FAST-zero '11

First International Symposium on
Future Active Safety Technology
toward zero-traffic-accident
September 5-9,2011, Tokyo, JAPAN

JSAE 2011

Logging Design for Vehicle Communication Field Operational Tests

Martin Goralczyk, Bernd Schaeufele, Ilja Radusch
Berlin Institute of Technology

TU Berlin, FR5-14, Franklinstr. 28-29, Berlin, 10587, Germany (E-Mail: martin.goralczyk@tu-berlin.de)

ABSTRACT: Large vehicle communication field operational tests with many cars and a lot of new functions under test produce vast amounts of log data. Since the logging runs on the system under test, the requirements are to make minimal use of CPU and RAM to not affect the test environment, be most flexible in configuration issues, and to be fail safe and secure. This paper covers the upcoming logging problems in big field operational tests and presents a solution on how to meet them on basis of the logging that has been implemented for the sim^{TD} field operational test.

KEY WORDS: logging, FOT, sim^{TD}, c2x

1. Introduction

In the last years a vast number of smaller field operational tests in the automotive industry has been performed to test applications on the basis of vehicle-to-vehicle or vehicle-to-infrastructure communication (V2X). These applications can generally be associated to the safety, traffic efficiency, or infotainment sector and have to be tested in the real world before parts of them can be introduced to customers. Most of the last years' projects like NoW [(1)], FleetNet [(2)], or PReVENT [(3)] had set their focus on a certain part of applications, like active safety or mobile internet access. As opposed to this the project sim^{TD}, started in late 2008, comprises over 20 applications from all sectors and is scheduled to test them in the Frankfurt-Rhine-Main area of Hesse with several hundreds of ITS Stations consisting of ITS Vehicle Stations and ITS Roadside Stations. The sim^{TD}Logging will showcase the restrictions, challenges, and the according solution.

In section 2 we will refer to related work in the area of individual logging solutions with specific requirements. Section 3 describes the project sim^{TD}, which provides the context for this paper. Section 4 offers a presentation of the chosen logging architecture in detail. In section 5 we present the results of the performance tests we made, regarding the database import. We complete our paper with a discussion in section 6 and a short summary in section 7.

2. Related Work

A great survey about logging was composed by Ruffin in 1994 [(4)], which describes all the fundamental concepts and definitions in the context of logging. It serves as a composition of the basic concepts of logging without special requirements on top of it.

When a new logging software gets developed, it seems to be certain, that existing solutions are not flexible enough to match a special demand. An example is shown, where Lee et al [(5)] describe, that if the aspect of battery life of mobile nodes is the field of attention, an algorithm that adapts the variable logging rate based on a situation analysis is able to extend the battery life for another 143 %.

Finlayson et al [(6)] describe Clio, a logging service that abstracts from the log files and the file system, so features like append-only and read-only logs can be system inherent, indifferent on what physical storage the files are written. Both approaches are valid for their specific requirements, but unsuitable for the Field Operational Tests (FOT) in sim^{TD}, where the specific requirement for the logging is the efficient and database-import friendly encoding of information. The examples show, that if there are critical conditions, like battery life or in sim^{TD} the vast amount of data to analyze, the logging has to accommodate to these challenging conditions.

3. Logging Requirements in sim^{TD}

As successor in a line of many other FOTs, which have analyzed only parts of automotive future, sim^{TD} stands out to test a large set of applications, all based on vehicle-to-vehicle and vehicle-to-infrastructure (V2X) communication to improve road safety and traffic efficiency, and also to integrate value-added services. The project started in August of 2008 and runs for four years. As a bottom-up approach the functions will be tested first on the test bench, then on a closed test area, and finally in the Frankfurt-Rhine-Main area of Hesse. In this context, a vehicle is called ITS Vehicle Station (IVS) and it communicates with other IVSs or one of the installed ITS Roadside Stations (IRS). The latter are stationary stations, located at strategic points in the test field to provide a way of communication between the IVSs and the ITS Central Station (ICS), a set of servers, which are providing central services like traffic or weather information.

There are two computing devices in the IVSs and IRSs. The Communication & Control Unit (CCU) handles all the communications and the Application Unit (AU) is the host of the applications. The stations use various transport media. sim^{TD} will evaluate the usage of the well-established WLAN 802.11b/g standard and also the 802.11p standard that is modified for automotive purposes. Moreover, cellular communication to the internet in form of HSPA is available on the CCU and trough it on the AU, and also in the region of the field-test.

