Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 60 (2006) 27–67
Science of Computer Programming
www.elsevier.com/locate/scico

An implementation framework for Software Architectures based on the coordination paradigm

George A. Papadopoulos*, Aristos Stavrou, Odysseas Papapetrou

Department of Computer Science, University of Cyprus, 75 Kallipoleos Street, P.O.B. 20537, CY-1678 Nicosia, Cyprus

Received 3 May 2004; received in revised form 6 April 2005; accepted 4 June 2005
Available online 24 August 2005

Abstract

Software Architectures have evolved considerably over the last decade and, partly also due to the significant progress made in component-based development, have become a major subfield of Software Engineering. The associated field of Architecture Description Languages (ADLs) has also evolved considerably, providing numerous approaches to the formal specification and representation of architectural designs. In this field, one of its most interesting (and rather recent) aspects has been the exploration of different ways to map architectural specifications down to executable representations. In this paper, we present a methodology for mapping the generic features of any typical ADL to executable code. The mapping process involves the use of ACME, a generic language for describing software architectures, and the coordination paradigm. More to the point, we show how the core concepts of ACME can be mapped to equivalent executable code written in the coordination language Manifold. The result is the generation of skeletal code which captures and implements the most important system implementation properties of the translated architectural design, thus significantly assisting the programmer in filling in the rest of the needed code.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Coordination languages and models; Software Architectures; Architecture Description Languages; Code generation

* Corresponding author. Tel.: +357 22 892693; fax: +357 22 892701.
E-mail addresses: george@cs.ucy.ac.cy (G.A. Papadopoulos), cs98sa2@cs.ucy.ac.cy (A. Stavrou), cspapap@cs.ucy.ac.cy (O. Papapetrou).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.06.002

1. Introduction

Software architectures [8] have evolved considerably over the last decade and, partly also due to the significant progress made in component-based development, have become a major subfield of Software Engineering. Software architectures are nowadays playing an important role in software development, not least because of the increasing complexity of software systems, covering both functional and nonfunctional aspects.

Tightly coupled with the progress in Software Architectures is the development of Architecture Description Languages (ADLs). ADLs [15] provide numerous approaches to the formal specification and representation of architectural designs. An ADL is typically associated with a conceptual framework but also offers a concrete syntax (textual and/or visual) for expressing the properties of a software architecture.

It is widely accepted that software architectures (and their associated ADLs) play a major role in the development of a system, acting as a bridge between the requirements that the planned architecture must satisfy and the actual implementation code [8]. Consequently, the existence of (semi-) automated processes that will generate executable code from an ADL-based specification is of paramount importance to reducing implementation time. In [17] it is stated that the generation of even skeletal code constructs could have significant positive impact to the overall development of a system. However, only a few contemporary ADLs are able to generate executable code, and even in these cases, the programmer must accept the preferred programming language that the ADL chooses to generate code in.

In this paper, we present a methodology for mapping architectural representations written in ACME [9,10], a generic language for describing software architectures, down to executable code. The mapping process involves the use of the coordination paradigm. More to the point, we exploit the fact that the core concepts of many ADLs are very similar to those of a particular class of coordination models and languages, namely the so-called "control-driven" family [21]. In particular, we show how the core concepts of a typical ADL, as expressed by ACME, can be mapped to equivalent executable code written in the coordination language Manifold. The result is the generation of skeletal code, which captures and implements the most important properties of the translated architectural design, thus significantly assisting the programmer in filling in the rest of the needed code. By virtue of the Manifold model, the skeletal code is effectively independent of any particular programming language; this is another advantage of the presented approach. Furthermore, the involvement of the coordination paradigm as an implementation route for architectural specifications produces executable representations where the computational components are clearly separated from the coordination ones, thus facilitating component reuse. Finally, the proposed framework also offers the coordination programmers the opportunity to use the ADL paradigm for modeling and designing the high-level aspects of a system before embarking on its actual implementation using a coordination language.

The rest of the paper is organized as follows. In the next two sections we present the coordination language Manifold and the ADL ACME; in the process we show that the family of coordination languages that Manifold is a member of shares many common features with ADLs. This fact is exploited in the fourth section, where we first present the mapping of the core concepts of ACME onto equivalent executable skeletal code in Manifold and then we use these basic mappings to illustrate how a fully-fledged software architecture can be implemented in a combination of Manifold code and code produced in some programming language. In the fifth section, we describe the way we have implemented a code generation tool based on our methodology. The paper ends with some conclusions where we dwell further into the issues raised in the previous paragraph, and reference related and further work.

2. Control-driven coordination and Manifold

The concept of coordinating a number of activities, possibly created independently from each other, such that they can run concurrently in a parallel and/or distributed fashion has received wide attention, and a number of coordination models and associated languages [21] have been developed for many application areas such as high-performance computing or distributed systems. All these models share some common purpose and aim at providing frameworks to offer suitable abstractions allowing a programmer, who is not necessarily proficient in parallel programming, to write programs that can run with reasonable efficiency on parallel and/or distributed architectures. However, they also support modularity, reuse of existing (sequential) software components, language interoperability, portability, etc.

In general, coordination models and languages fall into two main categories [21]. The first one we can call a “data-driven” or shared dataspace approach. Its main characteristic is the use of a notionally shared medium via which the processes forming a computation communicate. The most notable realization of this approach is of course Linda [3]. Linda introduces the so-called notion of *decoupled communication* whereby processes communicate with each other by either inserting into or retrieving from the shared medium the data to be exchanged between them. This shared dataspace is referred to as the *Tuple Space* and the information exchange between processes via the Tuple Space is performed by posting and retrieving *tuples.*

A different philosophy to developing coordination models and languages was also proposed, based on a “control-” or “event-driven” approach [21]. Contrary to what is happening in the shared dataspace approach to coordination, here 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. The coordinated components do not necessarily examine the data that is being transmitted through these point-to-point connections and therefore these components can be viewed as black boxes. The way software architectures are modeled in control- or event-driven coordination languages, with components acting as black boxes and point-to-point communication among them being realized by means of streams connecting input/output ports, resembles the ADL world where connectors set up communication paths among components. In [21] we argue that, in fact, many Software Architecture models, Architecture Description Languages (ADLs), Dynamic (Re-) Configuration Languages, etc. can be seen as instances of a general framework of control-driven coordination models and associated languages.

Manifold [20,22,19] in particular, is a typical member of this family, and is a realization of the Ideal Worker Ideal Manager (IWIM) coordination model [5]. 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. 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 or even as shell scripts in Unix-like operating systems.
- *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 instance `p`.
- *Streams*. These are the means by which interconnections between the ports of processes are realized. A stream connects a (port of a) producer (process) to a (port of 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 say that these processes are *tuned in* to the sources of the events they receive. 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) that 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 favor 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. Fig. 1 shows diagrammatically the infrastructure of a Manifold process.

The process `p` has two input ports (`in1`, `in2`) and an output one (`out`). Two input streams (`s1`, `s2`) are connected to `in1` and another one (`s3`) to `in2` delivering input data to `p`. Furthermore, `p` itself produces data, which via the `out` port are replicated to all outgoing streams (`s4`, `s5`). Finally, `p` observes the occurrence of the events `e1` and `e2` while it can itself raise the events `e3` and `e4`. Note that `p` need not know anything else about the

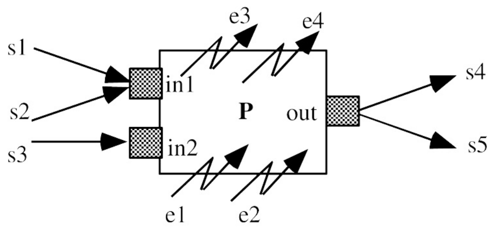

Fig. 1. The infrastructure of a Manifold process.

```
manifold PrintUnits() import.
manifold variable(port in) import.
manifold sum(event)
  port in x.
  port in y.
  import.
event overflow.

auto process v0 is variable(0).
auto process v1 is variable(1).
auto process print is PrintUnits.
auto process sigma is sum(overflow).

manifold Main()
{
 begin:(v0->sigma.x, v1->sigma.y,v1->v0,sigma->v1,sigma->print).
 overflow.sigma:halt.
}
```

Fig. 2. Manifold version computing the Fibonacci series.

environment within which it functions (i.e. who is sending it data, to whom it itself sends data, etc.).

For illustrative purposes, but also for the reader to be able to follow and appreciate the work presented later on, Fig. 2 shows the Manifold version of a program computing the Fibonacci series.

The code in Fig. 2 defines `sigma` as an instance of some predefined process `sum` with two input ports (`x`,`y`) and a default output one. The main part of the program sets up the network where the initial values (`0`,`1`) are fed into the network by means of two "variables" (`v0`,`v1`). The continuous generation of the series is realized by feeding the output of `sigma` back to itself via `v0` and `v1`. Note that in Manifold there are no variables (or constants for that matter) as such. A Manifold variable is a rather simple process that forwards whatever input it receives via its input port to all streams connected to its output port. A variable "assignment" is realized by feeding the contents of an output port into the input of a variable process. Note also that computation will end when the event `overflow` is raised by `sigma`. `Main` will then get preempted from its `begin` state and make a transition to the `overflow` state and subsequently terminate by executing `halt`. Preemption of `Main` from its `begin` state causes the breaking of the stream connections; the processes involved in the

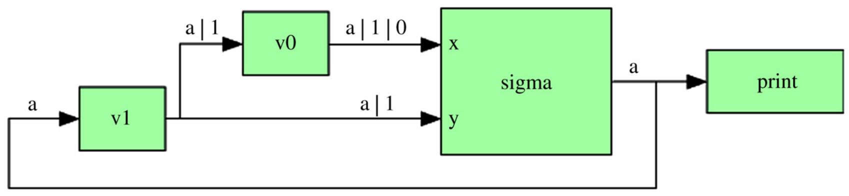

Fig. 3. The setup for a Fibonacci program in Manifold.

network will then detect the breaking of their incoming streams and will also terminate. Fig. 3 shows the process configuration that will be set up by executing the code presented in Fig. 2.

In this example note that both the computational process `sigma` (an instance of `Sum`) and the coordination process `Main`, treat each other as black boxes: `Main` is only concerned with the input/output dependencies of `sigma`, and `sigma` operates completely unaware of `Main`'s doings. Note also that the actual data being produced and transmitted between the components of this apparatus (namely the Fibonacci numbers) do not play any role in the setup. Therefore, both `Main` and `sigma` are black boxes: `Main` is an Ideal Manager that can coordinate any computational process `sigma` (assuming the coordination pattern is reusable and applicable in other cases), and `sigma` is an Ideal Worker that will compute according to its specification without any knowledge of its surrounding environment.

The computational component of the Fibonacci program, namely `Sum`, can in fact be written in any "Manifold-compliant" programming language (i.e. a language whose compiler has been enhanced with additional constructs making it possible to express at the level of this language the core Manifold concepts, i.e. ports, streams, events, etc.). Fig. 4 shows the implementation of `Sum` in Manifold-compliant C. Some explanatory comments are included, not only to help the reader understand the basic functionality of the code but also to clarify further the transformations introduced in the fourth section. In general, note that all entities starting with the `AP_` prefix are C extensions dealing with some core concept of the IWIM model and the language Manifold. Such software components, that are written in an ordinary programming language and represent the computational (as opposed to the coordination) parts of an application, are called *atomics*.

