A Database Integrity Monitor with Applications to Vehicular Technology*

K. H. Jones ‡ S. Olariu ‡ L. F. Rowell§ J. L. Schwing ‥ A. Willhite‡

August 26, 1999

Abstract

One of the challenges facing the designers of an integrated engineering system is to blend in a robust and efficient way a wide variety of independently developed programs, each with its specific requirements for input and output. To support a high-productivity design and engineering environment, a tool is needed to identify potential integrity violations introduced in the database as a result of activities involving manually changing the contents of some variables, and/or the subsequent execution of a number of analysis programs on the modified data. Such activities are routinely performed by design engineers in the aerospace industry, vehicular technology, and user mobility management in cellular systems.

It is therefore important to create an environment that allows the designers to perform “what if” experiments while, at the same time, monitoring the integrity of the underlying database. A fundamental task of such an environment is to make the user aware of potential integrity violations introduced in the database as a result of any data changes. A trigger technique is described for identifying, beforehand, the set of all variables in the database susceptible to inconsistency.

We offer a general solution to the integrity checking problem as defined above. Our integrity monitoring algorithms are transitive closure-based and they compare favorably with the state of the art both in simplicity and in efficiency. To illustrate, we show how our solution can be implemented in the Environment for Application Software Integration and Execution (EASIE, for short) developed at NASA Langley Research Center (LaRC).

*Work supported by NASA grants NCC1-99 and NASI-19858 and by ONR grants N00014-95-1-0779 and N00014-97-1-0526.
Corresponding author: Prof. Stephan Olariu olariu@cs.odu.edu
‡Analysis and Computation Division, NASA Langley Research Center
§Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162, U.S.A.
¶Advanced Vehicle Division, NASA Langley Research Center
‖Department of Computer Science, Central Washington University, Ellensburg, WA 98926, U.S.A.
©Advanced Vehicle Division, NASA Langley Research Center
**Index Terms:** integrity checking, databases, data engineering, mobility management, cellular systems, EASIE.

1 Introduction

Numerous papers have been written on specifying and monitoring integrity constraints in various database application ranging from query and transaction processing, to user mobility management in mobile cellular systems, to integrated engineering environments [3, 5, 7, 8, 11, 14, 19].

Modern approaches to integrity monitoring suggest generating triggers to alert the database user and/or manager of possible corruption of the data in the database [9, 10, 14]. For example, seamless mobility user management in a mobile cellular environment must constantly reconcile information in the user’s home registry (i.e. information about the user that exists in her home cell) and visitor registries in various cells that she is visiting [3, 4, 7, 11]. The purpose of this paper is to discuss transitive closure-based integrity monitoring algorithms.

In the context of the increased complexity of vehicular technology, engineering systems are crucial for supporting sophisticated design methodologies and analysis techniques. In order to increase productivity, the many iterations inherent in the design process have to be supported by a user-friendly, quick-turnaround, computer-based design system. The Environment for Application Software Integration and Execution (EASIE) developed at NASA Langley Research Center (LaRC) provides a methodology and a set of utility routines for a design team to build, maintain, and apply computer-aided design systems consisting of a large number of stand-alone analysis tools. Willhite et al. [16, 20] describe the development history of several integrated design systems and give an overview of needed future enhancements. EASIE implements these needed capabilities by configuring the entire system around a central database containing all the input and output variables for the integrated analysis programs, as illustrated in Figure 1.

Utilities exist for constructing the database schema, generating the routines to read from or write to the database, incorporating the analysis programs into the database, interactively reviewing and modifying values in the database, incorporating the analysis programs into an interactive executive for easy selection and execution, and building menus and procedures to assist the process.

The relationships between analysis programs and their input and output variables are described by a *template* as illustrated in Figure 2. An input template is, simply, a list of the input variables that are required for a particular program. Similarly, an output template
Figure 1: The central database and the application programs

is a list of the output variables which are to be inserted into the database by a particular program. These templates, once built, are associated by name with the particular analysis program to which they relate. This concept of program input and output templates enables the development of a single generic interactive editor which can display the input and output variables for any selected program by requesting the appropriate template [13, 15, 16, 17].

In typical design and engineering studies, use will be made of both commercially-available and in-house developed analysis tools. Among the techniques currently being used to combine independent analysis tools into design systems, loose-coupling of analysis programs is particularly attractive. The term loose-coupling describes a collection of programs, each of which has the capability to communicate with a central database and which access common data but which may be executed in any sequence.

