[2]. Those program fragments of production cell case study should be transformed and adapted for a combined component. In F-Logic this composition process which conducts transformation and adaptation is defined as a rule. Below are some of the rules for production cell case study:

```
_Belt(M):conveyorComponent(T)
         [inputHook->M,
          outputHook->M]
  :- M:T,
     _Belt(M):conveyorFragment(T).

R(M):rotaryComponent(T)
         [piece1inHook->M,
          piece2inHook->M,
          piece1outHook->M,
          piece2outHook->M]
  :- M:T,
     R(M):rotaryFragment(T).
```

Lets take a look at the first rule. It is the rule to transform an unbound fragment box `conveyorFragment` to a reusable component `conveyorComponent`. The symbol "`:`" represents typing declaration. It says that any instantiation of conveyor component, will have value M as its input and output hooks binder. That value M should have type of the expected type T. We put additional constraint of `_Belt(M):conveyorFragment(T)` to force the composition to find that we do have the object instantiation of conveyor fragment which may consist of some non-adaptable codes or other such as copy write watermarks. In F-Logic the existences of those object instantiations are defined as follow:

```
metalBlank_Atype::metalBlank
        [dimension*->(10,10),
         weight*->10,
         code*->a1].

blankA:metalBlank_Atype
       [price->100,
        vendor->intel].

feedBelt(blankA):
    conveyorFragment(metalBlank)
        [manufacture=mercedec,
         controlcode=feedbelt.code].
```

The symbol "`::`" represents sub typing. These codes are an example of using F-Molecule to represents object and facts. Those F-Molecule's define the data type of input which will be bound to the conveyor hook. In the last two lines, we define the object instantiation of feed belt and deposit belt. Those lines are not creating any component yet. It gives a name `feedBelt` for the conveyor belt which is bound to `blankA`. The `feedBelt` now inherits all unbound and adapted elements of the conveyor fragment box. They have nothing regarding the adaptation part of the component. The previous rules will create an instantiation of reusable component of type conveyor component. That rules will fill in the `feedbelt` with the necessary adapted code such as that the input hook and the output hook should have the value of "blankA". Due to the F-Logic machinery which is based of logic programming, this is possible. For example instead of just fill in the input and output hook, the composition may also modify some program elements in conveyor fragment as follows:

```
_Belt(M):conveyorComponent(T)
```

```
        [inputHook->M,
         outputHook->M,
         speed->S,
         feedingMethod->F]
  :- M:T,
     _Belt(M):conveyorFragment(T),
     rangeSpeed(M,S),
     feeding(M,F).
```

The rule above says, that the reusable component of conveyor has not only its input and output hook filled in, but also has the speed and feeding method modified. This modification, should follow from the other given predicate `rangeSpeed(M,S)` and `feeding(M,F)`. Those predicates provide possible value for the speed and feeding method. Later on this value may also be checked for constrain restriction with other component. Those program elements which are `speed` and `feedingMethod` may not be part of the hook. If it is part of the hook, than the treatment is the same, but if it is not declared as hook, but possible influenced by its composition than it will be the job of the component vendor to provide transformation template.

2.1.2 Software Component Composition Rules

The idea of invasive software composition is to transform or adapt the fragment box when we would like to develop a software composition. The previous explanation seems to let us think that we have to provide the re-usable component first before conducting the composition. It may seem correct in the imperative setting, but F-Logic is based on logic programming which is declarative. So actually, those reusable components will not be created, unless the transformation rules are executed. The transformation rules will not be executed unless there is a need for those reusable components. The need of those reusable components is defined by the description and composition rules. Below is an example of composition rule for the production cell case study.

```
productionCellSystem(T1,T2)
 [feedBeltComp=>conveyorFragment(T1),
  rotaryTableComp=>rotaryTable(T1),
  robotComp=>robot(T1,T2),
  pressComp=>press(T1,T2),
  depositComp=>conveyorFragment(T2)].

_P(M1,M2):productionCellSystem(T1,T2)
  [feedBeltComp->CompFeedBelt(M1),
   rotaryTableComp->CompRotaryTable(M1),
   robotComp->CompRobot(M1,M2),
   pressComp->CompPress(M1,M2),
   depositComp->CompDepositBelt(M2)]
:-
 CompFeedBelt(M1):conveyorComponent(T1),
 CompRotaryTable(M1):rotaryComponent(T1),
 CompRobot(M1,M2):robotComponent(T1,T2),
 CompPress(M1,M2):pressComponent(T1,T2),
 CompDepositBelt(M2):conveyorComponent(T2),
 M1:T1, M2:T2.
```