What the Manifold-compliant C code in Fig. 4 essentially does is to first define `sigma`'s three ports (two input ones and one output one), then retrieve from the input ports two integers, add them up and place the result in the output port. The whole process is repeated until the computed sum eventually exceeds the value `MAXINT`, in which case `sigma` raises the event `too_big` and terminates. The raising of `too_big` will be detected by the monitoring `Main` Manifold which then terminates execution (see code for `Main` earlier on).

An application typically consists of a number of atomics performing computations and being coordinated by one or more coordinators (manifolds). Atomics are distinguished as *atomic internals* or *atomic externals* according to the way that they are implemented. Atomic internals are written in Manifold-compliant (and otherwise ordinary) programming languages. Atomic externals are written as scripts of operating systems supported by Manifold. The ability to write atomics as operating system scripts offers great flexibility to

```
#include "AP_interface.h"
#include "fibo.ato.h"
#include "adid.h"

#define MAXINT 1000

void Sum(AP_Event too_big)
{
  int x = AP_PortIndex("x");             /* declare an index for */
  int y = AP_PortIndex("y");             /* every i/o port */
  int out = AP_PortIndex("output");
  int i1, i2, i3;         / declare local variables for addition */
 /* declare units for passing values to the i/o ports */
  AP_Unit u1, u2, u3;

  while (1) {
    AP_PortRemoveUnit(x, &u1, NULL); /* get data units from input */
    AP_PortRemoveUnit(y, &u2, NULL); /* ports 'x' and 'y' */
    AP_FetchInteger(u1, &i1);        /* and allocate them to */
    AP_DeallocateUnit(u1);           /* integer variables i1 & i2 */
    AP_FetchInteger(u2, &i2);
    AP_DeallocateUnit(u2);
  }

    i3 = i1 + i2;             /* calculate the sum */

    if (i3 > MAXINT) {        /* if max. Fib. Number reached */
      AP_Raise(too_big);      /* raise event overflow */
      return;
    }

    u3 = AP_FrameInteger(i3);
    AP_PortPlaceUnit(out, u3, NULL); /* put result in the */
    AP_DeallocateUnit(u3);          /* output port */
}
```

Fig. 4. Implementation of Sum in Manifold-compliant C.

the development of component-based systems using Manifold, since any operating system process can be included in the system as a black-box component.

3. Software Architectures and Architecture Description Languages

Software Architectures can be informally defined as software system blueprints, or, more formally, models that represent the design of the system at a high level of abstraction. Software Architectures manage to expose the system's gross organization as a collection of interacting components. Building a software architecture for a system promotes its understanding, thus aiding the design process. Software Architectures make the early detection of design errors possible, thus leading to improvements in software quality and correctness.

The main building blocks of software architectures are components, ports and connectors. Components are used to represent large parts of code with specific and distinct functionalities (not necessarily objects conforming to the component paradigm). Ports offer a way of communication between components and they are the only well-defined interface of each component with the rest of the system. We can have two kinds of ports, in-ports, and out-ports, the former getting information from the environment inside the components, and the latter delivering information from the component to the environment. Finally, connectors are used as pipelines, joining the components, eventually describing their collaboration and the flow of data and control.

Architecture Description Languages (ADLs) formalize a software architecture by offering a simple, yet flexible, representation. An ADL may comprise a formal or semi-formal descriptive textual language, a graphical language, or both. A state-of-the-art survey on ADLs is presented in [15].

3.1. The case of ACME

ACME [9,10] 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 as follows:

- *Components*: They represent the primary computational elements and data stores of a system. Typical examples of components include clients, servers, filters, blackboards and databases.
- *Ports*: They define the interface of a component. Each port identifies a point of interaction between the component and its environment.
- *Connectors*: They represent interactions among components. They mediate the communication and coordination activities among components. They can represent simple forms of interaction, such as pipes, procedure calls and event broadcast, or complex interactions such as clients–server protocols or SQL links between a database and an application.
- *Roles*: A set of roles specifies 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, the reading and writing 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.
- *Systems*: They represent configurations of components and connectors. A system includes a set of components and a set of attachments. Attachments define the relationship between a connector and a component.
- *Representations*: A component or connector can be described by one or more detailed, lower-level descriptions. These descriptions are called representations. Representations give ACME the opportunity to support hierarchical description of architectures.
- *Rep-map*: A rep-map defines the correspondence between an internal system representation and the external interface of the component or connector that is being represented.

Apart from these core concepts, ACME provides an open semantic framework in which architectural structures can be annotated in the form of properties. ACME itself does not provide the meaning of these properties. Properties become useful only when a tool makes use of them for analysis, translation and manipulation [10].

4. A methodology for mapping ACME to Manifold

ACME, being a generic architecture description language rather than a typical ADL with specific semantics, does not provide a single fixed way to describe the behavior of a system or its functional properties; nor does it generate executable code for any part of the modeled system. Instead, elements of the language representing parts of the architecture may be annotated with descriptive information dealing with, say, implementation aspects; it is then the responsibility of other tools, or even the programmer, to translate this information to some sort of executable representation.

It is precisely here that the paper offers a contribution, namely the development of a methodology for mapping the core concepts of ACME onto executable Manifold code. Other researchers have concentrated on the major role of ACME as an interchange language to map features of one ADL onto another with the purpose of integrating tools written for different ADLs; we, instead, aim at using the language's generic features and open semantic framework to derive an implementation route that will map ACME's core concepts (themselves part of the structure of any typical ADL) down to Manifold executable code.

Our methodology is associated with a tool for parsing the ACME code describing the software architecture of a system and automatically generating Manifold code based on the transformation rules proposed by our methodology. Although the Manifold code that is automatically generated covers most of the coordination aspects of a system, some coordination code parts implementing dynamic aspects are not generated due to ACME's limitation in adequately describing such behavior. This type of code has to be added manually by the programmer, along with the code that implements the actual computation parts of the system, the latter being outside the scope of this paper. The parts of code not automatically generated are clearly shown in the code examples included in our hospital case study (see Section 4.3).

The rest of this section, comprising the main contribution of the paper, consists of three parts. We first show how the basic building blocks of ACME can be mapped to Manifold structures. Then we show our approach towards extending ACME with some coordination-programming semantics in order to perform a more efficient representation. Finally, we present a fully-fledged example, which illustrates our mapping methodology.

4.1. Mapping the core concepts of ACME to Manifold

A rather basic knowledge of ACME's and Manifold's core features is enough to determine the clear mapping between most of the respectively equivalent concepts of the two models. For example, ACME's components can be directly represented in Manifold as processes. They both declare independent computational units having to execute specific and clear-stated tasks, using well-defined interfaces for interacting with the outside world. Ports have the exact same meaning in the two models: they both declare the way ACME components and the corresponding Manifold processes interact with their environment, with the aim to exchange information through well-defined interfaces.

More specifically, we now study the following ACME constructs and demonstrate their binding with equivalent Manifold constructs:

- *Components:* Components in ACME can be represented in Manifold as processes. The notion of Manifold's processes is exactly the same as that of ACME's components: they enclose a well-defined functionality and they communicate with the environment through the communication infrastructure (and not by a direct invocation).
- *Systems:* Systems in ACME are just syntactic wrappers of a number of component definitions and attachments. There is always a top-level `System` construct that defines the starting and ending point of our ACME description of a system and one `System` construct defining the starting and ending points for each representation included in our ACME description. In Manifold, the `System` construct is mapped to a special Manifold process called `Main` that must always exist in a Manifold application and defines the starting and ending points of the application execution. Furthermore, as we will see later, the lower-level system constructs, wrapping the representations, are also defined in Manifold by the Manifold's hierarchical structure.
- *Ports:* Ports in ACME can be represented in Manifold using the similar notion of the latter. ACME uses ports to represent the means of communication between a component and its environment. The only possible interaction between a component and its environment takes the form of reading from and writing data into the component's ports. This makes the component unaware of other components in its environment and the actual communication performed between them through connectors (this abstraction, while heavily restrictive, is mainly responsible for the representative power of ACME and its compatibility to a great subset of implementation languages). Manifold facilitates ports for the interprocess communication in the exact same way. Each process reads data units from its input ports, performs some processing and places the resulted data to its output ports. The process is not aware (and does not need to know) of which process has placed the data in its input ports or who will use the data that it placed to its output ports. The only difference between ports in ACME and Manifold is that ACME does not explicitly define ports as *input* or *output*. In order to resolve this difference, we use ACME's Open Semantic Framework to define a special property, namely `PortType`, which will hold the value of the port type (more details on our use of ACME's Open Semantic Framework are given later in this section).
- *Connectors and roles:* ACME's connectors and roles are equivalent in functionality to a set of Manifold streams. More precisely, an ACME connector or a set of ACME connectors, with their associated roles, represents the communication among a number of components in a system. In Manifold, the communication among a number of processes is performed indirectly, through a coordinator process. Specifically, a coordinator process, after the request from a worker process (by means of raising of an event or the satisfaction of a condition), dynamically creates the needed streams to transfer data among the processes that it coordinates. The simplest form of an ACME connector, having only two roles and acting as a pipeline from the first role to the second, can be represented in Manifold by a stream. More complex ACME connectors having

more than two roles can also be represented by means of a number of Manifold streams, depending on the number of roles attached to each side of the connector. For example, an ACME connector having two roles (**a** and **b**) attached to its right-hand side and two roles (**c** and **d**) attached to its left-hand side is equivalent to four binary connectors with roles **ac**, **ad**, **bc** and **bd** respectively. In short, this representation of a complex ACME connector with simple binary connectors enables us to represent any ACME connector with an arbitrary number of roles using simple Manifold binary streams.

- ACME's *representations* and *rep-maps,* which enable the software architect to define one or more lower-level descriptions of a component or connector, and thus establish the hierarchical descriptions of systems, can also be represented in Manifold. More to the point, while these two concepts cannot be mapped directly to concrete Manifold structures, the concept of having hierarchical structures of systems is clearly supported in Manifold. Recalling from Section 2, worker processes can themselves be managers of subgroups of other processes. Interpreting this in ACME, we get a component having a number of representations. ACME components attached to at least one representation can be mapped to manager processes, while ACME components that are not attached to any representation can be mapped to worker processes. However, it must be made clear that the concept of manager and worker processes in Manifold applies to a semantic (logic) level rather than a programming one. This means that we cannot directly map any manager process to a `Manifold` process at the programming level and any worker process to an `Atomic` process. Furthermore, an `Atomic` can be represented as an `Atomic Internal` or as an `Atomic External`, depending on the way that it is implemented. For these reasons, we defined a special property, namely `ComponentType`, that must be attached to every component of an ACME description in order to explicitly define if the component represents a `Manifold`, an `Atomic Internal` or an `Atomic External` process.

All the mappings described above can be clearly distinguished in Figs. 5 and 6 that represent the ACME architecture of a simple system and its corresponding model in Manifold: the first schema gives a high-level definition of a system's architecture and the second one gives a lower-level description of a component using the representation concept.

4.2. Towards a more powerful representation model

The power behind ACME, apart from its simplicity, is the Open Semantic Framework. Specifically, the system designer, having certain implementation language families in mind (in our case coordination languages), can consider extending the ACME ADL with new annotations and new semantics, which can later be converted to code. For example, if the system's designer knows that the implementation language will belong to the object-oriented family, he can add special annotations to represent the *constructor* and *finalizer* functionality in every component.