To learn the specific requirements to the logging in sim^{TD}, we started with the user of the logs, the evaluation and analysis. All the logs have to be analyzed against metrics to be defined during the project, so the analysis has to be as flexible as possible to set different logs in relation to each other.

The next step was the evaluation of the expected amount of data. Since, only regarding the field test without breaks, the testing interval totals on eight months and the number of ITS stations is determined, we had to estimate the daily amount of logs per station. These prognoses lead to a scenario, where we had to cope with about 30 terabyte of raw log data. Recently the prognoses were refined, what lead to an even higher value of about 65 terabyte.

Having this knowledge, it is most important to store and transfer the needed information as space-saving as possible to make it possible to cope with this big amount of data during logging on the ITS stations, transferring the logs to the data center, and at last analyzing all the data without the need of preprocessing or converting the files into another format.

Since the test software, of which the logging is one part, has no dedicated machine, but instead runs on the AU as well as the functions, the logging has to be very economizing in the means of CPU and RAM. Moreover, since logging makes use of hardware performance and storage space, logs have to be checked individually and discarded in special circumstances. Being part of the AU, logging should not only be able to log items that get pushed to it, it should also pull central available information for logging, following given rules. The log configuration defines, which logs are discarded and which information is pulled from other test software. These subscription rules in combination with the log white list comprise the log configuration.

Since in a field test the logs are not anonymized for obvious reasons, the ways of transportation have to be secured, to prevent unauthorized people to analyze the logs. On the other hand, the integrity has to be preserved to ensure, that the logs have not been changed since their creation.

4. Logging Concept

Derived from the requirements we describe the concept that we implemented for sim^{TD} beginning with the point of main effort: the homogenization of log data. To meet the requirement that all logs should be analyzed as flexible as possible, the logs are homogenized to all match one database table. This table has columns for the id of the ITS station, the time stamp the log was taken at, the id of the log type and a maximum of ten parameters, which are part of the definition of the log type. With this basic idea, a comprehensive definition of the measurands in sim^{TD} has to be made. We used a database with a simple web-interface to allow project partners to define measurands. Using the paradigm that only suppliers can define measurands, recipients have to demand their needs, if certain measurands are missing. Being defined, up to ten measurands can be assembled to logs. An example for measurands are the longitude, the latitude, the speed, and the heading a GPS receiver measures periodically. When it comes to logging, these values are combined to one log, but e.g. the position can be combined with an obstacle detection to another log. Generally spoken, a measurand is at least defined by its source and data type, and its definition is the prerequisite to be logged. To reduce the work of defining measurands and putting them in a second step as singles into logs, we defined, that measurands, not part of any logs, will be handled as logs, too.

Part of the definition of a measurand is the data type. We need strict definitions of measurands, because we cannot allow various data types in the analysis. To be there most flexible we had to focus on one data type. Regarding the given data type, the value has to be transformed to integer, because this is the most versatile data type within a size of 4 Bytes. Bigger types can handle more data, but since all measurands have to be converted to one common type, the choice fell onto integer. The data type long can store bigger values, but it would double the required disk space of the database system.

We support integer, float, and enumerations as data types. If a measurand is an integer, no transformation has to be done, but if a measurand is represented by a floating-point number, it has to be converted to integer. In our data, we expect most of the floating-point data to fit into the data type integer with sufficient accuracy, while the decimal place is part of the abstraction. For example if the measurand latitude is defined with an accuracy of six decimal places, the value *1.2345678* will be saved in the log as *1234567* and then decoded as *1.234567*.

The last type we support actively are enumerations. These objects are part of most programming languages and provide basically a mapping from a string to a number. This way system states, error codes, or other kinds of known countable strings can be encoded very efficiently. A very common example for an enumeration is the boolean state - false and true encoded as zero and one.

Using this transformation system, we support more data types for the definition of the measurands, even if the data handler in the background can only cope with integer. As a result the applications on the ITS stations as producers and the analysis software as consumer of the logging data do not have to take account of this abstraction, since it is transparent to them.

Figure 1 shows an overview of the reduction to integer. While the integer 123 can be directly used in the log, the decimal position of the float 12.34 has to be shifted two positions to the result 1234 and the value *"ok"* from the Enumeration *"ok", "warn"* will be transformed to zero.