2 The Integrity Problem

The loose coupling of EASIE allows the designer to iterate through system components as desired. During this process, the designer will use engineering insight to define and possibly modify those input parameters which may also be outputs defined by other analysis modules. This technique helps decrease development time but may introduce integrity violations into the database. The purpose of this work is to define techniques to identify those inconsistencies. Parameters susceptible of introducing inconsistencies will be flagged
in the database so they can be queried at any time to determine the consistency status.

The remainder of the paper will formalize the essence of the design process described above and will consider the solution of the database integrity problem in two cases. In the first case, the order in which the analysis programs will be executed is not known \textit{a priori} but is, rather, determined dynamically during the design process, perhaps depending on some intermediate value. In the second case, the designer has a predetermined ordering of analysis programs that will be involved in the current analysis.

The transitive closure is regarded as an important functionality of database systems [6], and considerable research has been devoted to devising algorithms for computing the transitive closure of a database relation [19].

One of our contributions is to use transitive closure-like algorithms for monitoring the integrity of data in an engineering database. The main feature of our algorithms is their simplicity and the fact that their performance compares favorably with the state of the art [1, 2, 6, 12, 18, 19].

For the purpose of this work, a central database is assumed, containing a set $V$ of variables. In addition, a set $A$ of loose-coupled analysis programs is also assumed. Every program $P \in A$ is specified by an input template $I_P$ containing the set of all the variables that are input to $P$, along with an output template $O_P$ describing the set of all the variables
that are output by \( P \), as illustrated in Figure 3.

\[
\begin{array}{ccc}
I_P & \longrightarrow & P \\
& \longrightarrow & O_P
\end{array}
\]

**Figure 3: Illustrating input/output templates**

For definiteness, a variable is referred to as \textit{in-out} if it is both input to some analysis program and output of some (possibly the same) program. A typical scenario inherent to the design activity, and arising in the process of verifying certain design hypotheses, calls for manually changing the contents of a set \( X \) of in-out variables in the database and for the subsequent execution of a set \( B \) of analysis programs. Needless to say, this course of action will result in a number of variables in the database whose values have been modified during the execution of the experiment. To help maintain the integrity of data in the database, a tool is needed to identify \textit{beforehand} all the variables whose values are potentially affected by this experiment. This capability would be useful in the very process of testing the design hypotheses since the engineer must become aware of the scope and of the potential impact of the changes performed. Of course, after the experiment is completed only those variables whose contents have actually been modified need be flagged in the database for later reference.

A variable \( w \) in the database is termed \textit{potentially inconsistent} if there exists an order of execution of the programs in \( B \) which results in a modification of the contents of \( w \). Similarly, a variable \( w \) in the database is termed \textit{inconsistent} if its value has actually changed during the experiment described above. It is useful to note that variables that are input variables but not in-out variables are never considered to be inconsistent. This is in accord with basic engineering principles that require a set of initial conditions to be input to the system under investigation.

In the remainder of this work it will be assumed that the set \( X \) contains in-out variables only. The procedure \texttt{Prune\_X}, illustrated in Figure 4, will be executed as a preprocessing step for the purpose of removing from \( X \) all variables which are not in-out.

It is easy to confirm that after executing \texttt{Prune\_X}, the set \( X \) contains in-out variables only. Furthermore, with the assumption that the database contains \( n \) variables and that there are, altogether, \( p \) application programs, procedure \texttt{Prune\_X} runs in \( O(np) \) time.

In general, the order in which the analysis programs in \( B \) are to be executed is dictated by the intermediate results of the experiment being conducted. Under these conditions is seems to be very hard to specify the set of inconsistent variables \textit{without} actually running
Procedure Prune:X;
{Input: a set $X$ of variables and a set $A$ of application programs;
Output: the same set consisting of in-out variables only}
0. begin
1. $I \leftarrow O \leftarrow \emptyset$;
2. for every $P \in A$ do begin
3. $I \leftarrow I \cup I_P$;
4. $O \leftarrow O \cup O_P$;
5. end {for}
6. return($X \leftarrow X \cap I \cap O$)
7. end; {Prune:X}

