Error Propagation Analysis of Real-Time Data-Intensive Applications*

Tei-Wei Kuo†, Doug Locke*, Farn Wang‡

†Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan 621, ROC
*Lockheed Martin Corporation, USA
‡Institute of Information Science, Academia Sinica, Taipei, Taiwan 115, ROC

Abstract

This paper proposes a methodology for high-level error propagation analysis of real-time data-intensive systems. A formal system in C-style programming language is proposed to provide a research framework for various issues on real-time system designs. A symbolic procedure is then presented to formally verify the amount of data errors tolerable to systems.

1 Introduction

Real-time data-intensive systems are often used to monitor and respond to the real world in a timely fashion. Theories and practice that apply to conventional databases do not directly apply to databases that deal with real-time data constantly on the change. This is mainly caused by the existence of an external environment that imposes new relations between the data objects in the systems and the real-world objects they model [8, 9, 16, 17]. Due to the dynamic nature of real world, data objects that model entities in the real world cannot in general be updated continually to perfectly track the dynamics of real-world entities. The time needed to perform an update alone necessarily introduces a time delay that introduces an error.

This paper proposes a methodology for high-level error propagation analysis of real-time data-intensive systems. Its aim is to provide a formal mechanism to justify the amount of data error tolerable to the system. We have designed a C-like description language called Structured Language for Interactive Data-intensive Systems (SLIDS) based on an event-driven model as a research framework for various issues on real-time system design, including language compilation, simulation, verification, and synthesis.

Distinct from the past work [11, 12, 13, 14], the methodology proposed in this paper lets users program both a real-time data-intensive system and the environment external to the system. Also, because of its very different goal in data error analysis, an automated verification process must be done directly on the system and environment programmed in SLIDS. The formal model and design purposes behind SLIDS distinguishes it from other work. For example, PERTS [11], the communication graph model [12], and the System Description Language (SDL) [14] let users explicitly define the timing constraints of processes, whereas certain timing constraints such as period are described implicitly in SLIDS (similar to the way in which many real-world event-driven systems have been built) in terms of regular clock events. The specification of available resources such as memory and the implementation details of the system are ignored in this paper because the error analysis methodology proposed in this paper is to investigate whether the design of the system can possibly satisfy a specified amount of data error (the feasibility problem). Finally, we formally define the problem of error propagation analysis in such a framework. The problem does not fall within the realm of past verification work, such as HyTech[1, 5], but rather demands new insight which leads to a new symbolic procedure for computing the tolerable data errors in system data-modification processes.

2 Real-time data objects and errors

A real-time database is a collection of data objects that are used to model the external environment. The
external environment consists of a collection of environment objects which represent the real-world entities of interest to the system. For the purpose of environment modeling, we assume the existence of certain physical processes which have write access to each individual environment object, and the processes change the status of the environment (objects). Note that an environment object may physically exist or be a composition of some other environment objects. For example, the distance of a train to a train station is an environment object.

Because of the physical limitation of the real world, no environment object can change its value at an unlimited speed\(^1\). We adopt the behavior description methods of giving constraints on the rate of value changing to continuous objects as in hybrid system research [5, 6]. Discrete objects can be treated as special continuous objects whose change rate is always zero.

Although rate constraints of environment objects provide engineers a way to formally analyze errors due to insufficient sampling speed of environment objects, there are still many kinds of errors happening in a system. The technical question is how to restrict and analyze the error propagation or even amplification in a system. In the following sections, we should provide a way to model a real-time system and a mechanism to analyze the error propagation in the system.

3 System model description

A system \( \Omega \) consists of a set of programs written in SLIDS. An execution of a program is called a transaction. Programs are classified into two types: environment and internal. In programs, we may use statements to manipulate variables and branch according to variables' values. Statements can be organized into composite statements from atomic statements.

