KriQL: a query language for the diagnosis of transition systems

Khaoula Es-Salhi, Siham Rim Boudaoud, Ciprian Teodorov, Zoé Drey, Vincent Ribaud

Pre-proceedings of the

15th International Workshop on Automated Verification of Critical Systems

https://sites.google.com/site/avocs15/

Gudmund Grov, Andrew Ireland (Editors)

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK

KriQL: a query language for the diagnosis of transition systems

Khaoula Es-Salhi[1], Siham Rim Boudaoud[2], Ciprian Teodorov[2], Zoé Drey[2] and Vincent Ribaud[1]

[1]Lab-STICC CNRS UMR 3128, Université de Bretagne Occidentale
20, avenue Le Gorgeu, 29200, Brest, France
Email: essalhi.khaoula.mii@gmail.com, ribaud@univ-brest.fr
[2]Lab-STICC CNRS UMR 3128, ENSTA-Bretagne
2 rue François Verny, 29200, Brest, France
Email: boudaoud.siham.rim@gmail.com, ciprian.teodorov,zoe.drey@ensta-bretagne.fr

Abstract: The formal verification of a concurrent system with a model-checker provides the user with a counterexample trace when a property is violated; however the problem diagnosis still remain a complex issue. Diagnosis is made difficult for several reasons: the trace conforms to a structure that is internal to the verification tool and hence hard to exploit, the trace yields low-level information, the trace size can be large. Traces cannot be understood without the Labelled Transition System (LTS) underlying the model-checker exhaustive exploration; we designed KriQL: a query language over the LTS graph featuring a blend of set filters and graph-based operations and we implemented some feasibility prototypes. Benchmark results indicate that a hybrid management system using a graph-based database and dedicated data structures should achieve sufficient performances.

Keywords: trace, transition system, query language

1 Introduction

System verification aims to establish that the system under study (SUS) possesses certain properties, such as fairness or reachability. Properties are generally issued from the system's specification. A defect is found once the system violates one or several of the specified properties. Model-checking is a verification technique that relies on the systematic analysis of all the states of the system to check if the system model (formalized using process algebra or concurrent automata) satisfies the specification (typically expressed using temporal logics). If a state is encountered that violates the specification, the model checker produces a counterexample - an execution path that leads to the undesired state - also called a witness trace.

In this context, the typical diagnosis process is simple: a human or machine troubleshooter provides a description of the problem that occurred, a kind of analysis is performed and the root cause identified in order to provide the user with a remedy to the problem. Yuan et al. resume the paradigm of problem diagnosis : problem description → root cause → solution [YLW+06]. When the problem is represented with a witness trace, problem description needs to interpret the trace and this interpretation is challenging for several reasons: the trace conforms to a structure that might be or might not be available with the trace, the interpretation has to deal with the

different levels of details from which the traces are built, in practical settings, the size of the trace can be very large.

Because the states space explored by a model-checker is a graph, graph querying that yields restricted sub-graphs or aggregated results is an helpful companion for problem understanding. The work presented in this paper relies on the research hypothesis that the capability to efficiently query the potential behaviors of the system will ease problem interpretation and will enable the development of diagnosis environments, providing specialized visualizations and analysis tools.

This paper presents KriQL a language for query-based diagnosis. This language focuses primarily on the manipulation of the labelled transition system (LTS) constructed during a model-checking process. KriQL enables the identification, extraction, composition, and inquiry of system states, execution traces and sub-clusters of the LTS (representing symbolically encoded sets of execution traces). At a high-level, our approach merges the graph-theoretic view over the system execution, obtained through model-checking, with trace and state-oriented analysis tools (used traditionally during test-debug approaches [PTP07]). The uniqueness of our approach stems from the reification of the systems behaviors enabling the homogeneous manipulations of execution traces through a dedicated query-language. From a practical point of view, KriQL should be seen as a kernel language offering the high-level facilities needed for understanding and diagnosing complex concurrent systems. The kernel can serve as basis for implementing automated fault-localization strategies, such as the work presented in [GSB07].

To validate our approach, we have integrated the KriQL language into the OBP (Observer-Based Prover) verification environment [DBRL12]. To expose the labeled transition system generated during model-checking in KriQL, there is a need for a storage back-end enabling fast query execution, requirement which is not needed during the model-checking process. As such, the LTS storage infrastructure used by OBP was not adapted to our needs. Hence, for our prototype implementation we have evaluated two alternative solutions using relational and graph databases, respectively. The benchmark results indicate however, the need of a more specialized LTS management system that achieves a better performance trade-off between the runtime of the "state-oriented" and the "transition-oriented" queries in KriQL. Nevertheless, the capabilities offered by KriQL facilitate the understanding of systems exhibiting large number of potential behaviors enabling bridging the gap between automated system verification and diagnosis.

