HAL
open science

SoC-Trace Infrastructure

Generoso Pagano, Vania Marangozova-Martin

To cite this version:

Generoso Pagano, Vania Marangozova-Martin. SoC-Trace Infrastructure. [Technical Report] RT-0427, Inria. 2012, pp.33. hal-00719745v2

HAL Id: hal-00719745

https://inria.hal.science/hal-00719745v2

Submitted on 28 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SoC-Trace Infrastructure

Generoso Pagano, Vania Marangozova-Martin

TECHNICAL REPORT
N° 427
July 2012
Project-Team MESCAL

ISSN 0249-0803 ISRN INRIA/RT--427--FR+ENG

SoC-Trace Infrastructure

Generoso Pagano *, Vania Marangozova-Martin †

Équipe-Projet MESCAL

Rapport technique n° 427 — version 2 — version initiale July 2012 —
version révisée November 2012 — 33 pages

Résumé : Les traces d'exécution sont largement utilisées pour la mise au point et l'optimisation des applications embarquées. Dans ce contexte, le projet SoC-Trace a pour objectif de fournir une infrastructure logicielle ouverte, capable d'exploiter de manière efficace les traces provenant d'exécutions de systèmes embarqués multi-cœur. Ce rapport décrit l'architecture système et logicielle du premier prototype de l'infrastructure. Ce prototype comprend, d'une part, un modèle de données qui représente les concepts intervenant lors de l'analyse de traces. D'autre part, il fournit une implémentation de ce modèle pour les bases de données relationnelles, ainsi qu'une interface logicielle pour son exploitation. Les utilisateurs peuvent travailler avec des traces de formats différents, les accéder à travers une interface unifiée et, après analyse, stocker les résultats dans la base de données. En fournissant des mécanismes partagés et réutilisables, l'architecture permet la collaboration de plusieurs outils d'analyse de traces, facilitant ainsi la création de chaînes d'analyse de traces innovantes et complexes.

Mots-clés : Traces d'exécution, débogage, profiling, systèmes embarqués, multi-cœur, source libre, gestion de traces, infrastructure, modèle de données, base de données, format de trace, SoC.

This research is supported by the FUI Project [1]

* generoso.pagano@inria.fr
† Vania.Marangozova-Martin@imag.fr

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

SoC-Trace Infrastructure

Abstract: Execution traces are a powerful instrument for debugging and profiling embedded applications. The SoC-Trace project aims at developing an open-source trace management infrastructure able to exploit multi-core embedded-systems execution traces. This document describes the first prototype of the trace management infrastructure, clarifying its objectives and describing its system and software architectures. The infrastructure provides, in the first place, a generic data-model representing the main concepts needed for trace management. Furthermore, the infrastructure provides a relational-database implementation of this model and a software library to interface with the database in a convenient way. The proposed prototype allows the user to deal with traces of different formats, to access them through a common interface and finally to save analysis results. Providing shared and reusable mechanisms, the trace management infrastructure facilitates the cooperation among different tools, thus creating an innovative and complex trace analysis environment.

Key-words: Execution traces, debugging, profiling, embedded systems, multi-core, open source, trace management, infrastructure, data-model, database, trace formats, SoC.

Table of contents

1 SoC-Trace Project **4**

2 Infrastructure Objectives **4**

3 System Architecture **4**
3.1 Multi-Trace DB . . . 5
3.1.1 Conceptual model . . . 5
3.1.2 Database architecture . . . 7
3.1.3 Database schema . . . 7
3.2 SoC-Trace Library . . . 13
3.3 SoC-Trace Management Tool . . . 14

4 Software Architecture **14**
4.1 Java interface to the DB . . . 15
4.1.1 Model . . . 15
4.1.2 Storage . . . 17
4.1.3 Query . . . 18
4.1.4 Search . . . 21
4.1.5 Utilities . . . 23
4.2 Management Tool . . . 23
4.2.1 Core . . . 23
4.2.2 Ui . . . 23

5 External Tools **24**
5.1 KPTrace Importer . . . 24
5.1.1 Parser architecture . . . 24
5.2 Test Tool . . . 25

6 Performance Evaluation **26**
6.1 Trace DB size . . . 26
6.2 Trace import time . . . 27
6.3 Buffered reading . . . 28
6.4 Complex parameters queries . . . 29

7 Future Works **30**
7.1 Database architecture . . . 30
7.2 Analysis result . . . 31
7.3 Tool management . . . 31
7.4 Other improvements . . . 31

8 Conclusions **31**

1 SoC-Trace Project

Execution traces are a powerful instrument for analyzing embedded systems behavior. They are used for understanding particular behaviors of a system, for debugging, for profiling and for optimization. The research on trace management is currently raising more and more interest in both industrial and academic world [2] [3].

The main objective of the SoC-Trace project [4] is the development of a generic trace management infrastructure and a set of tools able to exploit multi-core embedded-systems execution traces. The infrastructure should be able to store and analyze multiple traces of different formats whose size can be huge (i.e. of the order of TB). The infrastructure is to be open source, generic and efficient. The final goal is to have different tools, built on top of the infrastructure and providing new analysis methods, like data mining analysis, probabilistic analysis, richer visualization techniques, etc.

The trace management infrastructure, described in this document, has a critical importance in the whole project, since it defines the interfaces to access trace data and analysis results, thus being an integration point for the different analysis tools.

2 Infrastructure Objectives

The main objectives of the infrastructure are :

- The management of several traces in a database, using a shared generic data-model.
- The management of different tools able to work on traces.
- The management of the analysis results produced by the various tools.

The infrastructure shall be generic, since its usage must not be limited to a single trace format or a specific tool. Its data-model shall be able to reflect different trace formats and different analysis results. The infrastructure, being open source, shall provide extensible and reusable mechanisms. At the same time it shall be able to manage external black-box tools. The infrastructure shall be efficient, being in charge of dealing with huge amount of data : an execution trace, in fact, can easily exceed the size of several GBs or almost TBs. Finally, the infrastructure shall be complete, meaning that all the required functional and non functional requirements described so far shall be accomplished.

The following sections of this document are organized as follows. Section 3 presents the system architecture, with emphasis on the data-model. Section 4 provides some details on the infrastructure software architecture, with the description of the main modules. Section 5 describes two external tools developed to work with the infrastructure. Section 6 shows some first performance evaluation results obtained with the infrastructure prototype. Section 7 describes the future works we are planning. Finally, section 8 concludes.

3 System Architecture

The SoC-Trace infrastructure is built on top of the Eclipse Platform [5], which has been chosen because it is easily extensible and well designed in order to be used as an integration point for different applications. In order to take the greatest advantage of the Eclipse Platform capabilities, the Java programming language has been chosen. The general architecture of the SoC-Trace infrastructure is shown in Figure 1.

The elements strictly belonging to the SoC-Trace infrastructure are :

- The Multi-Trace DB : a database able to store (meta)information about traces, trace analysis tools and analysis results. The traces can be of different formats, the tools can

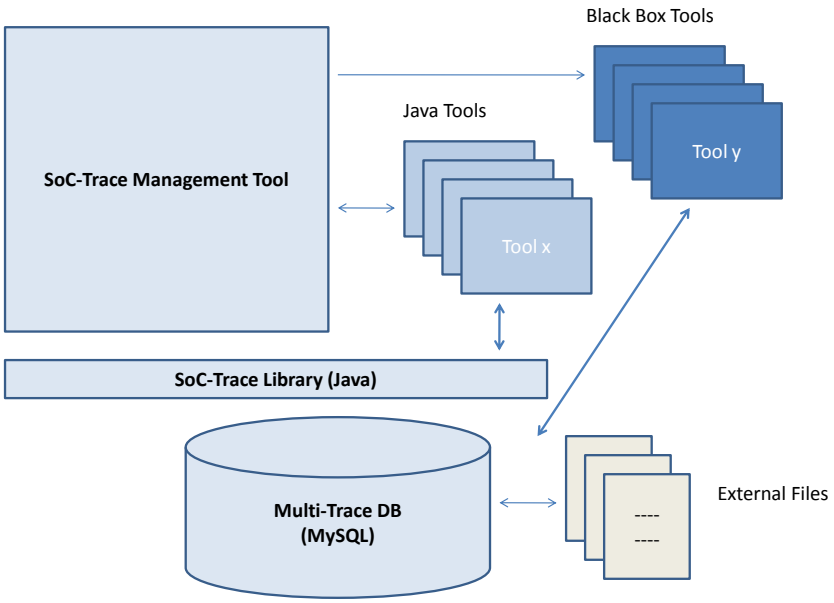


FIG. 1 – SoC-Trace Infrastructure Architecture

provide different types of analyses and the results may be of various types. The database stores also references to external files, in order to make it possible to organize the different files related to a trace.

- The SoC-Trace Library : a software library composed by a set of Eclipse plugins, providing a convenient interface to deal with the database and the entities of the data-model.
- The SoC-Trace Management Tool : a simple front-end tool used to graphically execute basic management operations on the whole system.

The infrastructure source code is written in Java, since the use of this language is, at the moment being, mandatory to write Eclipse plugins. However, the infrastructure is able to manage both Java and non-Java tools. The latter have the possibility to access directly the database and the external files, without necessarily passing through the SoC-Trace Library. The database management system (DBMS) we have chosen is MySQL [6]. The main motivations are its scalability and flexibility, its good performance and its open source license. In the following sections we describe in detail the three main blocks of the SoC-Trace infrastructure.

3.1 Multi-Trace DB

3.1.1 Conceptual model

The conceptual model of the SoC-Trace Multi-Trace DB has been designed with the aim of capturing the most significant concepts concerning the area of execution traces management and analysis. We have organized these concepts in three groups :

- Trace General Information : here we will typically gather information about the tracing configuration and the files related to a trace.
- Raw Trace Data : this includes the registered-execution data, i.e. the events and their format, as well as the information about their producer.

- Analysis Data : this concerns the meta-data about the used analysis tools and the different analysis results.

We can find these three groups, with their corresponding entities, in the Crow's Foot diagram shown in Figure 2.

