Appeared in Sixth Conference on Prestigious Applications of Intelligent Systems - PAIS 2010, Lisbon

A Fault-model-based Debugging Aid for Data Warehouse Applications

Peter Struss[1], Vikas Shivashankar[2], Mohamed Zahoor[3]

Abstract. The paper describes a model-based approach to developing a general tool for localizing faults in applications of data warehouse technology. A model of the application is configured from a library of generic models of standard (types of) modules and exploited by a consistency-based diagnosis algorithm, originally used for diagnosing physical devices. Observing intermediate results can require high efforts or even be impossible, which limits the discriminability between different faults in a sequence of data processing steps. To compensate for this, fault models are used. This becomes a feasible solution for standard modules of a data warehouse application along with a stratification of the data. Fault models capture the potential impact of faults of process steps and data transfer on the data strata as well as on sets of data. Reflecting the nature of the initial symptoms and of the potential checks, these descriptions are stated at a qualitative level. The solution has been validated in customer report generation of a provider of mobile phone services.

1 INTRODUCTION

One of the most urgent needs these days is to effectively support debugging of software, which becomes an ever increasing factor to determine both the industrial and commercial sphere and our personal lives. One of the most successful techniques of model-based problem solving is component-oriented consistency-based diagnosis (see [Struss 08]). Exploiting this technology, which has helped to localize and identify faults in physical systems, for software debugging has been pursued for quite some time (see [Mayer-Stumptner 07] for a comprehensive and systematic survey).

Component-oriented consistency-based diagnosis is based on the idea of using a system model structured along components and their connections which captures the dependencies of observable (intermediate) responses of the systems on its various components. Based on this, a deviation of an actual response from a predicted one can be assigned to a certain set of components. This idea of exploiting dependencies in the model has been exploited also for software systems, where "components" can range from the level of statements to higher-level functional[1] entities.

There are a number of obstacles that hamper a straightforward transfer of consistency-based diagnosis techniques to software debugging. The most fundamental one is the difference between diagnosis of (well-designed) artifacts and debugging of software: while the former aims at identifying or localizing the deviation of a faulty realization from a correct design, the latter is concerned with identifying or localizing the reason why an incorrect design fails to meet the specification.

The second obstacle is modeling itself: at the code level, a component-oriented model may become too complex and prevent a solution from scaling up to interesting programs, whereas at a very high level of software modules, the models tend to become very specific and are not reusable across different problem instances, which results in a (usually inhibitive) high cost of modeling.

Thirdly, while modeling the possible faults is often straightforward for physical systems (a shorted resistor is consistent with an increased current, but an open one is not), modeling faults in software is usually infeasible, because the space of programmers' faults is infinite.

The work we present here does **not** aim at a **general** solution to software debugging. Instead, it is guided by the idea that classes of software applications, which are configured from standardized modules do not suffer from the abovementioned obstacles, in providing an intermediate level of abstraction that allows for reusable models of standard software modules and, especially, for generic fault models – an approach we have not found in the literature on model-based software debugging.

In this paper, we address fault localization in data warehouse applications as an instance of such a class of standardized software applications.

The next section introduces the foundations of this application area and describes a specific example: a data warehouse application in customer report generation of a provider of mobile phone services. After a brief characterization of component-oriented consistency-based diagnosis, section 4 presents the core contribution of this paper, the foundations and examples of generic models for debugging of data warehouse applications. We then present the specialization to the example and briefly discuss an initial validation of the approach and future work.

