Evaluating the Use of ADLs in Component-Based Development

George A. Papadopoulos
Department of Computer Science, University of Cyprus
75 Kallipoleos Street, POB 20537, CY-1678, Nicosia, Cyprus
george@cs.ucy.ac.cy

Abstract. *In this paper we evaluate the use of software architectures in the development of component-based systems. The evaluation is based on the level of support provided by the software architectures formal representatives, namely ADLs, for four established component-based development principles. Specifically, we will describe and evaluate three representative ADLs: ACME, Unicon and Rapide. For each of the above ADLs we give a brief introduction to its purpose, capabilities and special features, and describe the semantics of its main building constructs.*

Keywords. Software Architectures, Architecture Description Languages (ADLs), Component-Based Development.

1. Introduction

In this paper, we will attempt an evaluation of the use of software architectures in the development of component-based systems. The evaluation will be based on the level of support provided by the software architectures formal representatives, namely ADLs, for four established component-based development principles. Specifically, we will describe and evaluate three representative ADLs: ACME, Unicon, and Rapide. For each of the above ADLs we give a brief introduction to its purpose, capabilities and special features, and describe the semantics of its main building constructs. We will also evaluate each ADL's modelling capabilities against the following criteria: encapsulation, concern separation, abstraction and decomposability ([1]).

In order to make the presentation and evaluation of ADLs more clear and understandable, we depict a simple software architecture of a component-based system. The architecture, presented in figures 1 and 2, comprises a part of a bank system's software structure. Figure 1 presents the high level structure of the system; it includes two components, the ATM and the Bank Server component, as well as the interaction among them. Figure 2 forms a decomposition of the Bank Server component to its constituent components. For each ADL, we will describe the structure of the system presented in the two figures, using the notation provided by the language. The evaluation of each ADL will also use the same example to clearly present the support that the ADL offers regarding the component-based principles discussed earlier.

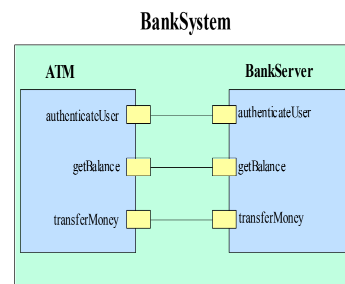

Figure 1. The Bank System Architecture

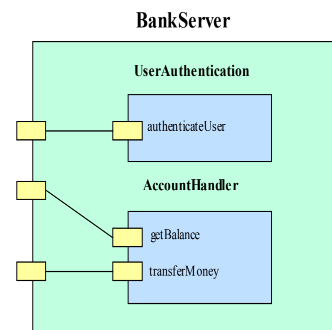

Figure 2. Decomposition of the Bank Server Component

2. ACME

ACME ([2]) is a generic language for describing software architectures. As is the case with any typical ADL, ACME provides constructs for describing systems as graphs of components interacting via connectors. Furthermore, the language provides representation mechanisms for decomposing

systems into subsystems and ways to describe families of components. In particular, the language's core concepts are Systems, Components, Connectors, Ports, Roles, Representations and Rep-maps.

The first three concepts have the usual meaning as in any component-based system. Ports define the interface of a component, identifying a point of interaction between the component and its environment.

Roles specify the interface of a connector. Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as the caller and the callee roles, or the sender and the receiver roles. A different kind of connector is the broadcast connector, which might have a single event-announcer role and an arbitrary number of event-receiver roles.

2.1. Modelling the example in ACME

```
System BankSystem = {
    Component ATM = {
Port authenticateUser,
getBalance, transferMoney;  };
    Component BankServer = {
Port authenticateUser,
getBalance, transferMoney;
        Representation {
          System BankServer = {
               Component
UserAuthentication = { Port
authenticateUser; };
               Component
AccountHandler = {
Port getBalance, transferMoney;};
             Bindings{

   authenticateUser→UserAuthenti
cation.authenticateUser;

   getBalance→UserAuthentication
.getBalance;
             transferMoney→
UserAuthentication.transferMoney;
};
          };
        };
    };
    Connector
authenticateUserConn = { Role
callee, caller;   };
    Connector getBalanceConn = {
Role callee, caller;   };
    Connector transferMoneyConn =
{ Role callee, caller;   };
   Attachments {
       ATM.authenticateUser to
authenticateUserConn.caller;
       ATM.getBalance to
getBalanceConn.caller;
       ATM.transferMoney to
transferMoneyConn.caller;

BankSystem.authenticateUser to
authenticateUserConn.callee;
       BankSystem.getBalance to
getBalanceConn.callee;
       BankSystem.transferMoney
to transferMoneyConn.callee;  };
};
```

