The Power of Methods With Parallel Semantics*

Karl Denninghoff
CSE C-0114
UC San Diego
La Jolla, CA 92093, USA
denning@cs.ucsd.edu

Victor Vianu
CSE C-0114
UC San Diego
La Jolla, CA 92093, USA
vianu@cs.ucsd.edu

Abstract

A model capturing the data manipulation capabilities of a large class of methods in object-oriented databases is proposed and investigated. The model uses a deterministic, parallel synchronous semantics with concurrent-read and concurrent-write. The results focus on the expressive power of methods and help understand various constructs and semantics associated with methods. Restrictions of methods providing various tractability guarantees are also discussed. The restrictions correspond closely to well-known relational query languages such as *relational calculus, Datalog,* the *fixpoint* queries, and the *while* queries. They provide complexity bounds such as constant parallel time, PTIME and PSPACE. Exact characterizations for some complexity classes are also obtained under certain assumptions. Our methods provide a model of database parallel computation which makes explicit the potential parallelism in databases. We compare our model to traditional parallel computation models such as PRAMs and Hardware Modification Machines and show mutual simulation results with reasonable cost. We also compare methods to a newer model of generic computation involving parallelism. We show that certain complexity classes defined using the two models are the same, which suggests that methods capture database parallel computation in a natural and robust fashion.

1 Introduction

Behavior encapsulations using methods is one of the important features of object-oriented databases. Methods provide a new programming paradigm which integrates data and computation. While methods are extensively implemented in object-oriented database systems [B88,A+90], there has been little study of their semantics and computational properties. This paper contributes towards a formal framework for understanding methods in object-oriented databases. We propose and study a model for deterministic methods, with parallel synchronous semantics. The model abstracts the data manipulation capabilities of such methods, much like relational calculus and algebra abstract the data manipulation capabilities of relational database systems. The results help understand various constructs and semantics associated with methods, particularly their impact on expressive power.

In the model we propose, each method is a simple line program which performs straightforward manipulations of printable constants and pointers to other objects. As in the relational model, values are uninterpreted and there is no computation on the values themselves (methods can only test for (in)equality among values). Individual methods contain no recursion. They can send messages to other objects, either by broadcasting to all objects of a given type, or by following pointers to other objects. The messages result in method invocations, and the call graph can be recursive. Methods can also create and initialize new objects. The semantics is a parallel synchronous semantics. Multiple computation threads associated with one object are allowed, with concurrent-read and concurrent-write (CRCW) semantics.

While parallelism has long been an underlying concern in databases, the model proposed here comes the closest so far to a true database parallel computation model. It captures explicitly the potential for parallel computation in databases. Of course, the model does not necessarily assume a truly parallel implementation. It can be implemented in a massively parallel environment, but also in a sequential environment which simulates true parallel semantics.

The results concern primarily the computational capabilities of the method paradigm. The focus is on the impact of various constructs and semantics

*Work supported in part by the National Science Foundation under grant IRI-8816078.

on expressive power. In order to evaluate formally the expressive power, we provide notions of computational completeness and compare the power of methods to that of classical relational query languages. This is done by mapping object-oriented database instances to relational representations, and conversely. The comparison is not straightforward due to the mix of constants and object identifiers in object-oriented databases. In particular, the ability of methods to create new objects complicates the comparison. For instance, there is a distinction between completeness of computation producing only printable constants, and computation producing new object identifiers in the result (for the latter, we adopt the notion of completeness proposed in [AK89]). In particular, we show that methods are complete for the first case, but not the second. This highlights important computational differences between pure value models and models with object identity. We also show that the power of the language is increased if complex objects allowing grouping of values into sets are used.

A second group of results involve restrictions of the model which provide tractability guarantees. Thus, we provide restrictions which limit the complexity of computations to constant parallel time, monotonic-PTIME, PTIME, and PSPACE. We show close connections with well-known relational query languages which provide similar tractability guarantees: relational calculus, *Datalog*, the *fixpoint* queries, and the *while* queries of [Ch81]. In particular, we obtain restrictions expressing *precisely* the PTIME and PSPACE transformations, with the additional assumption that objects have integer id's which can be compared by methods[1]. The restrictions considered concern: recursion, how new objects are created, bounds on the number of alternations of insertions and deletions from a type (a sort of stratification condition), the ability to modify values of attributes, and tests for inequality of values.

As mentioned above, our methods provide a model of database parallel computation. We compare the method model to traditional models of parallel computation, primarily the classical CRCW-PRAM [Pa87]. We also compare methods to the Hardware Modification Machine (HMM) of [Co81], since HMMs are closer in nature to methods. We provide mutual simulation results with PRAMs with reasonable cost (logarithmic time and constant space blowup). This shows that classical parallel complexity classes are closely related to corresponding classes defined wrt the method model. A primary difference with classical models is that methods treat objects *generically*, i.e. objects which are undistinguishable from each other in the database are always treated uniformly. In contrast, most traditional models provide a way of breaking symmetry. The distinction disappears if such a mechanism is provided in the database, for instance if object id's are integers which can be compared. With this assumption, complexity classes on PRAMs are generally the same as those defined wrt methods . In particular, this holds for the well-known class *NC* of "tractable" problems wrt to parallel time and space complexity (see [Pi79]). Lastly, we also compare the method model to a model which, unlike PRAMs or HMMs, captures generic computation. The model, introduced in [AV91] and called Generic Machine, involves parallelism. We show that PTIME and PSPACE complexity classes defined based on the two models coincide. This suggests that the method model provides an alternative robust basis for understanding the complexity proper to database parallel computation, even beyond the object-oriented framework.

There have been few previous formal studies of methods. The investigation that comes closest to ours is that by Hull and Su [HS89]. They also propose a formal model for methods, which captures data manipulation aspects, and look at issues of expressive power and complexity. However, the approach is fundamentally different. Indeed, their semantics of methods is non-deterministic, whereas ours is a parallel deterministic semantics. Hence, most of the results in this paper and [HS89] are incomparable. Furthermore, methods in [HS89] are used in conjunction with an external language, while we focus on the power of the methods themselves. The results in [HS89] emphasize queries, whereas we look at transformations of database states which emphasize behavioral aspects, and raise problems of a different nature. Finally, many of the results in [HS89] concern complex objects, which we do not consider here except incidentally.