In order to achieve a more meaningful and effective mapping of ACME code to Manifold code (in the sense that the produced Manifold code both exploits Manifold's features but also implements as many aspects of the original software architecture artifact as is possible), we have used the Open Semantic Framework of ACME to introduce some

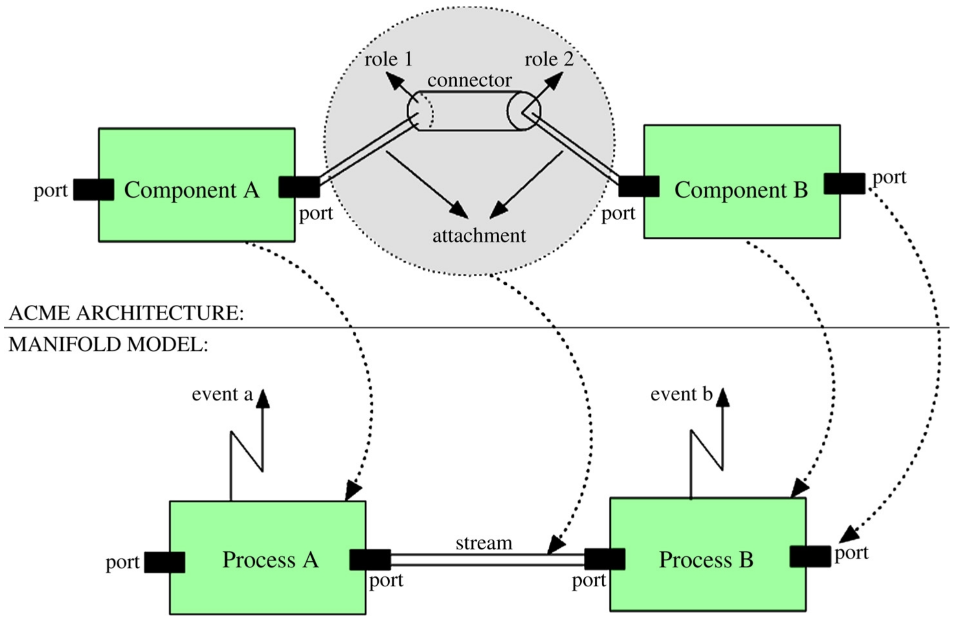

Fig. 5. High-level definition of a system’s architecture.

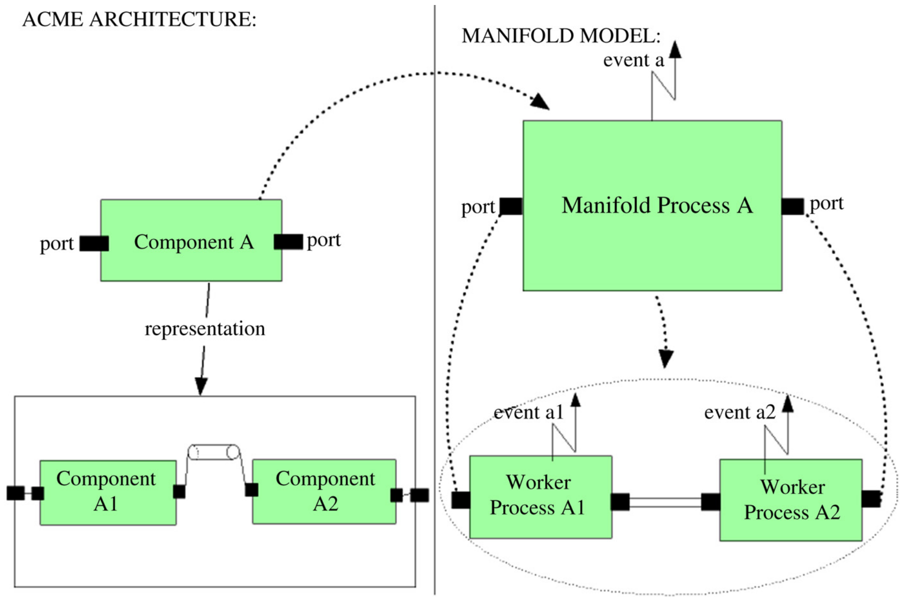

Fig. 6. Detailed description of component mapping.

new architectural elements in the form of ACME properties. These elements are related to the notions of events, state transitions and types of ports (input or output ones); these formalisms are not supported directly by ACME. However, using the Open Semantic Framework as shown below, we are now able to support them at the ACME level; the advantage of this undertaking is that the coordination part of some software architecture can now be separated from its computational part, thus assisting the developers in modeling and implementing different aspects of some application separately from the rest.

These "Manifold-compliant" properties we suggest to be defined at the ACME level, are summarized below (a detailed description of these properties and examples of their use is given in Appendix A):

- **`EventList : List`** = Declares the events that a component can raise to its environment.
- **`PortType : Enum {IN, OUT }`** = Declares the type of a port, i.e. if the port is an input or an output one.
- **`DataType : String`** = Declares the data type an input port can receive or an output port can send.
- **`DataSize : Integer`** = In case the **`DataType`** property of a port is of type string, **`DataSize`** declares the maximum size of the string that the port can receive or send at a time.
- **`ComponentType: Enum {ATOMIC_INT, ATOMIC_EXT, MANIFOLD}`** = Declares the type of Manifold process that a component represents, i.e. if the component represents an `Atomic Internal`, `Atomic External` or a `Manifold` process.
- **`Active_on: TupleList <event, ordering>`** = Declares the state in which a stream is activated (i.e. data is passing through it), and the order inside the state that the stream must be placed. In particular, stream connections with the same ordering are executed in parallel and those having different ordering are executed sequentially in the indicated order.

In particular, the `Active_on` property serves the need to complete the representation of the coordination logic at the ADL level. Streams in Manifold are the means of communication between two components; they act as *pipelines* transferring information between the ports of two components. Each stream in Manifold is explicitly activated/deactivated. However, since ACME *per se* does not explicitly activate/deactivate streams, the `Active_on` property is used to specify which of the ACME connectors (which will be implemented as Manifold streams) are active at some specific time/state. The idea here is to detect all the expected states in a given component configuration. Such a state definition could include the following scenario: in state A, port1 of component X sends some data to port2 of component Y. For an ACME representation of just one state, refer to Fig. 10 (the complete ACME architecture, with all states) and Fig. 11 (the same architecture for state=6).

As an illustration of using the above properties, we run over the transformation process performed by applying our mapping methodology on a simple system described in ACME. The ACME code for the toy example (Fig. 7) is automatically created by AcmeStudio [1], a tool offering a user-friendly graphical interface that enables users to easily create ACME-based descriptions of systems in a drag-and-drop way. The Manifold-compliant properties

```
1    System Example {
2     Component X = {
3      Properties {
4          ComponentType:
5             Enum {ATOMIC_INT, ATOMIC_EXT, MANIFOLD}=MANIFOLD;
6          EventList : List = <"ev_1">; }
7      Port Xin {  Properties { PortType:Enum{IN,OUT} = IN;  }}
8      Port Xout{  Properties { PortType:Enum{IN,OUT} = OUT; }}
9     }
10
11    Component Y = {
12     Properties {
13         ComponentType:
14            Enum {ATOMIC_INT, ATOMIC_EXT, MANIFOLD}=MANIFOLD;
15         EventList : List = <"ev_2">; }
16     Port Yin {  Properties { PortType:Enum{IN,OUT} = IN;  }}
17     Port Yout{  Properties { PortType:Enum{IN,OUT} = OUT; }}
18    }
19
20    Connector connXY = {
21      Role caller, callee;
22      Properties { Active_on:TupleList=<"ev_1,1">;}
23    };
24    Connector connYX = {
25      Role caller, callee;
26      Properties { Active_on:TupleList=<"ev_2,1"> }
27    };
28    Attachments {
29     X.Xout to connXY.caller;
30     Y.Yin  to connXY.callee;
31     Y.Yout to connYX.caller;
32     X.Xin  to connYX.callee;
33    };
34   };
```

Fig. 7. ACME code for the toy example.

that we have defined using the ACME's Open Semantic Framework are manually added by the user. In the following examples, these properties appear in **bold**.

The ACME description defines a system including two components with names `X` (line 2) and `Y` (line 11). The Manifold-compliant property `ComponentType` defines the type of Manifold process that each of the two components represent. For each component we define two ports (lines 7–8 and 16–17) that are distinguished as input or output ports by the Manifold-compliant `PortType` property. In lines 6 and 15, we use the `EventList` property to define the events that can be raised by each component. In lines 20–27, two binary connectors are defined for our system and in lines 28–33 attachments define the component ports attached to each role of the connectors. In order to define when these connectors will be activated, we use the `Active_on` property (lines 22 and 26) to define the event that will activate each of the two connectors. Mapping this code to Manifold we get two Manifold processes, each one having an input and an output port. We also get two streams connecting the input ports of each process to the output port of the other process. The first stream is dynamically created when event `ev_1` is raised by component `X`, while

```
//Main.m
manifold X(event ev_1)
   port in Xin.
   port out Xout.
import.

manifold Y(event ev_2)
   port in Yin.
   port out Yout.
import.

manifold Main() {
 event ev_1, ev_2.
 x is X.
 y is Y.

 begin: activate(x,y).
 ev_1: x.Xout->y.Yin.
 ev_2: y.Yout->x.Xin.
}
```

Fig. 8. The generated Manifold file for the example.

```
//X.m
export manifold X(event ev_1)
   port in Xin.
   port out Xout.
{
 event ev_1.

 begin:
}
```

Fig. 9. The code generated in the component definition file for component `X`.

the second stream is dynamically created when the event `ev_2` is raised from component `Y`. Since the two Manifold processes are at the first level of the Manifold application, they are activated and coordinated by the special `Main` Manifold process.

From the ACME code shown in Fig. 7 (which we recall that it was generated from a visual environment), we can actually generate not only the headers for each of the two processes in the Manifold code, but also the process definitions in the Manifold main file, the `main` method, the event declarations and the associated state transitions and communication streams. Fig. 8 shows the `Main` Manifold file that is generated using our methodology.

The skeleton of the component definition files is also produced, as illustrated in Fig. 9, for component `X`.

Although the mapping proposed by our methodology addresses all the features of ACME, the generated Manifold code is not complete, missing the mechanisms to activate processes, raise events within manifolds and perform dynamic reconfiguration of stream connections. This deficiency is due to the fact that ACME does not support the modeling of dynamically evolving systems and it would be a complicated exercise to attempt the specification of dynamic behavior at the ACME level by the way of introducing additional properties. We therefore leave the programmer to add manually these missing bits of code (the issue of dynamism is further discussed in the conclusions). Interestingly enough, the same situation arises with other cases where some ADL is mapped down to some executable representation that is based on a different model than that of the ADL in question [23].

An elaborated presentation of the mapping of an ACME description to skeletal Manifold code, as this is proposed by our methodology, is given in Appendix B.

4.3. *The hospital example*

The above-described methodology for generating executable Manifold code from an ACME representation will now be used to implement a fully-fledged software architecture. The system in question aims at addressing the needs of a typical hospital accounting service, using a component-based approach. The system supports account keeping for each patient. Each patient has his private account, and each user has an authorization level. The accounts can be charged or paid only by users with high authorization, while users with normal authorization are only able to print statements for the patients' accounts. For the payment of an account, a number of alternative payment methods such as credit card payment, cash payment and banking account transfers are provided.