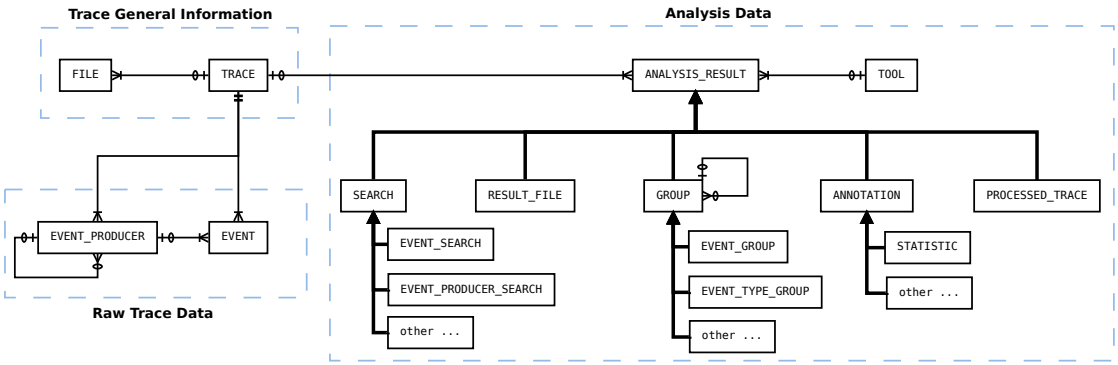


FIG. 2 – Conceptual Model

The system manages several execution traces (TRACE), which can be related to one or more files (FILE) (e.g. configuration files, raw trace files, IP-XACT [7] hardware description files).

Each trace is composed by several events (EVENT), an event being some change in the state of the traced system. An event is caused to happen by a single active entity we call an event producer (EVENT_PRODUCER). Examples of event producers are typically processes or threads. An event producer can generate several events and there can be a hierarchy among event producers (e.g. a process creating son processes).

The model also integrates information about the tools performing operations on traces (TOOL). Tools may save the results of their analyses (ANALYSIS_RESULT) in order to avoid their (possibly expensive) re-computation. In the current model, an analysis result involves a single trace. In the diagram, the concept of analysis result is modeled as an abstract entity, which has its concrete realizations in specific kinds of analysis results. We have defined namely :

- Searching/Filtering (SEARCH) Result : a result of a searching/filtering operation, which can be performed over the events or over the event producers of a trace.
- Custom File (RESULT_FILE) Result : this models a result which is stored in a custom file outside of the infrastructure. The database contains only the reference to the file, the semantics of the file content being understandable by the tool that produced the file.
- Grouping (GROUP) Result : a result structured in terms of groups of model entities (events, types of events or types of event parameters), possibly defining a hierarchy among such groups.
- Generic Annotation (ANNOTATION) Result : an annotation is an analysis result which describes itself (it uses the Self-Defining Pattern, discussed in section 3.1.3). It allows the tool to define both the structure and the value of the result.
- Processed Trace (PROCESSED_TRACE) Result : this result is a trace produced after treating another trace through, for example, adding additional information.

A detailed description of each kind of analysis result is provided in section 3.1.3.

By defining a standard interface for analysis results, the infrastructure enables the cooperation among different tools. For example, using the analysis result provided by the data-model, it is possible for a tool A to retrieve the results of a tool B, perform other elaborations and finally

produce another result. Furthermore, a shared set of analysis result types is also a base brick for the definition of complex workflows among different tools.

3.1.2 Database architecture

The Multi-Trace DB is designed as a two-level hierarchy of databases, as shown in Figure 3.

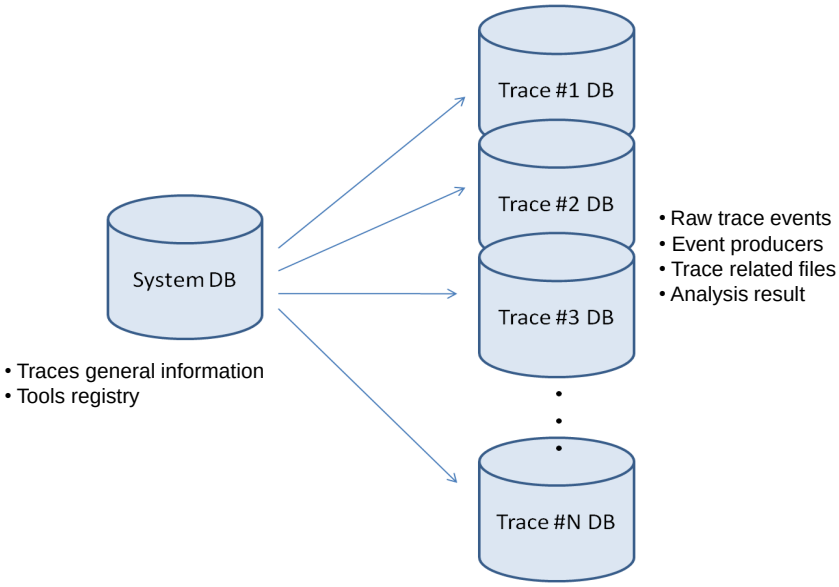


FIG. 3 – Multi-Trace DB architecture

The entry point of this database architecture is the System DB, which contains general information about traces and the registry of the tools registered to the infrastructure. Then, there are several Trace DBs, each containing the information related to a single trace. More precisely, a Trace DB stores raw events, event producers, file references and analysis results related to the concerned trace.

We have chosen such distributed architecture mainly for simplicity of implementation, scalability and efficiency. Indeed, the database schema of each Trace DB is simpler compared to a centralized solution. Yet, it is still possible to perform multi-trace (multi-DB) queries with MySQL. Other points that would impact the performance in a centralized solution are the space overhead to store a "trace id" in all rows of almost all tables, the time overhead to perform queries in presence of the above mentioned "trace id" and the presence of huge tables, which can lead to the necessity of table partitioning [8].

This distributed architecture leaves the place for extensions. In particular, if in the future multi-trace analyses will be performed, the System DB schema will be accordingly extended to support the storage of such multi-trace analysis results.

3.1.3 Database schema

Before explaining in detail the System DB and Trace DB table structures, we present the concept of Self-Defining Pattern which is used in several places.

Self-Defining Pattern

In both System DB and Trace DB, we have used a Self-Defining Pattern (SDP) in order to achieve the greatest genericity of representation. A SDP, as the name suggests, allows the representation of both *values* and their *structure (type) information*.

In order to illustrate this pattern, let us consider an *Element* (Figure 4).

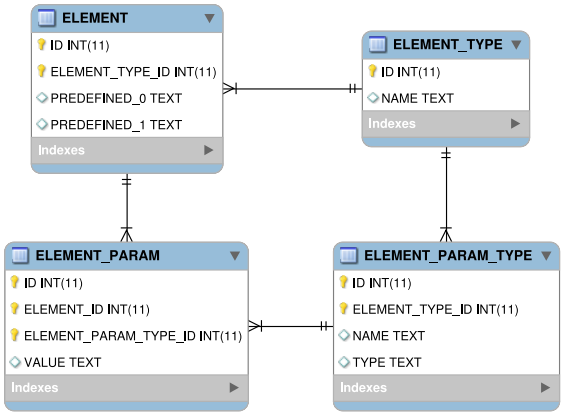


FIG. 4 – Self-Defining Pattern (SDP)

The SDP represents the Element concept using 4 tables.
An Element (ELEMENT) has a type (ELEMENT_TYPE) which is described by a given set of parameter types (ELEMENT_PARAM_TYPE). The values of these parameters for a given Element are stored in the ELEMENT_PARAM table. So the right part of the SDP describes the *type* of an Element (type and related parameter types), while the left part contains the *values* (the Element predefined attributes and parameters values). Each table contains the IDs (primary keys or foreign keys) necessary to perform the join with the other tables.

The Self-Defining Pattern is used in the DB schema to model the concepts of Trace, Event and Annotation. In the following, when talking about a generic concept modeled using the SDP, we will often use the name *Element* to identify such generic concept.

System DB and Trace DB schema

Figure 5 shows both System DB and Trace DB database schemas, using the Crow's Foot notation.

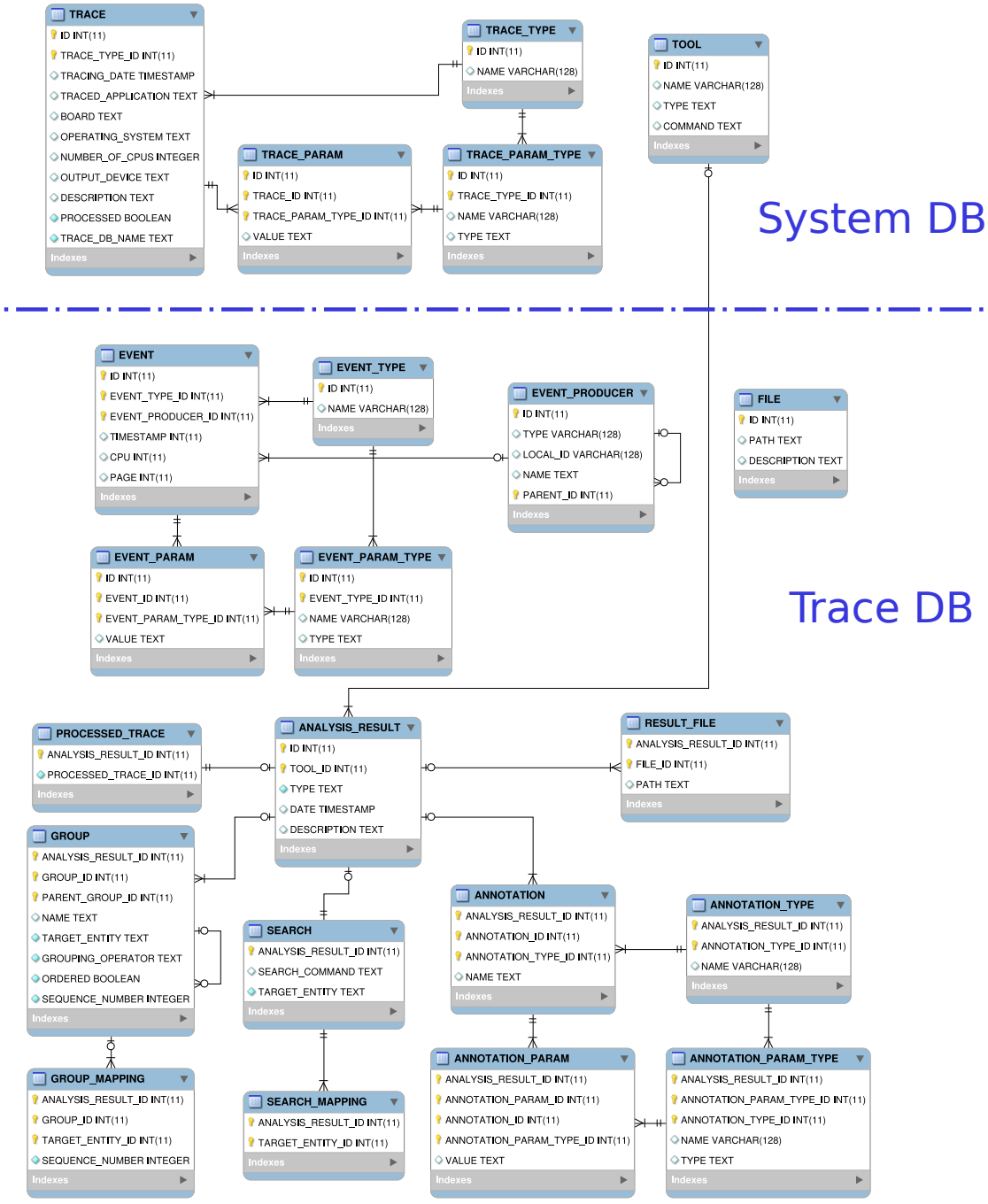


FIG. 5 – Database schema with both System DB and Trace DB

The System DB stores general information about traces and the tools registry. There are numerous trace formats, including KPTrace [9], OS21 [10], Pajé [11] or Open Trace Format (OTF) [12]. As the infrastructure should be able to store traces having different formats, we