Figure 4: Illustrating procedure Prune:X

the analysis programs in $B$. However, the problem of identifying the set $Y$ of all the potentially inconsistent variables in the database without actually running the programs in $B$ is solvable efficiently by using the input and output templates. In outline, the proposed algorithm does the following. To begin, the set $Y$ is initialized to $X$. Next, for all the programs $P$ in $B$ with $I_P \cap Y \neq \emptyset$, the following actions are taken: the output template for $P$ is added to $Y$ and $P$ is removed from $B$. To justify this, it is useful to note that all the potentially inconsistent variables contained in $O_P$ already belong to $Y$, making it unnecessary to add them again, at a later point. This process is continued until either $B$ is exhausted or else no remaining program in $B$ uses a variable in $Y$ as input. Note that the latter case can be easily detected using a simple boolean variable. The details of this simple algorithm are presented in Figure 5.

To obtain a bound on the running time it is convenient to assume that the number of the variables in the database is $n$ ($n \geq 0$) and that $B$ contains $k$ ($k \geq 0$) programs to be executed. Somewhat surprisingly, the running time of this procedure is independent of the number of variables in the set $X$. The following statement establishes the correctness of the procedure and argues about its running time.

Theorem 2.1. Procedure Find_Potentially_Inconsistent correctly determines the set of all the potentially inconsistent variables without actually executing the set $B$ of analysis programs. Furthermore, the running time of the procedure is $O(k^2n)$.

Proof. First, let $y$ be an arbitrary variable in $Y$. The proof of the fact that $y$ is potentially inconsistent proceeds by induction on the number $t$ ($t \geq 0$) of iterations of the while loop
**Procedure** Find\_Potentially\_Inconsistent($X$, $B$);

{Input: a set $X$ of variables and a set $B$ of analysis programs;}
Output: the set $Y$ of all the potentially inconsistent variables in the database}

0. begin
1. $Y \leftarrow X$;
2. done $\leftarrow$ false;
3. while ($B \neq \emptyset$ and not done) do begin
4. done $\leftarrow$ true;
5. for every $P \in B$ do
6. if $I_P \cap Y \neq \emptyset$ then begin
7. $Y \leftarrow Y \cup O_P$;
8. remove $P$ from $B$;
7. done $\leftarrow$ false;
10. end {if}
11. end; {while}
12. return($Y$)
13. end; {Find\_Potentially\_Inconsistent}

---

Figure 5: Illustrating Procedure Find\_Potentially\_Inconsistent

needed to add $y$ to $Y$ for the first time. If $t \geq 0$, then $y \in X$ and there is nothing to prove. Assume the statement true for all the variables added to $Y$ in fewer than $t$ iterations of the while loop. Since $y$ belongs to $Y$, it must have been added to $Y$ in line 7 of the procedure while the for loop in lines 5-10 was processing some program $P$ in $B$. However, $I_P \cap Y \neq \emptyset$, implying that some variable $z$ in $I_P$ must have been added to $Y$ in fewer than $t$ iterations of the while loop. By the induction hypothesis $z$ is potentially inconsistent, implying that $y$ is also potentially inconsistent.

Conversely, let $y$ be an arbitrary potentially inconsistent variable in the database. To show that $y$ must belong to $Y$, the proof proceeds by induction on the length $t$ ($t \geq 0$) of the shortest sequence of programs in $B$ which, when executed, result in a modification of the value of $y$. If $t \geq 0$, then $y \in X$ and line 1 in the procedure guarantees that $y \in Y$. Next, assume that every variable whose contents is modified by executing fewer than $t$ programs in $B$ is in $Y$. Let $P$ be the last program in the sequence of $t$ programs in $B$ whose execution modifies the contents of $y$. This happened because some variable $y'$ in $I_P$ was already potentially inconsistent. By the induction hypothesis, $y'$ belongs to $Y$. But now,
when \( P \) is processed in line 5, \( y \) will be added to \( Y \), as claimed.