The system described here can be viewed as part of a more complex system, connected to other components and services (this is why some ports in the ACME diagram appear not to be connected to other ports). The latter more complex system is effectively an amalgamation of a couple of other systems, designed for tele-medicine applications within the context of research projects. These systems have been developed over a period of a year by 2–3 programmers. They have been built using different technologies to those of this paper (one is Java-based and the other one is using .NET) and thus a direct comparison with the coordination approach is not a straightforward exercise. However, it is interesting to note that the approach developed in this paper allowed for the high-level modeling of the software architecture much faster and in a more natural way. Furthermore, the production of executable code was also a faster process, as much of the code was automatically generated from the tool described in Section 5.

For implementation, we use Manifold as the coordination language and the components (worker processes) are written in C. ACME is used for the system's architectural design. For the transition from the ACME code to the Manifold code, the methodology described in Sections 4.1 and 4.2 is used. Deriving the high-level architecture is a stepwise approach:

a. First, the components and their ports are identified. The only interaction of a component with its environment (i.e. another component) is through the ports. A port in the IWIM model as well as in most ADLs (including ACME) can be realized as a buffer, temporarily storing information, until it is read by another component (to be accurate, the connectors in ACME — the entities that connect two ports — can be used for

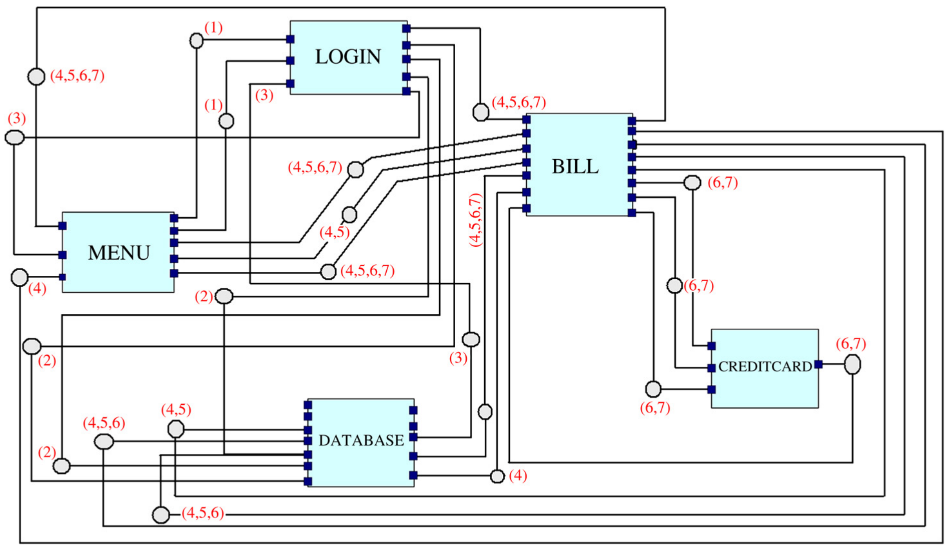

Fig. 10. The system architecture of the hospital example. The architecture is designed in AcmeStudio (Eclipse plug-in). For clarification reasons, we enhanced the diagram with state transition information (the `Active_on` property described in Section 4.2, which is saved in ACME code but not normally visible in the diagram). Numbers refer to the different states (described separately later). The colors of the original diagram as extracted from the AcmeStudio tool are altered for visibility purposes.

buffering the port values; they could even have some computational power). When a component needs to output a specific value-result to another component, it just pushes this value into the appropriate output port (not caring who will read it, when or why).

b. Second, the connectors are detected, that is, the actual interactions between two components through their ports. More specifically, we detect which components interact using which ports. We also detect which of these connectors are executed concurrently (in one state) and annotate them with the same `Active_on` number (as explained in Section 4.2). The idea is to detect all the expected states in a given component configuration. Such a state definition could include the following: e.g. in state `6`, component `Bill.Out_port_6` sends to component `CreditCard.In_port_1` some data. For an ACME representation of one state only please see Fig. 10 (the complete ACME architecture) and Fig. 11 (the same architecture for state=6).

c. Third, the ADL design is enhanced with the properties defined earlier in Section 4.2 (e.g. enhance the ports with their `DataType` property). This allows the generation of the better part of the coordination code, and leaves very few tasks for the programmer to complete manually.

d. Finally, the ACME code is parsed and the equivalent Manifold code is generated.

The ACME diagram (architectural design) of the example using AcmeStudio is shown in Fig. 10. The application consists of the following five components:

(i) Login: Verifies username and password and keeps the authorization level of the user.
(ii) Bill: Implements the basic accounting functions: insert/delete customer, pay/charge bill, print statement.
(iii) CreditCard: Communicates with appropriate bank systems to charge the account of a customer that pays using a credit card. This component is used by the Bill component in case the payment method selected by a customer is "credit card payment".
(iv) Database: Receives requests for data handling from the other components and executes them.
(v) Menu: Implements the interaction between the user and the system.

Each of the above components was implemented as a manager process coordinating an atomic process, which in turn implements the functionality of the component. The five manager processes are coordinated by the `Main` Manifold. It is interesting to note here that although the system could be implemented using five atomic processes coordinated by the `Main` Manifold, we have chosen to wrap every atomic process in a manager process in order to keep computational units (atomic processes) unchanged in case of a system change or evolution. In other words, we have separated the computational from the coordination concerns and in that respect we have generated code that can potentially have a higher degree of reusability than it would have had otherwise, had we generated only an equivalent fragment of executable code.

In the current state of the system, each of the manager processes simply passes data from its input ports to the input ports of the atomic processes that it coordinates and reversely from the output ports of the atomic processes to its own output ports. However, more sophisticated coordination scenarios can be supported.

The connectors represent Manifold streams. Each connector can be active in one or more states (presented in the diagram as numbers in brackets). Our example includes the following states:

(1) Menu_Requests_Login: The MENU component sends the username and password to the LOGIN component, so that the user's authorization is checked.
(2) Login_Requests_Verification: The LOGIN component asks from the DATABASE component to perform a query for the user's authorization.
(3) Login_Gets_Verification: The LOGIN component gets the authorization level from the DATABASE component, and returns the answer to the MENU component.
(4) Menu_Requests_Bill_Info: The MENU component requests information for a given customer from the BILL component, which in turn asks for some information from the DATABASE component and returns the answer to the MENU component.
(5) Menu_Requests_Bill_Create: The MENU component requests the creation of a bill from the BILL component for a new customer. The BILL component prepares and sends the related data to the DATABASE component. The DATABASE component sends back a confirmation to the BILL component, which returns the confirmation to the MENU component.
(6) Menu_Requests_Bill_Insert/Update: The MENU component requests the insertion or updating of a bill from the BILL component. The BILL component prepares and sends the related data to the DATABASE component. The DATABASE component sends

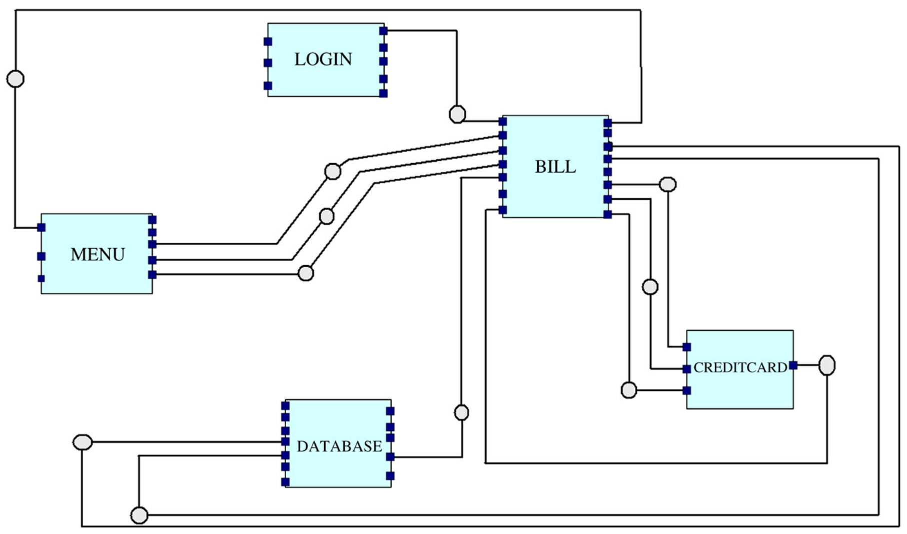

Fig. 11. The enhanced ACME providing a representation for a state-full model (current model is for state=6). The colors of the original diagram as extracted from the AcmeStudio tool are altered for visibility purposes.

back a confirmation to the BILL component, which returns the confirmation to the MENU component. Also, if credit card payment is included, BILL sends the relevant data to the CREDITCARD component, which makes the transaction and sends the result back.

(7) Menu_Requests_Bill_Delete: The MENU component requests the deletion of a bill from the BILL component. The BILL component sends the relevant data to the DATABASE component, and waits for deletion confirmation. It then notifies the MENU component that the deletion is successful.

The state-full diagram in ACME is made possible due to the `Active_on` extension, which was added using ACME's Open Semantic Framework (Section 4.2). Namely, this enhancement enables us to generate an important part of the coordination in the Manifold code (roughly, all the states and stream definitions). For example, the actual system architecture for state 6 (included in the original ACME design by means of the `Active_on` property) is depicted in Fig. 11. As this knowledge exists in our ACME model, the methodology is able to generate a more complete Manifold code, i.e. the state and all the streams.