have used the SDP for the representation of the Trace concept. In this case a trace type identifies a particular kind of trace (e.g. KPTrace trace) and the parameter types can be conveniently used to model specific configuration parameters, necessary to describe the environment where that trace has been produced (e.g. version of a driver, version of the tracing API, etc.). On the other hand, in the TRACE table, we store some attributes common to all kinds of trace (whatever trace type), like the date the trace has been produced and the number of CPUs.

The Tool concept is described by its name, its type and its launching command. The supported types of tools are currently : IMPORT and ANALYSIS. The first group concerns the tools in charge of importing raw traces into the infrastructure : the idea is to have a specific importer for each kind of trace format, which acts as an adapter from the specific format details to the SoC-Trace data-model. The second group of tools is quite large and includes all the tools able to perform whatever kind of analysis on a trace : pattern mining analysis, probabilistic analysis, etc.

Note that there is no explicit link between the TOOL table and the trace SDP tables, since this link passes through the Trace DB (whose name is one of the predefined attributes stored in the TRACE table). The Trace DB stores the link between the result of one analysis performed on the trace and the ID of the tool performing such analysis.

The Trace DB stores the raw trace data and the trace analysis results. For the ease of explanation, we can separate the Trace DB schema in two parts, the one related to the raw trace and the one related to the analysis results.

Trace DB : raw trace data

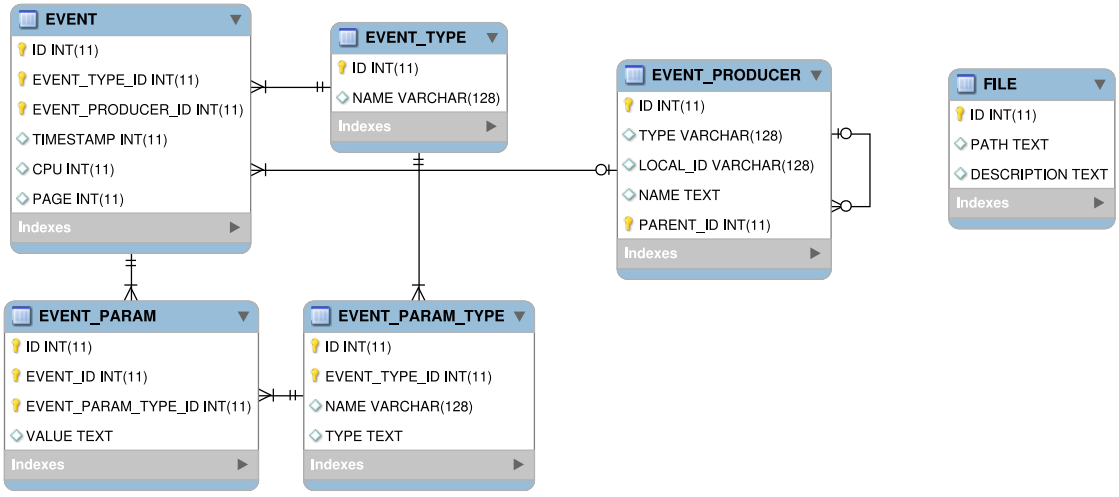


FIG. 6 – Trace DB : raw trace data

In Figure 6 we can see the raw trace data section of the Trace DB schema. This part of the Trace DB stores information about raw events of different types, event producers and files related to the trace. As a trace may contain different types of events (e.g. context switch events, memory allocations, etc.), we have represented the event concept using the SDP. Thus we can store in the EVENT table the attributes common to all event types (e.g. the timestamp or the CPU where the event has been produced), and use the other tables to describe the parameters typical of a

specific event type (e.g. a context switch event could be described by the new PID and the old PID). An event producer is described by its type (e.g. process, thread, etc.), its local identifier (e.g. the PID, if the producer considered is a process) and its name. The EVENT_PRODUCER table contains also a parent ID, intended to allow an importer tool to build a hierarchy among sources, while parsing the trace (e.g. threads created by a process). A file (FILE table) is simply described by its path and a custom textual description.

Example Let us consider how an event related to a context switch (name CS) could be mapped to the model. The event is traced with the following format :

```
<timestamp> CS <oldpid> <newpid>
```

A concrete event could be the following :

```
227.537525 CS 0 21
```

This line can be imported into the database filling the four Event-SDP tables as shown in Figure 7.

EVENT_TYPE	
ID	NAME
..	..
3	CS
..	..

EVENT_PARAM_TYPE			
ID	EVENT_TYPE_ID	NAME	TYPE
..	..	..	..
14	3	OLDPID	INTEGER
15	3	NEWPID	INTEGER
..	..	..	..

EVENT					
ID	EVENT_TYPE_ID	EVENT_PRODUCER_ID	TIMESTAMP	CPU	PAGE
..	..		..	..	..
9	3	54	227.537525	0	0
..	..		..	..	..

EVENT_PARAM			
ID	EVENT_ID	EVENT_PARAM_TYPE_ID	VALUE
..	..	..	..
37	9	14	0
38	9	15	21
..	..	..	..

FIG. 7 – Example of event importing

The event type mnemonic (`CS`) is used to fill the EVENT_TYPE table : the NAME attribute is `CS` and the corresponding ID is `3`. This event type is described by two event parameter types, the `OLDPID` and the `NEWPID`, both typed `INTEGER`. So in the EVENT_PARAM_TYPE table we have the corresponding two lines, where the EVENT_TYPE_ID attribute is `3` and the specific IDs of the parameter types are respectively `14` and `15`. Coming to the EVENT table, the `<timestamp>` value (`227.537525`) of the trace line is used to fill the predefined attribute with the same name, the EVENT_TYPE_ID is again `3` and the ID is `9`. The values given to the other attributes of the EVENT table depend on the context when the event was produced (what process, what CPU, etc.). Finally in the EVENT_PARAM table we find two lines, corresponding to the two parameter types describing this event. Both lines have the EVENT_ID set to `9`. The line corresponding to the `OLDPID` has the EVENT_PARAM_TYPE_ID set to `14` and the

VALUE set to `0`, the other line (`NEWPID`) has the EVENT_PARAM_TYPE_ID set to `15` and the VALUE set to `21`.

Trace DB : analysis result data

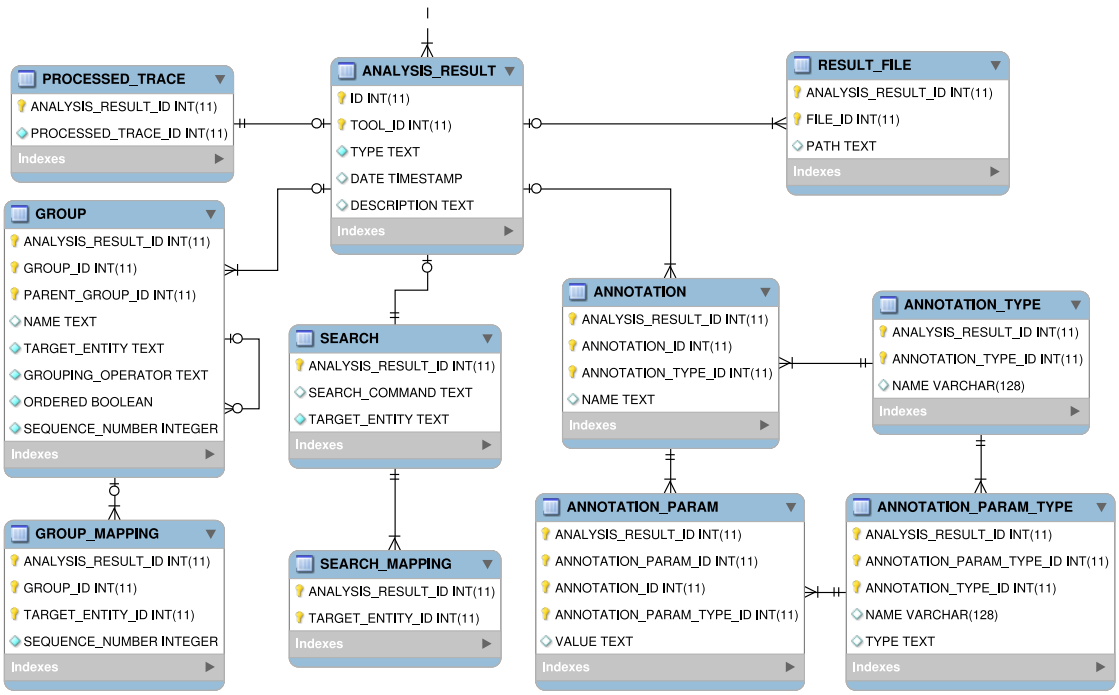


FIG. 8 – Trace DB : analysis result data

In Figure 8 we can see the analysis result part of the Trace DB schema. This part allows the representation of different analysis results produced by analysis tools. The entry point to understand this part is the ANALYSIS_RESULT table. Each analysis result is described by a unique identifier (ID), the identifier of the tool that performed the analysis (TOOL_ID), the type of analysis result produced (TYPE), the date of the analysis (DATE) and a custom description (DESCRIPTION). Considering the TYPE field, the data-model currently allows five different kinds of results to be stored in the DB :

Custom result file This result type is represented using the RESULT_FILE table, which stores the analysis result ID, the file ID (a result can contain more than one file) and the file path.