Other investigations of methods have been primarily related to typing issues. For instance, [AKW90] discusses compile-time detection of typing errors. Methods in this model are uninterpreted functions from types to types. The model retains only the typing information about methods. The results concern the tractability of compile-time typechecking with various assumptions on the schemas.

The paper is organized as follows. The model is presented in Section 2. The expressive power of methods is discussed in Section 3. Section 4 presents the restrictions on methods which provide various tractability guarantees. Connections with relational query languages are also shown. Section 5 discusses the relation of our model to PRAMs and other mod-

[1] This assumption is similar to the ordered domain assumption used sometimes with relational query languages.

els of parallel computation, and Section 6 looks at the connection with Generic Machines. Due to space constraints, the presentation is mostly informal in this abstract. More details are provided in [DV91].

2 Database Method Schemas

In our object-oriented database model, a schema consists of a finite set of *types*. Each type has a structural and a behavioral component. Types have names which identify them uniquely. Type names are denoted by $P, Q, R, T, \ldots$. Since the behavioral component of types is specified using methods, our schemas are referred to as *database method schemas (dms)* [2]. We first describe informally the structural component of types, then the behavioral one.

The structural component of types provide a "barebones" object-oriented specification mechanism. It specifies, for each type, a finite set of attributes and their *sorts*. Each attribute is of one of the following sorts: (printable) *constant*, or *pointer* to an object of a type in the schema. Attributes are denoted by $A, B, C, \ldots$,the constant sort by *const*, and the pointer-to-a-type-Q sort by $@Q$. For instance,

$$T : [A : const, B : const, C : @Q]$$

defines the structural component of a type T, assuming that Q is a type in the schema. Note that the model does not provide explicit inheritance, which is orthogonal to the issues discussed here. Neither does it provide complex objects.

The behavioral component of a type provides a finite set of *methods* attached to the type. The methods attached to a type are said to be *owned* by the type. A method can be owned by several types. Methods are line programs of simple instructions, whose syntax and semantics are described later in this section.

A type consists of a name and a pair $< S, B >$, where S is the structural specification for the type, and B is the finite set of methods owned by the type. Given a type $T =< S, B >$, we denote its structural component S by $str(T)$. Given a dms M, we denote by $str(\mathrm{M})$ the set of structural specifications of its types.

Instances of schemas are defined wrt the structural components of the types of the schema. In defining instances, we distinguish between object identifiers and printable constants. Let $\mathcal{I}$ be an infinite set of symbols called *object identifiers (id's)* and $\mathcal{C}$ an infinite set of symbols called *constants*. $\mathcal{I}$ contains a special value called *nil*. Consider a dms M. An instance I of $str(\mathrm{M})$ consists of:

- a mapping associating to each type T of M a finite subset T_I of $\mathcal{I} - \{nil\}$, and
- for each attribute A of the schema, a mapping which associates to each object i in T_I for some T in M with attribute A, a constant or id $i.A$ of the appropriate sort; if A is of sort $@P$, then either $i.A = nil$ or $i.A$ must be in P_I.

Note that the definition of instance allows distinct objects with precisely the same values for all attributes. Thus, the identities of objects are determined by their id's alone. Also, note that an object can belong to more than one type in a given instance. We refer to the membership of an object in a type as a *role* of the object. However, note that if the same attribute A occurs in more than one type containing a given object id i, the value of $i.A$ is the same regardless of the type. To emphasize the fact that the definition of an instance requires consistency conditions as above, we sometimes also refer to an instance as a *consistent* instance.

The set of instances of $str(\mathrm{M})$ is denoted $inst(str(\mathrm{M}))$. For brevity, we also use the notation $inst(\mathrm{M})$ whenever convenient.

We next describe the syntax and semantics of methods. A method owned by a type T is a procedure which can run on all objects of type T in any given instance. Methods run on objects *in their role as members of a given type*. We refer to an execution of a method on a given object (in its role as a member of a type) as a *thread* of the method. A method thread running on an object can only access directly the values of its parameters and declared variables, the id of the object (denoted *self*), and the values of attributes of the object (denoted *self.A*, *self.B*, etc). A method can do the following: test for equality and inequality among values to which it has direct access; transfer values among the variables and attributes it can directly access; send messages invoking methods of other objects, passing along some values as parameters; create and initialize a new object; create a role of *self*, as a specified type; delete itself.

We wish to capture methods which are deterministic and generic (i.e., they treat uniformly objects which are undistinguishable in the database). This naturally leads to a parallel, synchronous semantics for methods. A computation is initiated by an external call which consists of invoking a method, simultaneously, for all objects in some type owning the method. This generates one computation thread for each object of that type in the instance. In the course of the computation, objects invoke methods of other objects. In general, one or several methods of an object can be invoked simultaneously by distinct objects. Each such invocation results in a separate

[2] The term *method schema* was first used in [AKW90] with a different meaning.

computation thread. Thus, for a given object, there may be several methods, and several threads of the same method, running simultaneously. In this case, variables of the method are duplicated in each thread, and no conflict can arise by assignments to these variables. However, conflicts can arise if there are simultaneous attempts to change an attribute value of a given object to distinct values. This yields a run-time error which crashes (or makes undefined) the entire computation. This is a usual concurrent-read concurrent-write (CRCW) semantics.