Notice that the form of production cell description is also defined as F-Molecule. It allows us to have a hierarchy of component, so a system could be a fragment box as well. The F-Molecule above describes the production cell, as a fragment with two possible bindings. It has five elements which are all components. Those components should be instantiated according to the mentioned type and hook respectively. The second part of the code is our main concern. It is an F-Logic rule, which define the composition program to create a production cell system. It can be read as: to have a production cell system, we need an object instantiation for each required reusable component. For example, the description says that a production cell has `feedBeltComp` element of type `conveyorFragment` with a hook. The rule requires that the instantiation of a production cell system should have a reusable component of `conveyorComponent` which is the reusable component of type `conveyorFragment`. The rule also mentions that the hook of all of component should match accordingly.

2.1.3 Aspect Separation

Aspect is added as subclass of the previous composition rules. Again due to the partiality in F-Logic, it provides a nice separation from the core component. Any new aspect can be defined separately to each other and to the original code. The joint point is defined in term of implicit hooks which by default are defined in the modeling.

Below is the example to add tracing aspect for the production cell case study.

```
conveyorComponentTrace(_T)::
     conveyorComponent(_T)
 :- writeln('Trace Aspect weaving').
```

This rule will weave any object instantiation of `conveyorComponent` with the tracing code, if we decided that those components are of class `conveyorComponentTrace`. It provides a separation abstraction, which mean that the component vendor can prevent its component to be woven to illegitimate aspect by not providing its subclass declaration. So although the end user has defined it aspect requirement but if the component has not been allowed

to sub class the provided aspect trace weaving code, then the component will not do any tracing. The sub classing can be defined directly as a fact, or it can also be defined as rule. Below are the example in Production cell case study, we would also like to show that the tracing aspect can be propagated.

```
feedBelt(blankA)
    [method=>>methodFragment].
feedBelt(blankA)
    [method->>{methodOne,methodTwo}].
```

For example the `feedBelt` has several methods and usages of other class, as describe above. The below code define the weaving code of the tracing aspect. Notice that, these pieces of code are written after the previous code without changing anything from the core component. We define those methods and classes with type `Fragment`. It is important to allow them to be adapted in general manner since the non `Fragment` type of element are not by default meant to be modified.

```
methodTrace::methodFragment.

X:methodTrace:-
        X:methodFragment,
        write('add traces in method: '),
        writeln(X).

_Belt(M):conveyorComponentTrace(T):-
    _Belt(M):conveyorComponent(T),
    _Belt(M)[method->>X:methodTrace],
    _Belt(M)[class->>Y:classTrace],
    _Belt(M)[debugInfo->Notes],
    writeln(Notes).
```

Another motivating observation is that the F-Logic mechanization allows separation on aspects. Those aspects may have included other language constructs which are not part of the core component. The coding in F-Logic allow us to separate attributes in several F-Molecule's and yet always consider them as a whole. Of course the technical problem of *Object Schizophrenia*[13] may still occur on standard compilation scheme. However the F-Logic style of coding is quite similar to scheme migration[2] which could eliminate the problem.

3 Application: Checking Architectural Feature with F-Logic

The main aim of mechanizing Invasive Software Composition in F-logic is to allow reasoning on the composition architecture. The reasoning allows us to easily provide many additional checking. Those checking's may involved things such as cyclic check, reachability, and constraint check. We will discuss briefly one type of checking.