Grouping A grouping is composed by several groups of entities of the data-model (for the moment being it is possible to create groups of events, of event types or of event parameter types). A grouping can be possibly organized as a hierarchy of groups. For each group of a grouping it is possible to define several structural properties. First of all you specify the target entity being grouped (EVENT, EVENT_TYPE or EVENT_PARAM_TYPE) and the grouping operator (AND, OR). Then you specify if there is a total ordering among

the entities grouped (TRUE, FALSE). Finally it is possible to specify a sequence number, meaningful if the parent group is ordered. This result type is represented using two tables :

- GROUP : it stores the analysis result ID, the group ID (a grouping result can of course involve several groups), an optional parent ID (used to build hierarchies), the group name, the target entity name, the grouping operator, the ordered flag and the sequence number.
- GROUP_MAPPING : this table links each group (GROUP_ID) with the target entity instances belonging to that group (TARGET_ENTITY_ID). In a hierarchy, only leaf groups (the ones actually containing entity instances and not containing only other groups) are present here. The key of the table contains both the group identifier and the entity identifier, thus making it possible to have more entities in a given group and the same entity in more than one group.

Searching/Filtering A searching or a filtering of entities of the data-model produces as output a list of such entities. Currently it is possible to save the searching/filtering output related to events or event producers. This result type is represented using two tables :

- SEARCH : it stores the analysis result ID, the custom command used to perform the search/filtering operation (tool dependent) and the target entity name.
- SEARCH_MAPPING : this table lists the IDs of the entity instances being the result of the search/filtering operation.

Annotations This result type is intended to be as generic as possible, leaving to the analysis tool the freedom of defining a result format and saving the corresponding values. This kind of result is therefore implemented using the SDP. A tool can define an annotation type, specifying the corresponding parameter types. Then the tool can create concrete annotation instances of such type, computing the parameters values. An analysis tool, in a single analysis, can define several annotation types and for each type can save several concrete annotation instances. For example, a tool can define an annotation type representing a "memory statistic", being described by two parameter types : "memory used by function X" and "memory used by function Y". Then the tool can compute for each CPU the values for such parameters and save them in different annotation instances (one for each CPU).

Processed trace This result type is actually a link to another trace, being created starting from the current trace (the trace corresponding to the current Trace DB) and performing some kind of processing. The result type is saved in the PROCESSED_TRACE table that, in addition to the analysis result ID, simply contains the processed trace ID. In practice, the processed trace is stored in another Trace DB, thus being a completely normal trace : the single particular to consider is that in the TRACE table of that Trace DB, the PROCESSED flag shall be set to TRUE.

As anticipated above, the precise definition of common analysis result types is a powerful enabler for the cooperation among different analysis tools. For example, being the grouping result type format defined, a tool A can reuse a grouping defined by a tool B, in order to perform its elaboration. Let's imagine that the tool B identifies a pattern (saved as a group of event types) : the tool A can take into account this pattern definition to perform a filtering of the event instances for that pattern in order to do further analysis.

3.2 SoC-Trace Library

The SoC-Trace Library is a software library written in Java, providing a convenient interface to the Multi-Trace DB. The library is composed by a group of Eclipse plugins and allows Java external tools and the SoC-Trace Management Tool to interact with the database in a convenient

way. In fact it provides an object oriented vision of the data-model and several mechanisms to deal with the objects of the model without caring about SQL and databases low level issues. More details about this software library are given in section 4.1.

3.3 SoC-Trace Management Tool

This management tool is implemented in Java as a group of Eclipse plugins. Currently, the following functionalities are implemented.

- The creation of the SoC-Trace System DB (Figure 9). The system DB has a predefined name (SOCTRACE_SYSTEM_DB) and is created on localhost.

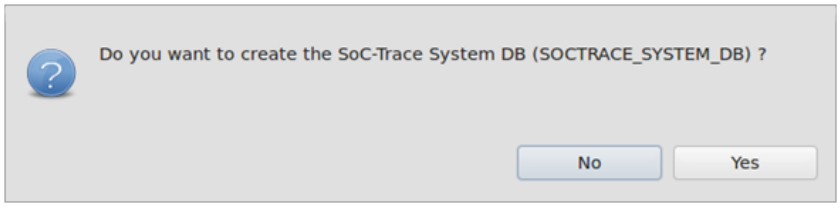


FIG. 9 – Dialog for creating the System DB

- The registration of a new tool to the infrastructure (Figure 10). The user specifies the tool name and the command to launch it, and selects the type. Currently the infrastructure defines two types of tools (import tools and analysis tool), as explained in section 3.1.3.

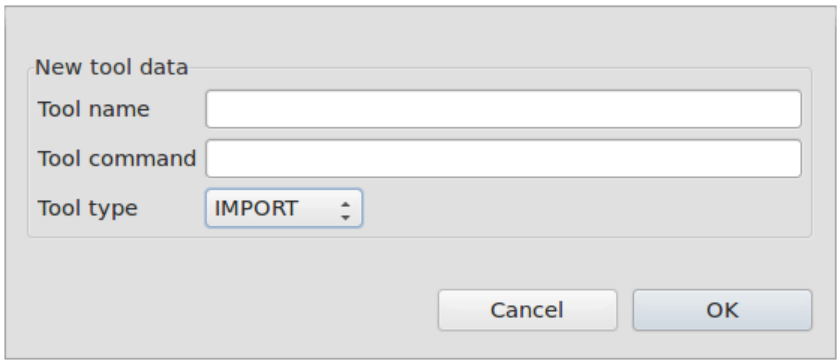


FIG. 10 – Dialog for registering a new tool to the infrastructure

- The importing of a new trace into the infrastructure (Figure 11). The user chooses the import tool to be used, selects the trace files and finally specifies other (optional) arguments for the trace import tool.

More details about the implementation of the SoC-Trace Management Tool are given in section 4.2.

4 Software Architecture

The entire SoC-Trace infrastructure is written in Java, using JDBC [13] to interface with the MySQL database management system. The code is organized as a group of Eclipse plugins. In

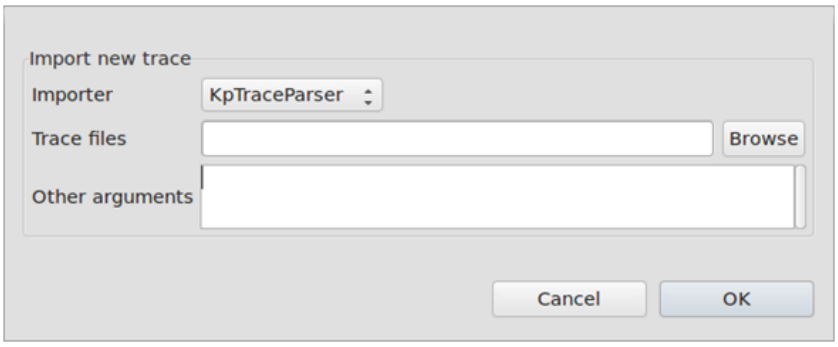


FIG. 11 – Dialog for importing a new trace into the infrastructure

the following sections the various plugins are described. We can separate the plugins forming the actual Java interface to the DB (SoC-Trace Library) from the ones forming the SoC-Trace Management Tool.

4.1 Java interface to the DB

This software library is composed by the following different plugins :

- `com.inria.soctrace.lib.model`
- `com.inria.soctrace.lib.storage`
- `com.inria.soctrace.lib.query`
- `com.inria.soctrace.lib.search`
- `com.inria.soctrace.lib.utils`

4.1.1 Model

The *data-model* is implemented in the plugin `com.inria.soctrace.lib.model`, which provides an object oriented representation of the DB schema. For all the tables of the DB (excluding some of the tables related to the concrete realization of specific analysis result, as explained below) there is a Java class that directly maps to it. These classes are implemented according to the following principles :

- The class stores all the attributes of the corresponding table, with the exception of the foreign key IDs (the IDs allowing the join with other tables) : in this case a reference to the corresponding foreign object is stored.
- Considering the SDP tables, an *Element* class always stores its *Element*Param references, and an *Element*Type class always stores its *Element*ParamType references. See for example Figure 12, which shows the high level UML diagram related to the Event SDP object oriented implementation.

The plugin defines four Java interfaces :

- *IModelElement*
  This interface is implemented by all the classes strictly modeling entities of the model.
  - The four SDP classes for the trace entity (Trace, TraceType, TraceParam, TraceParamType)
  - The four SDP classes for the event entity (Event, EventType, EventParam, EventParamType)
  - The classes for tool, file and event producer entities (Tool, File, EventProducer)

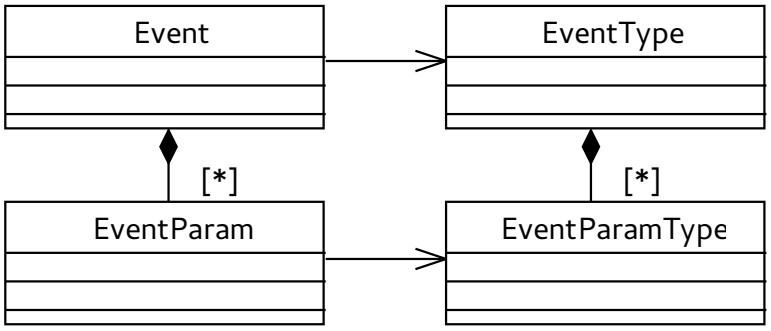


FIG. 12 – Event SDP UML diagram

  - The class for the analysis result entity (AnalysisResult)

  All these classes have an ID getter and can accept an *IModelVisitor* (see below).
- *ISearchable*
  This interface is implemented by all the classes that model entities that can be the target of a searching/filtering analysis. Currently Event and EventProducer classes implement this interface.
- *IGroupable*
  This interface is implemented by all the classes that model entities that can be the target of a grouping analysis. Among the classes already mentioned, currently Event, EventType and EventParamType classes implement this interface. This interface is implemented also by the class Group, which models a group of *IGroupable* objects (*Composite* pattern [14]). Actually the Group class is abstract and has two concrete implementations : OrderedGroup and UnorderedGroup.
- *IModelVisitor*
  This is the interface for all the visitors (see *Visitor* pattern [14]) for the classes implementing *IModelElement* interface. Currently, the *Visitor* pattern is used to save the different objects of the data-model into the DB (see section 4.1.2).