Some measurands cannot be transformed to integer. For example, there are various message formats with diverse structures used in V2X communication. These cannot be transformed into sim^{TD} logs, because they consist of much more than ten values. Stack traces and navigation destinations are two more obvious examples that cannot be predefined. Another problem are Ids of the data type long, which cannot be reduced to integer. These can be divided into two integers or treated as an individual string. To match all requirements, the data type string is supported but not recommended, because logs in strings cannot be analyzed as efficient as integers with a machine-readable definition and it uses more disk space in the database.

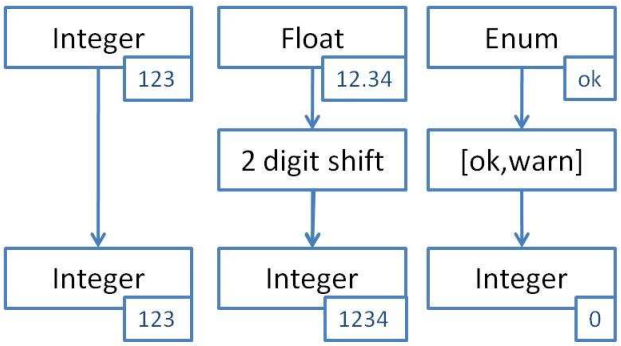

Figure 1: Overview of the reduction to integer

Summing up the definition part, everybody has to define every measurand he supplies and combine them to logs when they are measured at the same time. This definition serves as kind of a log manifest and also hosts the data type abstraction.

With the defined log manifest the rest of the process runs semi-automated. It was very important to us that changes in the definition lead to changes in the code. If we had decided for this type of abstraction but without the code generation, the logging would offer ten methods to log from one to ten parameters. Then the developers had to look into the definition and manually type in a log id and up to ten parameters in the right order, including the abstraction techniques decimal shift and enumeration mapping. This would be more than error-prone and errors like undefined log ids in the logs, not matching log ids to the number of given parameters, wrong shifting, or switched parameters are likely to happen and may even stay undetected in the analysis. Once the definition of measurands is finished and a non-generic logging is used, these features are provided without additional effort. Therefore, we built a code generator that uses the current definition, in our case a SQL database, and fetches all the needed information for the generating process. Product of the generator are on the one hand interface, implementation and all needed enumerations in Java for the logging parts running on the AU and also in C for the Logging demon, running on the CCU. Both components have to be built with the new sources and to be deployed onto the stations to let other components make use of new log definitions. All information about the abstraction from the database and from the code generating process is compiled into an XML file.

This file is used in another component, the log-decoder, to decode the log files to more valuable information. We have implemented sub-components to write different file types, like human readable log files, a Google Earth file for a view on a map, or a file containing just the decoded C2X messages.

The log decoder can also be used as a library by the analysts for very specific evaluations.

The generator itself works as follows. First, it gathers all defined enumerations and generates them. Next, the defined logging methods are generated. If the log or enumeration is called by a component on the CCU or the AU the corresponding method or enumeration will be added either to the Java code that runs on the AU or the C code that runs on the CCU. If a method is unknown, which means not part of the last known abstraction, a new log Id will be generated and used for this log. When supplying the code generator with the abstraction file of the last version, the new generation preserves known log Ids. This algorithm renounces a version control of every log. If a log changes in any aspect, it gets a new log id to prevent that two differently defined logs will be compared in the analysis directly, because of their matching log id. For obvious reasons an unchanged definition gets the same log id every time. In the last step for all measurands, which are not part of any log, log