2 APPLICATION DOMAIN: DATA WAREHOUSING

2.1 General Background

Data Warehousing and On-Line Analytical Processing (OLAP) are essential elements in decision support systems. Nowadays, there is not only a need to manage huge amounts of data, but also an equally, if not more, important requirement of analyzing this

[1] Technische Universität München, Germany, struss@in.tum.de
[2] IIT Madras, Chennai, India,
[3] Ericsson India Private Limited, Chennai, India

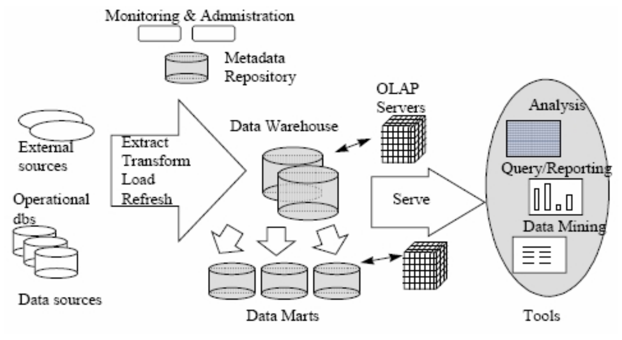

Figure 1. Architecture of a generic data warehouse system

data and extracting useful information, and data warehousing technologies support this. Many commercial products and tools in this area are now available, aiming at enabling faster and more informed decision making.

A data warehouse is a "subject-oriented, integrated, time-varying, non-volatile collection of data that is used primarily in organizational decision making."[Inmon 92]. The aim of data warehousing technologies is different from that of operational databases, which typically take care of day-to-day transactions. Unlike the latter, the focus in data warehousing is decision support, and, hence, summarized and consolidated data are more important than individual records. Data warehouses are orders of magnitude larger than typical databases and their main bottleneck is in answering complex ad-hoc queries involving scans, joins and aggregations typically over millions of records. Therefore, data warehousing technologies are becoming more sophisticated, complex and, as a result, more fault-prone, as well.

The general architecture of a data warehousing system is as shown in Figure 1 [Chaudhuri 97]. The major modules in such a system are:

- **Pre-processing** – This set of modules deals with the cleaning of data, normalization of certain fields and other pre-processing methods needed to bring the data to a common standard format.
- **Loading of the data warehouse** – This deals with the loading of the pre-processed data appropriately into the warehouse.
- **Summarization and consolidation using data marts** – This includes aggregating and consolidating the warehouse data and storing it into customized databases called data marts.

Therefore, a typical cycle in a Data Warehousing application is:

- Arrival of new data
- Pre-processing of the data
- Loading into the data warehouse
- Consolidation of new data with old data
- Storing consolidated data into data marts

2.2 Example: Report Generation Based on Call Data

The report generation tool is a system (Figure 2) used by the phone service provider to generate useful information from consumer usage records known as Call Data Records (CDR). The CDRs are generated by a number of network nodes operating in different regions and contain data such as duration of the call (in case of normal calls), data volume transferred (in case of a GPRS call), source and destination numbers, cost of the call, location identifiers of source and destination regions. The data is subject to various pre-processing steps in the Data Warehousing System (DWS) and then loaded into the data warehouse.