The analysis result part of the data-model deserves more explanations. Figure 13 shows a high level UML diagram of this part.

As we said, the AnalysisResult class implements the *IModelElement* interface. This class contains a reference to an AnalysisResultData, which is an abstract class whose concrete implementations store the necessary information for the different types of analysis result. Currently we have the following concrete classes :

- AnalysisResultProcessedTraceData
  It contains the references to the source Trace and the processed Trace objects.
- AnalysisResultGroupData
  It contains the reference to the root node of the grouping.
- AnalysisResultSearchData
  It contains a list of *ISearchable* objects.
- AnalysisResultAnnotationData
  It contains a list of Annotation objects and the corresponding AnnotationType objects.
- AnalysisResultFileData
  It contains a list of ResultFile objects.

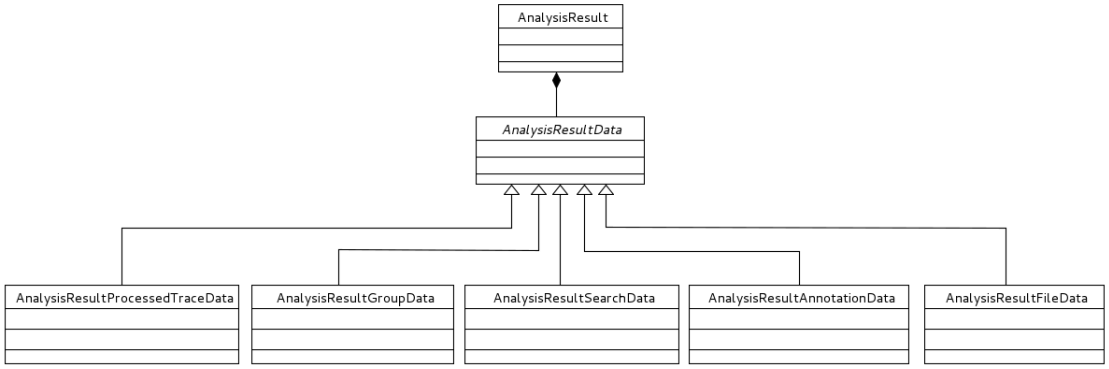


FIG. 13 – Analysis results class diagram

Finally the `com.inria.soctrace.lib.model` plugin provides also a custom exception class (SoCTraceException) to be used in the whole software library and a ModelConstant class to store some constants related to the data-model.

4.1.2 Storage

The `com.inria.soctrace.lib.storage` plugin contains classes dealing with SoC-Trace databases, allowing a user to easily create a new database, to open an existing one, to get the database connection and to perform SQL queries. The main classes are the following :

- SQLConstants
  It contains predefined SQL commands to create/modify SoC-Trace tables.
- StoredProcedures
  It contains the SQL code for some stored procedures that can be used for manually inspecting the database content.
- DBInitializer
  It contains the Java code to create the various SoC-Trace tables.
- SystemDBObject and TraceDBObject
  These classes allow the user to deal with the System DB and the Trace DB respectively. Both classes derive from the abstract DBObject class and allow the user to :
  - Create a new DB of the given type.
  - Open/Close a connection to an existing DB. The connection can then be used to give SQL commands to the DB using JDBC statements.
  - Commit/Rollback the current DB transaction (auto-commit mode is OFF for efficiency reasons).
- ModelElementCache
  It is a generic cache for objects belonging to classes implementing the *IModelElement* interface. The user can specify what kind of concrete classes is interested in storing in the cache and then he can save the objects of those classes. This cache is actually used in both DBObject concrete classes to store the *Element*Type and *Element*ParamType objects for Trace and Event SDP.
- ModelVisitor
  This class implements the *IModelVisitor* interface. It is an abstract class providing base

functionalities common to all the visitors that work on the DB, dealing with model objects. For the moment being, only the ModelSaveVisitor class has been implemented, in order to save the model objects into the DB. Each of the DBObject concrete class offers a *save()* method, that accepts an object implementing the *IModelElement* interface and calls the *accept()* method on such object passing an instance of the ModelSaveVisitor. This way, the interface to save a model object into the DB is very easy for the library-user. For example, to save a list of Event objects into the DB, you have to write only the code shown in Figure 14, where `events` is a list of Event objects and `traceDB` is the reference to a TraceDBObject.

```
for ( Event e: events ) {
        traceDB.save(e);
        for ( EventParam ep: e.getEventParams() ) {
                traceDB.save(ep);
        }
}
```

FIG. 14 – Saving Event objects into the DB

As we can see, all the complexity related to saving into a DB objects of different types (Event and EventParam in the example), is hidden behind the DBObject, which contains the save visitor. The visitor class contains a JDBC prepared statement for each supported object of the model. Like this, each call to the *save()* actually adds the object to the prepared statement batch, while the actual save operation into the DB is done in a buffered way, executing the prepared statement batch at the end.

4.1.3 Query

The `com.inria.soctrace.lib.query` plugin contains classes used to perform generic queries on the objects of the model. The plugin provides both query-classes and condition-classes. The basic idea is that to perform a query on an element of the model, you instantiate the specific query-class for that element and add to such query-object the required conditions to be fulfilled. Both query and condition classes are described in the following.

Condition-classes

To understand how query-classes work, first we describe the conditions that can be attached to them. We can have two kinds of condition-classes :

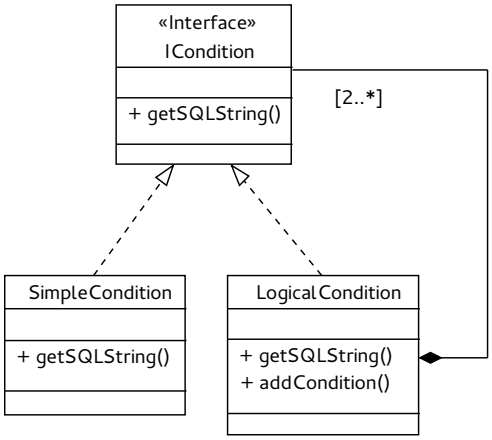
- The condition-classes implementing the *ICondition* interface, used to specify conditions for simple attributes of a generic table.
- The condition-classes implementing the *IParamCondition* interface, used to specify conditions on *Element*-SDP parameters. We remind that, here and in the following, by *Element* we identify a generic concept modeled as SDP.

Both kinds of condition-classes are built using the same design pattern (*Composite* [14]), as shown in Figure 15(a) and Figure 15(b).

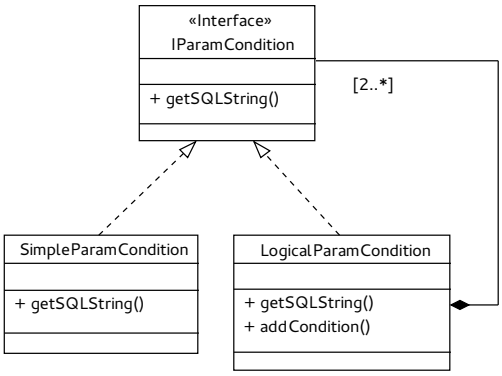
In both cases (*ICondition* and *IParamCondition* classes), *Simple* conditions are used to model expressions having the following format :

```
<name> <comparison_operator> <value>
```

It is possible to arbitrarily combine simple conditions using *Logical* conditions (AND/OR).



(a) ICondition classes



(b) IParamCondition classes

FIG. 15 – Condition-classes UML diagrams

```
<condition> <logical_operator> <condition>
```

For the classes implementing *ICondition* interface, <**name**> refers to the name of an attribute of a specific table and <**value**> refers to the value of this attribute. An example of SQL condition produced using *ICondition* and referring to the EVENT table could be the following :

```
( ( TIMESTAMP > 1000 ) AND ( CPU = 0 ) )
```

For the classes implementing *IParamCondition*, <**name**> refers to the *value of the attribute 'NAME'* in the *ELEMENT*_PARAM_TYPE table and <**value**> refers to the *value of the attribute 'VALUE'* in the *ELEMENT*_PARAM table. So, in this case the actual SQL condition is more complicated, since it requires information from the two tables above mentioned, plus the *ELEMENT*_TYPE table. However this complexity is hidden by the *IParamCondition* class, so the final SQL condition can be safely put in the WHERE clause of a query performed on the *ELEMENT* table.

Query-classes

A query-class is a class that can be used to perform queries on database tables, possibly using the condition-classes described above. The `com.inria.soctrace.lib.query` plugin defines three main abstract classes for query-classes :

- ElementQuery
  This class provides the basic functionalities to perform generic queries on simple (non-SDP) DB tables. For this kind of queries, only *ICondition* classes are used. The concrete classes implementing this abstract class are :
  - AnalysisResultQuery
  - EventParamTypeQuery
  - EventProducerQuery
  - EventTypeQuery
  - FileQuery
  - ToolQuery

  Note that the AnalysisResultQuery does not retrieve from the DB the analysis result data (AnalysisResultData abstract class). For this purpose, you should use the AnalysisResultDataQuery (see below) class explicitly. This is done for efficiency reasons : first you perform the query to get the AnalysisResult objects respecting some conditions, then you load the AnalysisResultData only into those objects that actually are of interest for you. A common use case is in fact to browse into analysis results, just to see the type or the date of the analysis itself, without being interested in loading the analysis result data (potentially a huge amount of data).
- SelfDefiningElementQuery
  This class extends the ElementQuery class, adding the support for *IParamCondition*. The class is in fact designed to be the base class of query-classes for SDP elements. The concrete classes implementing this abstract class are :
  - EventQuery
  - TraceQuery
- AnalysisResultDataQuery
  This class is designed to be the base of all the classes that are used to retrieve an analysis result from the DB, specifying the analysis result ID. This class does not support *ICondition* or *IParamCondition*, since the only purpose is to retrieve from the DB the analysis result data produced in a given analysis. Once the data is retrieved, it can be set in the corresponding AnalysisResult object. The concrete classes implementing this abstract class are :
  - AnalysisResultProcessedTraceDataQuery
  - AnalysisResultGroupDataQuery
  - AnalysisResultSearchDataQuery
  - AnalysisResultAnnotationDataQuery
  - AnalysisResultFileDataQuery

Example Figure 16 shows an example of usage of the EventQuery class. In this example we want to find all the context switch events (CS) produced after the timestamp 1000000 on CPU 0, where the old PID is 1205 and the new PID differs from 0.