The rest of this paper is organized as follows. Section 2 overviews the motivation behind our approach through a simple example. Section 4 presents the KriQL language semantics and pragmatics. Different implementation architectures are evaluated in Section 5. Section 3 situates our approach with respect to the state-of-the art. This study concludes (Section 6) indicating some future research directions.

2 A motivating example

We borrow our example from an invited talk given by Leslie Lamport's [Lam84] about two neighbours Alice and Bob sharing a yard in an exclusive manner because Alice and Bob pets cannot be together in the yard. Lamport's solution uses two threads sharing only two boolean variables (two flags in the story), each of which can be written by one thread and read by the other. The mutual exclusion algorithm proposed by Lamport is given in figure 1. Incidentally,

it's a chivalrous algorithm; Alice has priority over Bob.

```
Alice:
  while (true) {
    flagAlice = up;
    while (flagBob == up) skip;
    catInYard;
    flagAlice = down; }

Bob:
  while (true) {
    flagBob = up;
    while (flagAlice == up) {
      flagBob = down;
      while (flagAlice == up) skip;
      flagBob = up; }
    dogInYard;
    flagBob = down; }
```

Figure 1: Lamport's mutual exclusion algorithm

2.1 A typical model-checking approach

For verification purposes, the system based on the mutual exclusion algorithm should be translated into a model specification in a concurrent automata-like language. Alice and Bob automata are given in figure 2. Transitions between states bear Event-Condition-Action expression; for instance the expression {*rain*} *[catInYard == true / AliceCatGoesHome]* means that when the event *rain* occurs and if the condition *catInYard == true* is satisfied then the action *AliceCatGoesHome* is performed. Once a system model is specified, the property to be checked are formalized using a property specification language; for instance the mutual exclusion property may be represented with the predicate *not (catInYard and dogInYard)*. Finally the model checker is run to check the validity of the property in the model specification. In our example, the model checker would check that the property holds in all configurations (there is no state violating the assertion).

According to [BK08], whenever a property is falsified, the negative result may have different causes. There may be a modelling error, this implies a correction of the model. It might be a design error or a property error. In case of a design error, the verification is concluded with a negative result, and the design (together with its model) has to be improved. It may be the case that upon studying the exposed error it is discovered that the property does not reflect the informal requirement that had to be validated. This implies a modification of the property, and a new verification of the model has to be carried out.