We next describe methods in more detail. Let M be a dms. A method owned by a type T in M is identified by a *header* providing a method name and a list of parameters of the sorts *const* or $@P$ for some type P in M. Parameters, and variables local to the method, are denoted $x, y, z, \ldots$. For instance,

$$m\ (x : const,\ y : @P)$$

is a method header (for method m), if P is a type in the schema. The variables x and y are parameters whose values are supplied when the method is invoked. (An invoked method never returns values to the object invoking it.) The header may be followed by a statement declaring variables local to the method, of the form

$$var\ \ x_1 : sort_1,\ x_2 : sort_2, \ldots, x_k : sort_k;$$

where $sort_i$ is *const* or $@P$, where P is a type of the schema. All variables occurring in the method are either parameters or declared variables. The header and declaration statement are followed by the *body*, consisting of a finite sequence of statements. We next describe the statements allowed in methods and discuss their semantics. Below, l, r and the x_i may be variables, constants in C, the id value *nil*, or *self* or *self.A*, where *self* denotes the id of the object running the method, and A is an attribute of its type.

1. $l := r$;

   The assignment statement has the obvious semantics: l is assigned the value of r. Here l cannot be a constant, *self*, or *nil*. Note that, if a method m owned by type T modifies the value of some attribute A of an object with id i belonging to T, the modification occurs in all types to which i belongs (indeed, there is only *one* object with id i, although it may belong to several types). If simultaneous attempts are made to assign different values to the same attribute of one object, the computation crashes (the result is undefined).

2. *if condition then statement*;

   where *condition* is a boolean combination of tests of the form $x_1 = x_2$ or $x_1 \neq x_2$, and *statement* is not another *if* statement. This tests for (in)equality of the values of x_1 and x_2.

3. $[l :=]\ new_T(A_1 : x_1, \ldots, A_n : x_n)$,

   where the A_i are the attributes of T. This creates a new object of type T and initializes its attributes. The id of the new object may be assigned to l. The id is an arbitrary value from $\mathcal{I} - \{nil\}$, not occurring in the current instance. In general several such statements may be executed simultaneously, since several methods may be running in parallel. Then, we allow two possible semantics. The default semantics is that each *new* command results in the creation of a separate object, with a distinct id. However, for reasons discussed later, we provide the option of a different, "value-oriented" semantics, where only one object is created for distinct *new* commands with identical attribute values in the initialization, and no object is created if another object with the same attribute values already exists in the type. To indicate the latter semantics we use the notation

   $$[l :=]\ \ new^{val}_T(A_1 : x_1, \ldots, A_n : x_n).$$

   If no object is created and the assignment to l is present, l is assigned the value *nil*. If simultaneous attempts are made to assign different values (new id's or *nil*) to the same attribute of an object, the computation crashes (the result is undefined).

4. $send_{destination}\colon\ m(x_1, \ldots, x_n)$,

   where m is a method name, and the x_i provide values to the parameters declared in the header of m. The *destination* can be: (i) a type T owning method m, (ii) a variable of type $@T$ where T owns m, (iii) *self* if m is owned by the type of the object running the method, (iv) $x.A$ or $self.A$, if A is of sort $@T$ for some type T owning m. In cases (ii)-(iv), where the message is sent to an object of a specified type, only the method m owned by that type is executed. Thus, the message always affects the destination object in its role as a member of the specified type, although it may belong to several types. If the specified destination object does not exist in the destination type, the computation crashes (the result is undefined). Once again, simultaneous invocations can result in multiple threads of the same method running on an object, with CRCW semantics. As for the *new* command, we provide

a "value-oriented" version of the *send* command, denoted

$$send^{val}_{destination} : m(x_1, ..., x_n).$$

The semantics is that simultaneous invocation of a method on an object with identical parameter values results in a *single* computation thread.

5. $role_T(A_1 : x_1, ..., A_n : x_n)$,

   where T is a type in the schema, with attributes A_i. This inserts into type T the id *self*; the attribute values are initialized by the x_i. If the value of some attribute of *self* becomes ill-defined as a result of simultaneous inconsistent *role* commands or because *self* already exists in the type T with different attribute values, the computation crashes (the result is undefined).

6. *delete-self*;

   This removes the object from the type running the method and halts any computation threads running on the object in its role as a member of the type. If referential integrity is violated as a result of a deletion, the computation crashes (the result is undefined).

In addition, there are simple typing rules which prohibit assigning values of sort *const* to variables or attributes of sort $@P$, and conversely. However, assignments among different pointer sorts are allowed.

It is assumed that methods run synchronously. All statements take one unit of time. We elaborate this for statements 2 and 4. If statement 2 is run at time t, evaluating *condition* takes one unit of time[3]; if *condition* holds, *statement* is executed at time $t + 1$ and the instruction following the *if* statement is executed at time $t + 2$. If the test is not satisfied, the next instruction is executed at time $t + 1$. In statement 3, if the destination is a type, then the procedure invocation is broadcast to all objects of that type. If the destination is a particular object id, that object alone receives the message. If statement 3 is executed at time t, we assume the first statement of an invoked method is executed at time $t + 1$; the instruction following the *send* instruction is also executed at time $t + 1$ (thus, a method invoking other methods using *send* does not wait while the invoked methods are run, and no values are returned).

Clearly, programs are very sensitive to timing. Also, all programs interact, so they cannot be written independently. A real programming language based on this paradigm is likely to provide additional constructs which would render the programming task easier. We do not explore this issue here. However, relaxing the synchronicity while preserving determinism is an important and difficult problem.

As noted above, a computation of a dms may lead to inconsistent instances, and thus to undefined results. Possible inconsistencies are:

- violations of referential integrity due to deletions, or insertions by the *role* and *new* commands;
- crashes due to a CRCW conflict arising fom inconsistent simultaneous assignments, inconsistent *role* commands, or *send*s to non-existing destinations.

It can be shown that it is undecidable whether a given dms can lead to an inconsistency as above. While sufficient conditions ensuring consistency can be found, we do not explore them here. In the general case, consistency is the programmer's responsibility. However, the following important fact can be shown. Suppose that we allow intermediate inconsistent states which do not cause the computation to crash, as follows: violations of referential integrity are allowed, and types can contain several inconsistent versions of the same object as a result of *role* commands or CRCW conflicts arising from assignments to an attribute (in the latter case, one version of the object is included in the type for each value assigned to the attribute). Also, suppose *send* commands to non-existing destinations have no effect instead of crashing the computation. The following shows that allowing inconsistent intermediate states does not provide any additional computational power.