2.2. Evaluating ACME

Encapsulation: In the above description the details of each of the Bank System's components are hidden and only their provided and required functions (i.e. `authenticateUser`, `getBalance`, `transferMoney`) are exposed through the input and output ports of each component.

Concern separation: The communication part of the Bank System example which is described by connectors and attachments is clearly separated by the computational part of the system which is encapsulated in the component constructs.

Abstraction: When describing the software architecture of the above system we did not have to consider the algorithms to be implemented by e.g. the `Account Handler` component or even the protocols to be used for the communication of the system's components. This clearly presents the capability of ACME to adjust the level of abstraction according to the current development level.

Decomposability: The decomposition of the Bank Server component to its constituent components (e.g. `User Authentication` and `Account Handler`) though the representation construct, clearly illustrates the support of ACME decomposability.

3. Unicon

Unicon ([4]) attempts to support a large variety of real life applications and make the transition of system design to implementation code smoother. A system architecture described in Unicon consists of a number of components and connectors. Components represent

computational or data units of the system while connectors mediate the communication between components. Each component is associated with an interface and an implementation.

A component's interface defines the computational capabilities of the component, as well as a number of constraints on the way the component can be used. An interface must also include the component type, assertions that apply to the component and a number of players exposed by the component. Players are the units through which a component can interact, provide or request services. Their semantics is closely related to ACME ports described earlier. The specification of a player is given in the form of a property list. Each property includes an attribute name and its associated value.

The implementation of a component can be primitive or composite. Primitive implementations are specified in the code of some programming language. Composite implementations enable the building of progressively larger subsystems from components.

Unicon connectors mediate the interaction between components. They include a protocol specifying the type of interactions that are provided by the connector and an implementation.

3.1. Modelling the example in Unicon

```
COMPONENT ATM
    INTERFACE IS TYPE Computation
      PLAYER authenticateUser IS
RPCCall
          SIGNATURE ("char
*","int"; "char *" )
      END authenticateUser
      PLAYER getBalance IS
RPCCall
          SIGNATURE ("char *";
"float")
       END getBalance
      PLAYER transferMoney IS
RPCCall
          SIGNATURE ("char
*","char *", "float"; "float" )

      END transferMoney
    END INTERFACE
    IMPLEMENTATION IS VARIANT
atm_library IN "atm.jar"
       IMPLTYPE is (executable)
    END IMPLEMENTATION
END ATM
```

```
/* Definition of
UserAuthentication and
AccountHandler Components */
/* in the same way as above
*/
COMPONENT UserAuthentication  …
END UserAuthentication
COMPONENT AccountHandler … END
AccountHandler

COMPONENT BankServer
    INTERFACE IS TYPE Computation
      PLAYER authenticateUser IS
RoutineCall
         SIGNATURE ("char
*","int"; "char *" )
      END authenticateUser
      PLAYER getBalance IS
RoutineCall
         SIGNATURE ("char *";
"float")
      END getBalance
      PLAYER transferMoney IS
RoutineCall
         SIGNATURE ("char
*","char *", "float"; "float" )

      END transferMoney
    END INTERFACE
    IMPLEMENTATION IS
        /* Instantiate the parts
to use */
        USES userAuth is
INTERFACE UserAuthentication,
        USES accHandler is
INTERFACE AccountHandler
        USES authenticateConn
PROTOCOL ProcedureCall
        USES getBalanceConn
PROTOCOL ProcedureCall
        USES transferConn
PROTOCOL ProcedureCall
        /* Associate players of
parts to players of interface */
        BIND authenticateUser to
userAuth.authenticateUser
        BIND getBalance to
accHandler.getBalance
        BIND transferMoney to
accHandler.transferMoney
        /* Associate players of
roles */
        CONNECT authenticateUser
TO authenticateConn.caller
        CONNECT
userAuth.authenticateUser TO
authenticateConn.definer
        CONNECT getBalance TO
getBalanceConn.caller
```

```
        CONNECT
accHandler.getBalance TO
getBalanceConn.definer
        CONNECT transferMoney TO
transferConn.caller
        CONNECT
accHandler.transferMoney TO
transferConn.definer
    END IMPLEMENTATION
END BankServer

/* Defintion of BankSystem as a
component type in the */
/* same way as BankServer
Component.
*/
```

