Distributed Runtime Verification of Cyber-Physical Systems Based on Graph Pattern Matching

Gábor Szilágyi[1], András Vörös[1,2]
[1]Budapest University of Technology and Economics,
Department of Measurement and Information Systems, Budapest, Hungary
[2]MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary
Email: szilagyi@db.bme.hu, vori@mit.bme.hu

***Abstract*—Cyber-physical systems process a huge amount of data coming from sensors and other information sources and they often have to provide real-time feedback and reaction. Cyber-physical systems are often critical, which means that their failure can lead to serious injuries or even loss of human lives. Ensuring correctness is an important issue, however traditional design-time verification approaches can not be applied due to the complex interaction with the changing environment, the distributed behavior and the intelligent/autonomous solutions.**

In this paper we present a framework for distributed runtime verification of cyber-physical systems including the solution for executing queries on a distributed model stored on multiple nodes.

I. Introduction

The rapid development of technology leads to the rise of cyber-physical systems (CPS) even in the field of safety critical systems like railway, robot and self-driving car systems. Cyber-physical systems process a huge amount of data coming from sensors and other information sources and it often has to provide real-time feedback and reaction.

Cyber-physical systems are often critical, which means that their failure can lead to serious damages or injuries. Ensuring correctness is an important issue, however traditional design-time verification approaches can not be applied due to the complex interaction with the environment, the distributed behavior and the intelligent controller solutions. These characteristics of CPS result many complex behavior, huge or even infinite number of possible states, so design-time verification is infeasible.

There are plenty of approaches for monitoring requirements [6]. Runtime analysis provides a solution where graph-based specification languages and analysis algorithms are the proper means to analyze the behavior of cyber-physical systems at runtime.

In this paper a distributed runtime verification framework is presented. It is capable of analyzing the correctness of cyber-physical systems and examining the local behavior of the components. An open-source graph query engine being able to store a model in a single machine served as a base of the work [4]. It was extended to support distributed storage and querying: in case of complex specifications, the algorithm collects the information from the various analysis components and infers the state of the system. The introduced framework was evaluated in a research project of the department and proved its usefulness.

Figure 1 shows the basic approach to runtime verification. System development is started by specifying the requirements for the system. Then it is designed, according to the specification. From the specification and the system design, a monitor is created for observing the environment. The monitoring component stores the gathered information in a live model which is updated continuously to represent the actual state of the system. The runtime requirements can be evaluated on the live model and the solution can find if a requirement is violated. Various monitoring approaches exist, some observes data dependent behavior, others can analyze temporal behavior. In this paper the focus is on the runtime analysis of data dependent behavior which can be captured by a graph based representation.

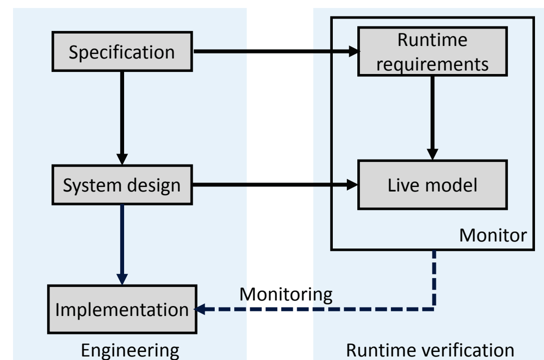

Fig. 1. Model-based runtime verification of a cyber-physical system.

II. Graphs as abstractions

To verify cyber-physical systems, we need to have information about its operation context. Various kinds of information might belong to the context such as the physical environment, computational units, configuration settings or other domain specific information. In modern cyber-physical systems, sensors provide a huge amount of data to be processed by the monitors, it is important to have a comprehensive image of the operation context which can be supported by graph-based knowledge representations.

The current snapshot of the system and its operational context can be formally captured as a *live model* which continuously gets updated to reflect relevant changes in the underlying real system [3]. This live model serves as an abstraction of the analyzed system. The framework uses graph representation to model the actual state of the system. These are directed, typed and attributed graphs. Their vertex types, edge types, and other constraints must be specified in a *metamodel*. The metamodel is also needed for the formalization of specification, since it also specifies the possible structure of the live model.

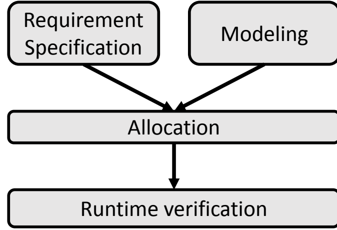

Fig. 2. The presented approach for runtime verification.

