7th International Conference on Artificial Intelligence and Soft Computing (ICAISC'04), Zakopane, Poland, 7 - 11 June 2004

FOOD: An Agent-Oriented Dataflow Model

Nicolas Juillerat and Béat Hirsbrunner

University of Fribourg, Department of Computer Sciences, 1700 Fribourg, Switzerland,
{nicolas.juillerat,beat.hirsbrunner}@unifr.ch,
http://diuf.unifr.ch/pai

Abstract. This paper introduces FOOD, a new Dataflow model that goes beyond the limitations of the existing models, and targets the implementation of multi-agent systems. The central notion of FOOD, the *Dataunit*, which expresses all the dynamics that are required in multi-agent systems, is presented in details. Then it is shown how the FOOD model, despite its simplicity, has an expression power that goes beyond object-oriented languages in term of dynamics and mutability.

1 Introduction

Traditional Dataflow models have advantages over textual languages, such as their implicit parallelism, which makes them good candidates for the implementation of multi-agent systems. Unfortunately, most of the existing Dataflow models are also limited to flows of data, which are static entities that are unable to express the dynamics required by multi-agent systems [11].

The FOOD model that is presented in this paper extends the traditional, demand-driven Dataflow model with a new notion that is able to express the dynamics and mutability of agents, and overcomes the limitations of the existing models. The expression power of FOOD is then stated, by presenting how it can express the notions of object-oriented languages without adding them explicitly (unlike most existing object-oriented Dataflow models [2]), and how it goes beyond them by allowing controlled mutation of both data and functionalities.

2 The traditional Dataflow models

Dataflow languages are visual languages where the different steps of computation are represented visually by black boxes, the *Units*[1]; and the flows of data are represented by *connections* between Units. The figure 1 shows an example of a Dataflow and its components. The various existing Dataflow models and languages, their structures, semantics and applications are described in the literature [3, 4, 7, 9, 10, 12].

Despite their advantages, traditional Dataflow models also have serious limitations. Figure 2 shows both an advantage and a limitation of the traditional

[1] Also called *function boxes*, *components* or simply *operators* in the literature.

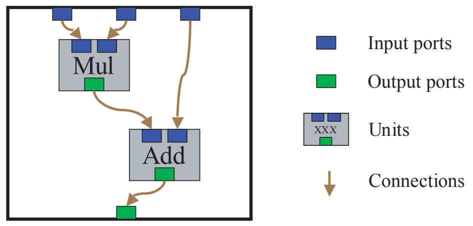

Fig. 1. A Dataflow and its components.

Dataflow models: The fact a Dataflow is a special case of a graph makes it possible for the topology of a Dataflow to match the topology of the problem it solves. However, distributed computing often implies high dynamics. The agent programming methodology reflect this requirement by encapsulating behaviors with data structures, i.e. using active entities. But in this example, an agent can only be implemented by a flow of data, a possibly structured, but passive entity. What we would like is to have flows of agents, that is, flows of entities consisting of *both data and functionalities*. We would also like agents to be able to adapt themselves, that is, both their data and functionalities should be *modifiable* at runtime. The FOOD[2] model presented in this paper addresses these wishes.

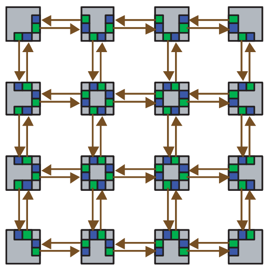

Fig. 2. A grid-like environment implemented by a Dataflow. Each cell is implemented by a Unit. Connections can transport the data representing an agent from one cell to another.

3 The FOOD model

While the traditional Dataflow models are based on the notions of Units, connections and data flows, a Dataflow in the FOOD model is built only with two notions: connections and *Dataunits*.

[2] First-class Object Oriented Dataflow

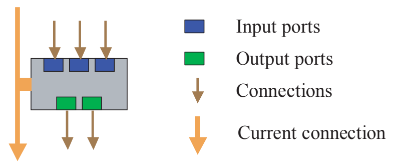

Fig. 3. A possible representation of a Dataunit call. The current connection holds the called Dataunit and the other connections supply arguments and collect results.