As the example shows, this query interface is quite powerful and generic, but has the drawback of being not so straightforward to use. Query-classes and condition-classes are therefore used to build a simplified higher-level query interface, described in section 4.1.4.

```
// Create Event query-object
TraceDBObject dbObject = new TraceDBObject(...);
EventQuery eventQuery = new EventQuery(dbObject);

// Event WHERE
LogicalCondition eventWhere = new LogicalCondition(LogicalOperation.AND);
eventWhere.addCondition(new SimpleCondition("TIMESTAMP", ComparisonOperation.GT, "1000000"));
eventWhere.addCondition(new SimpleCondition("CPU", ComparisonOperation.EQ, "0"));
eventQuery.setElementWhere(eventWhere);

// EventParam condition
ParamLogicalCondition and = new ParamLogicalCondition(LogicalOperation.AND);
and.addCondition(new ParamSimpleCondition("OLDPID", ComparisonOperation.EQ, "1205"));
and.addCondition(new ParamSimpleCondition("NEWPID", ComparisonOperation.NE, "0"));
eventQuery.addParamCondition("CS", and);

// Perform query
List<Event> elist = eventQuery.getList();
```

FIG. 16 – EventQuery example

4.1.4 Search

The `com.inria.soctrace.lib.search` plugin provides the means to perform predefined requests on the objects of the data-model without directly dealing with *ICondition* or *IParamCondition* objects, thus using a simplified interface. More precisely, the plugin provides a search interface (*ITraceSearch*) offering a rich set of predefined requests, allowing the user to perform queries on both databases (System and Trace DB) without actually caring about databases. The interface in fact can be used in a DB-agnostic fashion. The interface is implemented by the TraceSearch class. Before starting using the search interface, the user shall simply initialize the TraceSearch object and, at the end, the object should be uninitialized. The possible predefined requests provided by this search interface are listed below.

- Requests to get one or more Tool objects, respecting the given criteria :
  - Get the tool with a given name.
  - Get all the tools of a given type.
- Requests to get one or more Trace objects, respecting the given criteria :
  - Get the trace corresponding to a trace DB name.
  - Get all the traces produced between two dates.
  - Get all the traces where a given application has been traced.
  - Get all the traces produced on a given board.
  - Get all the traces produced on a given operating system.
  - Get all the traces produced on a given number of CPUs.
  - Get all the traces captured using a given output device.
  - Get all the traces having a given description.
  - Get all the traces of a given type.
  - Get all the traces whose type is among the ones specified.
  - Get all the traces of a given type, for which given parameters have given values.
  - Get all the traces (processed or not).
  - Get all the raw traces (non-processed traces).
  - Get all the processed traces.
- Request to get the File objects related to a given trace
- Requests to get the EventProducer objects belonging to a given trace, respecting the given

criteria :
  - Get the event producer, given the type and a local identifier.
  - Get the all the event producers of a given type.
- Requests to get the Event objects belonging to a given trace, respecting the given criteria :
  - Get all the events of a given CPU.
  - Get all the events of a given type.
  - Get all the events whose type is among the ones specified.
  - Get all the events with a given producer.
  - Get all the events whose producer is among the passed ones.
  - Get all the events whose timestamp is included in the specified interval.
  - Get all the events whose timestamp is included in one of the specified intervals.
  - Get all the events of a given type, for which given parameters have given values.
  - Get all the events of a given type, for which given parameters have given values and whose timestamp belongs (at least) to one of the intervals specified.
  - Get all the events whose type is among the ones specified, whose timestamp belongs (at least) to one of the intervals specified and whose producer is among the one specified.
- Requests to get the AnalysisResult objects related to a given trace :
  - Get all the analysis results produced by a tool.
  - Get all the analysis results produced by a tool and of a given type.
- Request to get the AnalysisResultData object related to a given analysis result.

Example Figure 17 shows an example of usage of the *ITraceSearch* interface. In this example first we get the Trace object corresponding to a given DB name. Then we search for all the context switch events of this trace, where the old PID is 1203 and the new PID is 1205.

```
// Create the TraceSearch object and initialize it
ITraceSearch traceSearch = new TraceSearch().initialize();

// Get the Trace, given the DB name
Trace trace = traceSearch.getTraceByDBName("KpTraceParser_20120509_152627_GMT");

// Perform the request
List<ParamDesc> paramsList = new LinkedList<ParamDesc>();
paramsList.add(new ParamDesc("OLDPID", "1203"));
paramsList.add(new ParamDesc("NEWPID", "1205"));
List<Event> elist = traceSearch.getEventsByParams(trace, "CS", paramsList);

// uninitialize the search object
traceSearch.uninitialize();
```

FIG. 17 – Search interface usage example

The `com.inria.soctrace.lib.search` plugin offers also an utility class (Printer) providing methods to print collections of data-model objects, as returned by the search interface methods. The Printer class offers also a method (`selectResult()`) to interactively browse the analysis results saved in the database, in order to visualize them in a simple, textual form. The analysis result browsing is performed according to the following steps :

- First select the trace where the analysis has been performed (e.g. trace imported on date x).
- Then select the tool performing the analysis (e.g. analysis tool y).
- Then select the type of result produced by that tool (e.g. searching, grouping, etc.).
- Finally select the result instance of that type (e.g. grouping performed on date z).

4.1.5 Utilities

The `com.inria.soctrace.lib.utils` plugin provides some utility classes, useful for developing client applications interacting with the SoC-Trace Library. The classes currently present are the following :

- IdManager : this class can be used to manage sequential IDs for model entities. In fact the burden of setting the ID values to the elements of the model (for the storage with the relational database representation) is currently up to the library-user : at the application level is in fact convenient to be aware of the IDs in order to manage objects efficiently (e.g. using maps of objects).
- Configuration : this is a Singleton [14] that wraps the SoC-Trace infrastructure configuration file (`.soctrace.conf`, located in the user directory). The class provides an enumerate (SoCTraceProperty) which contains the different properties defined in the configuration file and a *getter* method that allows the user to access such properties.
- Portability : this class provides some basic functionalities to ensure the portability of the infrastructure on both Unix and Windows systems.
- DeltaManager : simple class that can be used to measure time intervals, for example to monitor execution times.
- HeapMonitor : utility class providing facilities to measure the size of the used heap in a Java application.
- History : utility class that can be used to log runtime events in a thread-safe way, useful to debug multithreaded Java code without perturbating too much the execution.

4.2 Management Tool

The SoC-Trace Management Tool is implemented by two different plugins :

- `com.inria.soctrace.management.core`
- `com.inria.soctrace.management.ui`

The tool implementation is split over this pair of plugins in order to decouple what is related to visualization (UI) from what is related to functionality. Currently both plugins are very simple because of the prototype-nature of the tool itself.

4.2.1 Core

The core plugin (`com.inria.soctrace.management.core`) provides basic functionalities to the UI plugin. It encapsulates the SoC-Trace Library functionalities needed to provide services to the end user. For the moment being the main classes provided are :

- SoCTraceManagementConstants : it stores system level constants.
- SoCTraceManager : it provides the implementation of the operations accessible from the UI.
- ToolExecutionManager : it provides a simple interface to launch an external command as an Eclipse Job.

4.2.2 Ui

The UI plugin (`com.inria.soctrace.management.ui`) provides the graphical elements that form the SoC-Trace infrastructure user interface. For the moment being it contains classes for :

- UI constants
- The JFace [15] actions and dialogs for the operations described in section 3.3.

5 External Tools

In order to test and use the infrastructure with real use cases, we developed some external tools. First of all, to import real-life traces into the infrastructure, we wrote a parser for the KPTrace format (see section 5.1). Then, to test the different functionalities of the infrastructure, we developed also a simple test tool, described in section 5.2.

5.1 KPTrace Importer

This parser is simpler than the one provided by STMicroelectronics [16], since it contains very little semantics and is written using several simplifications. Typically, only a subset of event types is supported and no link is created between sibling events. Nevertheless the tool is useful since it allows the storage of traces with a real-life format into the SoC-Trace infrastructure and shows a possible usage of the SoC-Trace Library in order to write a Java importer.

5.1.1 Parser architecture

The KPTrace Importer is written as a normal Java program, using the SoC-Trace Library. This tool (KpParserTool) can be exported as an executable `jar` in order to be registered to the infrastructure tool registry. In the following the main classes of the tool are described.

- KpParserArgs : class containing the command line parameters necessary to the parser.
- KpRecord : simple class to store the raw information contained in a single raw-trace line.
- KpConstants : KPTrace parser constants (e.g. event producer types).
- KpParser : it is the actual parser class, which performs the parsing of all trace files in order to put them in the DB format. It uses the following utility classes to perform its job :
  - KpEventFormatManager : manages EventType and EventParamType objects for the specific KPTrace format. This class, starting from an intermediate description of the format, stored in the KpEventFormatDescription singleton, builds a map where each event type mnemonic is linked to the corresponding EventType object. The intermediate description stored in the KpEventFormatDescription singleton is a list of strings with the following format :

    ```
    <event_type_mnemonic> : <par0> <type0>, <par1> <type1>, [...]
    ```

    For example, for the context switch event it could be :

    ```
    CS : OLDPID INTEGER, NEWPID INTEGER
    ```

    The use of such generic intermediate format makes the importer code easily reusable for other trace formats.
  - KpEventProducerManager : manages EventProducer objects, for KPTrace specific event producer entities.
  - KpTraceInfoManager : manages the Trace element SDP, providing the predefined TraceType with the predefined trace parameter types. It provides also parsing functionality to fill the corresponding Trace and TraceParam objects.
- KpParserLauncher : open the System DB, creates the new Trace DB and launches the KpParser. At the end of parsing operations, commits all the changes to the databases.
- KpParserTool : it is the class containing the main method of the parser. This method simply handles command line arguments, checking them and then traslating them into an instance of the KpParserArgs class, which is then passed to the KpParserLauncher.