The steps of our approach to graph based runtime verification are illustrated on Figure 2. First we need to describe the metamodel which captures the domain information of the monitored system. According to the metamodel and the initial state of the system, a live model is created. This live model is used during the runtime analysis. Then requirements can be defined. After modeling, the system engineer shall specify the allocation i.e. how the elements of the live model are allocated to the computational units of the distributed system. After these tasks, the framework is able to generate the code for runtime verification of the system.

We illustrate this approach with an example of a simplified version of a train control system. First the metamodel shall be created for the system. (Figure 3). In our case, the model is composed of two types of elements: *Segment* and *Train*. Segments next to each other in the physical configuration are connected with *connectedTo* edges in the model. If a train is on a segment, the model represents it with the *onSegment* edge. An example live model of the system can be seen on Figure 4.

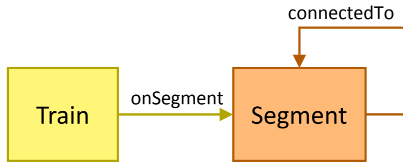

Fig. 3. The metamodel for the system

Safety requirements of the system can be described using graph patterns. A graph pattern is given by

1) a list of variables, each representing a vertex of the live model with a given type
2) a set of constraints, which must be satisfied by the variables, to match the pattern

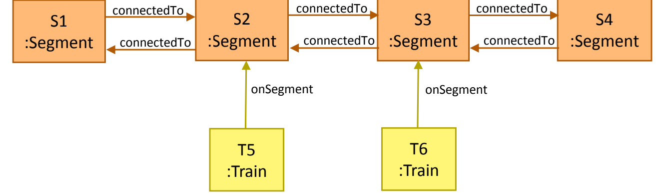

Fig. 4. An example live model for a train control system

Graph patterns in the framework are defined using the VIATRA Query Language (VQL) [1]. It has a rich expressive power capable of expressing constraints like:

- path expression – a specific reference, an attribute, or a path of references must exist between two variables.
- attribute equality – an attribute of a vertex must be a given value
- matching to a pattern – a list of given vertices must match to a pattern
- negative pattern matching – a list of given vertices must not match to another pattern
- check expression - an arbitrary expression containing attributes must be evaluated true

Graph patterns expressed as VQL expressions are evaluated on the input models. Graph pattern matching is reduced to a search for isomorphic subgraphs in the input model. The structure of the graph pattern yields the constraints during the search: the parameters of the graph pattern will finally be assigned to the corresponding graph nodes.

For example, if we want to find trains on adjacent segments, we can use the following pattern (given in VQL):

```
pattern NeighboringTrain(TA, TB)  // 1
{
  Train(TA);  // 2  TA is a train
  Train(TB);  // 3  TB is a train
  Train.currentlyOn(TA, SA);  // 4  TA is currently on SA
  Segment.connectedTo(SA, SB); // 5 SA is connected to SB
  Train.currentlyOn(TB, SB); // 6  TB is currently on SB
}
```

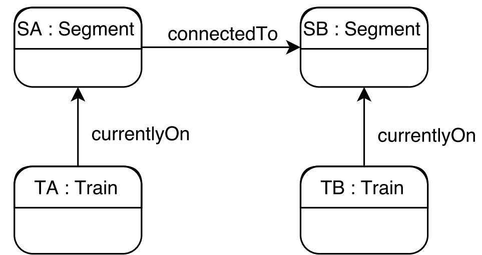

Fig. 5. Graphical visualization of the query.

The pattern's header (1) specifies its name and its parameters. Every statement in the body of the pattern is a constraint (2–6) for variables (SA, SB) and parameters (TA, TB). The visualized version of this pattern can be seen on Figure 5.

In the example model (Figure 4) there are 2 matches of this pattern. One is $\{TA = T5, TB = T6, SA = S2, SB = S3\}$ and the other is $\{TA = T6, TB = T5, SA = S3, SB = S2\}$.