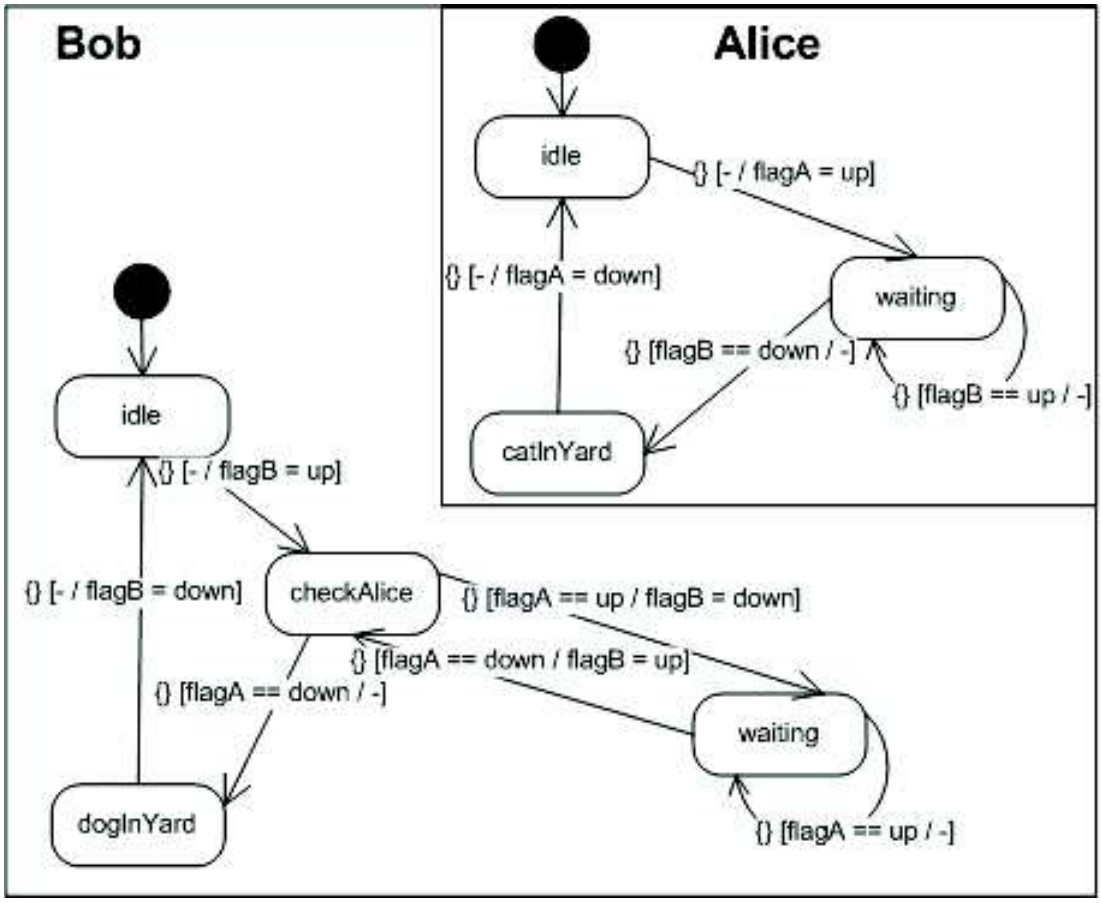

Figure 2: Automata of Alice and Bob behaviour

2.2 Diagnosis of a modelling error

Suppose that, in the state `checkAlice`, we mistyped the guard `flagA == down` in an assignment `flagA = down` and that we did not assert that Bob process cannot assign the shared variable `flagA`. Running the model-checker will violate the mutual exclusion property because there exists a path in the transition system where, once Bob has leaved the waiting state because Alice lowered her flag, Bob does not check Alice's flag (because of the mistyped instruction) before to unleash his dog, and if Alice has changed her mind at the same time, she might have raised her flag again, and unleashes her cat before Bob has raised his flag. Hence the model-checker will detect the violation and provides the user with a counterexample.

Suppose that we run a model-checker and got a labelled transition system (LTS), a graph resulting of the exploration of Alice and Bob system with a modelling error and the predicate `mutualExclusion`. Suppose that this cyclic graph is made of 14 configurations and 37 transitions. In order to distinguish LTS or process states, we call *configuration* an LTS state. A LTS configuration holds all information about processes including their state. The model-checker indicates that the mutual exclusion property has been violated for the first time in configuration

12. The counterexample trace is $0 \to 1 \to 3 \to 6 \to 9 \to 12$.
A textual and verbose detailed description of the counterexample is generally available where we find the value of each component of the automata: state, variable values ... However, the diagnostician might want to see only the value of variables `flagA` and `flagB` in the violation state 9 and its predecessor 6. In a pseudo-SQL language, we might write the query as

```
select LTS.confID, Alice.state, Alice.flagA, Bob.state, Bob.flagB
from myTrace where LTS.confID = 9 or LTS.confID.successor = 9,
```

and the results might be
$ID = 6; Alice : state = catInYard, flagA = true; Bob : state = checkFlagAlice, flagB = true$
$ID = 9; Alice : state = catInYard, flagA = false; Bob : state = dogInYard, flagB = true$

Such a result will help the diagnostician to notice that something happened to `flagA` (that went to `down` when the process `Bob` was in state `checkFlagAlice`) and probably will lead her to the modelling error.

2.3 Diagnosis of a design error

Mutual exclusion is only one of several properties of interest. Herlihy and Shavit introduce their book with the same example and list three other properties: deadlock-freedom, starvation-freedom and waiting [HS12]. A possible approach to implement properties in a model-checking approach is the use of observers. An observer is an automaton that monitors the model behaviour in order to verify faults. An observer is composed with the model through a synchronization product, i.e. the observer automaton is added to other automata and all possible states explored in a brute-force manner. The observer state is changing during the exploration and can reach special states, called reject states that denote a violation of the property monitored by the observer.

Alice and Bob are dead-locked if each of them has raised their flag and is waiting until the other lowers its flag. It does not happen in the Lamport's algorithm from figure 1 because as afore mentioned the algorithm is chivalrous. If Alice and Bob each raise their flags, Bob eventually notices that Alice's flag is raised, and defers to her by lowering his flag, allowing her cat into the yard [HS12]. The deadlock-free property can be implemented by a small automaton, presented in Figure 3, that changes state when Alice or Bob has raised their flag and enter in a reject state if Bob or Alice raises also their flag [1].

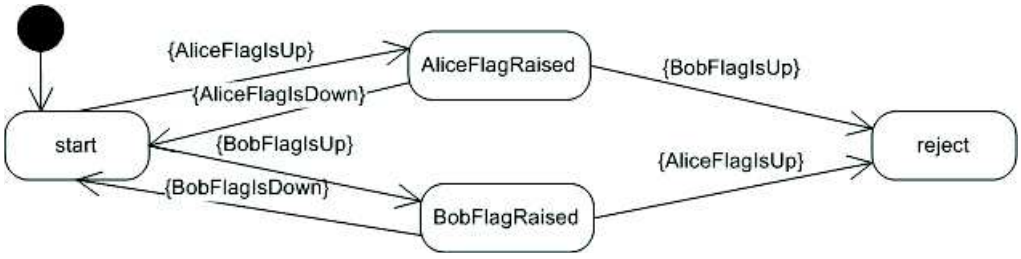

Figure 3: An observer automaton for checking Alice and Bob deadlock

[1] Actually, the observer rejects when Alice and Bob flags are both up and it might not be a problem if the algorithm can deal with this situation; we assume that the observer detects deadlock for simplicity sake.