The Composition Constraint Check is one of advantage of ISC mechanization in F-Logic. Due to the fact that F-Logic is based on Logic programming, the constraints check become very natural in coding. For example in production cell case study, one may want to have more constrains on the system based on the speed of each component. We can modify the composition code as follow

```
_Belt(M):conveyorComponent(T)
         [inputHook->M,
          outputHook->M,
          speed->S,
          feedingMethod->F]
  :- M:T,
     _Belt(M):conveyorFragment(T),
     rangeSpeed(M,S),
     feeding(M,F).

_P(M1,M2):productionCellSystem(T1,T2)
 [feedBeltComp->CompFeedBelt(M1),
  rotaryTableComp->CompRotaryTable(M1),
  robotComp->CompRobot(M1,M2),
  pressComp->CompPress(M1,M2),
  depositComp->CompDepositBelt(M2)]
:-
 CompFeedBelt(M1):conveyorComponent(T1),
 CompRotaryTable(M1):rotaryComponent(T1),
 CompRobot(M1,M2):robotComponent(T1,T2),
 CompPress(M1,M2):pressComponent(T1,T2),
 CompDepositBelt(M2):conveyorComponent(T2),
 M1:T1, M2:T2,
 CompFeedBelt(M1)[speed->S],
 CompRotaryTable(M1)[speed->S],
 CompDepositBelt(M2)[speed->S],
 CompRobot(M1,M2)[speed->S2],
 S2 is 2 x S,
 CompPress(M1,M2)[speed->S],
```

This composition code says that the speed element of all components should be the same (represented by variable S) unless for component robot which should has the speed twice as the other.

This constraint could be put as a restriction on the composition or it could only be put as a property to check. The above code is the example of the first case. On the second case, the composition rules are not modified, but those additional codes are defined as separated predicates which will answer "yes" or "no" related whether the constraint is fulfilled or not.

4 Conclusion

The F-Logic paradigm has been shown sufficient to mechanize the invasive software composition and to conduct additional checking and reasoning on the composition in a natural way. Those checking and reasoning are conducted in the same abstraction but language independent which make it a lot easier compare to Compost[2]. Compare to other approaches based on Prolog such as in [9, 3], the Object Oriented style of F-Logic provides a better abstraction. F-Logic is a good choice to do component software composition especially in the invasive software composition where components can to be transformed by F-Logic rules. The F-Logic partiality provides a separation of view to the system. It is possible to see the component on different view, because the component can be represented by combination of F-Molecule where each F-Molecule may represent each own view to the system or to the component. This situation serves well on providing aspect separation and code weaving modeling.

Acknowledgments: This research was conducted in Technical University of Dresden, Germany, within the AsiaLink Project –grant No. VN/ASIA-LINK/001 (79754). It is also partially funded by Hibah-B Fasilkom UI 2005. Author also thank to Prof. Uwe Assmann for the supervision and ideas.

References:

[1] Franz Achermann. Piccola white paper. Working paper, IAM, University of Bern, 1999.

[2] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag, 2003.

[3] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In *1st Conf. Generative Programming and Component Engineering*, volume 2487 of *lncs*, pages 110–127, Berlin, 2002. Springer-Verlag.

[4] M. Broy. Multi-view modelling of software sytems. In Hung Dang Van and Zhiming Liu, editors, *Proceedings of the Workshop on Formal Aspects of Component Software (FACS)*, 2003.

[5] Ivica Crnkovic and Magnus Larsson. Challenges of component-based development. *Journal of Software Systems*, December 2001.

[6] Laurent Dami. Functions, records and compatibility in the lambda N calculus. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 153–174. Prentice-Hall, 1995.

[7] Juliana KÃster Filipe. A logic-based formalization for component specification. *Object Technology*, 1(3):231–248, 2002.

[8] Holger Giese. Contract-based component system design. In Jr. Ralph H. Sprague, editor, *Thirty-Third Annual Hawaii International Conference on System Sciences (HICSS-33), Maui, Hawaii, USA (J. Ralph H. Sprague, ed.),*. IEEE Computer Press, jan 2000.

[9] Elnar Hajiyev, Neil Ongkingco, Pavel Avgustinov, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Datalog as a pointcut language in aspect-oriented programming. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 667–668, New York, NY, USA, 2006. ACM Press.

[10] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.

[11] Markus Lumpe. Gloo: A framework for modeling and reasoning about component-oriented language abstractions. In *CBSE*, pages 17–32, 2006.

[12] I. S. W. B. Prasetya, Tanja E. J. Vos, A. Azurat, and S. Doaitse Swierstra. A unity-based framework towards component based systems. In Teruo Higashino, editor, *Revised Selected Papers OPODIS 2004*, volume 3544 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2005.

[13] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, 1998.

[14] Guizhen Yang, Michael Kifer, Chang Zhao, and Vishal Chowdhary. *Flora-2: User Manual*, 0.94 (narumigata) edition, 2005.