After the requirements are specified, the user has to decompose the model and allocate it into computational units (see Section III). We call this the *allocation* of the live model. The computational units, the live model, and its allocation can be given in JSON format:

```
{
  "nodes" : [
    {
      "name" : "nodeA",
      "ip" : "127.0.0.1",
      "port" : 54321
    }
    ,
    ...
  ],

  "model" : [
    {":id": 0, ":node": "nodeA", ":type": "Segment",
      "connectedTo" :    [1] },
    {":id": 1, ":node": "nodeA", ":type": "Segment",
      "connectedTo" : [0, 2] },
    {":id": 2, ":node": "nodeB", ":type": "Segment",
      "connectedTo" : [1, 3] },
    ...
  ]
}
```

The allocation of a model element can be given by the *":node"* attribute. Model elements, like trains still must be assigned to a specific computational unit, although its physical place can change in time.

After the model elements are allocated to the computational units, and the framework generated the necessary artifacts, runtime verification can be started.

It works in a way depicted on Figure 6. The live model is continuously updated with the runtime information coming from sensors. Runtime requirements of the system – formalized as graph patterns – are verified on the live model continuously, as it is described in the next section, to ensure the system's correct operation.

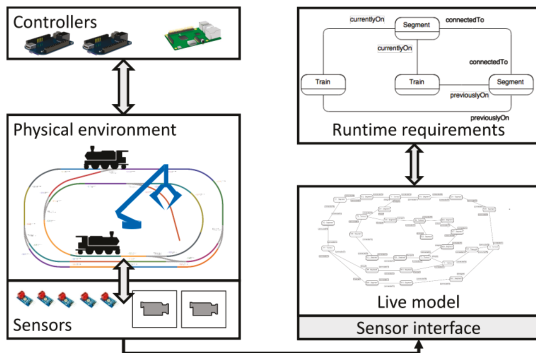

Fig. 6. Runtime verification of the system

III. Distributed Graph Queries

The distributed nature of cyber-physical systems makes runtime verification a challenging task. Various approaches exist regarding the model and query management. The main difference is the way they gather and process the information and evaluate the requirements:

- Centralized model and query management. It would require the sensor information to be transmitted to a central processing machine.
- Distributing the model to each computational unit. It would require model synchronization between nodes.
- Dividing the live model and the query processing tasks to the computational units.

Centralized approaches are not always viable due to various reasons, like the central machine can be easily overloaded, it can be a single point of failure (SPOF), which is undesirable in safety-critical systems. In the second case, model synchronization can introduce unwanted complexity, and overhead in network communication. We solve these problems by processing the sensor information on the corresponding computational units, and updating the local part of a distributed live model.

A. Distributing the storage of the model

After the metamodel is specified, which describes the types of vertices and edges, etc., an initial live model shall be created, representing the initial state of the system. As parts of the model are stored on different computational units, each vertex of the global model must be assigned to a given computational unit. References are stored where the source object for that reference is stored. Basically, the reference can only refer to a local object, i.e. a vertex assigned to the same computational unit. If the reference's destination vertex is not assigned to the same computational unit, we create a proxy object on the same computational unit. Vertices are identified with a globally unique identifier, which is portable between the computational units.

B. Distributed Query Evaluation Algorithm

The algorithm is based on the so-called local search algorithm [2]. To find matches of a given graph pattern, we start from a frame, i.e. a list of variables, unassigned at first. After that, we execute a given list of search operations (called search plan) being specific to the pattern.

To make the algorithm working in distributed systems, we examined the search operations that cannot be executed locally. There are basically two operations, that need to be handled differently from the single machine solution:

- Iterating over the instances of a given vertex type cannot be done locally, since there can be instances for that type on any of the computational units.
- Assigning a variable through a given reference cannot be done, if the source object is not present on the node.

At these operations we inserted a „virtual" search operation. It doesn't operate on the frame, but transmits the query execution to the other computational units of the system. To iterate over instances, first the query execution is distributed between units by the virtual operation, and after that, iterating over local instances can be done. In case of assigning variable via a reference the virtual search operation checks, whether the source object is present on the computational unit, then transmits it to the other units if the source object is not available.