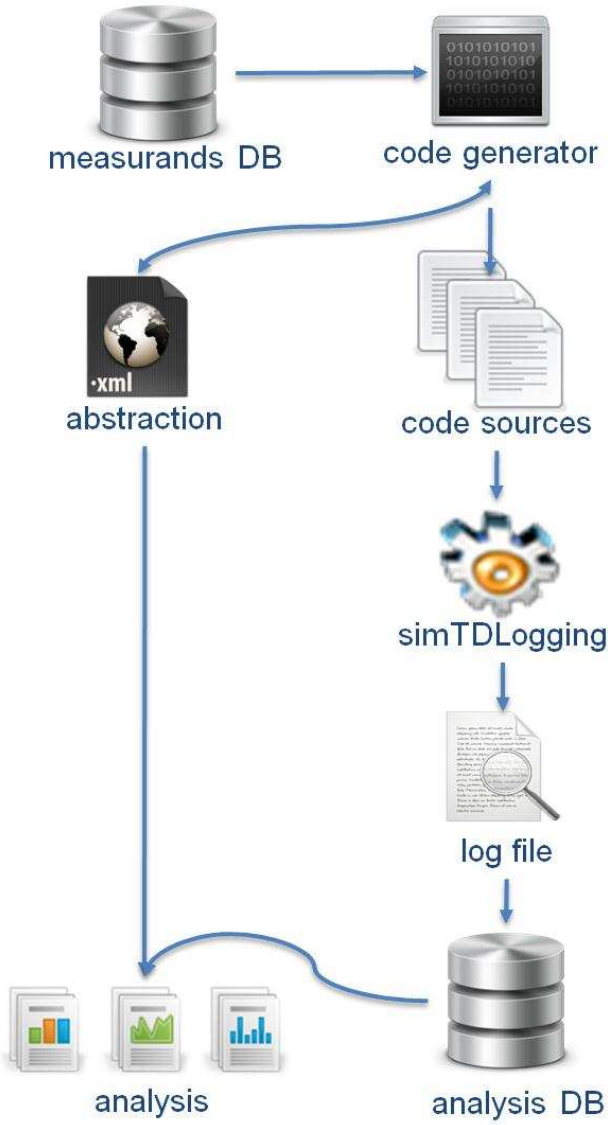

Figure 2: Overview of the log abstraction process

methods with only one parameter are generated. The last feature is just built in for being failsafe under the assumption that every defined measurand is meant to be logged.

Figure 2 shows an overview of the log abstraction process. The log definition is made in the measurands database. The code generator fetches it from there and produces all the needed code files and an abstraction in XML format. The code is compiled to the sim^{TD}Logging, which logs and writes the log files. These can be imported very easily and fast to a database for the analysis of the log data.

When releasing a new logging API (Java) or logging headers (C) all software developers in the sim^{TD} project now can use that API to call their defined and generated logging methods. They will be supported by their integrated development environment (IDE) with documentation and comments during this process, so this manual input process is as guided as possible to prevent careless mistakes.

After the log files are written, they have to get transported to a central data store safe and secure. To ensure that we are using hash files to check if log files have been changed since they were written. Moreover, log files are encrypted during the transportation from the IVS to the central file server, since the files are transported on USB drives. The wireless alternatives on the IVS cannot cope with the large amount of log data. The transport from IRS and ICS runs by wire and is secured by other means.

To ensure that log files, which were transported from the ITS stations to the central server, are not deleted on the ITS station until they are safe on the server, we implemented a file based protocol that uses so called delete instructions to delete only the successfully transported files. Files that did not get copied to the central file server successfully stay on the station that wrote them initially, and will get copied during the next transfer.

5. Logging Performance

When trying to measure the performance of the presented solution there are three values to be measured: the speed of the logging itself, the efficiency of the encoding, and the speed of the database import.

A big Java class with hundreds of log methods, like in our case, could react more slowly on method calls than a small class with a few generic log methods. To check, if our solution would sustain this important performance issue, we measured the return times of similar methods, once in a class with 5 methods and once in a class with 500 methods. The difference seems not to be measurable in milliseconds, because we could not find a performance advantage for one of the candidates.

The log files are zipped plain text files. We evaluated implementations for zip, gzip, bzip2 and 7zip by the means of packing efficiency. Therefore, we measured the time for packing a generated log file with a size of 1024KB and the resulting packed file size. The packed file size was nearly equal, but the zip implementation, which is part of the Java Runtime Environment (JRE), was the fastest, so we decided to zip the log files.

Moreover, we evaluated if a zipped text file with only 11 different characters (0-9 and ",") is competitive with binary encoding. We used a professional ASN1 encoder to get a binary representation of our generated test log file. Although the unpacked binary file was 64 % of the size of the text based log file, after zipping both files the binary zipped file was less than 1 % smaller than the zipped text file. So space-wise both approaches tie, but text files are more user friendly and most importantly can be used for very fast imports into a database.