5.2 Test Tool

In order to test and debug the SoC-Trace infrastructure, we developed a Test Tool, which provides several batteries of tests for the different functionalities. Basically the tool provides several classes, each providing different methods to test a specific part of the infrastructure. The main class of the tool allows the user to set up a working environment (creation of the System DB and import of sample traces) and to enable or disable the different tests on such environment. Currently we developed test classes for the following macro-areas of the infrastructure :

- low level queries using the `com.inria.soctrace.lib.query` plugin
- high level search requests using the `com.inria.soctrace.lib.search` plugin
- analysis results saving
- simple trace processing

In addiction to the possibility to produce and save sample analysis results, the Test Tool allows the user to interactively browse through the analysis results saved into the database, using the result browsing method provided by the Printer class (see 4.1.4). We finally observe that the source code contained in the Test Tool, besides offering testing functionalities, provides also a rich set of code snippets, useful for application developers aiming to add a tool to the SoC-Trace infrastructure.

6 Performance Evaluation

In this section we present the first performance evaluation results we obtained on the SoC-Trace infrastructure prototype implementation. To get these results we used both the KPTrace Importer (section 5.1) and some simple *ad-hoc* programs, written to test specific features of the SoC-Trace Library (section 4.1). The machine used to perform our performance evaluation tests has the following characteristics :

- RAM : 4 GB @ 1066 MHz
- CPU : dual-core @ 2.40 GHz
- OS : Fedora 16, 64 bit

The MySQL database is used with the InnoDB engine, with `auto-commit` disabled and all the other configuration parameters left with their default values. The version of Eclipse used is the 4.2.0 (Juno).

6.1 Trace DB size

First of all we studied the relation between the size of the raw trace and the corresponding Trace DB, in order to see how much overhead is added by the relational representation of data. We used raw traces (KPTrace format) whose size ranges from 10 kB to 100 MB and we computed an *increase factor* as the ratio between the Trace DB size and the raw trace size. Figure 18 shows the Trace DB size obtained, while Figure 19 shows the above mentioned *increase factor*. From the second histogram, we can note that, as expected, for small traces the overhead is huge, with the DB size more than 35 times bigger than the raw trace. Increasing the raw trace size, the overhead decreases until it reaches an asymptotic value, with an *increase factor* included between 3 and 4 times.

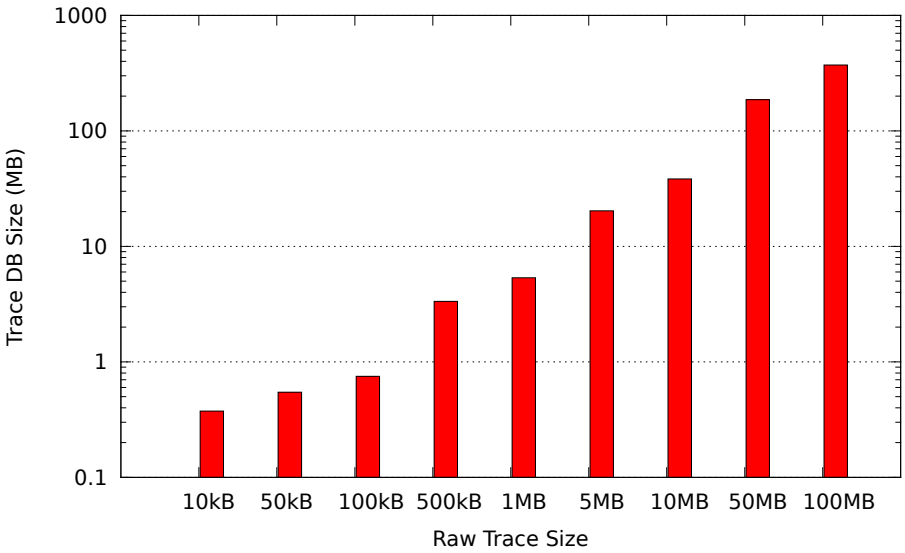


FIG. 18 – Trace DB size

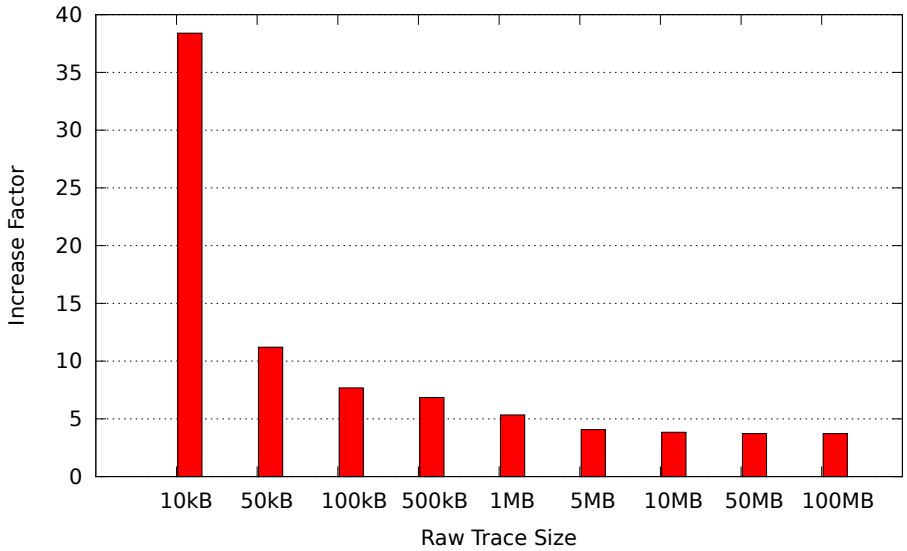


FIG. 19 – Database overhead

6.2 Trace import time

We measured the time needed to import the events of a raw trace into the infrastructure using our KPTrace Importer (section 5.1). The size of the raw traces used ranges from 10 kB to 50 MB. For each different configuration we performed 10 runs in order to have reliable values. The results are shown in Figure 20.

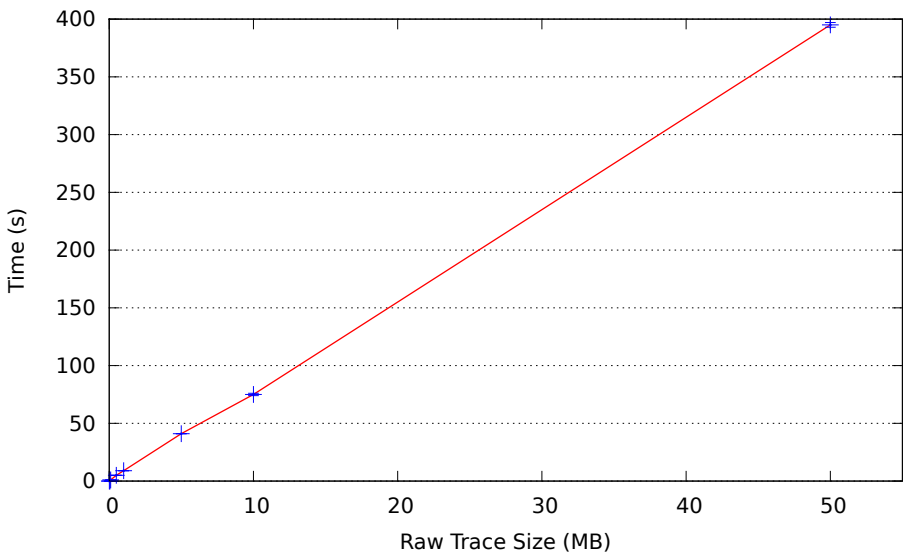


FIG. 20 – Time needed to import KPTrace traces into the infrastructure

We note that the time needed for the import operation increases linearly with the raw trace

size. This means that apparently there are no memory management issues and the system scales well. To further investigate the trace import time, we measured also the percentage of time spent parsing the raw trace files and the percentage spent writing the data into the database. The results are shown in Figure 21.

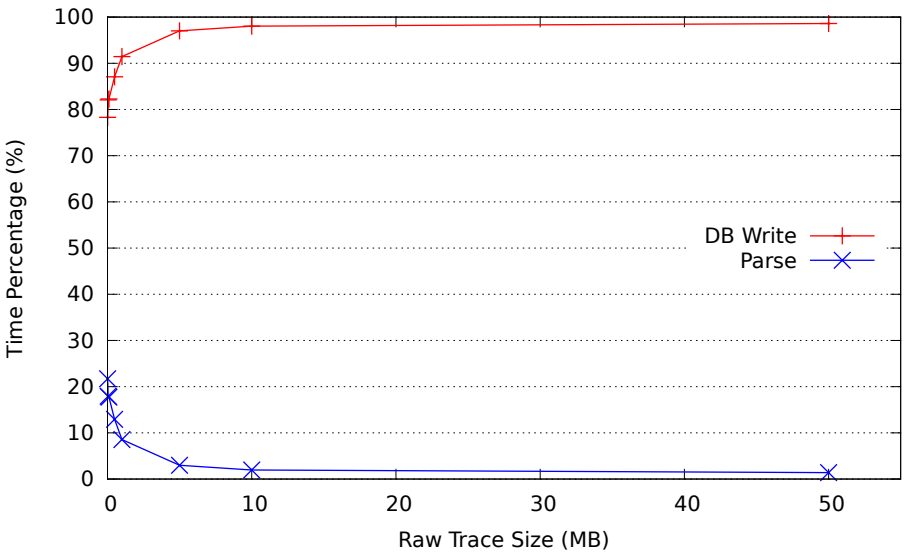


FIG. 21 – Percentage of time spent parsing and writing into the DB for trace importing

We note that for all trace size values the greatest percentage of time is spent in writing into the database. We observe that for smaller traces the parsing time percentage is higher compared with the parsing time percentage of bigger traces, since there is a certain amount of time spent parsing the processes (event producers) created during the execution traced : being this number of processes constant for all traces, the time spent for this parsing operation has a bigger weight on smaller traces. Being the write operation the most expensive one, we shall investigate the problem in order to improve writing performance.

6.3 Buffered reading

We studied the effect of a buffered reading to retrieve the events from the database. In each experiment we measured the time needed to retrieve from the database a total of 100000 events, using the EventQuery class (section 4.1.3) to perform the requests. At each request we retrieved only a page of events, namely a given number of events. It is straightforward that varying the size of the page, the number of needed requests varies too. The page size used ranges from 1000 events to 100000 events. For each different configuration we performed 20 runs. The results are shown in Figure 22.