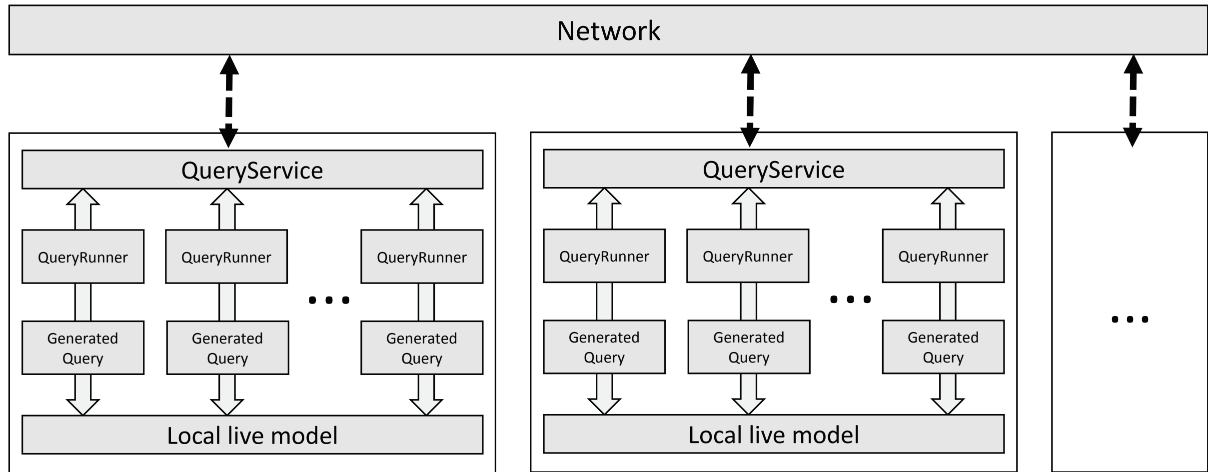

Fig. 7. Architecture of the distributed query evaluation.

C. Architecture

The architectural overview of the distributed query engine is depicted on Figure 7.

On every computational unit of the distributed system, a *QueryRunner* is set up for each generated query. Their role is to execute query tasks specific to their *Generated Query*, on the given local part of the model. An input for a query task consists of 1) a frame, containing the assigned variables, i.e. partial match, and 2) the index of the next search operation to be executed.

If an operation needs distributed execution, the QueryRunner uses the *QueryService* of the computational unit, which handles network communication and reach other computational unit. To serialize the data between different nodes, we used Protocol Buffers [5].

IV. Evaluation

The query evaluation time of the framework was measured in several configuration with the example railway control system, that was presented before, but with a more complex live model, containing 6000 elements. We split the model of the railway system into 2, 3, and 4 parts. First we ran the example query on each configuration, but every computational unit was run on the same machine. So practically, network communication had no overhead during the measurement (Figure 8).

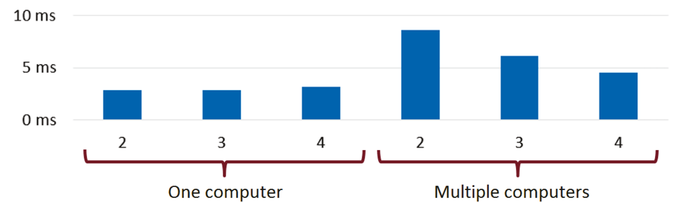

Fig. 8. Average time of query execution by computational units

After that, every computational unit of the system was run on different machines. This shows how network communication affects the speed of our implementation. We can conclude, that networking introduces overhead, but using more computational units makes the system's performance closer to single machine solution. The integration of sensor information in cyber-physical systems cause additional overhead, that can be prevented using the distributed solution.

V. Conclusion

In this paper, we presented a framework for distributed runtime verification of cyber-physical systems based on graph queries. Our approach represents the gathered information in a distributed live model and evaluates the queries as close to the informations sources as possible. A method for distributed model storage and query execution is developed based on a widely used search algorithm. In the future we plan to integrate incremental graph query algorithms to further improve the efficiency of the framework.

References

[1] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. VIATRA 3: A reactive model transformation platform. In *Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Proceedings*, pages 101–110, 2015.

[2] M. Búr. A general purpose local search-based pattern matching framework. masters thesis. 2015.

[3] I. Dávid, I. Ráth, and D. Varró. Foundations for streaming model transformations by complex event processing. *Software & Systems Modeling*, 2016.

[4] R. Dóczi. Search-based query evaluation over object hierarchies. Master's thesis, Budapest University of Technology and Economics, 2016.

[5] Google. Protocol buffers – data interchange format. https://github.com/google/protobuf.

[6] M. Vierhauser, R. Rabiser, and P. Grünbacher. Requirements monitoring frameworks: A systematic review. *Information & Software Technology*, 80:89–109, 2016.