After the execution of the field tests, the log data has to be analyzed for the evaluation of the tests. Analyses based on log files are not efficient, because for each analysis the files have to be parsed linearly. Thus, the log files are inherently designed to be imported into a database, so that complex queries can be performed efficiently. We describe how log files are imported and discuss the performance of the import process.

Normally in SQL databases, data is imported with the INSERT statement. As for each database entry an individual SQL query must be executed, this proceeding is slow for large amounts of data. Several database vendors have custom extensions to the SQL standard for importing large bulks of data. The securely established open source database management system PostgreSQL that is used in sim^{TD} offers the COPY statement for these purposes. With this command, a text file that contains comma-separated values can be imported as a whole into a database.

The data format of the files that are imported with the COPY statement must correspond to the schema of the database table it is inserted into. Therefore, the data within a file must be homogeneous, what can be guaranteed by the sim^{TD}Logging, see section 4. However, if the logs were as heterogeneous as if defined by the large number of functions and system components, and it was feasible to have an individual table in the database and an individual log file on the ITS stations for each log type, it would lead to the import of a large number of very small log files. The import of a large number of files into the database is very slow compared to the import of a smaller number of large files. Table 1 shows that the import of 10,000 logs it is about 1250 times faster from one log file than from 10,000 log files.

Table 1: Import Performance between 1 and 10,000 log files

number of logs	number of files	execution time (seconds)
10000	10000	935.847
10000	1	0.744

As the logs are homogenized, there are only two kinds of log files. The first contains logs with up to ten integer values, and the other logs with a string value. In order to match them to a database table, all logs have to contain the same amount of commas, even if containing fewer values. Keep in mind that these

files are stored zip compressed, and a series of the same byte can be compressed optimally.

The following listing is a section of an integer log file:

```
IVS001,1286264337230,610028,0,,,,,,,,
IVS001,1286264337231,610046,0,25,18,3000,,,,,,
IVS001,1286264337233,610038,30000,4,55,7,,,,,,
IVS001,1286264337234,610037,12,91758,230195,,,,,,,
IVS001,1286264337263,610033,0,143015,,,,,,,,
```

For comparison, here is a listing of a section of an string log file:

```
IVS001,1278515247,50004,"started"
IVS001,1278515248,50004,"activated"
IVS001,1278515249,50004,"running"
IVS001,1278515250,50007,"message received, ok"
IVS001,1278515250,50007,"new status: ""checked"""
```

The string values are set in quotation marks, because it is possible that the strings themselves contain commas. Quotation marks within the strings are escaped by a second quotation mark.

Accordingly, there are also only two kinds of tables in the database, the first one has ten columns for integer values, and the second one has one column for string values. Besides, each table has three more columns for the id of the station on that the log entry was created, the timestamp of the log entry and an id that identifies the type of log.

The log files are organized on the file server in a folder structure of the form <*date*>/<*stationid*>. The import is performed by a script that performs a batch import of log files into the database. It iterates over the folders of a given time span, unpacks the zipped log files of all stations, and imports them to the database.

The performance of the import was tested in order to compare the duration of imports with the COPY and the INSERT statement to demonstrate, why it was essential to adapt the logging format to be compatible with the COPY statement. The test files for the import were generated automatically and were randomly filled. The integer log entries contained between one and ten values, the number of values, and the values were generated randomly. The string files contained entries with a length of one to 50 characters. Both, the length of the entries and the characters were chosen at random, too. For generation, the Java random function was used.

Table 2: import performance between COPY and INSERT

test file (# lines,[int,string])	execution time COPY (seconds)	execution time INSERT (seconds)	speed up factor
10000, int	0.744	412.034	553.809
20000, int	0.748	848.889	1,134.878
30000, int	0.796	1,250.110	1,570.489
10000, string	0.712	346.430	486.558
20000, string	0.724	681.555	941.374
30000, string	0.776	990.090	1,275.889

Table 2 shows the results of the test. The column test file describes the number of lines in this file and whether the type is integer or string. The execution time for both import types, COPY and INSERT, is given in seconds. It is obvious, that the most amount of the execution time in the COPY command is independent from the number of lines to be imported, since the execution time for twice the data is only 0.5 % longer. In contrast to this the execution time in the INSERT command grows nearly linearly with the lines to copy. Having this in mind, the speed up factor of the copy command is rising with the file size to import and in our tests was about 500 for every 10,000 lines to import.