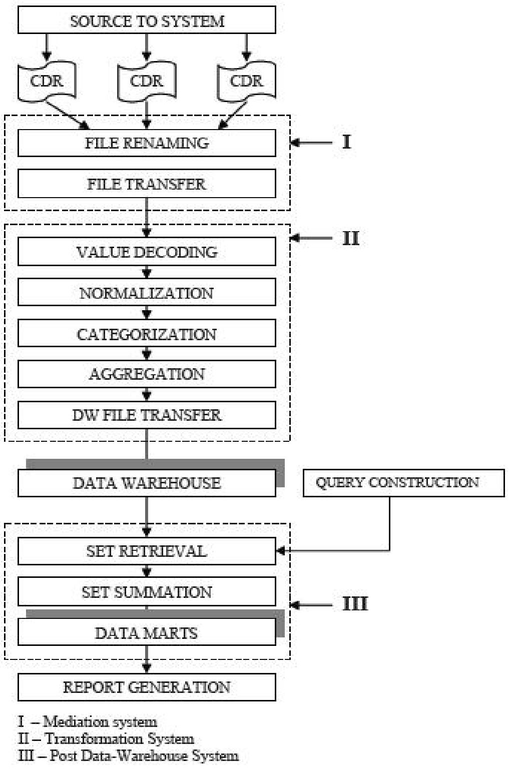

Figure 2. Report generation system based on call data

Extract-transform-and-load operations are then applied to the warehouse data to obtain customized figures, such as countrywide aggregate revenue for a given time period (e.g. a month), total revenue from a particular region, number of active subscribers in a given region, the liability of the service providers to the customers, the region-wise distribution of network usage etc., which are then stored in specialized data warehouses known as **data marts**. Updates to the data marts are typically done on a daily basis. From the data marts customized reports are generated. For instance, the **balance report** shows the total account balance of the subscriber base on a given date, thus used for reporting the operator's liability.
Another example is the **customer usage report**, which gives information about the usage statistics of the customer base for a given period of time, both for voice as well as GPRS calls.

A detailed process diagram is as shown in Figure 2. Once a CDR file is received from the source nodes, the **mediation** module processes it and renames the CDR file, assigning it a unique sequence number. After this, the CDR file is transferred via FTP to the **transformation system** for further pre-processing.

The **collection** engine of the transformation system monitors the directories for any incoming CDRs from the mediation system. Once a file of CDRs is received, the engine transforms each CDR into an internal data structure in the **value decoding** module. The **processing engine** checks the CDR for mandatory fields, the **normalization** module normalizes all numbers to a uniform format, and the **categorization** stage attaches tags to the CDR based on values of certain fields, such as tagging the records as local, national or international according to the source and destination numbers. The **aggregation** step performs the combination of multiple fields into one, deriving a new field based on certain existing fields etc. For instance, this step combines the local timestamp field and the time zone information in the CDR to generate a UTC timestamp. The CDR is now stored in another data structure and passed onto the **distribution** engine, which transfers all processed CDRs from the data structure to an output file. Once this is done for all the CDRs, they are stored in the data warehouse.

Extract-transform-and-load operations are carried out periodically on the data warehouse to populate customized consolidated values into the data marts. There are different kinds of data marts based on their functionality, such as financial, usage-level and subscriber-life-cycle data marts. The consolidated values in the data marts are then visualized using a customized **report generation** system as shown in Figure 2.

During the various processing and transfer steps, data can be corrupted in many ways and lead to missing or wrong data stored in the data warehouse and/or the data marts or appearing in the reports. For instance, a breakdown in the network connectivity during the transfer of CDRs into the warehouse might lead to incomplete data in the warehouse, thus leading to faults downstream. Usually, such defects are not detected until some results in the reports are identified as obviously incorrect, e.g. the total revenue for a time period being orders of magnitude smaller or larger than expected. Localizing the cause for this deviation in the entire process chain can be a tedious and time-consuming task for the staff. Some reasons for this are frequent changes in the structure and modules of the system, the fact that most intermediate results are not persistent and high efforts to rerun parts of the process.

The following is a typical fault scenario encountered in the application where the total **number of active subscribers** in the system according to a generated report did not match the expected value. To check whether the fault was produced during report generation, the data marts were inspected. When the same error was found in the data marts (thus implying that the fault was created upstream in the process), the warehouse data was then checked for errors. When the warehouse data was found to be **OK** (and yet the value in the data marts was wrong), it was concluded that there is an error with the **set retrieval module logic**. The code, after being checked, was indeed found to be buggy.

3 COMPONENT-ORIENTED CONSISTENCY-BASED DIAGNOSIS

The description of the system and the task suggests a perspective of "Localizing the fault in one component of the system as the possible cause of its misbehavior". Component-oriented consistency-based diagnosis (see [Struss 08]) has been developed as a solution to diagnosis of a broad class of physical artifacts. In a nutshell, it can be informally described as follows: the behavior of each component (type) of a system is modeled in a context-independent manner. Each component C_j can be in one of different behavior modes $mode_i(C_j)$. The correct or intended behavior mode (OK) is one of them, and others are either simply its negation or a list of specific (classes of) misbehaviors (such as "open" or "shorted" for a resistor). An overall system model is (automatically) configured according to the system structure (i.e. the interconnectivity of the components) for a mode assignment

$MA = \{mode_i(C_j)\}$,

which specifies a unique behavior mode for each component.