The ACME Code

A fragment of the ACME code produced for the example is presented in Fig. 12 (for brevity we present only the code for the LOGIN component; the code for the rest of the components is similar in nature).

```
System Hospital = {
   Component Type Login_Manager = {
    Properties {
     EventList : List = <"login_requests_verif">;
      ComponentType: Enum{ATOMIC_INT,ATOMIC_EXT,MANIFOLD}=MANIFOLD}

      Port i_username = {
       Properties {
         PortType: Enum{IN,OUT} = IN; } }

      Port i_password = {
       Properties {
         PortType: Enum{IN,OUT} = IN; } }

      Port i_security_level = {
        Properties {
        PortType: Enum{IN,OUT} = IN; } }

      Port o_username = {
        Properties {
         PortType: Enum{IN,OUT} = OUT; } }

      Port o_password = {
       Properties {
         PortType: Enum{IN,OUT} = OUT; } }

      Port o_security_level = {
       Properties {
         PortType: Enum{IN,OUT} = OUT; } }

      Port o_flag = {
       Properties {
         PortType: Enum{IN,OUT} = OUT; } }

    Representation = {
    System Login {
      Component Type Login_Worker = {
       Properties {
       EventList : List = <"login_worker_requests_verif">;
       ComponentType: Enum{ATOMIC_INT,ATOMIC_EXT,MANIFOLD} =
                    ATOMIC_INT }

      Port i_username = {
       Properties {
         PortType: Enum{IN,OUT} = IN;
         DataType: String = "string";
         DataSize: Integer = 40; } }

      Port i_password = {
       Properties {
         PortType: Enum{IN,OUT} = IN;
         DataType: String = "string";
         DataSize: Integer = 40; } }

      Port i_security_level = {
       Properties {
         PortType: Enum{IN,OUT} = IN;
         DataType: String = "integer"; } }
```

Fig. 12. Fragment of ACME code for the example in Fig. 10.

```
     Port o_username = {
      Properties {
        PortType: Enum{IN,OUT} = OUT;
        DataType: String = "string";
        DataSize: Integer = 40; } }

     Port o_password = {
      Properties {
        PortType: Enum{IN,OUT} = OUT;
        DataType: String = "string";
        DataSize: Integer = 40; } }

     Port o_security_level = {
      Properties {
        PortType: Enum{IN,OUT} = OUT;
        DataType: String = "integer"; } }

     Port o_flag = {
      Properties {
       PortType: Enum{IN,OUT} = OUT;
       DataType: String = "integer"; } }
    }

 Bindings {
   i_username to Login_Worker.i_username
   i_password to Login_Worker.i_password
   i_security_level to Login_Worker.i_security_level

   Login_Worker.o_username to o_username
   Login_Worker.o_password to o_password
   Login_Worker. o_security_level to o_security_level
   Login_Worker. o_flag to o_flag
  }
 }
}

  Connector Database_Login_Security =  {
    Role caller;
    Role callee;
    Active_on:TupleList=<"login_gets_verif,1">;  }
    Connector Login_Menu_Security = {
    Role caller;
    Role callee;
    Active_on:TupleList=<"login_gets_verif,1">;  }

  Connector Menu_Login_Username = {
    Role caller;
    Role callee;
    Active_on:TupleList=<"menu_requests_login,1">;  }
  Connector Menu_Login_Password = {
    Role caller;
    Role callee;
```

Fig. 12. (*continued*).

```
    Active_on:TupleList=<"menu_requests_login,1">;  }
  Connector Login_Database_Flag = {
    Role caller;
    Role callee;
    Active_on:TupleList=<"login_requests_verif,1">;  }
  Connector Login_Database_Username = {
    Role caller;
    Role callee;
    Active_on:TupleList=<"login_requests_verif,1">;  }
  Connector Login_Database_Password = {
    Role caller;
    Role callee;
    Active_on:TupleList=<"login_requests_verif,1">;  }

  /* The rest of the connectors for the login component and the
     rest of the components are defined here */

  Attachments {
    Database.o_security_level to Database_Login_Security.caller;
    Login.i_security_level to Database_Login_Security.callee;
    Login.o_security_level to Login_Menu_Security.caller;
    Menu.i_security_level to Login_Menu_Security.callee;
    Menu.o_username to Menu_Login_Username.caller;
    Login.i_username to Menu_Login_Username.callee;
    Menu.o_password to Menu_Login_Password.caller;
    Login.i_password to Menu_Login_Password.callee;
    Login.o_flag to Login_Database_Flag.caller;
    Database.i_flag to Login_Database_Flag.callee;
    Login.o_username to Login_Database_Username.caller;
    Database.i_username to Login_Database_Username.callee;
    Login.o_password to Login_Database_Password.caller;
    Database.i_ password to Login_Database_Password.callee;

  /*  The rest of the attachments for the login component and
      the rest of the components are defined here */
  }

}
```

Fig. 12. (*continued*).

The Manifold code

The ACME code produced was parsed with the suggested methodology, as described in Section 4.1. Following the suggested mappings, we were able to generate most of the code for the Manifold files (`.m`), the complete headers files (`.ato.h`), and a significant part of the component definition files (`.ato.c`).

Regarding the `Main` Manifold file, we were able to produce the complete component interfaces and the better part of the `Main` function. The code produced for the `Main` function includes the events definitions, the manager process definitions, the states definitions and the streams definitions placed in the right states and in the right order inside a state.

Part of the `Main` Manifold code that was produced from applying the methodology of Section 4.1 is presented in Fig. 13 (again, for brevity we show the LOGIN component

```
# include "MBL.h"
# include "rdid.h"

/* login component interface */
manifold login_manager(event login_requests_verif)
    port in i_username, i_password, i_security_level.
    port out o_username, o_password, o_security_level, o_flag.
    import.

/* bill, database, menu components interfaces defined here */

manifold Main()
{
 event menu_requests_login, login_requests_verify, login_gets_verif.
/* the rest of the events are defined here */
 process login_manager is login_manager(login_requests_verif).
/* the rest of the processes are defined here */

  begin : ( activate(pmenu, plogin, pbill, pdbase),
            preemptall, terminated(void) ).

  menu_requests_login : (
      menu_manager.o_username->login_manager.i_username,
      menu_manager.o_password->login_manager.i_password,
      terminated(void)   ).

  login_requests_verif: (
      login_manager.o_flag->dbase_manager.i_flag,
      login_manager.o_username->dbase_manager.i_username,
      login_manager.o_password->dbase_manager.i_password,
      terminated(void)   ).

  login_gets_verif:(
      dbase_manager.o_security_level ->
                        login_manager.i_security_level,
      login_manager.o_security_level ->
                        menu_manager.i_security_level,
      terminated(void) ).
/* the rest of the states are defined here */
}
```

Fig. 13. Manifold code for the `Main` file. The code which was manually added by the user appears in bold.

only). The code in bold had to be added manually by the programmer. The rest of the code was automatically produced.

We were able to generate almost the complete code of the Manifold files implementing the manager processes. The code produced includes the atomic process definition, the states definitions and the streams connecting the ports of the manager process to the ports of the atomic process.

The code produced for the Manifold file implementing the Login Manager is given in Fig. 14. The code in bold had to be added manually by the programmer; the rest of the code was produced using our methodology.

```
//pragma include "login.ato.h"

# include "MBL.h"
# include "rdid.h"

/* login component interface */
export manifold login_manager(event login_requests_verif)
    port in i_username, i_password, i_security_level.
    port out o_username, o_password, o_security_level, o_flag.
{
 event login_worker_requests_verif.
 process login_worker is login_worker(login_worker_requests_verify)

 begin : activate(login_worker);
         ( i_username->login_worker. i_username,
           i_password->login_worker. i_password,
           i_security_level->login_worker.i_security_level,
           login_worker.o_username-> o_username,
           login_worker.o_password-> o_password,
           login_worker.o_security_level ->o_ security_level);
           terminated(void).

login_worker_requests_verif: raise(login_requests_login).

}
```

Fig. 14. Manifold file for the LOGIN manager.

Our methodology was even more successful for the component definitions files (`.ato.c`). We were able to produce all the code that was related to Manifold; this allowed us to concentrate on the development of the functionality, rather than the synchronization with the Manifold environment. The methodology generated the functions for reading and writing variables from and to ports, and we were able to use them whenever they were needed. The events were defined at the top of each component, and we were able to raise them whenever needed, by copying the relevant lines with the `AP_Raise` commands. Part of the code for the login worker (atomic) process, produced by applying our methodology, is shown in Fig. 15.

Finally, we were able to create the complete header files. The header file produced for the login worker process follows:

```
# include "AP_interface.h"
extern void login(AP_Event login_requests_verif);
```

Summarizing, we see that the methodology not only visualized the Manifold system early in the development cycle, but also created almost all of the Manifold coordination code. More specifically, apart from the simple but time-consuming structural information such as component definitions and file headers, the methodology released the programmer from the very error-prone task of defining the streams in the proper states (this information was entered in a much easier visual environment, and was also schematically visible to the user). The methodology also created the skeleton

```
#include "AP_interface.h"
#include "login.ato.h"
#include <stdlib.h>

/* reads an integer from a port; returns the integer */
int readInteger(int port)
{
  int XVal;
  AP_Unit unitX = AP_AllocateUnit();
  AP_PortRemoveUnit(port,&unitX,NULL);
  AP_FetchInteger(unitX, &XVal);
  AP_DeallocateUnit(unitX);
  return XVal;
}

/* reads a string from a port; returns pointer to the string */
char* readString(int port, int string_size)
{
  char* XVal = (char*) malloc(sizeof(char) * string_size);
  AP_Unit unitX = AP_AllocateUnit();
  AP_PortRemoveUnit(port,&unitX,NULL);
  AP_FetchString(unitX, &XVal);
  AP_DeallocateUnit(unitX);
  return XVal;
}

/* writes an integer to a port */
void writeInteger(int port, int intVal)
{
  AP_Unit unitX = AP_AllocateUnit();

  unitX = AP_FrameInteger(intVal);
  AP_PortPlaceUnit(port,unitX,NULL);
  AP_DeallocateUnit(unitX);
}

/* called to write a string to a port */
void writeString(int port, char* myString)
{
  AP_Unit unitX = AP_AllocateUnit();
  unitX = AP_FrameString(myString);
  AP_PortPlaceUnit(port,unitX,NULL);
  AP_DeallocateUnit(unitX);
}
```

Fig. 15. Code for the LOGIN worker (`ato.c` file).

of all the required files, leaving no room for errors from the user. Furthermore, the difficult API of Manifold for the computational language — C in our case — e.g. reading from a component's port or placing data in a port, was replaced with easy-to-understand function invocations. While it would be very naïve to talk for percentage of the code generated, since the success of the methodology depends on the complexity of the scenario, in ordinary cases like the example presented here, almost the complete coordination code is produced, allowing the user to focus on the computational part of the program.

```
void login(AP_Event login_requests_verif)
{
  /* definitions of variables for reading and writing to
     the ports - created from the DataType and DataSize
     properties for each port */
  char username[40];
  char password[40];
  int security_level = 0;

  /* port definitions */
  int i_username= AP_PortIndex("i_username");
  int i_password= AP_PortIndex("i_password");
  int i_security_level= AP_PortIndex("i_security_level");
  int o_username= AP_PortIndex("o_username");
  int o_password= AP_PortIndex("o_password");
  int o_security_level= AP_PortIndex("o_security_level");
  int o_flag= AP_PortIndex("o_flag");

  /* whenever you need to read from a port you
    call the functions readInteger(int port),
    readString(int port, int string_size)
  */

  /* whenever you need to write to a port you
   call the functions writeInteger(int port,int intVal),
   writeString(int port, int string_size, char* myString)
  */
}
```

Fig. 15. (*continued*).

5. Code generation

While the actual development of a tool was not in the immediate target of this work, for the sake of proof-of-concept we have developed a parsing tool which generates Manifold code for an ACME specification using the methodology described in this paper. This tool makes use of AcmeStudio and technologies such as the Architecture Description Markup Language (ADML, [2]) and xAcme [24]. The tool was developed in Java, using Sun's JAXP v1.2 XML-parser, and generating Manifold code. Development took approximately 7 human workdays. The parser's implementation was nearly 1500 lines of code.

ADML and xAcme are XML-based representation languages for architectures, which were built based on ACME. AcmeStudio supports the exportation of an ACME design to ADML or xAcme representation. Having an ACME design described in an XML format makes it possible to use an XML parser to extract the features needed from the ACME description and generate the implementation code. For the implementation of our tool, we use AcmeStudio (Eclipse plug-in) to export ACME architectures to ADML code and Sun's JAXP v1.2 XML-parser to parse the ADML code and extract the needed information for code generation.