Suppose that Bob adopts the same algorithm as Alice: *a*) he raises his flag; *b*) when Alice's flag is lowered, he unleashes his dog; *c*) when his dog returns, he lowers his flag.

With this design, there are several paths in the transition system where Alice and Bob both raised their flag and are dead-locked. The model-checker will reach the reject state of the observer and produces a counterexample to this reject state.

Suppose that we run again a model-checker and got a LTS graph resulting from the exploration of Alice and Bob system with a design error and the observer `deadLock`. Suppose that this cyclic graph is made of 13 configurations and 32 transitions. The model-checker indicates that the deadlock observer reached reject for the first time in configuration 4. The counterexample trace is straightforward $0 \rightarrow 2 \rightarrow 4$, that is one of the shortest paths to achieve a deadlock: Alice raises her flag, Bob raises his flag, and they are deadlocked.

In a large trace, an useful query for the diagnostician will highlight the configurations where the observer did change its state (a pseudo-predicate called `changed`), and a way to figure out some information about the intermediate paths between these highlighted configurations, for instance the size of intermediate paths. In a pseudo-SQL language, we might write the query as

```
select LTS.confID, count(*) from myTrace where deadlock.changed
```

and the results might be

$LTS.confID = 0$

$LTS.confID = 2, count(*) = 1$

$LTS.confID = 4, count(*) = 1$

Such a result helps the diagnostician to focus on configuration 2 and she might query again to get Alice or Bob processes values in configuration 2, its predecessors or successors. The diagnostician proceeds with a mixture of navigation queries `changed`, `successors`; selection queries `select LTS.configuration` and aggregation queries `count(*)` until she is able to localize and fix the error.

3 Related work

3.1 Context-aware verification

Several model checkers such as SPIN [Hol97], Uppaal [LPY97], TINA [BRV04], have been developed to help the verification of concurrent asynchronous systems. In most if not all model-checking approaches, environmental conditions applying to the system execution (that we call contexts) are included in the system model. Our approach (that we called *c*ontext-aware verification) chooses to explicit contexts separately from the model. Context-aware verification focuses on the explicit modelling of the environment as one or more contexts, which are then iteratively composed with the System Under Study (SUS). Requirements are expressed either with predicates or with observer automata, as introduced in section 2.3. Requirements are verified within the contexts that correspond to the environmental conditions under which they should be satisfied, each context verification is orchestrated in a fully automatic divide-and-conquer algorithm. The interleaving of these contexts generates a labelled-transition system representing all behaviours of the environment, which can be fed as input to traditional model-checkers. The verification is performed by the tool OBP (Observer-Based Prover) [DBRL12]. As other model-checking research groups, a part of our research effort is dedicated to push further the state

explosion limit but we develop also others tools such as step-by-step simulation, trace search engine or graph visualization of the underlying labelled transition system. All these developments are implemented in a tool kit OBP *Observation Engine* and are freely available[2].

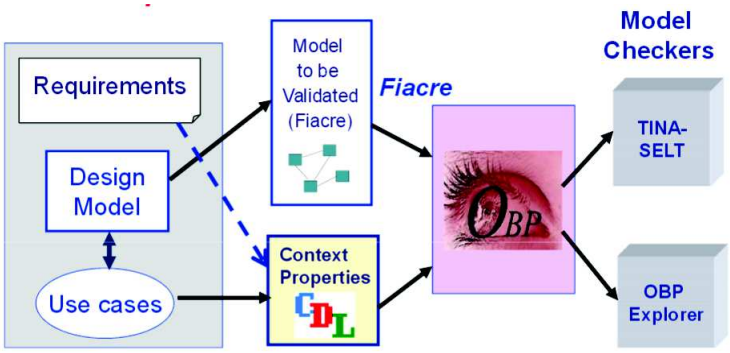

Figure 4: Context-aware overview

Fig. 4 shows a global overview of the OBP approach. The SUS is described using the formal language Fiacre [BBF+08], which enables the specification of interacting behaviours and timing constraints through timed-automata. The surrounding environment and properties are specified using the CDL formalism. Different model-checkers can process verification.

3.2 Trace query languages

Querying traces can be applied in any domain where execution traces can be recorded. There are two types of query-based debugging, those that operate a posteriori on traces and those where the query is weaved with the source program and parameterize the trace recording.

In [MLL05], authors propose PQL (Program Query Language), a language intended to query sequences of events associated with a set of related objects. They developed both static and dynamic techniques to find solutions to PQL queries. The static analyzer finds all potential matches conservatively using a context-sensitive, flow-insensitive, inclusion-based pointer alias analysis. Static results are also used to reduce the scope of dynamic analysis. The dynamic analyzer instruments the source program to catch all violations precisely as the program runs and to optionally perform user-specified actions. A PQL query is a pattern to be matched on the execution trace and actions to be performed upon the match. Subqueries allow users to specify recursive event sequences or recursive object relations. Such constructions are interesting and might be integrated in the KriQL language. However PQL does not yield aggregate queries as we need.