Lemma 2.1 Every transformation on instances expressible by a dms with inconsistent intermediate states is expressible by some dms such that all intermediate states in the computation are consistent.

Thus, no expressive power is lost by disallowing inconsistencies in intermediate states. Throughout the paper, we disallow inconsistent intermediate states in dms computations.

As mentioned above, a computation is started by an external invocation of some method of M, broadcast simultaneously to all objects of the type owning the method. We assume that no other external invocation is allowed before the computation triggered by the previous external invocation ends. Each external invocation defines a transformation on instances of M, i.e. a mapping on $inst(M)$. The semantics of a method invocation m is the transformation it defines,

[3] We could have assumed that each comparison takes one unit of time. This is a minor variation which does not affect the results.

called the *effect* of m and denoted $eff_{\mathbf{M}}(m)$. The semantics of a dms $\mathbf{M}$ can be viewed as the sum-total of the semantics of all invocations of its methods.

In general, we assume that, in addition to the types of the schema which are visible to users, there are other types which are "hidden", employed by methods for internal bookkeeping purposes. As we shall see, the use of such hidden types is essential for the expressive power of dms's. In general, we are only interested in the transformations defined on the visible types. Given a dms $\mathbf{M}$ containing another dms $\mathbf{V}$ (the visible portion), we define the semantics of an invocation m wrt $\mathbf{V}$ by the restriction to $\mathbf{V}$ of $eff_{\mathbf{M}}(m)$ (inputs over $\mathbf{V}$ are extended to inputs over $\mathbf{M}$ by making the types in $\mathbf{M} - \mathbf{V}$ empty). This is called the *effect of m wrt* $\mathbf{V}$, and denoted $eff_{\mathbf{M},\mathbf{V}}(m)$. Let S be the structure of a dms. A transformation τ on $inst(S)$ is said to be *expressible by a dms* if there exists a dms $\mathbf{M}$ and an invocation m of a method in $\mathbf{M}$, such that $S \subseteq str(\mathbf{M})$ and $\tau = eff_{\mathbf{M},\mathbf{S}}(m)$. If $S = str(\mathbf{M})$ (i.e. $\mathbf{M}$ uses no additional types), the transformation τ is said to be computed *in-place* by $\mathbf{M}$. We will elaborate on the expressiveness of "in place" computation later. We just mention here that the "in place" requirement generally restricts the expressive power of dms's.

Following is an example of a dms computing the transitive closure of a graph of objects.

Example 2.2 The following dms computes (in logtime) the transitive closure of a graph of objects whose nodes are in type N and edges in type E. To cause the transitive closure to be computed, the method *trans*() is externally broadcast to the type E. The structural and behavioral components of the schema are described below.

Type structures:

N:[ ] E:[A : @N , B : @N]

Methods for E:

trans()
  $send_E$: *connected*(*self.A* , *self.B*)

connected(x : @N , y : @N)
var z: @E
  if x = *self.B then*
    $z := new_E^{val}$(*self.A* , y)
  if x = *self.B* $\wedge$ $z \neq$ *nil then*
    $send_E$: *connected*(*self.A* , y)

□

We next elaborate briefly on some of the choices made regarding the semantics of methods. They concern the creation of objects (the *new* command), the treatment of computation threads, and method invocations (the *send* command).

Creating new objects: note that the two versions of the *new* command which are provided are non-redundant. Indeed, consider two types T and R with the same structure [A : *const*], and an input instance where T consists of n objects with the same value a for A. If only the $[l :=]new_T(...)$ is provided, then the transformation outputting in R *one* object with value a is not computable by dms's. Clearly, this is computable with the $[l :=]new_T^{val}(...)$ command. On the other hand, if only $[l :=]new_T^{val}(...)$ is provided, the transformation producing in R a copy of T (n new objects with value a) is not computable by dms's.

Computation threads: the semantics allows the *merging* of computation threads using the $send^{val}$ command. Without this command, the proliferation of threads may result in a combinatorial explosion. We note that this command can be simulated by the (non-merging) *send* and the new^{val} commands. However, the simulation is not straightforward.

Sending messages: the semantics allows an object to broadcast a message to all objects of a certain type, or to send messages to specific objects accessible by pointers. The latter semantics is clearly subsumed by the first. The converse is not true. We discuss this in more detail in Section 3.

3 Expressive Power

In this section we examine the expressive power of dms's, i.e. their ability to express transformations of instances. To this end, we first explore various notions of *completeness* appropriate to the object-oriented context. As we shall see, there are important differences with the classical relational framework. It will turn out that dms's are complete with respect to some notions and not others.

In the relational database framework, a query language is complete if it expresses all transformations of relational instances which are: (i) computable, and (ii) *generic*, i.e. data is treated uniformly. The genericity requirement is a natural and well-accepted consequence of data independence [AU79,CH80]. It says that a query can only use information about data which is provided at the conceptual level. Thus, a query language cannot access physical level information about data, so data items with the same logical properties are treated uniformly. Genericity is formalized as follows[4] : a relational transformation

[4] This assumes that no constants occur in the query; indeed,

τ is *generic* iff for each input instance I and each isomorphism[5] f on I, $\tau(f(I)) = f(\tau(I))$. The above definition cannot be directly extended to the object-oriented framework. The difficulty is that transformations of object instances typically result in the creation of new objects. This is different from the relational case, where the result of a query contains only values from the input. We use the notion of completeness for object-oriented databases proposed in [AK89]. In addition to genericity, the definition of [AK89] requires that the different possible results of a transformation differ only in the choice of new id's. Transformations satisfying these requirements are called, as in [AK89], *db-transformations*. A language or computation mechanism is said to be *sound* if it expresses *only* db-transformations and *complete* if it expresses *all* db-transformations.