All tests were performed on a machine with an Intel Core 2 Duo processor at 2.8GHz and 4 GB of RAM.

6. Discussion

The presented logging architecture has a lot of advantages and some disadvantages.

Without homogenizing the logs in a project with dozens of partners and maybe hundreds of distributed developers, everybody would define the same logs for themselves with the same semantics but diverse syntax. For example for an *"OK status"* of an application, there would be strings defined like, *"OK"*, *"ok"*, *"running"*, *"all fine"*, *"Status OK"*. Without central harmonization, the analysis would not be feasible.

On the downside, the process is very strict and does not allow logging new measurands spontaneously. For new logs, a new sim^{TD}Logging binary has to be compiled and installed, and this cannot be done by everyone. So usually the developer has to define new measurands, wait for a new logging version, implement those new logs in the own component, and at finally deliver this component officially to the integrator. According to the fact, that these steps take weeks, there is a big latency between the definition of logs and the logging of them.

To cope with this problem, we have developed a slightly different proposal. The code generating process could be built into a web application. Then the developer would define his measurands and logs in an XML document and supply it to the web application. If there were no errors in the XML, the developer would get a new version of his application specific log bundle directly as download, and could use the new defined logs instantly. These application specific bundles would only be used as stubs to compose the logs and forward them to the general logging, which writes the logs into files and transfers them. These generated bundles were only regarding the one application, they were built for, so there were no dependencies to other applications and therefore no periods of restriction within the integration. The XML files, which were delivered by the developers, would be, combined into one document, the log abstraction for further usage.

This variation would not improve any of the fundamental advantages of sim^{TD}Logging, but would be a help for the developers. Apart from that, it is to examine, if the proposed dispersion of logging functionality over many sub-components is tolerably slower, especially in the OSGI context.

Even though an abstraction is less performant than directly handling the raw data, the advantage in form of fast and flexible analysis due to structured and uniform log data is more appealing. Moreover, the log files do not waste more disk space than if they were coded binary and then zipped, and they are easy to handle due to their simple format.

We provide a log decoder that can be directly used to decode the log files to human readable ones or can be used as library to develop own analysis tools with transparent usage of the abstraction.

7. Conclusion

In this paper, we described a novel log architecture that is very typecast and standardized and requires a comprehensive definition of the measurands to be logged. We used that definition to generate source code for our log implementation to ensure that the defined abstraction rules are implemented without careless mistakes or flaws and to give maximal documentary support on calling those methods manually. The main objectives were to cope with a very big amount of log data during the logging, the transport, the storage, and the analysis. We have shown that with our solution we are very flexible and effective on handling big amounts of data. The draw-backs of our architecture are the mandatory definition of logs and the abstraction that is implemented into the logging.

Acknowledgement

This work was funded within the project sim^{TD} by the German Federal Ministries of Economics and Technology as well as Education and Research, and supported by the Federal Ministry of Transport, Building, and Urban Development.

References

(1) A. Festag, G. Noecker, M. Strassberger, A. Luebke, M. T.-M. B. Bochow and, S. Schnaufer, R. Eigner, C. Catrinescu, and J. Kunisch : Now - network on wheels: Project objectives, technology and achievements, in Proceedings of 5rd International Workshop on Intelligent Transportation (WIT), Hamburg, Germany, pp. 211–216. (2008).
(2) W. Franz, R. Eberhardt, and T. Luckenbach：“Fleetnet - internet on the road, in Proceedings of 8th World Congress on Intelligent Transport Systems, Sydney, Australia, (2001)
(3) Prevent homepage. [Online]. Available: http://www.prevent-ip.org/
(4) M. Ruffin：A survey of logging uses, INRIA (France), Tech. Rep. BROADCAST-36. Also available as Univ. of Glasgow (Scotland), Tech. Rep, vol. 2, pp. 94–82, (1994).
(5) C. oh Lee, M. Lee, and D. Han：Energy-efficient location logging for mobile device, in Applications and the Internet (SAINT), 10th IEEE/IPSJ International Symposium, pp. 84 –90, (2010).
(6) R. Finlayson and D. Cheriton：Log files: an extended file service exploiting write-once storage, SIGOPS Oper. Syst. Rev., vol. 21, pp. 139–148, (1987).