In [GOA05], authors propose Program Trace Query Language (PTQL), a language based on relational queries over program traces. Produced traces result from an automatically instrumentation by a tool PARTIQLE that monitors particular properties. Given a PTQL query and a Java program, PARTIQLE instruments the program to execute the query on-line. PTQL is a subset of SQL. It does not work on the LTS but on linear traces that play the role of relational tables (for

[2] OBP Language and Tools set website: http://www.obpcdl.org

instance, a trace can be joined with another trace). PTQL does not offer navigation queries as typical graph-based queries that we need.

Classical property specification languages, such as PSL[11], can also be considered as inputs to KriQL features. In this case the verification tools using these languages can be seen as the query execution runtime, and the witnesses produced would be considered as the query result.

4 KriQL: a graph-based query language

Generally, diagnosis encompasses any activity that provides information about the SUS, including analysis, observation, proofs, testing, etc. Merriam-Webster on-line dictionary defines diagnosis as an "investigation or analysis of the cause or nature of a condition, situation, or problem". However, for this study we use a narrower vision of diagnosis, which is restricted to the analysis of concurrent systems captured through Labelled Transition Systems, and considers that symptoms are materialized by traces. This setup corresponds to a model-checking verification approach. In this context, we consider that the semantics of the system was captured explicitly in a LTS, and that the symptoms are exhibited through counter-example traces. In his Turing Award lecture [CES09], Edmund Clarke emphasized that "interpreting long counterexamples" is still an open problem hindering the wide usage of model-checking (especially in industrial settings).

To address this problem, in this paper we propose the definition of a specialized query language that enables the uniform manipulation of the state-space and the counter-example as a set of traces. This approach complements the model-checking toolkit with the means for exploratory analysis (manual or automated) of model-checking results to facilitate the understanding, the localization and the isolation of the defects witnessed through the counter-examples.

In this section, we present KriQL, a query language operating on traces and the underlying transition system. KriQL is a front-end tool working on traces issued from our exploration engine called OBP, presented in the section 3.1. An overview of KriQL main concepts is presented in the section 4.1 as a meta-model, and its denotational semantics is sketched in the section 4.2. There are several candidate architectures to implement KriQL features that are presented in the section 5.1. KriQL implementation performances and architecture are discussed in sections 5.2 and 5.3.

4.1 KriQL overview

To ease the understanding of traces, we defined KriQL (for Kripke Query Language), a language that is aimed at expressing queries to extract relevant information from the traces produced by the exploration of a SUS. A trace (or path) is a part of the LTS – a Kripke structure [Var07] – representing the exploration graph of the SUS. Queries include the search of a path between two configurations, such as the shortest or longest path between two configurations according to criteria such as the change of variable values or the calculation of the range values of a variable in a given trace. The information resulting from the queries consists of either a subset of states (i.e., configurations) or a subset of transitions representing an excerpt of the traces. This information can be further filtered to show only the data of interest (such as process identifiers, or the evolution of a specific variable in a given trace).

KriQL meta-model

To enable the extraction and the representation of such information, we have defined a meta-model of KriQL, partly illustrated in Fig. 5. The meta-model provides the data structures (classes) that are necessary for an Application Programming Interface that will implement KriQL queries and answers. A ConfigurationSet is a set of configurations, each of which contains the status of a set of behavioral elements representing a subpart of the SUS. Since we work on data produced by the OBP explorer tool, the behavioral elements correspond to Fiacre elements, whether components (as the parallel combination of a set of processes) or independent processes. A TransitionSet is a set of transitions between configurations. A PathSet is a set of paths (i.e., traces), where a path corresponds to an ordered set of transitions. Finally, a Trail is a folded representation of a path. Specifically, it explicits only parts of a path according to filtering criteria given in a query.

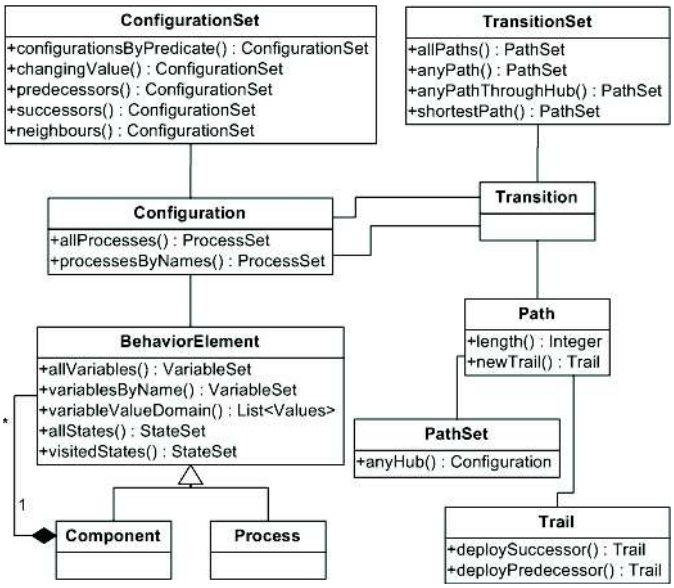

Figure 5: KriQL meta-model

4.2 The KriQL language

The *denotational definition* of a language consists of three parts [Sch97]: the abstract syntax definition of the language, the semantic domains, and a collection of valuation functions. The valuation functions provide the meaning of the language, by mapping the abstract syntax to the semantic domains.