To demonstrate the steps needed to generate Manifold code using our tool, consider the system appearing in Fig. 16. The connectors transmitting data from component `X` to

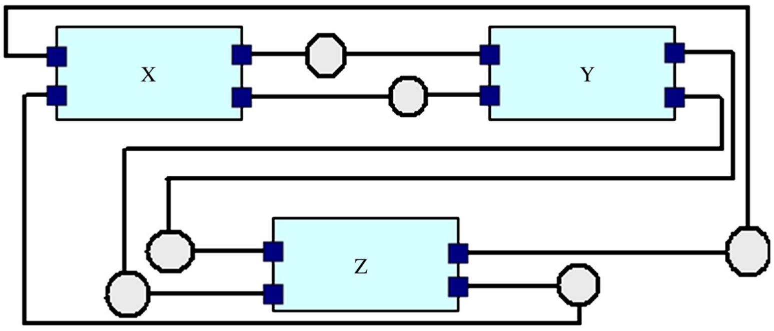

Fig. 16. A simple architecture configuration.

component `Y` and from component `Z` to component `X` are activated in reaction to the raising of event `ev_1` from component `X`. The connectors transmitting data from component `Y` to component `Z` are activated in reaction to the raising of event `ev_2` from component `Y`. Once the design of the system in AcmeStudio is complete the needed properties are added to components and connectors; we use the *Export Design* option to export our system description to ADML code.

The most relevant parts of the ADML code generated for this setup are shown in Fig. 17.

The ADML code is then fed into our parser that extracts the needed features and creates an object-model of the ACME design. Then, the object-model is passed to the code generation part of our tool in order to produce the Manifold code. The code produced from our tool for the `Main` Manifold file implementing this system is shown in Fig. 18.

A somewhat careful examination of Fig. 18 reveals that all the code related to the coordination aspects of implementing the main manager process is successfully generated, thus making the programmer's work much easier.

6. Related work

Architecture Description Languages are playing a major role in the development of component-based systems, offering an effective way in dealing with all the major issues of designing a complex software system, thus allowing the designers to assess at an early stage what is the best way to ensure that all the key requirements of a software architecture are satisfied. A natural evolution of the ADL field is the bridging of the gap between the design level, as this is expressed by some ADL, and the implementation level, as this is realized by some computational model. This need has been identified by many researchers; in particular, Monroe [17] claims that even the generation of skeletal code at the implementation level would significantly reduce implementation time. Garlan's positions [8] are along the same line of thinking. Towards that end, a number of ADLs support the generation of executable code from their specifications [15]. In particular, Aesop, C2, and Darwin generate skeletal code in C/C++

```
<System>
   <SystemDescription>
       <SystemStructure>
           <Component identifier='3287' name='X'>
               <ComponentDescription>
                   <ComponentBody>
                        <Property name='ComponentType'> ...
                           <PropertyLiteralValue value='MANIFOLD'/>
                        </Property>
                        <Property name='EventList'> ...
                           <PropertyLiteralValue value='ev_1'/>
                        </Property>
                        <Port name='X2out'>
                           <PortDescription> ...
                               <Property name='PortType'> ...
                                  <PropertyLiteralValue value='OUT'/>
                               </Property>
                           </PortDescription>
                        </Port>
/* The remaining ports are defined here in the same way */
                   </ComponentBody>
               </ComponentDescription>
           </Component>
/* The remaining components are defined here in the same way */
           <Connector name='XY1'>
             <ConnectorDescription>
               <ConnectorBody> ...
                  <Property name='Active_on'> ...

                    <PropertyLiteralValue value='ev_1,1'/>
                  </Property>
                  <Role identifier='3703' name='caller'> ... </role>
                  <Role identifier='3703' name='callee'> ... </role>
               </ConnectorBody>
             </ConnectorDescription>
            </Connector>
/* The remaining connectors are defined here in the same way */
            <Attachments>
               <Attachment portID='3447' roleID='3723'>
            </Attachment>
/* The remaining attachments are defined here in the same way */
            </Attachments>
       </SystemStructure>
   </SystemDescription>
</System>
```

Fig. 17. Sample of the ADML code representing the model in Fig. 16.

and Rapide executes the design code internally. Furthermore, MetaH [25] is supported at the implementation level by Ada, and ArchJava [4] effectively superimposes software architecture structures on top of Java. Finally, ZCL is mapped to LuaSpace, a CORBA-based distributed environment [23]. Only C2 and ZCL (the latter by virtue of the underlying CORBA environment) seem to be able to generate code for more than one programming language.

```
manifold X ( event ev_1 )
   port out X2out. port out X1out. port in X1in. port in X2in.
import.

manifold Y ( event ev_2 )
   port in Y1in. port in Y2in. port out Y1out. port out Y2out.
import.

manifold Z ( event ev_3 )
   port in Z1in. port in Z2in. port out Z1out. port out Z2out.
import.

manifold Main() {

 event ev_1, ev_2, ev_3.

 x is X(ev_1).
 y is Y(ev_2).
 z is Z(ev_3).

  begin:
  ev_1:(x.X1out->y.Y1in,x.X2out->y.Y2in);
       (z.Z1out->x.X1in,z.Z2out->x.X2in).
  ev_2:(y.Y1out->z.Z1in,y.Y2out->z.Z2in).
  ev_3:
}
```

Fig. 18. Code produced from our tool for the `Main` Manifold file.

In this paper, we have exploited the relationship between ADLs and control-driven coordination models, as this was understood in [21]. In particular, we build our work upon the fact that the fundamental concepts of ADLs (components, connectors, ports, black-box approach to component development, etc.) have very similar interpretation and realization in the world of control-driven coordination. We have therefore developed a blueprint for mapping in a methodical way the design code of any state-of-the-art ADL down to an executable representation that implements the modeled software architecture.

In a way, this paper complements the work reported in [12] where a methodology is proposed to build the software architecture of a system based on coordination information elicited from software requirements. More specifically, that paper describes an approach to analyze the use cases created during the software requirements phase in order to identify and model the elements of the system to be coordinated and the respective coordinators required to coordinate these elements. The elicited information, which is formally modeled using stereotyped UML diagrams, is then used to define the rules (policies) to be implemented by coordinators and finally to drive the description of the system's software architecture. A combination of the work reported in [12] with our methodology could lead to a complete framework for development of systems based on the coordination paradigm. The framework would identify the coordination requirements from the early software requirements phase and then generate the code implementing all the coordination aspects of a system using the system's architectural description as a bridge between the two phases.

7. Discussion

In this work we presented a novel approach, using high-level architectural models, for initially modeling a software system and subsequently generating executable code based on the coordination paradigm. Our technique is using on the one hand ACME, a generic language for describing software architectures, and on the other hand the coordination model IWIM and its associated language Manifold. In the process, we have shown that a significant part of the implementation code that must be eventually produced can be generated automatically from our specification framework. The fact that our approach integrates software architectures and coordination models enabled us to derive the advantages that both of them provide in reducing the costs of the software development process. The modeling of the system architecture using an ADL, which forms the first step of our approach, enables developers to define the more important properties and constraints of the system under development, but also to detect errors early at the design time, thus saving development time. The generated code, which is consistent with the previously modeled architecture, clearly separates the communication from coordination parts of the system, making the system maintenance much easier.

The reason for using ACME (rather than some particular concrete ADL) was precisely in order to show the generality of our approach: since ACME embodies the core features that any state-of-the-art ADL would support, by mapping ACME to Manifold we effectively provide the core of an implementation route for any other ADL.

Although our methodology for mapping an ADL down to executable code using the coordination paradigm has concentrated on the use of Manifold as the coordination language, we believe that the essence of our translation scheme can be used for any coordination language that adheres to the fundamental principles of the IWIM coordination model [5]. Consequently, a number of such control-driven coordination languages such as ConCoord [11] can use the mapping formalisms of Section 4 (and in particular the general principles of Sections 4.1 and 4.2) to derive different implementation paths from ADLs down to coordination-based executable code.

Furthermore, although we used C as the implementation language of the computational components, any other Manifold-compliant computational language could have been used; thus, our approach is language independent. The programs generated by our framework can run in a parallel/distributed fashion on a number of systems, by virtue of the Manifold environment which runs on top of PVM and has been successfully ported to a number of platforms including Sun, Silicon Graphics, Linux, and IBM AIX, SP1 and SP2. Finally, the opportunity to write the computational components of Manifold as operating system scripts (atomic externals) extends the applicability of our methodology to software architectures that involve entire existing systems as black-box components. In short, the presented methodology is reusable at all levels of software development: the ADL, coordination language, and computational language ones.

The fact that some coordination code parts have to be manually added by the programmer forms a drawback to the maintenance of the code generated by our approach, since the code transformation is now performed only one way (from ACME to Manifold). While this problem, up to now, was not limiting us to demonstrate the importance of this work, we are now in the process of enhancing our methodology with some more powerful

CASE tools that will further support our current work on coordination systems. More particularly, we are currently examining some code-block recognition methods used in existing code generation tools which can address the maintenance problem.

The integration of software architectures and ADLs for specification, with coordination models and languages for implementation, has a number of advantages and opens up several possibilities. From the ADLs point of view, coordination offers an alternative approach to code generation which enjoys some fundamental (in the coordination field) advantages such as programming language independence (components may be written in different languages even within the same application), and higher degree of component reusability (because of the clear separation of the coordination code from the computational one). These advantages are particularly noticeable for the case of control-driven coordination. From the coordination point of view, ADLs offer of course a way of modeling and analyzing a system well before its implementation begins. Furthermore, languages such as Manifold are very powerful but often difficult to be used effectively because of the plethora of features they support. Allowing a programmer to model his system first at the level of an ADL reveals significant information that can then be used at the level of coordination and, subsequently, implementation to derive an effective execution plan for his system that will essentially separate the coordination concerns from the computational ones, thus enjoying all the benefits of coordination.

The code generation process presented here can significantly reduce the cost for developing software with coordination languages. One of our initial motivations was that some of the current coordination languages failed to establish themselves not because of their weaknesses (since they offer a superset of any other procedural and object-oriented language) but mainly because of their high level of difficulty. As such, their use is currently restricted in scientific and research environments, where they can be facilitated and significantly reduce the cost of building large coordination-based systems to a minimum. However, this work enhances the coordination languages with a GUI front-end, which can very easily be learned and facilitated to generate most of the coordination-related code (which is actually the difficult part in the coordination language). This not only reduces the overall cost for developing the system since less code is needed to be written from the programmer, but more importantly it separates the coordination-communication from the computational logic from design time.

Of course, like any other approach, ours has also some potential risks and limitations that must be taken into consideration and dealt with. First of all, the software engineer must acquaint himself with the coordination approach, and more to the point, the control-driven one. As this latter approach is in many ways similar to the component-connector metaphor of software architectures (see [21] for a related discussion) we believe that this is not such a tiring exercise. Second, the tools and programming environments we have used in this work are all research and academic products; as such, there is the issue of continuous maintenance of them. Finally, although in theory any programming language can be used in association with Manifold, so far only C and Fortran are supported. However, we are aware of research that has been done in developing Java-based environments using IWIM, Manifold's underlying coordination model [6,16].

Our current work focuses on the issues of dynamic evolution; we note that ACME, being an interchange language rather than a fully-fledged ADL, has no dynamic capabilities.

```
/* when the state is filled (all processes working) */
filled: {
  /* create a new process */
  process merge<a, b | output> is AtomicIntMerger.
  /* and dynamically integrate it in the system */
  stream KK * -> (merge.a, merge.b).
  stream KK merge -> output.

  begin: (
     activate(merge),
  /* also assign it some work */
     input -> Sorter -> merge.a, atomsort -> merge.b,
     merge -> output).

  end | finished:.
}.
```

Fig. 19. Code extract from a distributed integer sort implementation with Manifold.