A diagnosis is obtained as a mode assignment MA whose model is consistent with the observations:

$MODEL(MA) \cup OBS \not\vdash \perp$.

Even if only the OK modes have an associated model, this yields fault localization. If models of the various fault modes exist, then fault identification can be performed and fault localization can be more confined.

Despite a number of obstacles, that were mentioned in the introduction, the principles and techniques of component-oriented consistency-based diagnosis can be exploited for fault localization in programs under certain conditions.

4 DIAGNOSTIC MODEL OF DATA WAREHOUSE APPLICATIONS

4.1 The Main Ideas

The overall process described in section 2.2 is a sequence of steps all data have to go through to ultimately yield a result in a report. If a wrong result is detected, each of these steps may be suspected to have caused it. A straightforward application of consistency-based diagnosis as described in section 3 (with each step modeled as a component in a linear structure) will produce exactly this result. Both for a human and a (semi-)automatic debugging aid, there are three basic ways to reduce the set of diagnostic candidates and finally obtain a fault localization:

- Collect **more observations**. In our application, this means checking intermediate data. Besides the data warehouse and the data marts, the only persistent data are the output of the mediation system. Inspecting more intermediate results requires re-running the steps, which is time-consuming and should be done only after having confined the location of the fault as precisely as possible by the following means.
- Use **fault models**. In contrast to physical systems, it is impossible to find a small set of models covering the abnormal behavior of pieces of software in the general case. However, at the abstract level of the functional description of a data warehouse application, it becomes feasible to describe some plausible improper behaviors of a module. This becomes even more powerful together with the third step.
- **Refine the structure**. This is achieved by stratifying the data according to their type and role in the process. Different

steps affect different fields of the record, and so do faults in these steps. For instance, a bug in normalization of a temporal representation may corrupt the time information, but leaves location information unchanged. And an incomplete transmission of data truncates a set of records, but leaves the content unmodified.

The last example illustrates the need to not only model the manipulation of the content of records, but explicitly represent and propagate properties of **record sets**. If the record, say, for a particular day is incomplete, then summing up some numerical information will yield a number which is too small.

This in turn motivates the modeling principle chosen: the models capture the **deviation** of properties of **data fields** or **sets** from those that would have been obtained if everything had worked as planned. Starting from an observed deviation of some report result, the system is going to identify models of the entire process that are consistent with this deviation. In this abstract representation, the references for the deviations remain implicit and dependent on the context: they are given by whatever are the outputs of the various steps that the respective report result depends on.

4.2 Partitioning of the Data

In this section, we present a general principle for partitioning the data for the debugging purpose. The rationale behind this is the fact that software modules only refer to certain parts of the data and also modify only certain fields on the data. Therefore, each module **induces** a **partition** of the data fields, basically into **relevant** and **irrelevant** to the function of the module. Relevant fields are those that are either referred to or modified by the module. Our strategy is, therefore, to construct a global partitioning that respects all local partitions.

This can be formalized as follows: For each module M_i and fields $f_j \subset F$ from the data records:

$\mathbf{A_i}$ is the set of fields $f_j \in F$ of the input whose content may **affect** the result, both under normal and abnormal behavior,

$\mathbf{E_i}$ is the set of fields $f_j \in F$ of the output that are **effects** of the processing of the module under normal and abnormal behavior.

In addition, each field $f_j \in F$ has a type $T(f_j)$ which influences the (description of the) potential deviations that it can exhibit such as **Numerical**, **String** etc. (see following subsection).

Based on the local partitionings, the global partitioning is defined as the one that respects all local partitions and the type, with the partitions being maximal:

$$\exists\, k, f_l, f_m \in P_k \Leftrightarrow (\forall i \quad (f_l \in A_i \Leftrightarrow f_m \in A_i) \\ \wedge (f_l \in E_i \Leftrightarrow f_m \in E_i)\ ) \\ \wedge (T(f_l) = T(f_m)))$$

For example, in case of the aggregation module, A_i represents the fields that are aggregated and E_i the aggregated field. Similarly, for the retrieval module, A_i is the set of keys to the query while E_i comprises the selected output fields.