4.2.1 KriQL abstract syntax

An excerpt of the abstract syntax of KriQL is given below. A query can be defined on any element of an exploration graph, such as configurations, processes or variables. It is composed

of the kind of element (Key) to be extracted from the graph, and on a set of conditions (Cond) on this element. Conditions can either be test equalities, or test on whether some variable value has changed (CondCh) or some process state has been visited (CondV) or not.

```
Query  ::= Get Key where Cond
         | Query union Query | Query inter Query
Key    ::= Configuration | BE | Process | Component
Cond   ::= ExprBE = ExprBE
         |  ( Cond or Cond )
         |  ( Cond and Cond )
         |  not Cond
         |  CondV
         |  CondCh
CondV  ::= ExprP.State[Identifier] Visited
CondCh ::=  ExprP.State[Identifier] Changed
         |  ExprBE.Variable[Identifier].Value Changed
ExprP  ::= Process[Identifier]
ExprBE ::= ExprP | Component[Identifier]
```

4.2.2 KriQL semantic domains

The meta-model of KriQL as defined in Sec. 4.1 provides a specification for the semantic domains of KriQL. We express an excerpt of them in a conventional notation as follows, where a set (e.g., ConfigurationSet) is expressed as a function, mapping an id (e.g., Configuration identifier) to a data structure (e.g., Configuration). Hence, the access to an element identified by i in a set/function s, is done with the call $s(i)$.

Domain $v \in EnvVariable = Id \rightarrow Value$
Domain $p \in Process = (CurrentState \times EnvVariable)$
Domain $cp \in Component = EnvVariable$
Domain $c \in Configuration = (Process + Component)$
Domain $envC \in EnvConf = Id \rightarrow Configuration$
Domain $t \in Transition = (Id_{confSource} \times Id_{confTarget})$
Domain $envT \in EnvTransition = Id \rightarrow Transition$

Additionally, we defined the *Result* domain to represent the result of queries, which depend on the Key provided by the developer in its query.

Domain $res \in Result = (EnvConf + EnvComponent + EnvProcess)$

4.2.3 KriQL valuation functions

The definition of valuation functions is provided in [ES15]. For simplicity, we provide here only a sketch of it. Given the abstract syntax of a query `Q = Get Key where C`, an example of a valid query is:

Query Q = **Get** `configuration` **where**
proc['Alice'].state=catInYard **and** proc['Bob'].variable['flagB']=up

We define three valuation functions **Q** (query), **C** (predicate), and **E** (expressions inside a predicate, applying on process states or variable values). When applied on a configuration, the valuation function **Q** iterates over a set of configurations (e_c), selecting those satisfying the predicate C : each configuration with identifier i in the set e_c is stored if C applied on it is true, otherwise an undefined value (the $\perp$ symbol) is returned.

$\mathbf{Q}[\![\mathrm{Q}]\!] : \mathit{EnvConf} \rightarrow \mathit{EnvTransition} \rightarrow \mathit{Result}$

$\mathbf{Q}[\![\texttt{Get}\ \text{configuration}\ \texttt{where}\ \mathrm{C}]\!]e_c\ e_t = \text{in}\mathit{EnvConf}(\lambda i.\mathbf{C}[\![\mathrm{C}]\!](e(i)) \rightarrow\ e(i)\ [\!]\ \perp)$

$\mathbf{E}[\![\mathrm{E}]\!] : \mathit{Configuration} \rightarrow \mathit{Value}$

$\mathbf{E}[\![\texttt{proc[Id].state}]\!]c = \mathit{getProcessById}([\![\mathrm{Id}]\!], c)\downarrow_1$ ($\downarrow_1$ *selects the first element of the process)*

$\mathbf{E}[\![\texttt{proc[Id].Id}_{var}]\!]c =$ let $p_v = \mathit{getProcessById}([\![\mathrm{Id}]\!], c)\downarrow_2$ in $p_v([\![\mathrm{Id}_{var}]\!])$

where $\mathit{getProcessById} : (\mathit{Id} \times\ \mathit{Configuration}) \rightarrow\ \mathit{Process}$
$\mathit{getProcessById} = \lambda(i,c).\text{cases}\ \ c(i)\ \mathit{of}\ \text{is}\mathit{Process}(p) \rightarrow\ p\ [\!]\ \mathit{Error}$

