Automatic Code Generation: A Practical Approach

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. *This work contributes in bridging the gap between software design and implementation of component-based systems using software architectures at the modelling/design level and the coordination paradigm at the implementation level. Exploiting the improvements realized by the latest version of UML, we present a methodology for automating the transition from software architecture design of component-based systems described in UML 2.0 to coordination code. The presented methodology is further enhanced with a code generation tool that fully automates the production of the complete code implementing the coordination-communication part of software systems modelled with UML 2.0.*

Keywords: Software Architectures, Coordination Models and Languages, UML 2.0, Code Generation.

1. Introduction

Our effort is focused on bridging the gap between software design and implementation of component-based systems using software architectures at the modelling/design level and the coordination paradigm at the implementation level. Our choice was based on the clear support of both software architectures and event-driven coordination models for Component Based Software Engineering and the similarities we have identified between the fundamental concepts of software architectures and the event-driven coordination model.

In [6] we have presented a methodology for mapping ACME ([2]), a generic language for describing software architectures, down to event-driven coordination code in the Manifold ([1, 7]) language. The reason for using ACME was precisely in order to show the generality of our approach: since ACME embodies the core features that any state-of-the-art Architecture Description Language (ADL) would support, by mapping ACME to Manifold we effectively provide the core of an implementation route for any other ADL Based on the results and experience of our first work and exploiting the improvements realized by the latest version of UML towards the support of software architecture descriptions, we propose a new methodology for modelling the software architecture of a component based system in UML 2.0 ([9]) and the automatic transition of this model to event-driven coordination code in Manifold. Our latest work targets an improved support for the dynamic aspects of the software architecture exploiting the powerful tools of UML for dynamic behaviour. Furthermore, we use the standards (UML2.0, XMI) and approach proposed by the new software development discipline, namely the Model Driven Architectures ([5]).

The presented methodology is further supported by a code generation tool that fully automates the production of the complete code implementing the coordination-communication part of software systems modelled with UML 2.0. The fact that our approach integrates software architectures and coordination models enables us to derive the advantages that both models provide in reducing the costs of software development. The modelling of the system architecture enables developers to define the more important properties and constraints of the system, but also to detect errors early at the design time. The generated code, which is consistent with the previously modelled architecture, clearly separates the communication from coordination parts of the system, making the system maintenance easier

1.1. Event-driven coordination

In general, coordination models and languages adhere in two main approaches, the "data-driven" or shared dataspace approach and "control" or "event-driven" approach. The main characteristic of the first approach is the use of a notionally shared medium via which the processes forming a computation communicate.

The most notable realization of this approach is Linda. In contrast to data-driven approach, in event-driven approach processes communicate in a point-to-point manner by means of well-defined interfaces. Such a system evolves dynamically by means of raising and receiving control events. Manifold is a typical member of this family, and is a realization of the Ideal Worker Ideal Manager (IWIM) coordination model ([1]). In Manifold, there exist two different types of processes: *managers* (or *coordinators*) and *workers*. A manager is responsible for setting up and taking care of the communication needs of the group of worker processes it controls (non-exclusively). A worker on the other hand is completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. Manager processes are written in Manifold whereas worker processes may be written also in Manifold or in some computational language (typically C, Fortran). In this latest case, these worker processes are called *atomics*. In particular, Manifold possesses the following characteristics:

- *Processes*. A process is a *black box* with well-defined *ports* of connection through which it exchanges *units* of information with the rest of the world. A process can be either a manager (coordinator) process or a worker. A manager process is responsible for setting up and managing the computation performed by a group of workers. Note that worker processes can themselves be managers of subgroups of other processes and that more than one manager can coordinate a worker's activities as a member of different subgroups. The bottom line in this hierarchy is atomic processes, which may in fact be written, in any programming language.
- *Ports*. These are named openings in the boundary walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (*input* port) or out of (*output* port) a process. We use the notation `p.i` to refer to the port `i` of a process `p`.
- *Streams* or *channels*. These are the means by which interconnections between the ports of processes are realised. A stream connects a producer process to a consumer process. We write `p.o -> q.i` to denote a stream connecting the port `o` of a producer process `p` to the port `i` of a consumer process `q`.
- *Events*. Independent of channels, there is also an event mechanism for information exchange. Events are broadcast by their sources in the environment, yielding *event occurrences*. In principle, any process in the environment can pick up a broadcast event; in practice though, usually only a subset of the potential receivers is interested in an event occurrence. We write `e.p` to refer to the event `e` raised by a source `p`.