4.3 Types of Fields and their Domains

The data fields and the data occurring in the query and report generation steps are categorized into **numerical** (such as duration of a call in our application), **categorical** (such as source and destination phone numbers), and **string** (such as a database query). We use the following domains, which capture the deviation of an actual value of a variable, X, from some reference value, X_{ref}:

Numerical = {Ok, -, --, +, ++, oppSign}, where

- **Ok** if $X = X_{ref}$
- **oppSign** if $(X * X_{ref} < 0)$
- **-** if $(X * X_{ref} >= 0) \wedge (X < X_{ref})$ $\neg (X << X_{ref})$
- **--** if $(X * X_{ref} >= 0) \wedge (X << X_{ref})$
- + if $(X * X_{ref} >= 0) \wedge (X > X_{ref})$ $\neg (X >> X_{ref})$
- ++ if $(X * X_{ref} >= 0) \wedge (X >> X_{ref})$

Categorical = {Ok, Wrong}, where

- **Ok** if $X = X_{ref}$
- **Wrong** if $X \neq X_{ref}$

String = {Ok, Null, Wrong, SynWrong}, where

- **Ok** if $X = X_{ref}$
- **Null** if $(X = null) \wedge \neg (X = X_{ref})$
- **Wrong** if $\neg (X = null) \wedge \neg (X = Xref) \wedge (X \text{ is valid})$
- **SynWrong** if $\neg (X = null) \wedge \neg (X = Xref) \wedge \neg (X \text{ is valid})$

The motivation for valid, invalid and null strings is predominantly to capture features of database queries: valid strings are those which are syntactically correct (i.e. which will execute without an exception on a database), whereas invalid strings are those which will result in an error when executed on a database. Null strings are also used to handle the case when the string construction module failed completely, resulting in an empty string.

As explained above, the model also captures explicitly how a set of data, DS, which is processed, is related to the data that should be processed in the proper process, DS_{ref}. The domain of the respective variable is

Set = {Ok, Empty, Subset, Superset, Wrong}, where

- **Ok** if $DS = DS_{ref}$
- **Empty** if $(DS = \{\}) \wedge \neg (DS = DS_{ref})$
- **Subset** if $\neg (DS = \{\}) \wedge (DS \subset DS_{ref})$
- **Superset** if $\neg (DS = \{\}) \wedge (DS \supset DS_{ref})$
- **Wrong** if $\neg (DS \subset DS_{ref}) \wedge \neg (DS_{ref} \subset DS) \wedge \neg (DS = DS_{ref})$

4.4 Models

Once the stratification of data into appropriate groups is established, models of individual components capturing both the desired and possible faulty behaviors can be designed, capturing the information about how a component treats the abovementioned **partitions** of a record. In the following, we present some examples from the model library.

File transfer component. If we consider the File Transfer component (which, in our application, handles the transfer of files containing CDRs across a network), we know that only the 'record set' property can be affected, i.e. if the transfer is not successful, either the file transfer was incomplete (nevertheless preserving the integrity of an individual record) or nothing at all was transferred, resulting in a completely unsuccessful transfer. A full description of the model of this component is shown in Table 1.

As can be observed from the table, in the OK mode of the component, the set property of the CDR file is simply propagated, i.e. output of the component is identical to its input. However, in the fault mode when the FTP connection is broken, the model captures the fact that no matter what the nature of the input, the output could be either a Subset of the original data (resulting from a partial loss in connectivity) or an Empty set (resulting from a complete loss of connectivity). In addition, if the input is a Superset, the output after truncation can be a Wrong set (which means, we ignore the highly unlikely case that transaction incidentally produces the proper set).