In the following, we shall use boldface font to denote reserved words in SLIDS. Also square brackets ('[', ']'), parentheses ('(', ')'), arithmetic operators ('+', '-'), relational operators ('\leq', '<', '\geq', '>', '=\', '\neq\'), and assignment operator (':=\') are reserved tokens with special meaning in SLIDS. Comma (',') is used to separate two integers in an interval expression. Semicolon (';') is used as statement terminator as in C-programming. We also use character \( x \) for derived variables, \( y \) for environment variables, \( w \) for port variables, and \( z \) for general variables.

\(^1\)Only environment objects guarded by the law of inertia are considered in this paper.

3.1 Constant, variable, expression, and condition

Variables in SLIDS can be of type integer, real, or boolean. Constants are either rational or boolean. Boolean constants are either \textit{true} or \textit{false}. Variables are either introduced for the purpose of computation inside transactions or used to denote data objects in the system. Variables corresponding to environment objects are called environment variables; variables not corresponding to environment objects are called derived variables. Expressions are linear combinations of derived variables and constants. No environment variable is involved in any expression. An expression \( \eta \) can be constructed from finite applications of the following rules:

\[
\eta ::= c \mid cx \mid \eta + \eta \mid \eta - \eta \mid (\eta)
\]

where \( c \) is a rational constant, \( x \) is a real number variable. For example, both \( x + 5y - 3 \) and 2.8 are expressions.

Any condition \( \theta \) can be constructed from finite applications of the following rules:

\[
\theta ::= \eta \circ \eta \mid \neg \theta \mid \theta \lor \theta \mid (\theta) \mid \text{true} \mid \text{false} \mid \emptyset(w)
\]

\( \circ ::= \leq \mid < \mid \geq \mid > \mid = \mid \neq \)

where \( \theta \) is an expression, and \emptyset(w) is a function which returns the status of a communication port (to be defined later). \emptyset(w) is \textit{true} if Port \( w \) is empty; otherwise, it returns \textit{false}. Note that standard shorthands on Boolean conditions like \( \theta_1 \land \theta_2 \equiv \neg((\neg \theta_1) \lor (\neg \theta_2)) \), \( \theta_1 \rightarrow \theta_2 \equiv (\neg \theta_1) \lor \theta_2 \), and \( \theta_1 \leftrightarrow \theta_2 \equiv (\theta_1 \rightarrow \theta_2) \land (\theta_2 \rightarrow \theta_1) \) are accepted. For example, both \( \neg (x + 5 \leq 20) \) and 5 \( \leq 8 \) are conditions.

3.2 Atomic statements

3.2.1 Assignment

Any assignment statement \( S \) can be constructed from finite applications of the following rules:

\[
S ::= x := \eta; \mid \quad \text{reset environment } y := a \text{ with rate } [c, d]; \mid \quad \text{sample environment } y \text{ to } z;
\]

where \( \eta \) is an expression, \( x \) and \( y \) are a derived variable and an environment variable, respectively, and \( a, c, \) and \( d (c \leq d) \) are rational numbers. Only derived variables can be involved on both sides of a assignment statement of type "\( y := \eta \)."

Statement "\text{reset environment } y := a \text{ with rate } [c, d]\)" resets the value of environment variable \( y \) to
value $a$ and sets up the value changing rate of $y$ (ranging from $c$ to $d$) henceforth. Intuitively speaking, henceforth at any moment, the time derivation of $y$ is between $c$ and $d$. When the change-rate is zero, we may omit the with rate clause. When only the change-rate is modified at the execution, we may write “reset environment $y$ with rate $[c, d]$”. The value-resetting and rate-setting of environment variables can be done through the operations of proper actuators.

We intend to use this statement for both continuous and discrete objects. For discrete objects, their reset-statement never have a “with rate” clause.

Statement “sample environment $y$ to $z$;” reads from environment variable $y$ and writes into derived variable $z$. The sampling of environment variables can be done through proper sensors. We assume that assignment statements always execute successfully without possibly causing suspension or termination of the executing transactions.