It is fairly easy to see that dms's are sound wrt db-transformations. The genericity is a consequence of the symmetric, uniform treatment of objects. This is ensured by the synchronicity and the CRCW semantics. Indeed, objects which are undistinguishable at the conceptual level are always treated identically.

Can we expect dms's to be complete? This seems likely at first glance, since dms's are *computationally* complete. Indeed, they can simulate Turing Machines whose tapes are encoded as sequences of objects (one for each tape cell), as in Figure 1.

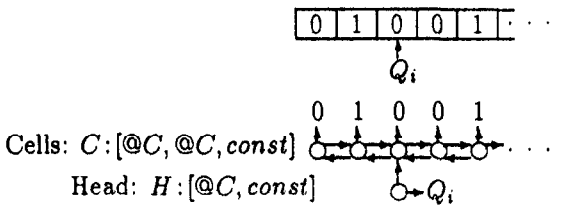

Figure 1: Encoding of a Turing Machine in a dms instance.

Rather surprisingly, it turns out that dms's are *not* complete. We show this by exhibiting a db-transformation which is not expressible by dms's (see [DV91]). That particular transformation becomes computable if complex objects are allowed as values of attributes of objects (in fact, sets alone are sufficient). This shows that complex objects would add to the expressive power of dms's. However, it can be shown that, even with complex objects, dms's would not be complete. This follows from recent results on the language IQL of [AK89], which has complex objects but was shown to not be complete. These difficulties disappear if the generic treatment of objects by methods is circumvented. This can be done by supposing that object id's are integers which can be accessed and compared by methods.

Theorem 3.1 Each db-transformation on instances with integer id's is expressed by some dms with integer comparison.

We note that no deterministic language for object-oriented databases is known which is complete without an assumption equivalent to integer id's, except by including an unnatural primitive construct which amounts to checking graph isomorphism [AK89].

In order to better understand the expressive power of methods, it is useful to compare them to known relational languages. To this end, we establish a straightforward correspondence between object instances and relational instances[6]. We associate to each db-transformation τ a relational transformation $rel(\tau)$, and conversely, to each relational transformation ν a db-transformation $obj(\nu)$. We first define $rel(\tau)$. Consider a dms M. To each type T in M with structure $T : [A_1, ..., A_n]$, associate a relational schema $sch(T)$ with attributes $T_{id}, A_1, ..., A_n$. To each object of type T with id i and attribute values $a_1, ..., a_n$, associate the tuple $rel(i) = [i, a_1, ..., a_n]$ over $sch(T)$. The mapping rel extended to instances over $str(\mathrm{M})$ is clearly one-to-one. For a given db-transformation τ, denote by $rel(\tau)$ the relational transformation mapping $rel(I)$ to $rel(\tau(I))$ for each dms instance I. Conversely, suppose ν is a relational transformation. We can "lift" a relational instance to a dms instance by associating to each relation in the schema a type, and to each tuple in a relation an object in the corresponding type. This is done by assigning an id to each tuple; the attribute values of the id are specified by the tuple. For a relational instance I, the corresponding dms instance is denoted $obj(I)$. The db-transformation associating $obj(\nu(I))$ to $obj(I)$ is denoted by $obj(\nu)$.

We focus on a category of relational languages which provide a construct analogous to the creation of new objects: the *invention* of new values from the domain [AV88a,AV88b]. One such language is an extension of Datalog which allows: (i) negations in bodies of rules, (ii) negations in heads of rules interpreted as deletions, and (iii) the ability to invent values using variables occurring only in heads of rules. Rules are fired in parallel with all applicable valuations, until no

constants specified in the query can be distinguished from others. The definition can be easily extended to accommodate such constants.

[5] An isomorphism on I is a one-to-one mapping on constants, extended to I.

[6] We note that the correspondence between relational databases and instances of object-oriented or semantic databases has also been discussed in [HS89,LV87].

rules can be fired. This language, denoted $Datalog^{\neg *}_{\infty}$, is defined precisely in [AV88a,AV88b]. It turns out that dms's and $Datalog^{\neg *}_{\infty}$ have essentially the same expressive power.

Theorem 3.2 A db-transformation τ can be expressed by a dms iff $rel(\tau)$ can be expressed by $Datalog^{\neg *}_{\infty}$.

It was shown in [AV88a,AV88b] that $Datalog^{\neg *}_{\infty}$ is complete for relational transformations for which outputs do not contain invented values. Similar results can be obtained for dms's using Theorem 3.2. Consider db-transformations which: (1) do not create new objects in the output; (2) are over schemas with types whose attributes are all of type *const*. Db-transformations satisfying (1) *manipulate* pointers and constants among existing objects. We refer to these as *m-transformations*. Db-transformations satisfying (2) involve instances which are essentially *relational* tuples of constants, "lifted" to an object instance by the *obj* mapping described above. Thus, there are no pointers among objects. Such "relational" transformations are called *r-transformations*. It turns out that (i) each m-transformation can be expressed by a dms, and (ii) each r-transformation can be expressed by a dms.

We lastly look at the expressive power for two important special cases: in-place computation, and dms's which do not perform internal broadcast of messages.

In-place computation. As stated earlier, there is a loss of expressive power if in-place computation is required, i.e. the dms cannot use additional types. Indeed, consider the type $P : [A : const]$. Consider the transformation *even* over instances I of P defined by: $even(I) = I$ if I has an even number of objects, and $even(I) = \phi$ otherwise. It can be shown that the transformation *even* is not computable in-place by dms's. Thus, additional types are necessary for full expressive power. However, it turns out that it is sufficient to add *one* particular type with structure $T : [A : @T, C : const]$ in order to recover the full power of dms's [DV91]. In some sense, this provides a normal form for the hidden structural component of dms's. It follows almost immediately that, if **V** contains a type T as above, then any db-transformation over **V** which is computable by dms's is computable in-place. The cyclicity involved in the structure of T is essential. The following more general condition can be shown. It involves the *reference graph* of a structural schema, i.e. the graph whose nodes are the types of the schema, and where an edge from type P to type Q indicates that type P has an attribute of sort $@Q$. Let **V** be a dms whose reference graph has some cycle such that there exists a type in the cycle with an attribute of sort *const*. Then each db-transformation over **V** which is computable by dms's is computable in-place. The above follows from the observation that the type T can be encoded using the types occurring in a cycle as above.