However, modeling the dynamic evolution of a system (also known as dynamic (re-) configuration), is one of the most important aspects an ADL could support (although, admittedly, very few of them actually do). On the other side, the control-driven coordination models and languages such as Manifold have very powerful dynamic features [22]. Namely, in Manifold we are able not only to define state transitions (as we have modeled them in our enhanced ACME) but we can also dynamically reconfigure the whole setup, by adding further component instances. Both state transition and dynamic reconfiguration are visible in the Manifold extract in Fig. 19 (taken from a distributed integer sorter, [7]). This code is executed when all the current component instances have been overloaded. It dynamically creates new component instances and distributes them to the virtual machine, also assigning them a workload.

This functionality cannot be fully supported presently by our enhanced ACME environment. Instead, we have added a limited support for dynamism in our ACME descriptions by defining the `Active_on` property that specifies which event triggers the construction of each possible connection of a Manifold system architecture. This enabled us to place the Manifold code that dynamically creates the corresponding streams in the right Manifold states. We are currently investigating how the aspect of dynamism, as this is expressed in languages that explicitly support dynamic architectures such as C2 [14,18] and Rapide [13], can be mapped from the ADL level down to the coordination level. This mapping will also enable us to generate the complete coordination code of the system and therefore solve the problem of maintaining the handwritten extensions that currently the programmer has to manually apply to the generated code in order to add any arbitrary dynamic reconfiguration behavior of the system.

Appendix A

Here we give a detailed description of the Manifold-compliant properties we have defined by exploiting the ACME's Open Semantic Framework. We also present examples of the use of these properties in an ACME description but also the respective Manifold code generated based on these properties.

`EventList : List` = Declares the events a component can raise. **`List`** takes the form <**`event_name_1`** ... **`event_name_N`**> and results in the declaration of **`event_name_1`**, ..., **`event_name_N`** within the component in which it appears. The exact code that is generated depends on the nature of the component in question (whether it is a Manifold or an atomic process — see Section 4.1).

`PortType : Enum {IN, OUT }` = Declares the type of a port, i.e. if the port is an input or an output one. Depending on the nature of the port, the code generator will produce a set of calls to relevant Manifold routines which will extract data from this port (if it is an input one) or place data into the port (if it is an output one). For example, for an input port, calls to the routines **`AP_PortRemoveUnit(...,&unit...)`** will be made and the data received will be placed in the placeholder **`unit`**.

`DataType : String` = Declares the data type an input port can receive or an output port can send. Depending on the type of the sent or received data, calls to the appropriate Manifold routines will be produced by the code generator. For example, if the port is an input one and the data type is integer, then a call to the routine **`AP_FetchInteger(unit,...)`** will be made which will transform the data that has been received from the execution of the **`AP_PortRemoveUnit`** call to an integer.

`DataSize : Integer` = In case the **`DataType`** property of a port is of type string, **`DataSize`** declares the maximum size of the string that the port can receive or send at a time.

`ComponentType: Enum {ATOMIC_INT, ATOMIC_EXT, MANIFOLD}` = Declares the type of Manifold process that a component represents, i.e. if the component represents an `Atomic Internal, Atomic External` or a `Manifold` process.

The following example demonstrates in an obvious way the use of the above properties (a trivial component, receiving a string in the input port and delivering the string capitalized in the output port):

```
Component ConvertStringToUpper = {
 Properties {
  EventList : List =<"component_activated","capit_finished">;
  ComponentType: Enum {ATOMIC_INT, ATOMIC_EXT,MANIFOLD} = ATOMIC_INT;
  Port InputString = {
    Properties {
      PortType:Enum{IN,OUT } = IN;
      DataType : String = "string";
      DataSize : Integer = 40;
    }
  }
  Port OutputString = {
    Properties {
     PortType:Enum {IN, OUT } = OUT;
     DataType : String = "String";
     DataSize : Integer= 40;} }
   }
  }
 }
}
```

The above ACME fragment will generate the following pieces of Manifold/C code:
In the Manifold file:

```
manifold ConverStringToUpper( event component_activated,
                              event capit_finished      )
  port in InputString.
  port out OutputString.
atomic { internal. }.
```

In the header file:

```
# include "AP_interface.h"
extern void ConverStringToUpper(AP_Event component_activated,
                                AP_Event capit_finished      );
```

In the atomic file:

```
#include "AP_interface.h"
#include "ConverStringToUpper.ato.h"
#include <stdlib.h>

void ConverStringToUpper(AP_Event component_activated,
                        AP_Event capit_finished      ){
char InputString[40];
char OutputString[40];

int input = AP_PortIndex("InputString");
int output= AP_PortIndex("OutputString");

}
```

Active_on: TupleList <event, ordering> = Declares the state in which a stream is activated (i.e. data is passing through it), and the order inside the state that the stream must be placed. In particular, stream connections with the same ordering are executed in parallel and those having different ordering are executed sequentially in the indicated order.

To explain in an intuitive way the `Active_on` property, we consider the following trivial example: We have a numeric calculator accepting two numbers and an operator (`+-*/`) and returning the result. To maximize efficiency, since disk I/O is slow, a buffer component is used to feed the calculator component. However, the buffer component has a limitation of storage space for only two numbers and an operator (thus being able to feed the calculator component for only one processing cycle). The buffer component is fed from the `diskIO` component, whose responsibility is to keep the buffer always filled, by reading the two numbers and the operator from a file, and posting them into the buffer's component ports. The `diskIO` component is also responsible for receiving the result from the calculator component and storing it to another file.

The system initializes with the creation of the three components. Then, `diskIO` delivers the two numbers and the operator into the buffer component, and finally, the loop starts. Fig. A.1 shows the setup in AcmeStudio. An extract from the ACME code produced for this configuration is presented in Fig. A.2. Note that the code is enhanced with Manifold-compliant properties, as explained in Section 4.2.

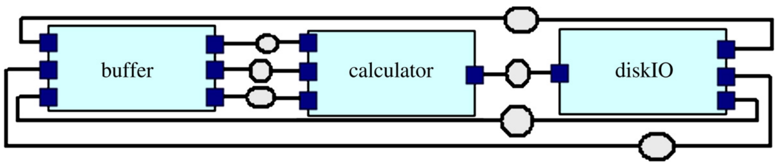

Fig. A.1. Numeric calculator example in ACME.

```
Component buffer = {
  Port in_number1, in_number2, in_operator;
  Port out_number1, out_number2, out_operator; };

Component calculator = {
  Properties {
  EventList : List = <"result_ready">; } // raised when result is
                                       // ready and I need more numbers
  Port in_number1, in_number2, in_operator, out_result; };

Component diskIO = {
  Properties {
  EventList : List = <"numbers_ready">; } // raised when numbers are
                                     // ready and I wait for the result
  Port in_result, out_number1, out_number2, out_operator; };

Connector buffer_calculator_number1 = {
  Role caller, callee;
  Active_on:TupleList=<"numbers_ready,1">; };

Connector buffer_calculator_number2 = {
  Role caller, callee;
  Active_on:TupleList=<"numbers_ready,1">; };

Connector buffer_calculator_operator = {
  Role caller, callee;
  Active_on:TupleList=<"numbers_ready,1">; };

Connector calculator_diskIO_result = {
  Role caller, callee;
  Active_on:TupleList=<"result_ready,1">; };

Connector diskIO_buffer_number1 = {
  Role caller, callee;
  Active_on:TupleList=<"begin,1", "numbers_ready,2">; };

Connector diskIO_buffer_number2 = {
  Role caller, callee;
  Active_on:TupleList=<"begin,1", "numbers_ready,2">; };

Connector diskIO_buffer_operator = {
  Role caller, callee;
  Active_on:TupleList=<"begin,1", "numbers_ready,2">; };

Attachments {
// here we define the attachments
};
```

Fig. A.2. ACME code extract for the toy configuration in Fig. A.1.

Regarding the use of the `Active_on` property, we now show the relevant extract from the Manifold code that our methodology produces for this example (the line with *italics* is not produced from the methodology; however, it is included to help in better understanding the example):

```
begin:
      ...
      (diskIO.out_number1 ->buffer.in_number1,
       diskIO.out_number2 ->buffer.in_number2,
       diskIO.out_operator->buffer.in_operator);     /* seq op */
      raise(numbers_ready).

numbers_ready:
      (buffer.out_number1 ->calculator.in_number1,
       buffer.out_number2 ->calculator.in_number2,
       buffer.out_operator->calculator.in_operator); /* seq op */
      (diskIO.out_number1 ->buffer.in_number1,
       diskIO.out_number2 ->buffer.in_number2,
       diskIO.out_operator->buffer.in_operator).

result_ready:
(calculator.out_result-> diskIO.in_result).
//numbers_ready event is raised from inside the diskIO component
//result_ready event is raised from inside the calculator component
```

Appendix B

Here we describe in detail the Manifold code our methodology produces from an ACME description of a system, with respect to the three separate kinds of files a Manifold system consists of. We have assumed that the programming language to be used along with Manifold is C; however, as has already been stated, by virtue of the Manifold model any other "Manifold-compliant" language could be used.
In the `atomics` files code is generated as follows:

1. The `atomic` header, including the events raised on each `atomic` component:

ACME code	C code with Manifold Libraries

`Component X = {`
`Properties {`
`ComponentType:`
`Enum {ATOMIC_INT, ATOMIC_EXT,`
`MANIFOLD}=ATOMIC_INT;`
 	
`EventList : List =`
`<"event_1"..."event_n">;`
`}}`
	`void X(AP_Event event_1, ...`
	`AP_Event event_n) { }`
	`// creates the function skeleton`
	`// for manifold`

2. The `Port` definitions for the input and output ports of each `atomic` component:

ACME code	C code with Manifold Libraries
`Port X = { ...}`	`int X = AP_PortIndex("X");`

3. Functions for reading from specific ports (the example is for reading an integer from a port):

ACME code	C code with Manifold Libraries

Port X = {
Properties {
PortType : Enum {IN, OUT } = IN;
DataType : String = "int";} }
	int readInteger(int port){
	int XVal;
	AP_Unit unitX= AP_AllocateUnit();
	AP_PortRemoveUnit(port,&unitX,NULL)
	;
	AP_FetchInteger(unitX, &XVal);
	AP_DeallocateUnit(unitX);
	return XVal;}

4. Functions for writing to output ports (the example is for writing a string to an output port):

ACME code	C code with Manifold Libraries

Port X = {
Properties {
PortType : Enum {IN, OUT }= OUT;
DataType : String = "string";} }
	writeString(int port, char* str){
	AP_Unit unitX=AP_AllocateUnit();
	unitX = AP_FrameString(str);
	AP_PortPlaceUnit(port,unitX,NULL);
	AP_DeallocateUnit(unitX); }

In a similar manner, we construct the functions for reading other data types from ports and the corresponding functions for writing into ports.
5. The raising of events by means of the `AP_Raise` command:

ACME code	C code with Manifold Libraries

Component X = {
Properties {
EventList : List =
<"event_1"..."event_n">;
}}
	AP_Raise(event_1);
	...
	AP_Raise(event_n);