### 3.2.2 Interprocess communication

Transactions may declare ports for interprocess communication (IPC). The receiver of a port is always the owner or creator of the port. There can be more than one writer for a port if there exist more than one transactions possibly writing into the port. Ports can be declared and created through a variable-style declaration such as “Port $p_1$”. Port $p_i$ of transaction $T_j$ can be accessed with identification $T_j : p_i$. Every port is associated with a size-one buffer. Each write operation overwrites the contents of the buffer and always executes successfully. Each read operation reads from the buffer$^2$. IPC statements can be constructed from finite applications of the following rules:

$$S ::=\ \text{read } w \text{ to } x; \ | \ \text{write } w \text{ from } x;$$

where $w$ is a port. For read-statement, $x$ is a derived variable. For write-statement, $x$ is either a derived variable or constant. Note that no environment variable can be involved in IPC statements. Statement read/write can read/write values from/into the corresponding buffer of the port. All IPC statements are non-blocking statements. When statement read reads from a port, it empties the port. When statement write writes into a port, the statement overwrites the buffer of the port if the port is not empty. Statement read will return a null symbol $\perp$ if it tries to read from an empty port. Note that empty($w$) returns the status of Port $w$.

### 3.2.3 Timed delay

Transactions may wait for a given amount of time. Two types of wait statements are supported in SLIDS:

$$S ::= \text{wait } [c, d] ; \ | \ \text{wait until } \theta ;$$

where $c \leq d \in \mathcal{N}$, and $\theta$ is a condition. Statement “wait $[c, d]$”, lets the executing transaction suspend itself for a randomized amount of time ranging from $c$ to $d$. Statement “wait until $\theta$,” lets the executing transaction suspend itself until condition $\theta$ is satisfied. As soon as condition $\theta$ is satisfied, the transaction is ready. Note that condition $\theta$ in a “wait-until” statement is the only exception in SLIDS which may contain an expression involving environment variables such as “Clock”.

### 3.3 Control flow and composite statements

SLIDS allows composite statements to control the execution branching of programs. A composite statement $S$ can be constructed from finite applications of the following rules:

$$S ::= SS \ | \ \text{if } \theta \ \text{then } S \ \text{else } S \ | \ \text{while } \theta \ \text{do } S \ | \ \{S\}$$

where $\theta$ and $S$ are a condition and a statement, respectively.

Statement “SS” denotes a sequence of statements. Statement “if $\theta$ then $S$ else $S$” is a standard if-then-else statement. If the evaluation of condition $\theta$ is true, statement $S$ following reserved word “then” will be executed; otherwise, statement $S$ following reserved word “else” is executed. Statement “while $\theta$ do $S$” will repeatedly execute statement $S$ until condition $\theta$ fails. Composite statement $\{S\}$ is semantically equal to statement $S$.

### 3.4 Program

Programs are classified as either internal or environment. (Internal) programs are used to model the system operations inside a computer system. Environment programs are used to model the operations of the
A complete system is a collection of internal programs, environment programs, and declarations of environment variables.

### 3.4.1 Internal program

Each (internal) program in SLIDS is a (logical) transaction in the system. This means that an implementation of the system can merge several programs/transactions into one physical transaction. However, the implementation must not violate the semantics of program/transactions defined in terms of SLIDS. Any verification of properties of the system will be based on the definitions of SLIDS programs/transactions.

Each program can be derived from the following rule:

\[
P ::= \text{Program name } \{ \text{declarations } S \}
\]

where name is a string of characters which uniquely identifies the program, declarations is for definitions of local derived variables, and \( S \) is a composite statement. Declaration of derived variables is not strongly enforced.

### 3.4.2 Environment program

The operations of environment objects/variables can be modeled as a collections of environment programs. However, the composite statements of environment programs shall not access any derived variable in the system.

\[
P ::= \text{Environment Program name } \{ S \}
\]

As can be seen, environment programs share the same syntax rules with internal programs.