$\mathbf{C}[\![\mathrm{C}]\!] : \mathit{Configuration} \rightarrow \mathit{Boolean}$

$\mathbf{C}[\![\texttt{E1 = E2}]\!]c = ([\![\mathrm{E}_1]\!]\ c)\mathit{equals}([\![\mathrm{E}_2]\!]\ c)$

$\mathbf{C}[\![\texttt{C1 and C2}]\!]c = ([\![\mathrm{C}_1]\!]\ c)\mathit{and}([\![\mathrm{C}_2]\!]\ c)$ (*equals* and *or* are the standard boolean operators)

5 Implementation issues

5.1 Implementing a LTS in a database

Each system to be verified is represented by a design and the LTS structure is related with the design structure. The design structure allows the user to understand traces because it provides the user with structural information about the design: process names, variables names, constants and so on. Because we want traces to be stored in a database, the design structure yields the database structure. In the next subsections, we discuss the possible implementations of the design structure.

5.1.1 Relational DBMS

A relational DBMS stores data as table rows conforming to a database schema. Binding the design structure to a database schema can be accomplished in a generic or a specialized way.

A *generic binding* is applied to any design structure in the same manner; one set of tables is suitable for hosting data independently of the design structure. Database structure is issued from the structure of model-checker languages (Fiacre and CDL in our case, see 3.1). For instance, a single table `Process` will host any instance of any process of the SUS. Reference to any particular design construct such a process or a variable name, are provided via parameters as input of any operation.

A *specialized binding* produces a different set of tables upon a specific design structure. Hence the set of operations is dedicated to the design structure instead of being passed via parameters. A specialized binding needs to be generated each time the design of the SUS is updated. For instance, a specialized binding for the motivating example of section 2 uses two tables `AliceProcess` and `BobProcess`, each table hosting all instances of the specified process.

The generic binding will hold all the data in a few tables, while the specific binding will use many smaller tables. Auto-joins and recursive queries will suffer of a bias related to the table size. Hence, despite its conceptual advantage, the generic binding was discarded.

5.1.2 Graph-based database

A graph database stores data as vertices and edges. Because we used the Neo4j system, we will use the Neo4j vocabulary, *nodes* for vertices and *relationships* for edges. A graph database does not have a schema as a relational database has. Neo4j language for querying graph database is called Cypher. Cypher queries find data that matches a specific pattern. A Cypher query anchors one or more parts of a pattern to specific locations in a graph using predicates and then flexes the unanchored parts around to find local matches [RWE13]. Cypher queries are small graphs made from real nodes and relationships. Hence domain modelling in a graph database is isomorphic to graph modelling. According to [RWE13], "*in a graph database what you sketch on the whiteboard is typically what you store in the database.*"

5.1.3 Querying data

Classical graph queries such as shortest path or node reachability depend only on the graph structure. However, recent applications using graph databases require novel queries such as graph pattern matching, keyword search or graph aggregation. In [KWY12], authors state that novel queries raise several challenges: queries integrate both the structure and the attributes of the network; when graphs become complex and large, scalability becomes an issue; due to the lack of fixed schema, it might be infeasible to use conventional SQL or SPARQL framework to answer these queries. Three categories are proposed for novel graph queries: mining queries, matching queries, selection queries [KWY12]. However implementing a LTS in a graph database leads to a special case: the LTS respects an underlying schema that stems from the system and properties models. Hence we divided queries in two categories, those using essentially graph attributes and those using essentially the graph structure.

5.2 Node queries

We call node queries the KriQL operations that process essentially nodes information but might in some cases use transitions information. We classified node queries in three categories:

side-effect free restriction such an operation applies a predicate to a `ConfigurationSet` source and returns nodes where the predicate is `true`.

side-effect restriction such an operation monitors an element over a `ConfigurationSet` source and needs neighbourhood information to process results.

set construction such an operation gathers all values reached by a single or a list of a configuration element.

Relational queries use joins to combine data and dedicated structures such as indexes to improve join performances. Restrictions are typically very efficient in a relational implementation. Set construction operations require auto-join or recursive queries over the same table; it might suffer of poorer performances due to the number of auto-join or related to the tables size.

The concept of query in a graph database is *graph traversal*. A *traversal* is the operation of visiting a set of nodes by moving between nodes connected with relationships. The traversal stops when rules stop apply such as a depth size. Traversals are well-adapted for navigation along a path. When the whole graph needs to be traversed because the operator needs to process all nodes from a certain type (such as needed by a restriction operator), we can expect poorer performances of a graph database vs. a relational one.

5.3 Edge queries

We call edge queries the KriQL operations that process essentially edges information but might in some cases use nodes information. We classified edge queries in three categories:

partition such an operation splits a `Path` in a a sequence of `Paths`.

path existence such a graph traversal operation searches a path between two nodes over a `TransitionSet`.

path computation such a graph traversal operation searches exhaustively all paths between two nodes over a `TransitionSet`.