3.2. Evaluating Unicon

Encapsulation: Unicon hides the implementation details exposing the computational capabilities and requirements of each component through the `Interface` and `Player` constructs. In the example each component includes an interface constituting of a number of players. Each player represents a function of the component giving its signature, i.e. the number and type of input and output parameters. For example, the signature of the `getBalance` method given in the above example ("char *"; "float") specifies that an input parameter of string type is required which is the account number and a return value of float type is returned which is the current balance of the account.

Concern separation: A superficial examination of the above code may lead to the conclusion that the communication and computational parts of the system are not separated since both are included in the component implementation construct of Unicon. However, separation is still achieved since the computational part is described by component interfaces, players and primitive component implementations in contrast to the communication part that is described through connectors, protocols and bindings included in a composite component implementation. In the above example, primitive component implementations are the ones of ATM, `UserAuthentication` and `AccountHandler` components which include information about the libraries implementing the components and could also include code in some programming language. Composite component implementation is the one given for the Bank Server component which includes component instantiations, connector instantiations and the connections between them. Each connector is attached to a built-in protocol defining the type of communication implemented by the specific connector. Bindings are also included in the component implementation, specifying the links between the internal parts of the component and the players of its interface.

Abstraction: Although a software engineer can use Unicon to give a detailed description of the system including code in specific programming languages, the level of abstraction can be adjusted, i.e. during the first stage of the system description one can only use the component, interface and players constructs in addition to the communication constructs to give a high level description of the system and elaborate this description at a following stage, giving more implementation details.

Decomposability: Decomposability is also supported by Unicon, by first defining the constituent components, e.g. the `UserAuthentication` and `AccountHandler` components that constitute the Bank Server component in the above example and then specify instantiations of these constituent components definitions in the component implementation part of the parent component.

4. Rapide

Rapide ([3]) is an executable event-based ADL, intended to describe but also simulate the behaviour of systems' architectures. Rapide models computations and interactions of a system as partially ordered event sets (or "posets"). An architecture described in Rapide consists of interfaces, connections and constraints.

An interface describes the functionality provided and/or required by a component of the system. The main elements that constitute an interface are actions, functions and behaviour. Actions represent asynchronous "one-way" messages to be sent or received by the interface, while functions represent synchronous communication. Functions and actions can be grouped into Services that can be reused in different interfaces. The behaviour of an interface can be described by an implementation module, by a set of reactive rules or by defining an architecture that implements the interface. A module can be described using elements of

conventional programming languages provided by Rapide, Reactive rules are pieces of code written in a rule-based approach and executed when certain preconditions are satisfied. Preconditions are defined in the form of event patterns

Rapide connections are the assembly units of the language. Unlike some other languages, Rapide connections are active elements associated to a behaviour. They can be seen as special case of reactive rules that when triggered by events of specific patterns, generate another set of events