Internal broadcasts. The *send* instruction allows sending a message either to individual objects using pointers, or by broadcasting the message internally to all objects of a given type. It is useful to understand the difference between these constructs. There is a loss of expressive power if internal broadcasts are not allowed. Indeed, let a *pointer schema* be a dms which uses no internal broadcasts (however, an external broadcast is allowed to start the computation). The transformation *even* defined above is not expressible by pointer schemas. Intuitively, a message sent by an object can only reach objects reachable by pointers. It turns out that pointer schemas can simulate internal broadcasts within strongly connected components (of the graph whose nodes are objects and directed edges indicate the existence of pointers among objects). The simulation of broadcast is nontrivial due to timing requirements (all objects must know when the global simulation of the broadcast has been completed). Pointer schemas are as powerful as full dms's on strongly connected inputs. However, this does not follow from direct simulation of broadcast, since intermediate results may not be strongly connected. Instead, the proof requires a more subtle technique involving the construction of pointer chains among new objects representing the objects in the input. This builds in effect orderings of the objects in the input, which are used in the computation in a manner similar to integer id's. These orderings are constructed in polynomial time and space. Since building orderings dominates the complexity of some hard transformations, strong connectivity influences complexity in general. For example, the hard (exponential space) transformation *even* is in polynomial time and space under strong connectivity.

4 Restricted Dms's

We have seen that dms's are very powerful computational tools. In particular, there is no bound on the complexity of dms computations. As for relational query languages, it is of interest to identify tractable restrictions of dms's. In this section we present several such restrictions, which provide tractability guarantees to various degrees: constant parallel time, polynomial time, polynomial space, and monotonicity. We also show connections with traditional query languages which provide analogous tractability guar-

antees, such as relational calculus/algebra (the *first-order* queries FO), *Datalog*, the *fixpoint* queries, and the *while* queries. See [Ch88] for a description of these languages.

We consider various ways of limiting the complexity of dms computations. We first consider dms's without recursion. Let the *call graph* of a dms M be the graph whose nodes are the methods for each type in the schema; there is a node from method m of type P to method n of type Q if m contains a *send* command whose destination is in type Q and which invokes method n. Dms's with acyclic call graphs are called *non-recursive*. It is easy to see that a db-transformation τ is expressed by a non-recursive dms iff τ is computable in constant time by some dms. Intuitively, non-recursive dms's correspond closely to relational calculus/algebra. However, methods can create new objects, while the calculus cannot invent new values. To compare the two, we use the m-transformations defined in the previous section, since they do not output new objects. Thus, for m-transformations τ, $rel(\tau)$ does not involve invented values. We also look at relational transformations lifted to dms transformations. We now have the following.

Theorem 4.1 (i) An m-transformation τ is expressible by a non-recursive dms iff $rel(\tau)$ is in FO.
(ii) A relational transformation ν is in FO iff $obj(\nu)$ is expressible by a non-recursive dms.

Note that Theorem 4.1 provides a characterization of FO relational transformations as those computable in constant time by a dms. This is of particular interest. Indeed, the parallel complexity of FO has been known to be constant time (using a polynomial number of processors). However, this was shown via a rather artificial connection with AC_0, a circuit complexity class consisting of circuits of bounded depth [I87]. Furthermore, AC_0 provides just an upper bound but not a precise characterization, since there are queries computable in AC_0 but not in FO (in the absence of an order assumption on the domain) [G90]. On the other hand, with dms's the characterization is exact. This suggests that dms's provide a natural model of parallel computation for databases. The connection of dms's to traditional models of parallel computation is described in more detail in Section 5.

We next consider restrictions of dms's which allow recursion. We limit the creation of new objects and the proliferation of computation threads in order to keep the computation within PSPACE. Here and in the later restrictions, we need to disallow certain commands from occurring in a computation loop. We call a command *cyclic* if it occurs in a method which can be reached from some cycle in the call graph of the dms. (Note that the command itself need not occur in a cycle.) Otherwise, the command is called *acyclic*. Note that if a command is acyclic, there is a constant bound on the number of times it is executed in any computation of the dms. Next, a command includes a *self-reference* if *self* occurs in the command (*self.A* is not a self-reference). Consider dms's such that: no cyclic command includes a self-reference; messages are sent only by the $send^{val}_{destination}$ command; and, new objects are created only by the new^{val} command (without the assignment option). We call such dms's *while* dms's, due to the close correspondence with the relational *while* queries. We can now show:

Theorem 4.2 (i) Each computation of a *while* dms takes polynomial space in the size of the input instance.
(ii) An m-transformation τ is expressed by a *while* dms iff $rel(\tau)$ is a *while* query.
(iii) A relational transformation ν is a *while* query iff $obj(\nu)$ is expressed by a *while* dms.

Theorem 4.2 (i) places a PSPACE upper bound on the space complexity of *while* dms computations. From (ii), (iii) and [V82], it follows that there are transformations computable (by a Turing Machine) in PSPACE which are not expressible by *while* dms's (e.g., *even*, introduced earlier). However, it can be shown that *while* dms's express *exactly* the PSPACE transformations if integer id's and comparison are assumed. The proof is similar in spirit to that for showing that the relational *while* queries express exactly PSPACE on *ordered* databases [V82].

