ON CORRECTLY CONFIGURING VERSIONED OBJECTS

Rakesh Agrawal and H. V. Jagadish
AT&T Bell Laboratories, Murray Hill NJ

We consider the problem of configuring a system in software and design database domains, where a system comprises a version for each of its constituent objects. We present a syntactic characterization of a correct configuration, tied to a transaction model, that makes it possible to generate automatically all correct configurations of a system. One can also generate configurations that satisfy some selection criteria such as the absence and presence of specified features, or check whether a user-specified configuration is correct.

1. INTRODUCTION

Software and design databases invariably consist of versioned objects (*cf.* [2,4,5,7-9,14]). Versions of an object often represent alternatives or revisions. A system comprising a set of objects is *configured* by selecting a version for each of the objects that constitute the system. A configuration is also treated as a versioned object, so that more than one configuration can coexist.

An important issue in configuring a system is that the constituent versions must be compatible [12,15]. We all know that the version of a module compiled for Motorola 68000 should not be linked with the version of another module compiled for Intel 386. All the systems we know of provide very little by way of support for deciding what can constitute a correct configuration, leaving this decision to the user. In a system consisting of m objects, each with v versions, there can be up to v^m possible configurations. As systems become large, relying on user intuition to decide what constitutes a correct configuration is at the very least error-prone, if not altogether impossible.

This paper is an attempt to correct this deficiency by introducing a formal notion of the correctness of a configuration and mechanisms for generating and verifying correct configurations. We take inspiration from the rich database literature on transaction management (*cf.* [6]) and suitably extend the transaction model. Updates to the database and dependencies between versions are encapsulated in transactions that transform the database from one consistent state to another. Correct configurations are equated with consistent database states. However, consistent database states are not limited to the set of states that the database actually goes through during the execution of a set of transactions, but also includes states that could have been generated using different serialization orders consistent with dependencies in the transaction set. We do not require versions to be totally ordered and admit a much weaker notion of serializability than the 1-copy serializability [6] used in classical multiversion concurrency control theory.

We thus have a "syntactic" characterization of a correct configuration that is tied with the transactions that create and update versioned objects. Using this characterization, it is possible to generate automatically all correct configurations of a system. One can also generate those configurations that satisfy some selection criteria such as configurations that incorporate specified features, or check whether a user-specified configuration is correct.

There is often a distinction made between the *interface* of an object and its *implementation* [3]. Sometimes only the implementation is allowed to change and not the interface. At other times both change. These are often distinguished and sometimes given different names, one called a version, and the other a revision or an alternative. In this paper, we will not make this distinction, and will use the single term "version" to refer to all different implementations and interfaces of an object. We also do not differentiate between types of versions such as public, private, transit, working, etc. [7-9]. If necessary, the reader can imagine that we deal only with publicly released committed versions.

The idea of composite objects being obtained as configurations of primitive objects has been explored in [2]. In the terminology of [11] these primitive objects are *shared independent* constituents of a configuration. Whenever a new version of a primitive object is created, new configurations are also obtained, by means of *version percolation*. The number of composite object versions obtained grows in a combinatorial fashion, as pointed out in [7]. The solution proposed in [7], based on time-stamps, saves system storage by passing on to the user the burden of examining the new configurations possible and recording them in the database, if desired. Our approach, in this paper, avoids the need for storing configurations altogether by providing an efficient mechanism for putting configuration together on the fly.

Katz and Chang [10] also propose a system in which configurations can be composed from versioned objects. However, they rely upon a logic program (or rule base), provided by the user, to ensure correctness rather than using syntactic characteristics of the transaction model itself.

The organization of the rest of the paper is as follows. In Section 2, we introduce our transaction model and define what we mean by a correct configuration of object versions. In Section 3, we show how one can obtain all correct configurations from a given initial set of correct configurations. In Section 4, we discuss how to determine whether a user-specified configuration is correct. We also discuss how user requests for configurations containing (or not containing) certain specified features can be satisfied. We present our conclusions in Section 5. We assume that the reader is familiar with the basic notions of transactions and serializability; see [6] for a tutorial introduction.

2. BASIC CONCEPTS

We first introduce our transaction model, and then formally define what we mean by a correct configuration of object versions.