The notion of connection in FOOD is similar to that of traditional Dataflow models. The notion of Dataunit is between the notions of data and Unit, and has the following characteristics:

- Dataunits are used like data: they flow through the connections of the Dataflow, they can be used as argument to a function and returned as a result. Like data, Dataunits can only exist in connections and ports.
- Dataunits can be called (evaluated), like Units. As a Dataunit can only exist in connections and ports, a call to a Dataunit occurs in a connection that is holding it, which is named the *current connection*. Other connections are used like in the traditional Dataflow models to transmit arguments and results (which are Dataunits too). An example is shown in figure 3.
- Dataunits can be simple, in which case they have a single functionality, whose implementation is given by a *reference to a Dataflow*. But they can also be *structured*: a structured Dataunit is composed of zero or more fields that are Dataunits as well. Like with structured data, any field of a Dataunit can be replaced at runtime by another compatible Dataunit. A call to a structured Dataunit can only evaluate one of the simple Dataunits it contains and must therefore statically define which one. A possible representation of a call to a structured Dataunit is shown in figure 4.
- Dataunits, like data structures, are typed. The type of a simple Dataunit is given by the types of its input and output ports. The type of a structured Dataunit is given by the types of all its fields.
- Assignment compatibility between Dataunits is defined as follow: a Dataunit is assignment compatible with another one if they have the same type, or if the later *contains* a field that have the same type as the former. This definition is less restrictive than the common definition of compatibility between structured data types.
- Dataunit can optionally have a static property: the *enclosing type*. A Dataunit that has an enclosing type can only be used as a *field* in a structured Dataunit whose type is compatible with the enclosing type. This Dataunit is the *enclosing Dataunit* and can dynamically change at runtime. While a

Datunit without an enclosing type can only access its own fields, a Dataunit with an enclosing type can also access the fields of its enclosing Dataunit.[3]

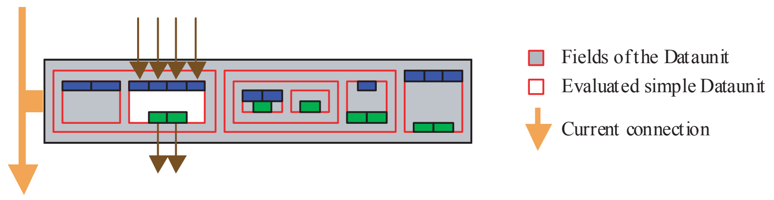

Fig. 4. A possible representation of a call on a structured Dataunit. In this representation the entire structure of the Dataunit is displayed.

4 The expression power of FOOD

In this section, we first show that the notion of Dataunit in FOOD can be used to represent, or simulate an object and therefore that FOOD expresses all the semantics of object-oriented languages. In the second part we present other possibilities of FOOD that can not be expressed directly by most object-oriented languages[4] but are required for the implementation of multi-agent systems.

4.1 Object-oriented semantics

A Dataunit is composed of fields that are Dataunits themselves. This is the only way of structural composition that exists in FOOD. An object, on the other hand, is structured in four ways that all have different restrictions and characteristics [8]: an object has *fields*, *parents* (inheritance), *methods* and *inner classes*. Fields and inheritance are *mono-directional* compositions: an object has access to its parents and fields, but the reverse is not true. Methods and inner classes are *bi-directional* compositions because they both have access to their enclosing object. Inheritance, methods and inner classes are all *static* compositions, i.e. they can not be changed at runtime. Fields, whose values are modifiable, therefore constitute the only dynamic composition in object-oriented languages.

Fields are implicitly expressed by ports and connections in FOOD, just like variables in traditional Dataflow models [4, 6]. The three other ways of composition in object-oriented languages can be expressed using the single way of

[3] The assignment compatibility between Dataunits includes the compatibility of the enclosing types, if any. The absence of enclosing type is defined to be compatible with any enclosing type.

[4] Except, in a limited way, using Aspect programming [1], which goes in a similar direction.

composition of the Dataunits in FOOD: an object that inherits one ore more parent objects is expressed by a Dataunit that is composed by one or more "parent" Dataunits that have no enclosing type. The methods and inner classes of an object are expressed in the same way, but in that case, the enclosed Dataunits (expressing the methods or inner classes) have an enclosing type that matches the type of the enclosing Dataunit (expressing the enclosing object). Therefore, they can access the fields of the enclosing Dataunits, just like a method or an instance of an inner class can access the fields and methods of their enclosing object.