To further limit the complexity of dms computations to polynomial time, we make two additional restrictions. First, no assignments to attributes are allowed. Next, we limit the alternations of object insertions and deletions from each type. To this end, we place the following syntactic restriction. For each type T in the schema, the following cannot occur simultaneously: there is a cyclic command which deletes an object from T; and, there is a cyclic command which inserts an object in T (by a new^{val} or *role* command). A *while* dms satisfying the above conditions is called a *fixpoint dms*, due to its close connection with the *fixpoint* queries. Note that the absence of assignments and the above condition limit to polynomial time the "useful" portion of the computation i.e. the portion which causes changes in the database. However, there is nothing that prevents non-terminating vacuous computations. Termination can be forced externally, by stopping the computation if no change to the database occurs for a certain number of steps (polynomial in the size of the database).

Syntactic conditions with the same effect can also be added, but we do not elaborate on these here. Instead, we simply assume that the computation stops once a vacuous infinite loop is reached (this is similar to usual fixpoint semantics). We can show:

Theorem 4.3 (i) Each computation of a *fixpoint* dms takes polynomial time in the size of the input instance.
(ii) An m-transformation τ is expressed by a *fixpoint* dms iff $rel(\tau)$ is a *fixpoint* query.
(iii) A relational transformation ν is a *fixpoint* query iff $obj(\nu)$ is expressed by a *fixpoint* dms.

The proof of (ii) and (iii) uses the normal form for the *fixpoint* queries provided by the language $Datalog^{\neg}$, shown in [AV88a,AV88b].

Theorem 4.3 (i) places a PTIME upper bound on the computation of *fixpoint* dms computations. However, the *fixpoint* dms's do not express all PTIME db-transformations. As for *while* dms's, exact expressiveness is achieved with integer id's and comparison. The proof is similar to that for showing that the relational *fixpoint* queries express exactly PTIME on *ordered* databases [I86,V82].

We finally consider a further restriction which guarantees monotonicity of computation and corresponds closely to *Datalog*. Let a *Datalog* dms be a *fixpoint* dms with the following restrictions: (i) there are no *delete-self* statements, and (ii) conditions in *if* statements are conjunctions of equality tests ($x = y$). We can now prove an analog of Theorems 4.2 and 4.3 for *Datalog* dms's and *Datalog* relational queries [DV91].

5 Dms's and Parallel Computation Models

Dms's provide a parallel model of computation which is database-oriented. It is fundamentally distinguished from other known models of parallel computation by its generic, symmetric treatment of data/processors. Indeed, in the traditional models of parallel computation, genericity is not an issue because of various assumptions, such as integer id's for processors, integer-numbered memory cells, etc. However, there are close connections between dms's and some of these models. A close match occurs only when dms's use integer id's. We compare dms's with one of the main models of parallel computation, the Parallel Random Access Machine with Concurrent Read and Concurrent Write semantics (CRCW-PRAM, or simply PRAM) [Pa87]. In the process, we also point out the connection with a model closer in form to ours, the *Hardware Modification Machine (HMM)* of Cook [Co81]. We conclude that many parallel complexity classes, including the well-known *NC* (Nick's class), are the same with respect to PRAMS and dms's with integer id's.

We begin with a brief review of the PRAM model (details can be found in [Pa87]). A PRAM consists of an unbounded tape and unbounded processor pool. Processors have unique integer id's which they can refer to. A processor can be activated by other processors having its id number. Each processor has a fixed finite number of registers storing binary encodings of arbitrary integers, as does each cell of the tape. The standard sequential RAM instruction set is used [AHU76], subject to restrictions on growth rates of integers. Thus, multiplication is not an instruction, but shifts (or multiplication/division by 2) are. Each processor can access the tape to load/write an integer to/from any of its registers, whose addresses can be accessed. Conflicting concurrent writes cause the computation to crash.

We show that dms computations can be simulated by PRAMs within a logarithmic time and constant space factor, assuming that the input database is encoded in some standard way on the PRAM tape. (Note that the encoding is necessarily ordered.) This cost is standard for such simulations. Indeed, a log factor time difference exists between different PRAM models. The precise result and proof sketch are given in [DV91].

We next look at the converse simulation, of PRAMs by dms's. However, instead of a direct simulation, we use a third model called Hardware Modification Machine (HMM). Thus, we show how dms's can simulate HMMs, and use the known result that HMMs can simulate PRAMs with time within a logarithmic factor and space within a polynomial factor of the PRAM [D80]. HMMs are of particular interest to us because the computational paradigm is closer to that of dms's.

An HMM consists of a set of identical finite-state transducers computing in parallel. Each finite-state transducer is called a *unit*. Each unit owns a fixed number k of pointers to other units called *taps*. In each finite-state transducer, transitions occur synchronously based upon its current state and the outputs of the units it taps. A tap can be moved to any unit not more than two units away from its owner and each unit may activate and initialize one new unit per step. Note that units are similar to objects (or computation threads running on objects) and the taps are somewhat similar to pointers among objects. An important difference with dms's is that HMMs are not viewed as computing transformations from HMM configurations into other HMM configurations. Instead, an input to a HMM **H** with k taps per unit is

essentially a tape of arbitrary length l, encoded as the leaves of a k-ary tree of units (of depth $log_k(l)$). This circumvents genericity, since inputs are in effect ordered. However, the HMM computation itself treats the unit machines in a generic fashion.

Our result (stated precisely in [DV91]) is that dms's can simulate HMMs within a constant time and space factor.

The above results, and the simulation of PRAMs by HMMs [D80], show that PRAMs and dms's can simulate each other within a logarithmic time factor and polynomial space factor. The mutual simulation of PRAMs and dms's allows us to conclude that parallel complexity classes defined on the two models are closely related. Note first that, in the simulations above, the encodings of dms instances as PRAM instances were assumed given, and were not factored into the cost of the simulation of PRAMs by dms's. Obviously, we cannot generally assume that a dms can simulate such encodings since this may violate genericity. However, the encodings can be computed with integer id's with logarithmic cost. Now we can show that parallel time and space complexity classes defined on PRAMs and dms's with integer id's are identical, as long as they are insensitive to logarithmic factors in time complexity and polynomial factors in space complexity. In particular, the class NC of "tractable" parallel problems are the same in the two models. More precisely, let PRAM-NC be the functions computable in a PRAM with log-time and polynomial-space cost, and DMS^{int}-NC be the db-transformations on instances with integer id's computable by some dms (with integer comparison) with log-time and polynomial-space cost. Then we have:

Theorem 5.1 DMS^{int}-NC = PRAM-NC.