2.1 Transaction Model

A database is a collection of objects, each of which is independently versioned. For ease of exposition, we assume that the database starts with a "pre-creation" version of every object in the database. Subsequent versions are created by transactions that read and update database objects. Whenever an object is updated by a successful transaction, it results in the creation of a new version of the object. A transaction may update more than one object, but may create only one version of an object. All versions are created when the transaction commits. Deletions result in the creation of "post-deletion" versions.

An object version is created by exactly one transaction. Each object version is stamped with the signature of the transaction that created it. Given an object version x, the function $oid(x)$ returns the identity of the object of which x is a version, and the function $generator(x)$ returns the identity of the transaction that created x.

Associated with each transaction T_i is a *read set* R_i of object versions read by T_i and a *write set* W_i of object versions created by T_i. We differentiate between "reading" and "browsing " an object version. When a transaction T_i reads an object version x, it ensures that x, unless updated by T_i, will be configurable with whatever versions T_i creates. Browsing on the other hand is a "free" operation. When T_i browses an object version, it is not placed in R_i. A transaction can browse an object version x and then write object versions that are not compatible with x. Finally, transactions can read "pre-creation" versions but may not read "post-deletion" versions.

We put the following additional requirements on transaction behavior:

i. A transaction T_i that writes an object version must read at least one previous version of the same object, i.e., $\forall$ object versions $x \in W_i, \exists\, y \in R_i$ such that $oid(x) = oid(y)$.

ii. A transaction T_i reads no more than one version of an object that it updates, though it may read multiple versions of objects that it does not update, i.e., $\forall$ object versions $x, y \in R_i, \forall\, z \in W_i$: $oid(x) = oid(z), x \neq y \Rightarrow oid(x) \neq oid(y)$.

Thus, for each version of an object, one can identify a unique version of the same object from which it has been directly derived. We can, therefore, create a *version graph VG* for each object with one node for each version of the object and an edge from each version to the versions that are derived directly from it. More than one version can be derived directly from a version, and hence the version graph is a tree[1], as in most software and design databases.

We say that a version x of an object is *derived* from version y of the same object iff there is a path from y to x in the version graph of the object.

Simultaneous reads of the same object version by two different transactions do not conflict. Simultaneous updates to the same object in parallel by two different transactions do not conflict either, since they create different object versions.[2] Browsing also does not conflict with either simultaneous reading or writing of the same object version. However, a transaction cannot read any object version before it has been created. This requirement gives rise to a dependence between transactions, which can be represented in a *transaction dependence graph.*

The transaction dependence graph TG has a node corresponding to every transaction in the system, and there is an edge from a node corresponding to transaction T_i to a node corresponding to transaction T_j iff $R_i \cap W_j \neq \phi$. A transaction T_i is said to *depend* on

1. Allowing a transaction to read more than one version of an object that it updates, thereby allowing a version to be directly derived from more than one version, is equivalent to allowing the version graph to be a directed acyclic graph (DAG). We do not consider version graphs that are DAGs in this paper since trees are considered adequate in most practical applications [Won Kim, personal communication, 1988]. Note that we allow a transaction to browse multiple versions of an object that it updates. Thus, if a transaction has to peruse multiple versions of an object that it updates, it can "read" one from which branching will occur in the version graph and "browse" the remaining ones.

2. Note that in our transaction model, the system never aborts a transaction.

transaction T_j iff there is a path from T_i to T_j in TG.

Lemma (Acyclicity of the Transaction Dependence Graph): There is no cycle in the transaction dependence graph, as defined above.

Proof: Versions are created only after a transaction has committed. □

We can, therefore, topologically sort the transaction dependence graph, i.e., the transactions can be totally ordered in such a way that if T_i depends on T_j then T_i occurs later in the order than T_j. Each such total order constitutes a *serialization* of the transactions.

Read-only transactions are not significant in this model: while they do have a dependence on the transactions that write the object versions they read, they themselves write nothing and no other transactions depend on them. We, therefore, drop all read-only transactions from the transaction dependence graph for the rest of the paper.

2.2 Correct Configurations

A *configuration* is a set of object versions with no more than one version of any object, i.e., it is a set of object versions C with the property that $\forall\ x, y \in C : x \neq y \Rightarrow oid(x) \neq oid(y)$.