Edge queries are path walks and will require a massive use of joins in a relational implementation because edges are essentially couples of node identifiers (the source and the destination of the edge) stored in a single `Transition` table. We can expect that the longer the path walk is, the poorer the performance will be because each step along the walk requires a join.

The strength of a graph database is its ability to move between nodes connected with relationships without performance loss whatever the graph size. Thus we can expect excellent performances of a graph database vs. a relational one.

5.4 Benchmark results

We performed benchmark measurements about node and edge queries with 3 typical queries in each case. We tested 2 different implementations: a relational database (Postgres) and a graph database (Neo4J). For the benchmarks we used several LTS ranging from 100 000 to 1 million states. Table 1 synthesizes the results emphasizing the slowest (dark red cells) and the fastest (light green cells) execution times. The *time explosion* cells represents queries that exceeded the allocated execution time. In [BERT15] we detailed these results, focusing on a realistic case-study from the automotive domain.

Unfortunately, no implementation were successful for all test cases and we concluded for the necessity of a blended implementation: a specific implementation for node queries and a graph-based implementation for edge queries.

Table 1: Query performances

	Relational DB	Graph DB
Node queries		
side-effect free restriction		
side-effect restriction		time explosion
set construction		
Edge queries		
partition		
path existence		
path computation	time explosion	

6 Conclusion

A major advantage of model-checking is the production of a counterexample, a trace that provides a detailed witness of how the model violates the property. However, without diagnosis tools, the task is hard to relate the counterexample to its roots cause and progress toward a solution to fix the problem. We presented KriQL a query language over a transition system, and the purpose of this work relies on the research hypothesis that efficient traces query will support better visualization and ease problem interpretation. We need now to perform several usability studies to figure out if and how KriQL features are achieving our objectives.

Bibliography

[BBF+08] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat. Fiacre: an intermediate language for model verification in the TOPCASED environment. In *ERTS 2008*. 2008.

[BERT15] R. Boudaoud, K. Es-Salhi, V. Ribaud, C. Teodorov. KriQL a query language for labelled transition systems. In *IEEE EuroCon 2015*. University of Salamanca, 2015.

[BK08] C. Baier, J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BRV04] B. Berthomieu, P.-O. Ribet, F. Vernadat. The tool TINA - Construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research* 42(14):2741–2756, 2004.

[CES09] E. M. Clarke, E. A. Emerson, J. Sifakis. Model Checking: Algorithmic Verification and Debugging. *Commun. ACM* 52(11):74–84, Nov. 2009.

[DBRL12] P. Dhaussy, F. Boniol, J.-C. Roger, L. Leroux. Improving model checking with context modelling. *Adv. in Software Engineering*, 2012.

[ES15] K. Es-Salhi. Un langage de requete pour un systeme de transitions. Technical report, Universit de Rennes I, France, 2015.

[GOA05] S. F. Goldsmith, R. O'Callahan, A. Aiken. Relational Queries over Program Traces. In *Proc. of the 20th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05, pp. 385–402. ACM, NY, USA, 2005.

[GSB07] A. Griesmayer, S. Staber, R. Bloem. Automated Fault Localization for C Programs. *Electronic Notes in Theoretical Computer Science* 174(4):95 – 111, 2007. Proceedings of the Workshop on Verification and Debugging.

[Hol97] G. Holzmann. The Model Checker SPIN. *Software Engineering* 23(5):279–295, 1997.

[HS12] M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., 2012.

[KWY12] A. Khan, Y. Wu, X. Yan. Emerging Graph Queries in Linked Data. In *Proc. of the 2012 IEEE 28th Int. Conf. on Data Engineering*. ICDE '12, pp. 1218–1221. IEEE Computer Society, Washington, DC, USA, 2012.

[Lam84] L. Lamport. 1983 Invited Address: Solved Problems, Unsolved Problems and Non-problems in Concurrency. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*. Pp. 1–11. ACM, 1984.

[LPY97] K. G. Larsen, P. Pettersson, W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer* 1:134–152, 1997.

[MLL05] M. Martin, B. Livshits, M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proc. of the 20th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05, pp. 365–383. ACM, NY, USA, 2005.

[PTP07] G. Pothier, E. Tanter, J. Piquer. Scalable Omniscient Debugging. *SIGPLAN Not.* 42(10):535–552, 2007.

[RWE13] I. Robinson, J. Webber, E. Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.

[Sch97] D. A. Schmidt. *Denotational Semantics - A Methodology for Language Development)*. David A. Schmidt, 1997.

[Var07] M. Y. Vardi. Automata-theoretic Model Checking Revisited. In *Proceedings of the 8th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*. Pp. 137–150. Springer-Verlag, 2007.

[YLW+06] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, W.-Y. Ma. Automated Known Problem Diagnosis with Event Traces. In *Proc. of the 1st ACM SIGOPS European Conf. on Computer Systems*. Pp. 375–388. ACM, NY, USA, 2006.