Without the integer id assumption, the equality no longer holds. Let DMS-NC be the class of db-transformations (without integer id's) computable by some dms with log-time and polynomial-space cost. It can be shown that DMS-$NC \subset$ PRAM-NC. Indeed, the db-transformation *even* is clearly in PRAM-NC. However, it is not in DMS-NC, since it can be shown that computing *even* with dms's without integer id's requires exponential space.

6 Dms's and Generic Machines

We have seen in the previous section the connection of methods with classical parallel computation models. In particular, we have seen that, without the integer id assumption, there is a mismatch between parallel complexity classes defined by dms's and those defined by PRAMs. Intuitively, this is due to the fact that the PRAM model does not capture generic computation, and cannot provide an appropriate basis for measuring its intrinsic complexity. Similar remarks apply to sequential complexity and classical sequential devices such as Turing Machines, which circumvent the genericity issue (Turing Machines always work on an ordered tape!).

A computational model called *Generic Machine (GM)*, designed to capture the complexity proper to generic computation, has been defined in [AV91]. A GM is a Turing Machine (TM) augmented with a finite set of fixed-arity relations forming a *relational store*. The relational store can be viewed as associative access storage supporting the generic portion of the computation, while standard computation is carried out on the tape. GMs compute relational transformations, with no invented values in the result. Designated relations contain initially the input, and others hold the output at the end of the computation. Communication between the tape and the relational store is provided. GM allows spawning other GM's which then compute synchronously in parallel. There is a mechanism for merging parallel machines. The output is only obtained after all machines are merged into a single one.

Based on GM, complexity classes proper to generic computation are defined: GEN-PTIME, and GEN-PSPACE, obtained by polynomial restrictions on time and space resources used in the computation. We can define analogous complexity classes based on the method model. Let DMS-PSPACE be the set of db-transformations expressible by some dms such that, at each point in the computation on input I, uses a number of objects and active threads which is polynomial in the size of I. Let DMS-PTIME be the set of DMS-PSPACE db-transformations expressible by some dms such that the computation on every input I terminates in a number of steps polynomial in the size of I. It turns out that these generic complexity classes essentially coincide in the two models. To make the classes comparable, we need to restrict the comparison to db-transformations whose relational representation do not involve invented values. The following can be shown:

Theorem 6.1 (i) An m-transformation τ is in DMS-PTIME (DMS-PSPACE) iff $rel(\tau)$ is in GEN-PTIME (GEN-PSPACE).
(ii) A relational transformation ν is in GEN-PTIME (GEN-PSPACE) iff $obj(\nu)$ is in DMS-PTIME (DMS-PSPACE).

The agreement of dms's and GMs wrt complexity is significant. It speaks to the robustness and naturalness of the generic complexity classes based on these

models. Since DMS-PTIME and DMS-PSPACE include transformations with new objects in results, they can be viewed as extensions of the GEN-PTIME and GEN-PSPACE. Dms's provide an elegant, practically motivated alternative to GM as a model of database parallel computation.

References

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive, to appear in *JACM*. Preliminary version in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 159–173, 1989.

[AKW90] S. Abiteboul, P. Kanellakis, and E.Waller. Method Schemas. In *Proc. Ninth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pp. 16-27, 1990.

[AV88a] S. Abiteboul, V. Vianu, Procedural and Declarative Database Update Languages. *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pp.240-250, 1988.

[AV88b] S. Abiteboul, V. Vianu, Datalog Extensions for Database Updates and Queries, I.N.R.I.A. Technical Report No.715 (1988). To appear in J. of Computer and Systems Science (in press).

[AV91] S. Abiteboul and V.Vianu. Generic Computation and Its Complexity. In *Proc. ACM Symp. on Theory of Computing*, pp. 209-219, 1991.

[AHU76] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1976).

[AU79] A.V. Aho and J. Ullman. Universality of Data Retrieval Languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pp. 110-117, 1979.

[A+90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pp. 395-396, 1990.

[B88] F. Bancilhon. Object-Oriented Database Systems. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 152-162, 1988.

[Ch81] A. Chandra. Programming Primitives for Database Languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pp. 50-62, 1981.

[Ch88] A. Chandra. Theory of Database Queries. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp.1-9, 1988.

[CH80] A. Chandra and D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21(2):156–178, Oct. 1980.

[Co81] S. A. Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathématique*, 27(1-2):75–100, 1981.

[DV91] K. Denninghoff and V. Vianu. The Power of Methods With Parallel Semantics. *UCSD Technical Report CS91-184*, 1991.

[G90] Y. Gurevich, personal communication.

[D80] P. W. Dymond. *Simultaneous Resource Bounds and Parallel Computation.* PhD thesis, Univ. of Toronto, August 1980. TR 145/80.

[HS89] R. Hull and J. Su. On Accessing Object-Oriented Databases: Expressive Power, Complexity, and Restrictions. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pp. 147-158, 1989.

[I86] N. Immerman. Relational Queries Computable in Polynomial Time. *Inf. and Control*, 68:86–104, 1986.

[I87] N. Immerman. Expressibility as a Complexity Measure: Results and Directions. Yale Univ. Res. Rep. DCS-TR-538, 1987.

[LV87] P. Lyngbaek and V. Vianu. Mapping a Semantic Database Model to the Relational Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pp. 133-142, 1987.

[Pa87] I. Parberry. *Parallel Complexity Theory.* London: Pitman; New York: Wiley, 1987.

[Pi79] N. Pippenger. On simultaneous resource bounds. *Proc. 20th IEEE Symp. on the Foundations of Computing Science*, pp. 307-311, 1979.

[V82] M. Vardi. The Complexity of Relational Query Languages. In *Proc. 14th ACM Symp. on Theory of Computing*, pp. 137–146, 1982.