A *complete configuration* selects exactly one version of every object in the database, i.e., it is a configuration C such that for every object p in the database, $\exists\ x \in C$ such that $oid(x) = p$.

We are interested in *correct configurations*. We equate correct configurations with consistent states of the database. However, consistent database states are not limited to the set of states that the database actually goes through during the execution of a set of transactions, but also includes states that could have been generated using different serialization orders consistent with the dependencies in the transaction dependency graph. To this end, we introduce the notion of qualification of a configuration for a transaction.

A configuration C *qualifies* for a transaction T_i iff $\forall\ x \in R_i,\ \exists y \in C$ such that $y \in R_i \wedge oid(y) = oid(x)$. In other words, a configuration qualifies for a transaction if the following holds for all objects that the transaction reads: i) if the transaction reads only one version of an object, this version is included in the configuration; and ii) if the transaction reads multiple versions of an object, any one of the versions is included in the configuration. In addition, the configuration may contain versions of objects not in the read set of the transaction. For example, let p_i be a version of the object p, q_i a version of q, etc. Then, given a transaction with a read set $= \{p_1, p_2, p_3, q_1\}$, the configuration $\{p_1, q_1, r_1\}$ qualifies for it, while the configuration $\{p_2, q_2, r_2\}$ does not. Note that the set $\{p_1, p_2, q_1\}$ is not a configuration since it includes two versions of the object p.

If configuration C qualifies for transaction T_i, then T_i is said to be *applicable* to C. A new configuration D is said to be *generated* by *applying* T_i to C, as follows:

$$D = W_i \cup \{x \in C \mid (\forall\ y \in W_i : oid(x) \neq oid(y))\}$$

i.e., D is obtained by substituting in C the updated versions of the objects written by the transaction T_i.

We can now inductively define the set of correct configurations as follows:

1. The *initial complete correct configuration*, consisting of the "pre-creation" versions of all the objects, is a correct configuration.
2. Every subset of a correct configuration is also a correct configuration.
3. Given a transaction, T, and a qualifying correct configuration C, the configuration D generated by applying the T to C, is also correct.

Sometimes, we will find it necessary to work backwards from a configuration to determine how it could have been generated. To this end, we introduce the notion of the reverse transformation of a configuration. A configuration C is said to *reverse qualify* for a transaction T_i, iff the transaction's access set A_i is such that $\forall\ x \in A_i,\ \exists y \in C$ such that $y \in A_i \wedge oid(y) = oid(x)$. This definition of reverse qualification is the same as the definition of qualification with the substitution of the access set of the transaction for the read set. The *access set* A_i of transaction T_i is defined as:

$$A_i = W_i \cup \{x \in R_i \mid (\forall\ y \in W_i,\ oid(x) \neq oid(y))\}$$

i.e., the access set is the write set plus the versions of objects in the read set that do not occur in the write set. It can also be thought of as the read set with the updated versions replacing the old ones for objects that were updated by the transaction.

To *reverse apply* T_i to D, substitute in place of each element $x \in W_i$ in D, an element $y \in R_i$ such that $oid(y) = oid(x)$. That is, the resulting configuration C is obtained as

$$C = \{x \in D \mid x \notin W_i\} \cup \{x \in R_i \mid (\exists\ y \in W_i \text{ such that } oid(x) = oid(y))\}$$

This definition parallels the definition of applying a transaction. If configuration D is obtained by applying a transaction T to a qualifying configuration C, then D reverse qualifies for T, and C is obtained on reverse applying T to D, and vice versa. Note that the result of reverse application of a transaction to a reverse qualifying configuration is unique.

3. GENERATION OF ALL CORRECT CONFIGURATIONS

We now examine how to obtain all correct configurations from a given initial set of correct configurations. While this in itself may not be a

problem of interest directly, the results developed in this section can readily be adapted for several problems of interest, some of which are discussed in Section 4.

3.1 A Naive Method

A straightforward technique to obtain all correct configurations is to create all possible serializations of the transaction dependence graph, and for each serialization to compute a set of correct configurations as each transaction "moves the database from one consistent state to another". However, a transaction can "access" not just the immediately preceding consistent state of the database, but any previous consistent state as well. When a transaction executes, the following procedures are executed to augment the set of correct configurations:

Transaction-Apply: Apply the given transaction to every qualifying correct configuration obtained previously from this serialization to generate new correct configurations.