Recall that in FOOD, each field of a Dataunit can be replaced at any time with a compatible one. Overriding a method of a parent object can therefore be expressed in FOOD by replacing the field corresponding to the method in the field corresponding to the "parent" Dataunit by a new compatible Dataunit that implements the new version of the method. The current connection, which is associated to each Dataunit call, can be used to express the notion of the current object. As this connection holds a structured Dataunit, and that the parent objects are expressed by fields, they can be accessed as well.

A first interpreter of a language following the FOOD model has been realized, with functionalities to convert any Java object into an equivalent Dataunit (as discussed above) whose implementations are provided by the methods of the underlying Java object. Work is still in progress to provide Dataunits with implementations based on real Dataflows.

4.2 Mutability semantics

The only way of composition used in FOOD allows any field to be replaced at *runtime*, therefore FOOD can express even more dynamics than object-oriented language: each functionality of a structured Dataunit can be modified at any time by replacing the corresponding field with another compatible Dataunit that provides a different implementation. The use of fields for composition, static typing and compatibility rules, ensure that these "mutations" are not chaotic. Other programming practices such as self-modifying code can express the same level of mutability but are very hard to control and are only possible in assembly languages and, surprisingly, in LISP[5]; these practices are usually strongly discouraged [5]. The main difference between the FOOD approach and self-modifying code is that in FOOD only Dataunits are mutable; that is, only *references* to Dataflows are mutable; the Dataflows themselves are not.[6]

If it were possible to express the possibilities of FOOD in an object oriented language, like we have previously transposed the possibilities of object-oriented languages to FOOD, we would have an object oriented language with facilities

[5] LISP can manipulate lists, and then evaluate a list as if it were a LISP expression using the **eval** keyword

[6] An indirect, but crucial consequence is that FOOD is a Dataflow model where, although the Dataunits are passed by value, their real contents (that is, their implementations, i.e. the referenced Dataflows) are in reality passed by reference.

such as: replacing one of the parent of an object at runtime by another one that is compatible, or replacing the method of an object at runtime by another one with the same signature.

5 Conclusion

The FOOD model inherits the advantages of traditional Dataflow models, which makes it a well-suited model for the implementation of distributed applications. But it also overcome their limitations by introducing a new notion, the *Dataunit*, which can express all the dynamics and mutability that are necessary to implement the notion of agent. The expression power of the Dataunit has been proved to go even beyond the possibilities of object-oriented languages for capturing some advanced features of multi-agent systems, namely structural dynamics, adaptability of behaviors and composition [11]. This makes FOOD a new and modern computation model of choice for the implementation of multi-agent systems, in particular dynamical and re-configurable systems.

References

1. The AOSD Steering Committee: Aspect-oriented Software Development, `http://aosd.net/` (last visited on January 2004)
2. M. Burnett, A. Goldberg, T. Lewis: Visual Object-Oriented Programming, Manning Publications (1995)
3. M. Burnett et al.: Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm, CiteSeer Scientific Literature Digital Library (2001)
4. M. Boshernitsan, M. Downes: Visual Programming Languages: A Survey, CiteSeer Scientific Literature Digital Library (1997)
5. Commodore Business Machines: Amiga ROM Kernel Reference Manual: Libraries, Addison-Wesley, 3rd edition (1991)
6. G. Gao, L. Bic, J.-L. Gaudiot: Advanced Topics in Dataflow Computing and Multithreading, Wiley-IEEE Computer Society Press (1995)
7. D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, K. Doyle: Fabrik, A Visual Programming Environment, ACM/SIGPLAN 00PSLA '88 Conference Proceedings, 23, (1988) 176–190
8. B. Meyer: Object-Oriented Software Construction, Prentice Hall, 2nd edition (2000)
9. R. Mark Meyer, T. Masterson: Towards a better Programming Language: Critiquing Prograph's Control Structures, The Journal of Computing in Small Colleges, Volume 15, Issue 5 (2000) 181–193
10. K. Pingali and Arvind: Efficient Demand-Driven Evaluation, ACM Transactions on Programming Languages and Systems, Volume 7, Issue 2 (1985) 311–333
11. A. Tafat-Bouzid, M. Courant and B. Hirsbrunner: A Coordination Model for Ubiquitous Computing, Proceedings of the 3rd WSEAS International Conference on Multimedia, Internet and Video Technologies (2003)
12. W. Wadge, E. Ashcroft: Lucid, the Dataflow Programming Language, Academic Press (1985)