Activity in a Manifold configuration is *event driven*. A coordinator process waits to observe an occurrence of some specific event (usually raised by a worker process it coordinates) which triggers it to enter a certain *state* and perform some actions. These actions typically consist of setting up or breaking off connections of ports and channels. It then remains in that state until it observes the occurrence of some other event, which causes the *preemption* of the current state in favour of a new one corresponding to that event. Once an event has been raised, its source generally continues with its activities, while the event occurrence propagates through the environment independently and is observed (if at all) by the other processes according to each observer's own sense of priorities. The figure below shows diagrammatically the infrastructure of a Manifold process.

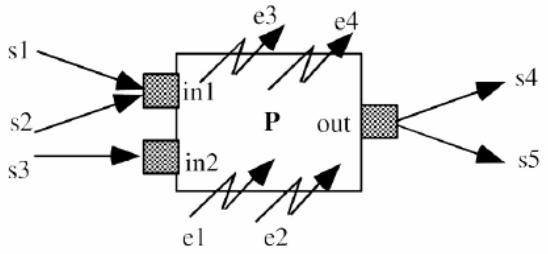

Figure 1. A Manifold process

2. The basic rules of code generation

The general steps for the construction of the diagrams are the following ones:

1. Identify the top-level components of the system architecture. Create a top level diagram and add a special Main component. (This will represent the special manifold process that every system in manifold should include). Add the top-level components of the system as sub-components of the Main Component.

2. For each component identify the different operations that are provided by this component.
3. For each operation identify the different parameters that the component needs to execute this operation and the possible values returned by this operation. Create an interface for each operation and add the specific operation with its parameters and return values.
4. Identify the possible signals sent by the component providing this operation to its environment in response to a call on this function. Add the signals to the created interface.
5. Identify possible main variables related to the operation that can be identified at this stage and may affect the setup of the architecture. Add these attributes to the created interface.
6. Identify the required operations and create an interface for each of them in a similar way as above. For each required interface add a signal sent by the component requesting the call of the related operation to its parent component that coordinates it in order to create the needed setups (connections).
7. For each component add a port for each provided or required operation of the component and attach it to the corresponding required or provided interface.
8. Identify all possible connections between the sub-components of a first level component.
9. Identify all possible connections from the top level component to its parts (sub-components, classes).
10. Decompose each of the sub-components to another diagram. Add in the new diagram the specific sub-component as the top level component and add all sub-components and classes that this component is composed of.

Figure 2 presents the top level architecture model of a small part of a bank system where an ATM component sends the requests accepted from users to the central bank server, realized by the BankServer component, for displaying the balance of its account.

2.1. From architecture to Manifold code

A Component can be exactly mapped to a Manifold coordinator process. An Active Class is mapped to a Manifold atomic process. Passive Classes will be used in our architecture modelling to represent the different data types supported by Manifold such as string, integer, tuple, etc.