Subset-Expand: For every new correct configuration, include every subset in the set of known correct configurations.

Thus, a transaction examines all possible preceding consistent states of the database, selects the ones that qualify for it, and from these generates new consistent states (subsets of which are also marked consistent).

For each serialization, the starting set of correct configurations consists of the initial complete correct configuration comprising of "pre-creation" version and all subsets thereof. These are all the correct configurations known before any transaction executes. The *processing* of a serialization is the execution of the procedure *Transaction-Apply* for each transaction in the serialization, in order, starting with a set of initial correct configurations. The *total processing* of a serialization is its processing with the procedure *Subset-Expand* executed between successive executions of the procedure *Transaction-Apply*. Thus, from the definition of correct configuration, the total processing of every serialization of the transaction graph, starting from the given initial set of correct configurations, yields the desired set of all correct configurations.

The problem with considering all possible serializations is that there are exponentially many of them. Moreover, each serialization generates an exponential number of correct configurations. This approach, therefore, is impractical. The rest of this section is devoted to finding techniques that will do better.

3.2 Incomplete Correct Configurations

Every subset of a correct configuration is a correct configuration, and new correct configurations are generated by applying transaction to qualifying correct configurations. To ensure that all correct configurations were found we had to execute procedure *Subset-Expand* between successive executions of procedure *Transaction-Apply*, thereby generating a large number of duplicates.

For example, consider a system in which there is a single transaction with a read set of $\{p_1\}$ and a write set of $\{p_2\}$, and an initial complete configuration of $p_1q_1r_1$. First applying *Subset-Expand*, we get the set of correct configurations $\{p_1q_1r_1, p_1q_1, p_1r_1, p_1, \phi, q_1, r_1, q_1r_1\}$. The first five of these qualify for the transaction[3]. Applying the transaction to each of these, we obtain as new correct configurations $\{p_2q_1r_1, p_2q_1, p_2r_1, p_2, \phi\}$. On applying *Subset-Expand* individually to each of these new configurations, we obtain 8, 4, 4, 2, and 1 correct configurations respectively, for a total of 19, whereas only 8 of them are distinct, and only 4 not included in the set we had before applying the transaction.

The lemma below helps us to eliminate this duplication.

Lemma (Subset Completion): Every correct configuration is a subset of a complete correct configuration[4].

Proof: If a correct configuration qualifies for a transaction, so does a complete correct configuration that is its superset, by definition of qualification. Since the only mechanism for generating correct configurations, beginning with subsets of a complete correct configuration, is to apply transactions for which it qualifies, we have a proof by induction. □

The consequence of this lemma is that it is sufficient to have a procedure to generate all complete correct configurations. All other correct configurations can be found directly as subsets of these.

Lemma (Complete Generation): A configuration, D, generated by applying a transaction T to a qualifying configuration C, is complete iff C is complete.

Proof: By definition of the application of a transaction and generation of a new configuration, B has all the

3. The empty set, ϕ, represents a configuration comprising no objects. It is included here for formal correctness.

4. This lemma is not as trivial as it may at first appear. It has an implication in the reverse direction from the one in the definition of a correct configuration, which said "Every subset of a correct configuration is a correct configuration". If we consider a slightly different definition of qualification, the lemma does not hold. For example, let a configuration C qualify for a transaction T_1 provided $\forall\ x \in R_i\ \forall\ y \in C\ oid(x)=oid(y) \Rightarrow x=y$. Consider a single transaction system with a read set of $\{p_1, q_2\}$ and a write set of $\{p_2, q_3\}$, and an initial complete correct configuration of p_1q_1. Clearly, this configuration does not qualify for the transaction so that the only complete correct configuration is the initial one. However, applying the procedure *Subset-Expand*, we get $\{p_1q_1, q_1, p_1\}$ all as correct configurations, the last of which now qualifies for the transaction according to our new rule, generating new correct configuration(s) that are not subsets of the initial complete correct configuration.

elements of Λ except that some objects have their versions updated. □

By virtue of the Complete Generation Lemma, we are guaranteed that we need only consider complete configurations if we wish to generate complete correct configurations. In conjunction with the Subset Completion Lemma, it implies that we need not totally process a serialization to obtain all correct configurations: it is enough to process a serialization beginning with the initial complete correct configuration, and then to apply the procedure *Subset-Expand* once at the end. Returning to the example just before the Subset Completion Lemma, we could first create $p_2q_1r_1$ by applying the transaction to the initial configuration $p_1q_1r_1$, and then run *Subset-Expand* at the end to generate all the correct configurations, with significantly fewer duplicates.