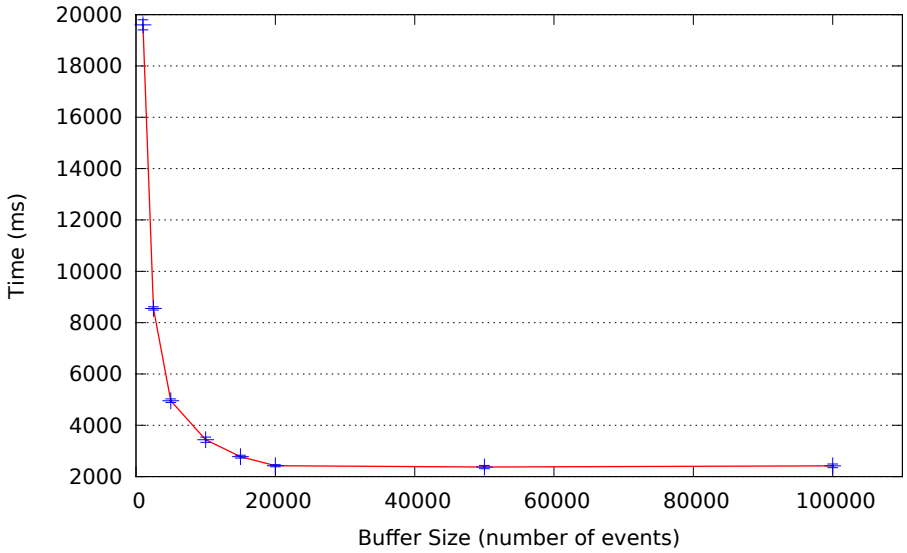


FIG. 22 – Time needed to read a given number of events using different buffer size

It is clear from the curve that performing a bigger number of requests affects negatively time performance. With a page size of 1000 events (100 requests) the time spent to read all the events is about 19 seconds, then increasing the page size (reducing the number of requests) the time decreases up to 2.4 seconds, which is the time obtained with a page size of 20000 events. Continuing to increase the page size the time needed to retrieve the events does not change significantly.

6.4 Complex parameters queries

Finally we performed complex queries using the *ITraceSearch* interface (section 4.1.4), in order to see how increasing the number of SDP-parameter-conditions involved in the queries affects requests duration. For this purpose, we used a trace containing 100000 events, performing queries concerning the Event-SDP tables, with a variable number of SDP-parameter-conditions (`parameter = value`) linked by the AND logical operator. The number of events of the result is always the same for each configuration (for each number of different parameter-conditions). The number of parameter-conditions used ranges from 1 to 50. For each configuration we performed 20 runs. The results are shown in Figure 23.

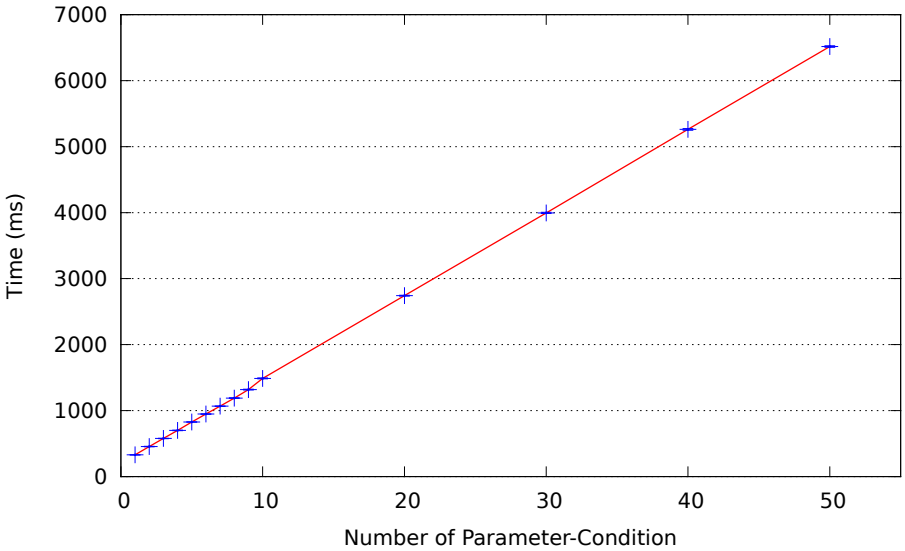


FIG. 23 – Time needed to perform a request varying the number of SDP-parameter-conditions

We note that the amount of time needed to perform the requests increases linearly with the number of parameter-conditions, meaning that there is no visible issue in memory management and the system scales well. It will be interesting to try to optimize the system in order to decrease the slope of the curve, for example using custom DB indexes on the SDP tables.

7 Future Works

The SoC-Trace infrastructure described in this document is a first prototype which will evolve in the near future. In this section some of the main works that we are planning are described.

7.1 Database architecture

As described in section 3.1.2, currently the database has a distributed architecture, with a single System DB and several Trace DBs. This has been done basically for scalability reasons. However, from a logical point of view, all Trace DBs have the same schema. Furthermore, each new trace added to the infrastructure needs the creation of a new database. These observations made us think about a *centralized* solution, with a single database that stores all the information, without suffering from the single-DB issues discussed in 3.1.2. Instead of replicating all the database tables for each trace, we can simply replicate only the tables that contain specific-trace raw information : EVENT, EVENT_PARAM and EVENT_PRODUCER tables. Obviously, for each trace, these tables shall have a different name : for example there can be a suffix with the trace ID in the name of the table (e.g. EVENT_1). Like this, when a new trace is imported into the infrastructure, only these three tables shall be created and not a whole database. This solution has the additional advantage that it is no longer necessary to rewrite the trace format description (EVENT_TYPE and EVENT_PARAM_TYPE table information) for each trace of a given type : it is enough to write the format once, then all the trace of that format can refer the same event type and event parameter types. For these reasons we started implementing

a centralized version of the database architecture, in order to better compare the pros and the cons of the two different solutions to finally choose the best one.

7.2 Analysis result

The current data-model is designed to support single-trace analysis results, however future analysis tools will be able to perform complex, multi-trace, analyses. For this reason, a crucial enhancement for the infrastructure will be the support for multi-trace analysis results. This enhancement can be done either keeping the *distributed* DB architecture or using the *centralized* DB architecture described in 7.1. In the former case, we can extend the System DB schema in order to save multi-trace analysis result. In the latter case, we simply need to add a new table, mapping the analysis results with the traces involved in the analysis. Furthermore, we are already thinking about adding new predefined types of analysis result, like for example *Graph Results*, which could be useful to represent Bayesian networks.

7.3 Tool management

A possible point of improvement for the infrastructure is to enhance the tool management features. First of all, the SoC-Trace Management Tool should be enhanced in order to provide support for workflow definition. It should be possible for a user to define a trace-management-workflow in order to accomplish a specific goal. For example a possible workflow could be the following : import a raw trace using *this* importer, perform a pattern mining analysis with *this* analysis tool and finally visualize the saved results with *this* visualization tool.

Then it will be interesting to explore the possibility to add specific support for tools written as plain Eclipse plugins and not necessarily as external applications. This will bring the advantage of making the tool integration easier and will let the tools use to the greatest extent Eclipse functionalities. As we said, now only generic external tools are managed, storing them into the TOOL table, which acts as a tool registry. Eclipse has actually a built-in registry mechanism for plugin management, that automatically recognizes and loads (if necessary) the requested plugins. In order to have an integration between the two kinds of tools, we should be able to keep the advantages of Eclipse plugins automatic discovering and loading, while preserving the possibility to manage non-Eclipse tools.

7.4 Other improvements

We are planning to do some implementation enhancements in order to make the SoC-Trace Library richer and easier to test. First of all we plan to add new *Visitor* classes to the `com.inria.soctrace.lib.storage` plugin, in order to support other common DB operations, like the *update* or the *delete* of an existing data-model entity. Then we want to improve the test tool, making it easier to add new tests and possibly adding an *installation-checking* utility, able to check if the SoC-Trace infrastructure has been correctly installed.

8 Conclusions

In this document we described the first prototype of the SoC-Trace infrastructure. After clarifying the objectives of the project and the infrastructure, we described both the system and the software architecture of the prototype, showing how the solutions are a solid starting point for the fulfillment of the desired goals. We also described some external tools we developed in order to start using the prototype with concrete use cases. We provided some first performance

evaluation results, useful to have a critical view on the technical choices done so far. Finally we discussed some of the future works we are planning to do in order to get an enhanced version of the infrastructure. The possibility to easily use the infrastructure with real-life traces, concrete analysis and result saving, showed that the proposed data-model, though being open to improvements, is actually generic enough to deal with arbitrary formats and non-trivial analyses. On the other side, performance evaluation results showed that there is room for enhancements in order to make the infrastructure more efficient. Our future efforts will be therefore on the improvement of both functionalities and performance.

References

[1] FUI (Fonds Unique Interministériel). http://competitivite.gouv.fr.

[2] Carlos Prada-Rojas, Miguel Santana, Serge De-Paoli, and Xavier Raynaud. Summarizing Embedded Execution Traces through a Compact View. In *Conference on System Software, SoC and Silicon Debug – S4D 2010*, Southampton, UK, September 2010.

[3] Damien Hedde and Frédéric Pétrot. A non intrusive simulation-based trace system to analyse Multiprocessor Systems-on-Chip software. In *International Symposium on Rapid System Prototyping*, pages 106–112, 2011.

[4] Miguel Santana. Présentation du Projet SoC-Trace, 2011.

[5] Eclipse. http://www.eclipse.org.

[6] MySQL. http://www.mysql.com.

[7] IP-XACT. IEEE 1685-2009, ISBN 978-0-7381-6160-0.

[8] MySQL Table Partitioning. http://dev.mysql.com/doc/refman/5.6/en/partitioning.html.

[9] KPTrace Trace Format. http://www.stlinux.com/stworkbench/interactive_analysis/stlinux.trace/kptrace_traceFormat.html.

[10] OS21 Trace Format. http://www.stlinux.com/stworkbench/introduction/STAPISDK_OS21.html.

[11] Jacques Chassin De Kergommeaux and Benhur De Oliveira Stein. Paje : An Extensible Environment for Visualizing Multi-Threaded Program Executions. In *Proc. Euro-Par 2000, Springer-Verlag, LNCS*, pages 133–144, 2000.

[12] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang Nagel. Introducing the Open Trace Format (OTF). In Vassil Alexandrov, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 526–533. Springer Berlin / Heidelberg, 2006.

[13] JDBC. http://www.oracle.com/technetwork/java/javase/jdbc/index.html.

[14] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] JFace. http://wiki.eclipse.org/index.php/JFace.

[16] STMicroelectronics. http://www.st.com.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803