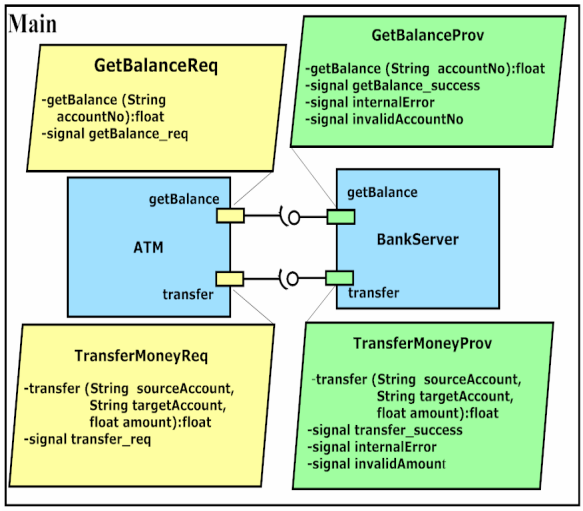

Figure 2. ATM Example – Top level architecture diagram

An interface is not directly mapped to a specific Manifold construct but the set of operations, attributes and signals defined for the specific interface are separately mapped. For every operation that is defined in a provided or required interface attached to a port, we create an input port for each input parameter and an output port if the specific operation returns a value. A special input control port is also created for each operation and a guard is installed on this port to notify the owning manifold process for requests received for the specific operation. The set of signals defined in provided and required interfaces attached to the ports of a component or class are defined to be the events that can be raised by the corresponding manifold or atomic process. Attributes of an interface attached to a component or class are mapped to local variables of the corresponding Manifold coordinator or atomic processes.

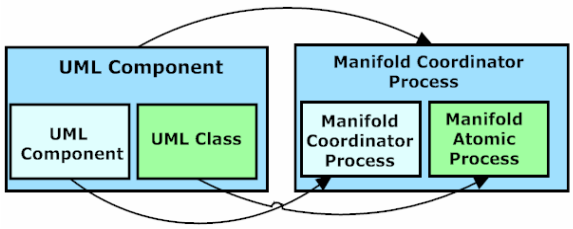

Figure 3. Mapping components and classes

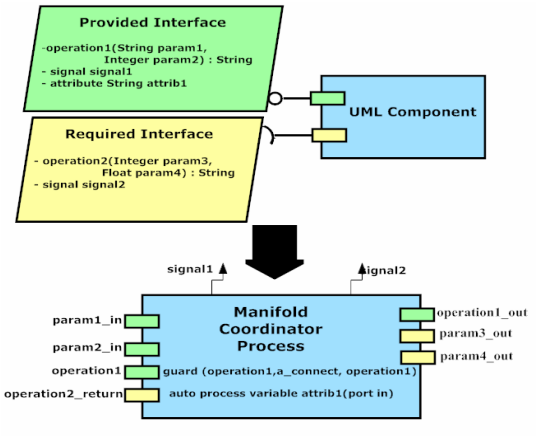

Figure 4. Mapping ports and interfaces

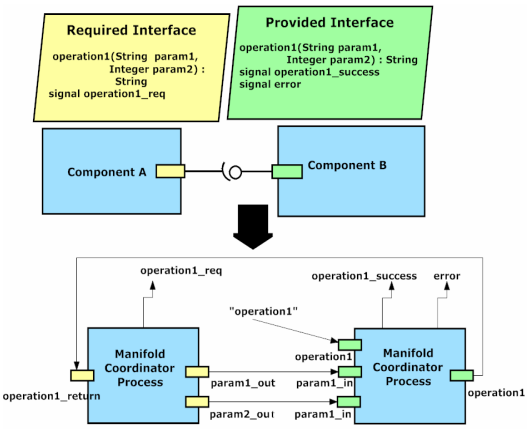

Figure 5. Mapping assembly connectors

2.2. Creating the scenario model

Scenario modelling is realized by a number of sequence diagrams describing the dynamic aspects of a component-based system's architecture, i.e. the:

- interactions taking place between components, realized by message exchanges,
- activation/deactivation of component and class instances,
- conditions under which the above actions take place,
- sequence within which the above actions take place.

As soon as the different scenarios are identified, the software architect can create in a hierarchical top-down approach the sequence diagrams of each scenario as follows:

1. Create a top level sequence diagram and include a lifeline for the "Main" component and a lifeline for each instance of the first level components/classes that are involved in the execution of the first execution scenario.
2. Use the constructs for scenario modelling described above to define the interactions – messages taken place during the execution of the first execution scenario.
3. Decompose every decomposable lifeline to another sequence diagram, describing the message exchanges taking place for the current scenario at a lower lever (i.e. between the specific component and its part's instances).
4. Add all "signal" and incoming "operation" call messages of the higher level sequence diagram that are attached to the lifeline currently being decomposed.
5. Between the already created messages, add all message exchanges taking place between the decomposed lifeline (i.e. the parent component) and the other lifelines.
6. For each component, create a new sequence diagram with a special name "Component name - Init" in order to describe the initialization process of the component such as the creation of process instances.