<u>In the `header` files code is produced as follows:</u>

1. The function prototype of each `atomic` (we produce a separate file for each atomic):

ACME code	C code with Manifold Libraries

Component X = {
Properties {
ComponentType:
Enum {ATOMIC_INT, ATOMIC_EXT,
MANIFOLD}=ATOMIC_INT;
 	
EventList : List =
<"event_1"..."event_n">;
}}
	# include "AP_interface.h"
	
	extern void X(AP_Event event_1, ...
	AP_Event event_n);

In the `Manifold` files code is produced as follows:

1. The `Main` Manifold file skeleton, the events and the processes definitions:

ACME code	Manifold code
`System X {`	`manifold Main()`
`Component Y = { Properties {`	`{ event ev_1, ev_2, ev_3.`
`EventList : List =`	`process pY is Y(ev_1, ev_2);`
`<"ev_1", "ev_2">;}}`	`process PZ is Z(ev_3);`
`Component Z = { Properties {`	`}`
`EventList : List = <"ev_3">;}}`	
`}`	

2. The file skeletons of manager processes including the interfaces of managers, definition of events raised by workers coordinated by managers, definitions of worker processes, definition of streams connecting ports of managers to ports of workers, and definition of states:

ACME code	Manifold code
`System A = {`	`export manifold X(event ev_1,`
`Component Type X= {`	`event ev_2)`
`Properties {`	
`EventList : List =`	
`<"ev_1", "ev_2">;}}`	
`ComponentType:`	
`Enum {ATOMIC_INT, ATOMIC_EXT,`	
`MANIFOLD}=MANIFOLD;`	
`}`	
`Port Xin = {`	`port in xin.`
`Properties {`	`port out xout.`
`PortType:Enum{IN,OUT} = IN;`	
`}`	`{`
`}`	
`Port Xout = {`	
`Properties {`	
`PortType:Enum{IN,OUT} = OUT;`	
`}`	
`}`	
`Representation = {`	`event ev_3, ev_4.`
`System X_rep {`	`process X_worker is`
`Component Type X_worker = {`	`X_worker (ev_3, ev_4);`

ACME code	Manifold code
`Properties {`	`begin:`
`EventList : List =`	`activate(X_worker);`
`<"ev_3", "ev_4">;`	
`}`	`/* if the 'begin'state involves`
`Port Xworker_in = {`	`the setting up of stream`
`Properties {`	`connections, then the involved`
`PortType:Enum{IN,OUT}=IN;`	`processes are activated */`
`}`	
`}`	
`Port Xworker_out = {`	
`Properties {`	
`PortType:Enum{IN,OUT}=OUT;`	
`}`	
`}`	
	`(Xin->X_worker.Xworker_in,`
`}`	`X_worker.Xworker_out->Xout).`
`}`	
`Bindings= {`	`ev_3:`
`Xin to X_worker. Xworker_in;`	
`X_worker.Xworker_out to Xout;`	`ev_4:`
`}`	
`}`	`}`
`}`	

3. All interfaces of processes that are coordinated by a manager process (this mapping can be applied for both the `Main` file and the managers' Manifold files):

ACME code	Manifold code
`Component X = {`	`manifold X(event event_1, ...`
`Properties {`	`event event_n)`
`EventList : List=`	`port in Xin.`
`<"event_1"..."event_n">;`	`port out Xout.`
`ComponentType:`	`atomic { internal. }.`
`Enum {ATOMIC_INT, ATOMIC_EXT,`	`/*`
`MANIFOLD}=ATOMIC_INT;`	`If` *`ComponentType=ATOMIC_EXT`*
`Port Xin = {`	`the keyword` *`internal`* `in the`
`Properties {`	`produced code is replaced with`
	`the keyword` *`external`*
`PortType:Enum{IN,OUT } = IN;`	
`}`	`If` *`ComponentType=MANIFOLD`*
`Port Xout = {`	`The last line of the produced`
`Properties {`	`code (` *`atomic {internal.}`*`)`
`PortType:Enum {IN, OUT } = OUT;`	`Is replaced with the keyword`
`}`	*`import`*
`}}`	`*/`

4. The state and streams definitions (this mapping can be applied for both the `Main` Manifold file and for the managers' Manifold files that coordinate a number of worker processes):

ACME code

```
Component X = {
 Port X1in, X2in, X1out, X2out};
Component Y = {
 Port Y1in, Y2in, Y1out, Y2out};
Component Z = {
 Port Z1in, Z2in, Z1out, Z2out};

Connector connXY1 = {
  Role caller, callee;
  Active_on:TupleList=<"ev_1,1">;
};
Connector connXY2 = {
  Role caller, callee;
  Active_on:TupleList=<"ev_1,1">;
};
Connector connYZ1 = {
  Role caller, callee;
  Active_on:TupleList=<"ev_2,1">;
};
Connector connYZ2 = {
  Role caller, callee;
  Active_on:TupleList=<"ev_2,1">;
};
Connector connZX1 = {
  Role caller, callee;
  Active_on:TupleList=<"ev_1,2">;
};
Connector connZX2 = {
  Role caller, callee;
  Active_on:TupleList=<"ev_1,2">;
};

Attachments {
  X.X1out to connXY1.caller;
  X.X2out to connXY2.caller;
  Y.Y1in to connXY1.callee;
  Y.Y2in to connXY2.callee;
  Y.Y1out to connYZ1.caller;
  Y.Y2out to connYZ2.caller;
  Z.Z1in to connYZ1.callee;
  Z.Z2in to connYZ2.callee;
  Z.Z1out to connZX1.caller;
  Z.Z2out to connZX2.caller;
  X.X1in to connZX1.callee;
  X.X2in to connXY2.callee;
};
```

Manifold code

```
manifold Main()
{
  process x1 is X().
  process y1 is Y().
  process z1 is Z().


ev_1: (x1.X1out-> y1.Y1in,
       x1.X2out-> y1.Y2in);
      (z1.Z1out-> x1.X1in,
       z1.Z2out-> x1.X2in).


ev_2: (y1.Y1out-> z1.Z1in,
       y1.Y2out-> z1.Z2in).
}

/*
If components in the left are
included in a representation of a
manager process, the Main
manifold interface (first line of
code produced) is replaced by the
specified manager's interface */

/* Note the use of the sequential
   operator ';'in the stream
   connections. The Active_on
   property indicates which
   component connections are
   performed in parallel and
   which ones sequentially by
   means of the <event,ordering>
   notation */
```

References

[1] AcmeStudio Homepage by Carnegie Mellon University, School of Computer Science. http://www-2.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html.

[2] ADML Home Page, by The Open Group. Available at http://www.opengroup.org/architecture/adml/adml_home.htm.

[3] S. Ahuja, N. Carriero, D. Gelernter, Linda and friends, IEEE Computer 19 (8) (1986) 26–34.

[4] J. Aldrich, C. Chambers, D. Notkin, ArchJava: Connecting software architecture to implementation, in: Twenty Fourth International Conference on Software Engineering, ICSE'02, 19–26 May 2002, Orlando, FL, USA, IEEE Press, 2002, pp. 187–197.

[5] F. Arbab, The IWIM model for coordination of concurrent activities, in: First International Conference on Coordination Models, Languages and Applications, Coordination'96, 15–17 April 1996, Cesena, Italy, in: LNCS, vol. 1061, Springer Verlag, 1996, pp. 34–56.

[6] S. Chachkov, D. Buchs, From abstract object-oriented model to a ready-to-use embedded system controller, in: Twelfth International Workshop on Rapid System Prototyping, 25–27 June 2001, Monterey, CA, IEEE Press, 2001, pp. 142–148.

[7] K. Everaars, F. Arbab, An introduction to the coordination language Manifold, Technical Report, June 2000.

[8] D. Garlan, Software architecture, in: Encyclopedia of Software Engineering, John Wiley & Sons Inc., 2001.

[9] D. Garlan, R.T. Monroe, D. Wile, ACME: An architectural description of component-based systems, in: Foundations of Component-Based Systems, Cambridge University Press, 2000, pp. 47–68.

[10] D. Garlan, R.T. Monroe, D. Wile, ACME: Architectural description interchange language, in: IBM Centre for Advanced Studies Conference, CASCON'97, Toronto, Canada, November 1997, pp. 169–173.

[11] A.A. Holzbacher, A software environment for concurrent coordinated programming, in: First International Conference on Coordination Models, Languages and Applications, Coordination'96, 15–17 April 1996, Cesena, Italy, in: LNCS, vol. 1061, Springer Verlag, 1996, pp. 249–266.

[12] P. Inverardi, H. Muccini, Coordination models and software architectures in a unified software development process, in: Fourth International Conference on Coordination Models, Languages and Applications, Coordination 2000, 11–13 September 2000, Limassol, Cyprus, in: LNCS, vol. 1906, Springer Verlag, 2000, pp. 323–328.

[13] D.C. Luckham, J.J. Kenney et al., Specification and analysis of system architecture using Rapide, in: Software Architecture, IEEE Transactions on Software Engineering 21 (4) (1995) 336–355 (special issue).

[14] N. Medvidovic, D.S. Rosenblum, R.N. Taylor, A language and environment for architecture-based software development and evolution, in: Twenty First International Conference on Software Engineering, ICSE'99, 16–22 May 1999, Los Angeles, CA, USA, ACM Press, 1999, pp. 44–53.

[15] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93.

[16] S. Mitra, S. Aggarwal, Infrastructure for the synchronization and coordination of concurrent Java component programs, in: Thirty Third Hawaii International Conference on System Sciences, HICSS-33, 4–7 January 2000, Maui, Hawaii, vol. 8, IEEE Press, p. 8052 (electronic version).

[17] R.T. Monroe, Rapid development of custom software architecture design environments, Ph.D. Thesis, Carnegie Mellon University, 1999.

[18] P. Oreizy, N. Medvidovic, R.N. Taylor, Architecture-based runtime software evolution, in: Twentieth International Conference on Software Engineering, ICSE'98, 19–25 April 1998, Kyoto, Japan, IEEE Press, 1998, pp. 177–186.

[19] G.A. Papadopoulos, Distributed and parallel systems engineering in Manifold, in: Coordination, Parallel Computing 24 (7) (1998) 1107–1135 (special issue).

[20] G.A. Papadopoulos, F. Arbab, Coordination of systems with real-time properties in Manifold, in: Twentieth Annual International Computer Software and Applications Conference, COMPSAC'96, 19–23 August 1996, Seoul, Korea, IEEE Press, 1996, pp. 50–55.

[21] G.A. Papadopoulos, F. Arbab, Coordination models and languages, in: Marvin V. Zelkowitz (Ed.), Advances in Computers, vol. 46, Academic Press, 1998, pp. 329–400.

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

[23] V.C.C. de Paula, T.V. Batista, Mapping an ADL to a component-based application development environment, in: Fundamental Approaches to Software Engineering, FASE 2002, 6–14 April 2002, Grenoble, France, in: LNCS, vol. 2306, Springer Verlag, 2002, pp. 128–142.

[24] B. Schmerl, xAcme: CMU Acme extensions to xArch, http://www-2.cs.cmu.edu/~acme/pub/xAcme/, 23 March, 2001.

[25] S. Vestal, MetaH User's Manual, Version 1.27, HoneyWell Inc., Minneapolis, MN, 1998.