Búr, M., & Varró, D. (2019, September). Towards WCET Estimation of Graph Queries@ Run. time. In 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS) (pp. 233-238). IEEE. DOI: https://doi.org/10.1109/MODELS.2019.00007

Towards WCET Estimation of Graph Queries@Run.time

Márton Búr[*] and Dániel Varró[*][†]
[*]McGill University, Montréal, Canada
[†]MTA-BME Lendület Research Group on Cyber-Physical Systems, Budapest, Hungary
Email: marton.bur@mail.mcgill.ca, daniel.varro@mcgill.ca

Abstract—**Recent approaches in runtime monitoring and live data analytics have started to use expressive graph queries at runtime to capture and observe properties of interest at a high level of abstraction. However, in a critical context, such applications often require timeliness guarantees, which have not been investigated yet for query-based solutions due to limitations of existing static worst-case execution time (WCET) analysis techniques. One limitation is the lack of support for dynamic memory allocation, which is required by the dynamically evolving runtime models on which the queries are evaluated. Another open challenge is to compute WCET for asynchronously communicating programs such as distributed monitors. This paper introduces our vision about how to assess such timeliness properties and how to provide tight WCET estimates for query execution at runtime over a dynamic model. Furthermore, we present an initial solution that combines state-of-the-art parametric WCET estimations with model statistics and search plans of queries.**

I. Introduction

A. Motivation and Problem Statement

In the past decades, model-based systems engineering has been used in many traditional safety-critical (SC) applications such as cars or aircrafts. However, modern cyber-physical systems (CPS), like self-driving cars and autonomous robots, pose several new challenges as they need to interact with a continuously evolving environment over a heterogeneous computing platform while still complying with safety regulations.

In traditional SC systems, models, queries and transformations offer great expressive power in modeling tools (like Capella, Papyrus, Artop), but their use has been restricted to design-time, i.e., no model queries or transformations run on an aircraft at runtime. A main reason for this is that any piece of software executed at runtime in a SC system needs to satisfy various extra-functional requirements to ensure deterministic, predictable behavior. For example, the calculation of worst-case execution time (WCET) or schedulability analysis is compulsory for real-time SC software deployed at runtime.

While the models@run.time initiative [1] has been promoting the use of models, queries and transformations at runtime with major recent advances [2]–[5], *existing approaches provide no timeliness guarantees* required for any critical applications. For example, while [3] claims "near real-time" for online data analytics, a timely response is not guaranteed. Similarly, the query-based runtime monitoring approach [2] is unable to ensure that a query will justifiably complete on time even in the absence of communication errors.

Such lack of guarantees is hardly surprising. On the one hand, traditional real-time SC systems have been able to compute such timeliness guarantees like WCET, but the deployed software uses static memory allocation and a priori bounded, low-level data structures - none of which provides sufficient flexibility for modern autonomous or self-adaptive CPS. Furthermore, fixed worst-case bounds obtained from analyzing the computational complexity of an algorithm largely overestimates execution time, thus they are impractical.

B. Challenges and Contributions

In this paper, we present the high-level research challenge for assessing timeliness properties for model queries used at runtime (referred to as *queries@run.time*). In particular, we present a research agenda to *assess worst case execution time for graph-based query techniques* used at runtime over dynamically evolving graph-like runtime data and a heterogeneous computing platform with resource constraints that is characteristic to many critical CPS applications. Existing applications of such queries@run.time already include online analytics [3] or runtime monitoring [2].

Moreover, we present initial results on *how to compute high-level (architecture-independent) WCET [6] for graph queries executed on a single platform node*, which is already a major research challenge, and a major cornerstone of the long-term research agenda. Since the dynamic growth of data prevents global WCET estimates, our key idea is to provide WCET guarantees relative to the size of the model (i.e., the number of model objects and links between them) by exploiting model statistics. As such, we can use existing tools for static WCET estimation until the size of the model exceeds a certain limit, or the structural characteristics of the model change significantly.

Up to our best knowledge, our proposal is the first attempt to investigate timeliness properties of *graph-based query techniques used at runtime* in a real-time CPS setting.

II. Motivation

Since many IoT and CPS applications rely upon graph-like knowledge representation, our concepts are generally applicable to different domains. Nevertheless, we illustrate these concepts in the context of runtime monitoring in an open-source CPS educational demonstrator of a model railway network, which is a representative application of using graph models and graph queries at runtime [2], [3].