3.3 Regular Systems

We first introduce the notion of *weak dependence.* Transaction T_i *weakly depends* on transaction T_j iff

i. for every object in the read sets of T_i and T_j, at least one common version of the object exists in the two sets; and
ii. there is at least one common version that is read both by T_i and T_j, but is only updated by T_i and not T_j ; and
iii. there is no common version that is read both by T_i and T_j, which is updated by T_j; and
iv. T_j does not depend on T_i.

Formally, T_i weakly depends on T_j iff

$\forall\, x \in R_i$, $y \in R_j$ such that $oid(x) = oid(y)$
$\quad \exists\, z \in R_i \cap R_j$ such that $oid(z) = oid(x) = oid(y)$ $\wedge$
$\exists\, x \in R_i \cap R_j$, $y \in W_i$ such that $oid(x) = oid(y)$ $\wedge$
$\nexists\, u \in R_i \cap R_j$, $v \in W_j$ such that $oid(u) = oid(v)$ $\wedge$
T_j does not depend on T_i.

The intuition behind weak dependence is that T_j can create a database state in which T_i can execute, while T_i cannot create a state in which T_j can execute. Therefore, even though there is no dependence between the transactions, in that T_i does not read something that T_j writes, we should execute T_j before T_i.

The weak dependence relation can be represented in a *transaction weak dependence graph* in which there is an edge from the node corresponding to the transaction T_i to the node corresponding to the transaction T_j iff T_i weakly depends on T_j. A set of transactions is said to constitute a *regular* system iff the transaction weak dependence graph is cycle-free.

The definitions of a regular system and weak dependence may appear to be rather arcane. We wanted to have the widest possible scope for our definition of a regular system. In a practical implementation, a simpler definition of weak dependence may be used, with some of the conjuncts removed, resulting in a narrower scope for what is a "regular system".

We argue that most systems in practice are likely to be regular systems. Consider why a transaction that updates one object (by creating a new version) would want to read versions of other objects. The most probable reason is because the updated object relies upon some properties of the objects read but not updated, of which only the versions read are guaranteed to possess the properties relied upon. One can then draw a "relies upon" graph over the objects. (In a UNIX® system, this graph could be obtained through a tool such as Cscope [13] where the objects are files and the "relies upon" relation is based upon functions called). If this "relies upon" graph is acyclic, which is often the case, then we have a regular system. Even if this graph has cycles in which pairs of objects mutually depend upon each other, we still have a regular system. In fact, it can be shown that if this graph has no simple cycles of length greater than 2, then we have a regular system. If we do have a situation in which object p relies upon object q which in turn relies upon r, and then r relies upon p without r relying on q or q on p, we may still have a regular system, but that is no longer guaranteed.

Define a *transaction graph* to be the union of the transaction dependence and weak dependence graphs so that the transaction graph has an edge from node T_i to node T_j iff there is an edge from T_i to T_j either in the transaction dependence graph (dependence edge) or the transaction weak dependence graph (weak dependence edge).

Lemma (Acyclicity of the Transaction Graph): The transaction graph in a regular system is cycle-free.

Proof: Suppose there is a dependence edge from T_i to T_j. There cannot be a dependence edge from T_j to T_i, since the transaction dependence graph is acyclic. There cannot be a weak dependence edge from T_j to T_i, since such weak dependence requires that T_i not depend on T_j. Now suppose that there is a weak dependence edge from T_i to T_j. There cannot be a weak dependence edge from T_j to T_i, since the transaction weak dependence graph in a regular system is acyclic. There cannot be a dependence edge from T_j to T_i by the definition of weak dependence once more. □

Thus, in a regular system, one can obtain one or more *regular serializations*, which are total orderings of all executable transactions such that if T_i depends or weakly depends on T_j then T_i occurs later in the order than T_j.