For our example, the "getBalance" and "transfer" scenario can be identified. The sequence diagrams for the "transfer" scenario as well as the special "init" diagram for the "Main" component are presented below:

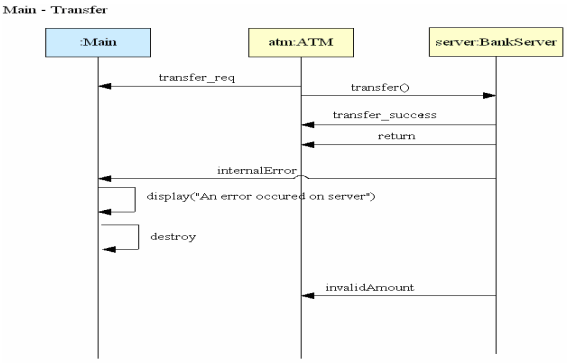

Figure 6. First level sequence diagrams

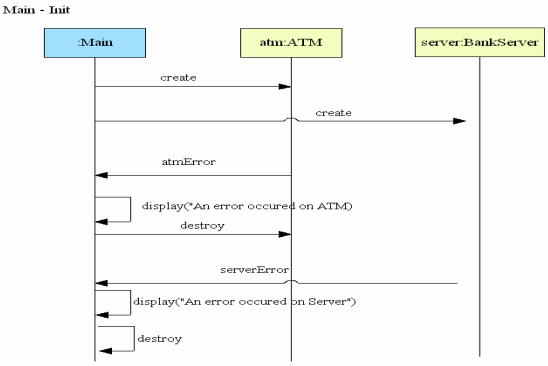

Figure 7. Special "init" sequence diagram for the "Main" component

3. Code generation

We have developed a tool that automatically generates the Manifold code implementing the coordination-communication part of software architectures modelled with UML 2.0. Our code generation tool takes as input an XMI document describing the architecture model of a system and outputs the full Manifold code implementing the coordination part of the system. The full route of creating and transforming a software architecture model to Manifold code is shown below:

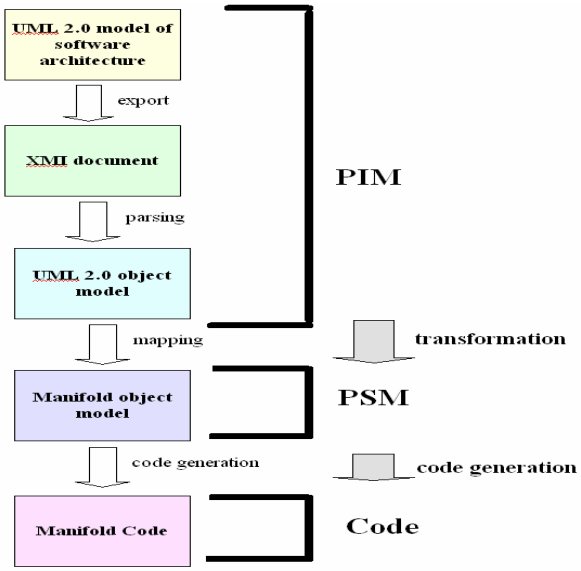

Figure 8. Code generation tool – Transformation/code generation route. Following the MDA approach, we first create a Platform Independent Model (PIM) in UML2.0, then we apply our mapping rules to create a Platform Specific Model (PSM) and finally we create the coordination code by applying our code generation rules on PSM. In our case "platform" is the specific event-driven coordination language, e.g. Manifold.

The creation of the software architecture of the system forms the first step. For the modelling of the software architecture, we use the Sparx Enterprise Architect modelling tool ([8]). Using the "export" function of Enterprise Architect, we then export the modelled software architecture to an XMI (v.1.1) document.

Since the latest version of XMI (v.2.1) that corresponds to UML 2.0 has recently been released, the few tools that provided support of UML 2.0 after its official release on 2003 have used previous versions of XMI format to export the models and added custom extensions to cover the needs not supported by these versions. Additionally, since XMI has to be general enough to represent not only UML models but every kind of model, there are specific needs of UML tools that may not be supported. As it is stated in [4] "the XMI standard itself doesn't support all that is needed, and vendors unfortunately implement it differently". In order to make our code generation tool more independent from specific UML modelling tools we first parse XMI generated by Enterprise Architect and create an intermediate, tool independent, representation of the model. The intermediate representation consists of generic UML 2.0 Java classes that represent the elements of our software architecture model.