Table 1. Model of the File transfer Component

STATUS	Input.set	Output.set
OK	Ok	Ok
	Wrong	Wrong
	Empty	Empty
	Subset	Subset
	Superset	Superset
CONNECTION DISRUPTED	*	Subset
	*	Empty
	Superset	Wrong

Table 2. Model of the Query construction Component

STATUS	qStrTemplate	qCriteria	qString
OK	Ok	Ok	Ok
	Ok	Wrong	Wrong
	Wrong	*	Wrong
	Wrong	*	SynWrong
FAULTY	*	*	Wrong
	*	*	SynWrong

Table 3. Model of the Set retrieval Component

STATUS	qString	inputSet	selectKey	outputSet
OK	Wrong	*	*	Wrong
	*	Wrong	*	Wrong
	*	*	Wrong	Wrong
	Wrong	*	*	Subset
	*	Subset	*	Subset
	*	Wrong	*	Subset
	*	*	Wrong	Subset
	Wrong	*	*	Superset
	*	Superset	*	Superset
	*	*	Wrong	Superset
	Wrong	*	*	Empty
	*	Empty	*	Empty
	*	Wrong	*	Empty
	*	*	Wrong	Empty
	Ok	Ok	Ok	Ok
FAULTY	*	*	*	Empty
	*	*	*	Subset
	*	*	*	Wrong
	*	*	*	Superset

However, an assumption made while building this model is that the file transfer component never spoils the integrity of the data and only can disrupt the set property, which is indeed true in our case study.

In our application, this model is used in different places in the process: the data transfer to the transformation system and the transfer into the data warehouse.

Query construction component. This takes as input a query template, qStringTemplate, with placeholders for variables and categorical variables, qCriteria containing values for these placeholders, and produces a query string, qString. It is used to construct queries automatically in order to retrieve desired information from the data warehouse. The model of this component is described in Table 2. In the OK mode of operation, if both inputs are Ok, the output is Ok. If not, the output takes appropriate values for different input cases as shown in the table.

In the **FAULTY** mode of operation, no matter what the values of the input are, the output string can take the values Wrong or SynWrong.

Set retrieval component. As a final example, we consider the component that retrieves relevant data from the data warehouse for a particular operation (e.g. to calculate total revenue for a particular period, this module extracts the per-CDR revenue data) which then may be given as input to a module that performs an operation on this data (such as the summation component). The inputs to this component are the query string for the actual retrieval, **qString**, the data set on which the query operates, **inputSet**, and **selectKey**, which determines the required field (e.g. the revenue per CDR), and generates the relevant subset of data, **outputSet**. A complete description of the model is given in **Table 3**.

In a similar manner, the other components are modeled, capturing both the normal and deviant behavior with appropriate fault modes.

It should be noted as an important disadvantage that the global partitioning, being dependent on the local ones, may have to be changed if new modules are introduced or the records are modified. In order to obtain truly generic models, in a future solution, they should be stated in abstract terms of their sets A_i, E_i, $F \setminus (A_i \cup E_i)$ and the mapping to the record fields should be represented separately.

5 STRUCTURING THE CALL DATA

Based the principles of section 4.2, the fields of the CDR were grouped into the following 9 groups:

- **CDR Information** – this group deals with CDR-specific information such as CDR identifier.

- **Account Information** – this deals with the account information of the subscriber, such as the plan being used, the base location of the subscriber etc.

- **Call-Information** – this gives information about the source and destination phone numbers, whether they are roaming or not etc.
- **Cost-Information** – this gives information about the rates that the subscriber will be charged for this call
- **Duration of Call** – gives the duration of the call
- **Location-Information** – gives the location identifiers of the subscribers
- **Data Volume** – gives the data volume transferred in case of a GPRS call
- **Timestamp of call** – gives the time at which the call began
- **Final-charge of call** – gives the final amount that the subscribers are charged.

In addition, the models propagate

- **Set Information** - dealing with the set property of a file of CDRs.