3.4 An Efficient Method for Generating All Correct Configurations

Lemma (Order Independence): The set of complete correct configurations generated by the processing of a serialization is not altered by switching the order in the

serialization of two transactions that have no dependence and no weak dependence between them.

Proof: Consider two transactions, T_i and T_j, neither of which depends on the other. Let $r_k = \{oid(x) \mid x \in R_k\}$, and $w_k = \{oid(x) \mid x \in W_k\}$ be the sets of objects (not object versions) read and written by transaction T_k. Since T_i and T_j are not dependent, we know that $R_i \cap W_j = R_j \cap W_i = \phi$. (We use the absence of weak dependence later).

Consider a complete configuration C known to be correct before either transaction has been executed. We have four cases to consider:

1. C qualifies for neither T_i nor T_j.
   No new configurations are generated irrespective of the order of executing the transactions. Therefore the lemma holds for all such configurations, trivially.
2. C qualifies for T_i but not for T_j.
   Let C_i be generated by applying T_i to C. T_i does not write anything that T_j reads ($W_i \cap R_j = \phi$). Therefore, C_i cannot qualify for T_j, since C did not. Thus C_i is the only new complete configuration generated, irrespective of the order of transactions.
3. C qualifies for T_j but not for T_i.
   Using arguments similar to the previous case, we show that C_j is the only new composition generated irrespective of the order of execution of the transactions.
4. C qualifies for T_i and T_j, both.
   Now we have to consider several subcases.
   a. $r_i \cap w_j = r_j \cap w_i = \phi$.
      That is, neither transaction reads any objects written by the other. (However, both transactions may read some objects in common that they do not update). It follows that $w_i \cap w_j = \phi$. Let C_i be the configuration generated by applying T_i to C. C_i differs from C only in the objects included in w_i, none of which are in r_j. Therefore, since C qualifies for T_j, so does C_i. Let C_{ij} be generated by applying T_j to C_i. Similarly, we can obtain C_j, show that it qualifies for T_i, and obtain C_{ji}. Note, however, that since $w_i \cap w_j = \phi$, the changes made from C by the two transactions are in different sets of objects. Therefore, $C_{ij} = C_{ji}$. Thus, either order of application of the transactions to C and its derivatives, yields the same new configurations: C_i, C_j, and C_{ij}.
   b. $r_i \cap w_j \neq \phi \quad r_j \cap w_i \neq \phi$.
      In this case, clearly, C_i cannot qualify for T_j and C_j cannot qualify for T_i. Therefore the only new complete correct configurations produced are C_i and C_j.
   c. $r_i \cap w_j \neq \phi \quad r_j \cap w_i = \phi$.
      If $r_i \cap w_j \neq \phi$, then we must have $r_i \cap r_j \neq \phi$. Consider an object p read by both transactions and updated by T_j. Both transactions must read the same version of p, otherwise we would not have found a C that qualifies for both transactions. Furthermore, p is not updated by T_i; otherwise, we would have found p to be a member of $r_j \cap w_i$. Therefore, T_j weakly depends on T_i, a contradiction given the premise of the lemma.
   d. $r_i \cap w_j = \phi \quad r_j \cap w_i \neq \phi$.
      As in the previous case, we can show that T_i weakly depends on T_j, a contradiction. □

Lemma (Serialization Indifference): The processing of every regular serialization results in the same set of complete correct configurations.

Proof: Follows directly from the Order Independence Lemma and the definition of a regular serialization. (Recall that a regular serialization can be obtained only in a regular system). □

Lemma (Weak Dependence Ordering): In a regular system, given T_i weakly depends on T_j, the processing of a serialization with T_i occurring before T_j results in a subset of the complete correct configurations generated by a serialization that is identical except that T_j occurs before T_i.