## 4 Symbolic error propagation analysis

Intuitively, given a real-time data-intensive system \( S \), a variable \( x \in X_S \), and an error bound \( \beta \), the corresponding temporal propagation analysis problem instance asks if during any computation, the error \( \epsilon_x \) between \( x \) and its real intended expression value can be greater than \( \beta \).

In our approach, we use symbolic procedure\[5, 6\] for the analysis of temporal error propagation. Thus we can translate our programs into hybrid automatas and map program statements into assignment statements performed at each transition.

In the verification of hybrid systems\[1, 4, 5, 6, 7\], the state space is partitioned into regions which can be defined by a conjunction of linear inequalities. In a multidimensional space, such a conjunction represents a convex hull. The verification of hybrid systems relies on the search of state graphs whose nodes are these regions and arcs are transitions between states in these regions.

It turns out that our error-propagation analysis problem falls beyond the regime of traditional symbolic analysis procedure\[1, 2, 3, 4, 5, 6, 7, 10\]. The issue here is how we define intended expression value of each variable. According to our observation, the intended expression value of a variable is dynamic with respect to the computation of the systems. Each assignment to a variable dynamically changes the variable's intended expression value and thus its error. Due to this dynamic nature, we adopt the following rules to compute the intended expression value of a variable along a given computation.

- The set of variables in a system may include:
  - environment variables. Such a variable may change its value along the time line with a rate interval.
  - derived variables. A derived variable's value may not change until an assignment statement to it is performed.
  - port variables, which are considered as a special type of derived variables. A read operation is considered as an assignment from a port variable. A write operation is considered as an assignment to a port variable.

- Initially, the intended expression of each derived variable is zero while the intended expression of each environment variable is itself.

- After the execution of a statement like “write \( x \) to \( P \)” the intended expression of \( P \) is changed to that of \( x \).

- After the execution of a statement like “read \( P \) to \( x \)” the intended expression of \( x \) is changed to that of \( P \).

- After the execution of a statement like “\( x := f(z_1, \ldots, z_n) \)” where \( f(z_1, \ldots, z_n) \) is a linear term over variable \( z_1, \ldots, z_n \), the intended expression of \( x \) is changed to \( f(\xi_1, \ldots, \xi_n) \) where \( \xi_i \) is the intended expressions for variable \( z_i \) right before the assignment.

Thus the symbolic analysis procedure defined in \[5, 6\] can be adapted to incorporate a conjunct like...
in the conjunction of every region for every derived variable in the system, such that \( \zeta_x \) is a linear term representing the intended expression value of \( x \). Such conjuncts define intended expression of derived variables and also participate in defining the convex hulls of regions.

The relation among \( x, \zeta_x \), and \( \zeta_x \) must be maintained by doing adjustment described in the just-mentioned rules. The error-propagation analysis problem instance can be answered by seeing if there is a convex hull region with \( \zeta_x > \beta \) reachable from the initial convex hull region.

5 Conclusion

With the scale and the complexity of computer systems built today, we are quickly approaching the limit in managing sophisticated software systems. This paper proposes a methodology for high-level error propagation analysis of real-time data-intensive systems. It aims at providing a formal mechanism to justify the amount of data error tolerable to the system. We propose a C-like description language SLIDS based on an event-driven model as a research framework for various issues on real-time system designs. We must point out that SLIDS is not intended to be a stand-alone description language. Rather, SLIDS is a component of a prototyping, analysis, synthesis, and simulation environment, and any system programmed in SLIDS can be easily transformed into an executable on many platforms for simulation. The most important idea, however, is that we now have a way to formally verify the data error of a real-time data-intensive system. The problem does not fall within the realm of prior verification work, such as HyTech[1, 5], and demands new insight which leads to a new symbolic procedure for the analysis of the tolerable data error in system data-modification processes. We expect that the formal problem definition and new symbolic analysis procedure can be implemented as an automatic tool and bring in the advantage of computer-aided verification.

References