For parsing the XMI document and creating the UML 2.0 object model, we use Apache Commons Digester ([3]). Having an intermediate representation of the software architecture enables the support for additional modelling tools in the future with minimum effort. If we wanted to add support for a modelling tool other than Enterprise Architect that has a different implementation of XMI format, then we would only have to add another set of digester rules for parsing the XMI document exported by this tool (or just the rules for parsing the XMI parts that are implemented differently in this tool) and transform it to the common UML2.0 object model.

The next step is the transformation of the UML2.0 object model to the equivalent Manifold object model by applying the mapping rules of our methodology. The Manifold object instances are finally processed to generate the Manifold code by applying the syntax rules of Manifold. Part of the code generated by our tool for the "Main" manifold (`Main.m` file) is presented below:

```
manifold ATM(event
getBalance_req,
event transfer_req ) import.

manifold BankServer(
  event getBalance_success,
 port in getBalance. import.

manifold Main(){
 begin:
(activate(atm),activate(server)
).
atmError.atm:
    "An error occurred on ATM" ->
out, deactivate(atm).
serverError.server:
    "An error occurred on Server"
-> out; halt.
```

```
//transfer scenario
transfer_req: "transfer"-
>server.transfer,
  atm.sourceAccount_out ->

server.sourceAccount_in,
  atm.targetAccount_out ->

server.targetAccount_in.
  atm.amount_out →
server.amount_in.

  "An error occurred on Server"→
out;
   halt.  }
```

4. Discussion and further work

Some advantages of this work are the following ones:

- The use of a standard, broadly accepted and established modelling language for describing software architectures.
- The two types of diagrams that are used in our methodology can be perfectly interrelated, thanks to the new feature of UML 2.0 for structure and behaviour gross integration.
- By virtue of XMI, the software architecture descriptions can be exchanged and used/edited by many modelling tools.
- Adhering to the main principles of the MDA approach, we tried to keep the software architecture model constructed by our methodology "platform" independent.

The software developer that will use our methodology and the associated code generation tool will face a common, in the field of automatic code generation, problem: the maintenance of the generated code. Although in our latest methodology the coordination code that can be generated is more complete limiting the need for the programmer to manually add missing bits of coordination code, if the software architecture of the system changes in a subsequent stage (e.g. the system is extended with new functionality and subsequently new components) the code has to be generated again. However, the problem is limited to the atomics files that the tool generates for the coordination-related code and where the programmer manually adds the computational code.

Our future work involves the enhancement of our code generation tool by:

- addressing the problem of code maintenance; we are currently in the process of considering code-block recognition methods used in other code generation tools,
- supporting additional modelling tools apart from Sparx Enterprise Architect,
- adding enhanced mechanisms for consistency checking and validation of the imported software architecture model.

5. References

[1] F. Arbab, I. Herman and P. Spilling, "An Overview of Manifold and its Implementation", *Concurrency: Practice and Experience* 5 (1), 1993, pp. 23-70.

[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, pp. 47-68, 2000.

[3] Jakarta Commons Digester Website, http://jakarta.apache.org/commons/digester, accessed Feb 2006.

[4] C. Laird , XMI and UML combine to drive product development, IBM Whitepapers, available at http://www-128.ibm.com/developerworks/xml/library/x-xmi/, October 2001.

[5] OMG Model Driven Architecture Website, http://www.omg.org/mda/, accessed Feb 2006.

[6] G. A. Papadopoulos, A. Stavrou, and O. Papapetrou, "An implementation framework for Software Architectures based on the coordination paradigm", *Science of Computer Programming* 60(1): 27-67 (2006).

[7] G. A. Papadopoulos and F. Arbab, "Configuration and dynamic reconfiguration of components using the coordination paradigm", *Future Generation Computer Systems* 17 (8) (2001) 1023-1038.

[8] Sparx Systems Website, available at http://www.sparxsystems.com.au/, accessed Feb 2006.

[9] OMG, Unified Modeling Language: Superstructure version2.0, August 2003.