Proof: Following arguments similar to those used in the proof of the Order Independence Lemma, we can dismiss the cases in which a configuration qualifies for no more than one of the two transactions. Since T_i weakly depends on T_j, we know that there are configurations that could qualify for both. Consider any such configuration C. Let C_j be generated by applying T_j to C. C_j differs from C in the objects that are updated by T_j. But we know that w_j includes no objects that are in $r_j \cap r_i$. Since $w_j \subseteq r_j$, it follows that $w_j \cap r_i = \phi$. Therefore, C_j qualifies for T_i as well. On the other hand, consider C_i generated by applying T_i to C. By definition of weak dependence, $w_i \cap r_j \neq \phi$. But since T_j does not depend on T_i, we must have $W_i \cap R_j = \phi$. In other words, C_i contains a version of at least one object that is different from the version(s) read by T_j. Therefore, C_i cannot qualify for T_j. Thus, executing T_i before T_j yields C_i and C_j from C, while executing T_j before T_i yields C_i, C_j, and C_{ij}. □

Theorem (Exhaustion): The processing of any regular serialization generates all possible complete correct configurations from a given initial complete correct configuration.

Proof: Follows from the Serialization Indifference Lemma, the Complete Generation Lemma, and the following facts:

i. If transaction T_j depends on T_i, then T_j must execute after T_i. Therefore only serializations need be considered. AND
ii. If transaction T_j weakly depends on T_i, then, from the Weak Dependence Ordering Lemma, we need only consider serializations in which T_i is executed before T_j. Therefore, only regular

serializations need be considered. □

Theorem (Uniqueness): In the processing of any serialization, each correct configuration is generated at most once.

Proof: Consider the generators of the object versions in the given configuration. There must be one that occurs last in the serialization. Call it T. The given configuration, C, is generated by applying T to some other configuration, D, which can be obtained by reverse-applying T to the given configuration. C cannot be generated by any previous transaction in the serialization, since T is the latest of the generators of the elements of C. For the same reason, C cannot be generated by any later transaction. If D can be obtained uniquely from C, then C is generated only once by transaction T. But D is unique by definition of reverse-application. □

In view of the Exhaustion Theorem and the Uniqueness Theorem, it follows that one can generate all complete correct configurations in a regular system by processing any one regular serialization starting from the given initial complete correct configuration. Moreover, during this procedure each complete correct configuration is generated exactly once, and one complete correct configuration is generated by every transaction application to an existing qualifying complete correct configuration. Therefore we have obtained a very efficient method of generating all complete correct configurations in a regular system.

4. GENERATION OF SELECTED CONFIGURATIONS

In the previous section, we discussed how to generate all possible correct configurations of a system. While the ability to do so is important, usually only a subset of all possible configurations need be generated, or a membership question answered. In this section, we look at these two types of queries. We first consider the problem of determining whether a user-specified configuration is correct. We then discuss how to satisfy user requests for configurations containing (or not containing) certain specified features.

4.1 Correctness Determination

Given a regular system, the following algorithm determines whether the given complete configuration is correct:
Choose any regular serialization of the transactions.
Repeatedly execute the following steps until the initial complete correct configuration is obtained, establishing the correctness of the complete configuration given, or until a configuration is obtained that does not reverse qualify for its last generator, establishing the incorrectness of the proposed complete configuration:

i. Find the generator of every object version in the configuration.
ii. Pick from this set of generators, the generator (transaction) that occurs last in the serialization.
iii. Reverse apply the transaction to the configuration, provided that the configuration reverse qualifies for the transaction, to yield the new configuration for the next iteration.

This algorithm is linear in the number of objects and in the number of transactions in the system. Its correctness follows directly from the exhaustion theorem, the fact that exactly one transaction writes an object version, and the uniqueness of the configuration generated by reverse applying a transaction to a configuration that reverse qualifies.

If the system is not regular, the above algorithm can be used with the modification that Step 2 must now be repeated for all possible serializations of the transaction dependence graph before declaring a configuration incorrect. The algorithm is terminated if the correctness of the given configuration is demonstrated in any serialization.

4.2 Configurations Containing Specified Features

Transactions encapsulate meaningful changes to the systems, and each transaction identifier can be mapped into a string that identifies corresponding changes. Thus, there could be a transaction called <add-feature-A> that possibly updates several objects. A system configuration "has" <add-feature-A>, if for each object updated by the transaction <add-feature-A>, the configuration uses a version that is reachable in the version-graph from the version generated by <add-feature-A>. The user can then ask for configurations that have <add-feature-A>, <add-bell-B>, <add-whistle-C>, etc. The following algorithm determines such configurations in a regular system:
Start with the initial complete correct configuration.
Choose any regular serialization of the transactions.
Repeatedly execute the following steps until all the transactions have been processed:

i. Take the next transaction in the serialization and apply it to all the qualifying configurations for this transaction to generate new complete correct configurations.
ii. If the transaction just applied corresponds to one of the features selected, delete all previously known complete correct configurations, and retain only those generated as a result of applying this transaction.