6 VALIDATION OF THE DIAGNOSTIC MODEL

So far, the models were validated against a small set of real problem scenarios encountered by the end user, which were considered typical and representative, and the fault localization of the diagnosis tool under the available observation was compared to the manual debugging steps. A comparison to another debugging tool could not be performed, simply because there is none. We present two of these cases in the following.

6.1 Scenario One: Consumer Usage Amount Less than Expected Value.

In this scenario, it was observed that the customer usage amount displayed in the report generated by the system is less than the expected value.

The steps taken to manually localize the fault were as follows:
1 **Generate report** – erroneous value present in report
2 **Probe data marts** – erroneous value present in data mart (implying that the cause for the fault is upstream)
3 **Query data warehouse** – correct **duration** values are present in the data warehouse (implying that something is wrong with the selection criteria in the query or selectKeys, in this case, the timestamps)
4 **Analyze the number of CDRs in result set** – does not match with expected value
5 Analyze timestamp of a CDR and compare with output of mediation system – does not match

Therefore, the diagnosis was 'Erroneous timestamp calculation' and indeed, the aggregation component containing the timestamp calculation code was found to be buggy.

The steps taken to localize the fault using the model-based diagnosis system were:
1 **Initialize given evidence**, i.e. **Total duration** as observed in data marts is '–'(step number 2 in the manual debugging).
2 **Output of Set retrieval module is Wrong** (step number 4 in the manual debugging) - exonerates the **Set Summation** module (since the fault has occurred before this component was used).
3 **Time Info in the data warehouse is Wrong** (step number 3 in the manual debugging) - eliminates a number of candidate diagnoses leaving 4 diagnoses.
4 **Time Info at output of Mediation module is Ok** (step number 5 in the manual debugging) - exonerates the 'Source to System' component.

This leaves us with three suspect modules for more detailed probing and debugging, including the component that was actually found to be faulty, namely the aggregation component.

6.2 Scenario Two: Number of Active Subscribers not Matching Expected Value.

In this scenario, the starting point is an error in the report summarizing the active subscriber statistics. The manual debugging procedure required **4** probes to narrow down onto the module causing the fault, the **Set retrieval** component, which are:
1 **Generate report** – erroneous value in report
2 **Probe data marts** – erroneous value present in data mart (implying that the cause for the fault is upstream)
3 **Run query on data warehouse** – correct value is obtained, indicating the problem is downstream from the data warehouse.
4 **Analyze the Set Retrieval component** – found to be buggy.

It took the model-based system 5 steps (including the initial symptom) to generate the same result.

The cases provide some evidence that component-oriented consistency-based diagnosis provides the basis for a useful debugging aid. More specifically, the level of abstraction of the component models appears to be expressive enough for the task. The tool is **not** meant to provide an **automatic** localization of the fault, but to guide a human debugger without requiring him to have deep detailed knowledge about the system structure, the modules, recent modifications etc. any more. This is possible since this knowledge about the system is now incorporated in the model. Therefore, at least for a set of common sources of errors, a person not too experienced with the data warehouse system can perform debugging, which was previously impossible.

7 FUTURE WORK

In this paper, we described the models for consistency-based debugging of a data warehouse application and its validation. So far, only the diagnostic part has been realized. For a real debugging aid, a module has to be integrated that proposes "probes", i.e. inspection of persistent data and rerunning process steps. More scenarios will be treated to establish the basis for making a business case that justifies the development of a tool for everyday use in this area.

REFERENCES

[Chaudhuri 97] S. Chaudhuri, U. Dayal: An overview of data warehousing and OLAP Technology. In: ACM SIGMOD Record, 1997.

[Inmon 92] W.H. Inmon: Building the Data Warehouse. John Wiley, 1992.

[Mayer-Stumptner 07] W. Mayer, M. Stumptner. Model-Based Debugging – State of the Art and Future Challenges. Electronic Lecture Notes in Theoretical Computer Science, 171(4), 2007

[Struss 08] P. Struss: Model-based Problem Solving. In: van Harmelen, F., Lifschitz, V., and Porter, B. (eds.). Handbook of Knowledge Representation, Elsevier, 2007