4.1. Modelling the example in Rapide

```
type ATM is interface
   requires function
getBalance(accNo: String) return
balance;
   requires function
authenticateUser(cardNo: String,
pin:Integer)
  return accNo;
   requires function
authenticateUser(sourceAcc:
String, targetAcc: String,
  amount:Float) return
sourceAccBalance;
behavior ….. end;
/* Definition of
UserAuthentication,
AccountHandler and  */
/* BankServer in the same way as
above
*/

with UserAuthentication,
AccountHandler, BankServer
architecture BankServerArch is
    bankServer: BankServer
    userAuth: UserAuthentication;
    accHandler: AccountHandler;
connect
    ?cardNo, ?accNo, ?sourceAcc,
?targetAcc :String;
    ?pin: Integer;
    ?balance: Float;

bankServer.authenticateUser(?card
No, ?pin) =>
userAuth.
authenticateUser(?cardNo, ?pin);

    bankServer.getBalance(?accNo)
=> accHandler.getBalance(?accNo);
bankServer.transferMoney(?sourceA
cc,?targetAcc,?amount) =>
accHandler.transferMoney(?sourceA
cc,?targetAcc,?amount);
end BankServerArch;

with ATM, BankServerArch
architecture BankSystem is
    atm: ATM;
    bankServer: BankServerArch;
connect
    ?cardNo, ?accNo, ?sourceAcc,
?targetAcc :String;
    ?pin: Integer;
    ?balance: Float;
    atm.authenticateUser(?cardNo,
?pin) =>
bankServer.
authenticateUser(?cardNo, ?pin);

    atm.getBalance(?accNo) =>
bankServer.getBalance(?accNo);

atm.transferMoney(?sourceAcc,?tar
getAcc,?amount) =>
accHandler.bankServer
(?sourceAcc,?targetAcc,?amount);

end BankSystem;
```

4.2. Evaluating Rapide

Encapsulation: The details of each component are encapsulated behind the interface type definition of each component. Each interface type defines the provided and required functionality of the component in the form of function declarations. In the Bank System example the `Account Handler` interface defines the function: "`transferMoney (sourceAcc: String, targetAcc: String, amount:Float) return sourceAccBalance`" which declares three input and one output parameter. The first two input parameters are of string type corresponding to the source and target accounts depicted in the transfer transaction, while the third input parameter is of float type, corresponding to the amount to be transferred. The output parameter is also of float type corresponding to the new balance of the source account after the execution of the transfer transaction.

Concern separation: The communication part of the system is explicitly defined by connections included in the architecture element provided by Rapide notation. The connections take the form

of mapping between functions or actions required by a component and corresponding functions or actions provided by another component. One such mapping in the above connection is the following: "`atm.getBalance(?accNo) => bankServer.getBalance(?accNo);`".

Abstraction: Abstraction is also supported, although the behaviour and posets construct can be used to give a detailed description of a system's computation. As is the case with Unicon, the level of abstraction can be easily adjusted according to the needs of each stage.

Decomposability: The support of Rapide here is very similar to that of Unicon. There is no explicit construct that is used to define the decomposition of a component to its constituent components but decomposability can be defined in the same way as in Unicon, i.e. by defining an interface type for each constituent component and then define instances of these types into the description of the parent's component.

5. Summary and conclusions

The four main principles of component-based development are supported by all ADLs, although this support is not straightforward in all cases. The description of required or provided interfaces exposed by components is modestly supported. The description of interfaces supported by the rest of the ADLs is implemented either by providing specific constructs and syntax such as Rapide and Unicon, or by providing just a framework such as ACME that offers an open semantics framework enabling users to define their own properties to describe different aspects of the system.

Table 1 summarizes the support of the ADLs presented above for the four principles of component-based development. The symbol "√" used in the table indicates clear or explicit support of the specific principle while the "?" symbol indicates implicit or weak support.

We can see that most of the ADLs do not clearly support all of the component-based development principles and this is because most of the ADLs emphasize on specific aspects such as the communication part. On the other hand, ADLs that support all principles, such as ACME, lack in support of dynamic configuration descriptions.

Criteria vs ADLs	ACME	Unicorn	Rapide
Encapsulation	√	√	√
Concern Separation	√	?	√
Abstraction	√	√	?
Decomposability	√	?	?

Table 1. Support of ADLs for component-based development principles

6. References

[1] I. Crnkovic and M. Larsson, "Challenges of Component-based Development", *Journal of Software Systems*, Vol. 61 (3), 2001, pp. 201-212.

[2] D. Garlan, R. T. Monroe and D. Wile, "ACME: An Architectural Description of Component Based Systems", *Foundations of Component-Based Systems*, Cambridge University Press, 2000, pp. 47-68.

[3] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan and W. Mann, "Specification and Analysis of System Architecture Using Rapide,", *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, Vol. 21(4), 1995, pp. 336-355.

[4] M. Shaw, R. DeLine, D. Klein, T. L. Ross, D. M. Young and G. Zelesnik, "Abstractions for software architecture and tools to support them", *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, Vol. 21(4), 1995, pp. 314-335.