The set (possibly empty) of complete configurations finally left reflect the features added by all the specified transactions.
The correctness of this algorithm also follows directly from the exhaustion theorem and the fact that exactly one transaction writes an object version.

4.3 Configurations Not Containing Specified Features

If correct configurations are required that "do not include" certain features, the removal of the corresponding transactions from the serialization will ensure that no configuration is generated that includes these features. It may be more efficient not just to remove the undesirable transactions, but also all transactions that can be reached from them in the transaction dependence graph. The desired configurations can then be generated by starting with the initial correct configuration and applying all transactions, in order as per the truncated serialization, to the qualifying configurations.

5. CONCLUSIONS

We considered the problem of configuring a system comprising one version for each of its constituent objects. We presented a transaction model and a syntactic characterization of a correct configuration tied to this model, which equates correct configurations with consistent database states. We allowed version graphs that are trees. We also presented algorithms for generating all correct configurations and those configurations that satisfy some selection criteria such as absence and presence of specified features, and for checking whether a user-specified complete configuration is correct.

We feel that the theory and algorithms presented in this paper can provide the basis for developing automated system configuration tools, and help advance the current state of art in which very little is available to users by way of support in deciding and verifying what constitutes a correct configuration. Indeed, we plan to incorporate these algorithms in Ode [1] — an object-oriented database and environment being developed in our laboratory.

We intend to address in the future two limitations of the current work. One is the requirement that version graphs be trees. In some cases, especially in software engineering environments, DAGs are more appropriate. The second limitation is that the algorithm presented in Section 4.1 can be applied efficiently only to check whether a complete configuration is correct. Extension of the algorithm to incomplete configurations is non-trivial.

REFERENCES

[1] R. Agrawal and N. H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model", *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989.

[2] T. M. Atwood, "An Object-Oriented DBMS for Design Support Applications", *Proc. IEEE 1st Int'l Conf. Computer-Aided Technologies*, Montreal, Canada, Sept. 1985, 299-307.

[3] D. Batory and W. Kim, "Supporting Version of VLSI CAD Objects", *IEEE Trans. Software Eng.*, 1986.

[4] D. Beech and B. Mahbod, "Generalized Version Control in an Object-Oriented Database", *Proc. IEEE 4th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1988, 14-22.

[5] P. A. Bernstein, "Database Support for Software Engineering", *Proc. 9th IEEE Int'l Conf. Software Eng.*, March 1987, 166-178.

[6] P. A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[7] H. T. Chou and W. Kim, "A Unifying Framework for Version Control in a CAD Environment", *Proc. 12th Int'l Conf. on Very Large Databases*, Kyoto, Japan, Aug. 1986, 336-344.

[8] K. Dittrich and R. Lorie, "Version Support for Engineering Database Systems", Rep. RJ4769, IBM Research Lab., San Jose, California, July 1985.

[9] R. Katz, E. Chang and E. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986.

[10] R. Katz and E. Chang, "Managing Change in a Computer-Aided Design Database", *Proc. of the 13th Int'l Conf. on Very Large Databases*, Brighton, England, Sep. 1987, 455-462.

[11] W. Kim, E. Bertino and J. Garza, "Composite Objects Revisited", *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989.

[12] D. E. Perry, "Version Control in the INSCAPE Environment", *Proc. 9th IEEE Int'l Conf. Software Eng.*, March 1987, 142-149.

[13] J. L. Steffin, "Interactive Examination of a C Program with Cscope", *Proc. USENIX Winter Conf.*, Dallas, TX, 1985, 170-175.

[14] W. Tichy, "Tools for Software Configuration Management", *Int'l Workshop Software Version and Configuration Control*, Grassau, FRG, Jan. 1988.

[15] J. F. H. Winkler, "Version Control in Families of Large Programs", *Proc. 9th IEEE Int'l Conf. Software Eng.*, March 1987, 150-161.