Data & Knowledge Engineering 67 (2008) 293–329

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Twenty-five years of Mokum: For 25 years of data and knowledge engineering: Correctness by design in relation to MDE[1] and correct protocols in cyberspace

Reind P. van de Riet *

Emeritus, Department of Computer Science, Vrije Universiteit in Amsterdam, Department for Informatic, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Available online 29 April 2008

Keywords:
Model driven architecture
Knowledge base systems
Correctness of communication protocols

ABSTRACT

The use of the Mokum system for correctly designing Information Systems (in the old days called **Data** Base Systems) and how these designs can automatically be translated into implementations, is one theme of this paper. Theoretical considerations will not be avoided. Similarities and differences with tools from the field of Model Driven Architecture, such as UML, will be demonstrated. How to deal with Mokum in a situation with many sites, as in real Cyberspace is demonstrated as a second theme. Special attention is given to a protocol: MAP, which guarantees that a certain access constraint is being maintained. The constraint being that for an object in one site to obtain access to an attribute of an object in another, a third site must give permission. The technique is based on a new notion: the Triple,[2] to be used in a **Knowledge** based **Engineering** environment.

0. Introduction

In the old days one believed in the possibility to design correct Information Systems (IS) and to implement them. Currently, one is more realistic, is more concerned with tools to find errors and to remedy these systems.

The Mokum system is from these old days (see also [16,18]); the acronym MOKUM stands for Manipulating Objects with Understanding in Mokum. The latter "Mokum" stands for "Amsterdam", as it is its nick name. In Appendix B we will explain a bit more about this name. We want to show in this paper what the principles are of this system, and why we believe that these principles are still valid, because one can automatically generate an implementation from a design in the form of a few diagrams, which can be shown to be semantically meaningful and correct with respect to access control. In the first chapter we will deal in great detail with aspects of the Mokum system itself; in a second chapter we will demonstrate the connection with Model Driven Engineering/Architecture (MDE) and in a third chapter the protocol for the communication between Mokum sites in Cyberspace will be studied from the aspect of Security and Privacy. In the first chapter the notion of type *tree* will be introduced; we will see that *triggers* are the main theme in the second chapter and that *triples* will have the same significance in the third chapter.

So the paper will be about **trees, triggers and triples in Mokum.**

* Tel.: +31 20 4447757; fax: +31 20 4447653.
E-mail address: vdriet@cs.vu.nl

[1] The acronym MDA is a trademark of the OMG; MDE is the alternative.

[2] "Triple" may have a connection with the author's farewell lecture: "Does 3 exist?" in 2000.

0169-023X/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2008.04.006

The following principles are being used:

1. **The IS encompasses all players in the field**, not just the data. So in a hospital IS, these are the patients, the doctors, the nurses, the financial people, etc. In an IS for a garage, not only cars to be repaired or sold are represented but also the technicians, the selling people, their bosses, the managers and the administrative employees.The ground for this principle is that we want to be able to prove correctness of access and usage of software tools.
2. **Design and implementation are one and cannot be considered independently**.
3. **All players and data to be defined are ordinary objects**, in the realm of Object-Oriented programming.One final remark: we will talk about "Mokum" when we mean the ideas and principles of Mokum; we will talk about a "Mokum system" when we mean an implementation of a real system; we will also talk about a "Mokum system" for a particular application, which actually could be better named a "Mokum IS", but unfortunately we will use the simple term "Mokum system" or even "Mokum" and hope that the context makes clear what we mean.

1. The Mokum system

1.1. Introduction to Mokum

During design phase and implementation phase the **type tree** is of crucial importance. The type tree is a tree in which all the types of the objects to be considered are defined. It is used to define access from one object to another, and also to pieces of software.

This last criterium is translated in the rule that an object can only execute a piece of software in the form of a **trigger**. All triggers are defined within a type(s) of the object and react upon a message sent by another object (or itself). Dependent on the state an object is in it will/can react. Suppose it is asleep, then only a timer message can awake it. Suppose it is waiting for a reaction of a special object, it may be only susceptible to a message of this other object by being in another state.

We will see that it is (in principle) easy and straightforward to automatically translate in Mokum a design, in the form of WorkFlow diagrams, on a high level into the definitions of these triggers. That means that with a correct design comes a correct implementation.

How do we achieve the goal of having a correct design? There are two reasons: due to the simple principles upon which access control is based, it is possible to use simple tools (e.g. in Prolog) to prove that access w.r.t. security and privacy is guaranteed; the second is that we use semantic knowledge, in the form of Ontologies, to validate the design. This validation does not have the power of a proof of correctness, but it helps to ascertain a high quality of a design.

As said above, Mokum is an object-oriented system, with special notions to handle access control. There is a type hierarchy, with the usual inheritance of properties. Each type has a number of attributes, which themselves may be of simple nature, such as text and integers, or of a newly defined type, such as for the father of a person. In addition a type has several states and a script, in which triggers are defined which react on incoming messages, by changing states, values of attributes, and sending messages to (other) objects. There is the notion of time, so that timers can be set. The `is_a` relationship defines the inheritance of properties. If T `is_a` S then type T has all the properties of type S, not only attributes but also states and script. So an object being an instance of type T is automatically an instance of type S. All types are `is_a` related to the special type *thing*. The only property of *thing* is that it defines unique identifiers for its instances, the so-called object identifiers: oid. As has been said above, all types can be put in a tree, the **type tree**, structured according to the `is_a` relationship as hierarchical principle. In the diagram for the type tree the `is_a` relationship is indicated by means of an arrow with a closed head. The `is_a` relationship is transitive.

What makes Mokum special is the notion of **collection** and its **keeper**. In order to make it possible to define access control, these notions have been introduced. Suppose that an object, like a doctor in a hospital system, needs single access to a set of other objects, of a certain type, in this case type patient, we make this object keeper of that set. So in the example case: we make the doctor keeper of the collection of (his or her) patients, to which only he or she has access. To indicate the relationship between keeper and collection, there is a special relationship between types the **`coll_of`** relation, indicated as follows: suppose the keeper has type T and the collection is a set of objects of type S, then in the type tree there is a special arrow from T to S, an arrow with open head. The `coll_of` relationship is non-transitive. In a case where there is only one `coll_of` relationship for a certain type, the drawing of the `coll_of` arrow is enough, otherwise each `coll_of` relationship needs to get a unique name. In principle, a compiler can now already prevent many accesses: when object O_T, as an instance of type T, wants access to an attribute A of object O_S, as an instance of type S, and T and S are not `is_a` or `coll_of` related, then access can immediately be denied. We say that A is not visible to T and therefore to O_T. Of course, one has to take into account that the `coll_of` relationship can be inherited:

$$T\ \textbf{is\_a}\ T_1\ \textbf{is\_a}\ T_2\ \textbf{is\_a}\ \ldots T_n\ \textbf{coll\_of}\ S_1\ \textbf{is\_a}\ S_2\ \textbf{is\_a}\ \ldots S$$

So an algorithm for visibility is (note that the basic facts all have a "_" in their names and are simply given beforehand, e.g. in a diagram; in Prolog "," means: "go on if OK" or "and", ";" means "or", or "else" and "." "end of defintion")

```
isa(T,S):- T=S;(is_a(T,Tl),isa(Tl,S)).
attr(A,T):- isa(T,S),attr_of(A,S).
```