To address the complexity, note that the while loop in lines 3-11 is executed at most \( k \) times. In each iteration of the while loop, the for loop in lines 5-10 takes

\[
O\left(\sum_{P \in B}(|I_P| + |O_P|)\right) \subseteq O(kn)
\]

time, if the appropriate data structure is used to maintain sets. Consequently, the entire procedure runs in \( O(k^2n) \) time, and the proof of Theorem 2.1 is complete. \( \square \)

To be able to identify and flag the set of all the inconsistent variables in the database when the analysis programs in the set \( B \) have been executed, it is assumed that the integrated system has a logging capability. Specifically, it is assumed feasible to determine the exact order of execution of the programs during the experiment. Now identifying the set of all the inconsistent variables can be done efficiently by the following procedure. It is assumed, without loss of generality, that the analysis programs in \( B \) have been executed in the sequence \( P_1, P_2, \ldots, P_m \), with all repetitions removed as a preprocessing step. Clearly, this can be accomplished by scanning the sequence once. The details are spelled out in Figure 6. The following result argues about the correctness and running time of the procedure

**Procedure** Find\_Inconsistent \((X, B)\);

\{Input: a set \( X \) of variables and a sequence \( P_1, P_2, \ldots, P_m \) of programs in \( B \);
Output: the set \( Z \) of all the resulting inconsistent variables\}

0. \begin{align*}
1. & \quad Z \leftarrow X; \\
2. & \quad \text{for } j \leftarrow 1 \text{ to } m \text{ do} \\
3. & \quad \quad \text{if } I_j \cap Z \neq \emptyset \text{ then} \\
4. & \quad \quad \quad Z \leftarrow Z \cup O_j; \\
5. & \quad \quad \text{return}(Z) \\
6. & \quad \text{end; \{Find\_Inconsistent\}}
\end{align*}

Figure 6: Illustrating Procedure Find\_Inconsistent

Find\_Inconsistent.

**Theorem 2.2.** Procedure Find\_Inconsistent correctly determines the set of all the inconsistent variables after having executed the set \( B \) of analysis programs. Furthermore, the running time of the procedure is \( O(mn) \).

**Proof.** To settle the correctness, it suffices to prove that a variable \( z \) is inconsistent after having executed the programs in \( B \) in the sequence \( P_1, P_2, \ldots, P_m \) if, and only if, \( z \) belongs
to $Z$. First, assume that $z$ belongs to $Z$. To show that $z$ must be inconsistent it suffices to use induction on the number $t$ ($t \geq 0$) of iterations of the for loop needed to add $z$ to $Z$ for the first time. If $t \geq 0$, then $z \in X$ and there is nothing to prove. Assume the statement true for all the variables added to $Z$ in fewer than $t$ iterations of the while loop. Since $z$ belongs to $Z$, it must have been added to $Z$ in line 4 of the procedure while $R_1$ was processed. However, $I_t \cap Y \neq \emptyset$, implying that some variable $z'$ in $I_t$ must have been added to $Z$ in fewer than $t$ iterations. By the induction hypothesis $z'$ is inconsistent, implying that so is $z$.

Conversely, let $z$ be an inconsistent variable in the database resulting from the execution of the sequence $P_1, P_2, \ldots, P_m$. To show that $z$ must belong to $Z$, let $j$ be the the smallest subscript in the range $1 \leq j \leq r$ such that after executing the sequence $P_1, P_2, \ldots, P_j$ the value of $z$ is modified. The previous choice of $j$ guarantees that $z \in O_j$ and so $z$ will belong to $Z$ after line 4 has been executed in the $j$-th iteration of the for loop.

To address the complexity, note that the for loop in lines 2-4 is executed $m$ times and that each iteration takes at most $O(n)$ time, for a total complexity of $O(mn)$, as claimed. □

3 Conclusions

To support a high-productivity design and engineering environment, a tool is needed to identify potential integrity violations introduced in the database as a result of activities involving manually changing the contents of some variables, and/or the subsequent execution of a number of programs on the modified data. Such activities are routinely performed by design engineers in the aerospace industry, vehicular technology, and user mobility management in cellular systems [16, 20] and are a key ingredient in modern CAD applications.

It is therefore important to create an environment that allows the designers to perform “what if” experiments while, at the same time, monitoring the integrity of the underlying design database.

In this paper we addressed two integrity-related problems, namely:

(1) the problem of identifying beforehand the set of variables which may result in integrity violations in the database, and

(2) the problem of generating triggers to alert the users of potential inconsistencies.

Our solutions are transitive closure-based and involve flagging the affected variables in the database.

Two efficient algorithms have been proposed. The associated tools are currently being implemented in EASIE. These tools will give the designer a handle on the scope and impact of the changes involved. In addition, the problem of flagging all the affected variables in
the database after having run the analysis programs on the modified data is also discussed.

References