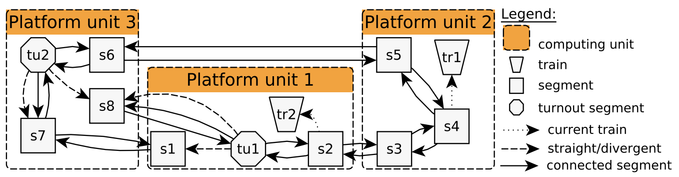


Fig. 1. Distributed runtime model for an open CPS demonstrator [2]

A. Runtime Monitoring by Graph Queries

Recently, various expressive rule or query-based specification languages have been proposed [2], [7], [8] to specify monitoring goals and to drive monitor execution over a continuously changing runtime graph model following the models@run.time principle [1]. For example, safety monitors captured by graph queries aim to automatically prevent trains from collision and derailment in a model railway network reported in [2]. The railway track is equipped with various sensors (cameras, shunt detectors) capable of sensing trains on a particular segment of the track, and these sensors are connected to heterogeneous embedded platform units (e.g. *Arduinos*, *Raspberry Pis*, *BeagleBoards*). These units also control actuators that stop trains to guarantee safe operation.

We assume that runtime data is captured by *runtime graph model* $G = (N, E)$ where each node $n_i \in N$ is deployed on some platform unit potentially in a distributed way. Changes in the system are reflected by (periodically) updating the runtime model based on sensor reads. As such, one obtains a high-level and dynamically evolving data model, which offers increased flexibility for modern CPS applications compared to pre-allocated data buffers in a traditional real-time programs.

Example 1: The runtime graph model shown in Figure 1 captures the knowledge base of the three platform units (Unit 1 – Unit 3), the domain elements (s1–s8, tu1, tu2, tr1, and tr2) as well as the links between them. Each platform unit hosts model elements contained within them in the figure, e.g. Platform unit 2 is responsible for storing attributes and outgoing references of objects, i.e., hosts s3, s4, s5, and tr1.

Runtime monitoring programs are deployed to the same physical platform. We assume that the (safety) properties of interest are captured by high-level graph queries as in [2]. Formally, a *graph query* $\varphi(v_1, \ldots, v_n)$ is a predicate that consists of a conjunction of constraints expressed over relational logic, which is more expressive than using low-level statemachines or temporal logic formulae used in monitors of traditional SC systems. Related languages provide a baseline for other complex event processing or trace analysis techniques [8]–[10] defined over graph models.

The execution of runtime monitors can potentially be *hierarchical* and *distributed*. Monitors may observe the local runtime model hosted by a platform unit, and they can collect information from runtime models hosted by different platform units. Moreover, one monitor may request information from other monitors, thus yielding a hierarchical monitoring network.

Example 2: Railway safety standards prescribe a minimum distance between trains on track [11], [12]. The closeTrains monitor definition captures a (simplified) description of the minimum headway distance to identify violating situations where trains have only limited space between each other. A runtime monitor needs to detect if there are two different trains on two segments, which are connected by a third segment. Any match of this graph pattern highlights track elements where passing trains need to be stopped immediately. Listing 1 presents the monitoring query closeTrains as a logic formula.

Listing 1. The closeTrains monitoring goal as a formula

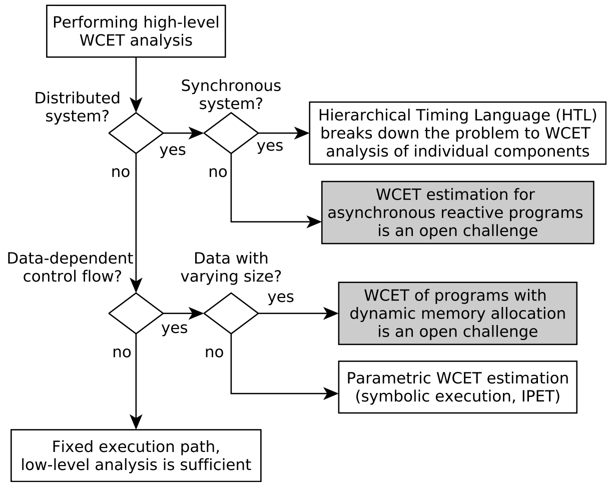
$\texttt{CloseTrains}(start, end) =$
(1) $\quad \texttt{Segment}(start) \wedge$
(3) $\quad \exists middle : \texttt{ConnectedSegment}(start, middle) \wedge$
(4) $\quad \texttt{ConnectedSegment}(middle, end) \wedge$
(2) $\quad \exists train_1 : \texttt{CurrentTrain}(start, train_1) \wedge$
(5) $\quad \exists train_2 : \texttt{CurrentTrain}(end, train_2) \wedge$
(6) $\quad \neg(train_1 = train_2)$

B. Local Search-based Graph Pattern Matching

Efficient graph query evaluation has decades of research results frequently categorized into *local search-based* [13] or *incremental* [14] approaches. As query-driven runtime monitors are deployed over a physical computing platform with resource constraints (e.g. CPU, memory), the increased memory usage of incremental approaches may be a critical factor. Therefore, we assume that real-time query-based programs follow a local search-based pattern matching approach (as in [2]). *Distributed* graph query evaluation over fragmented data was first presented in [15] while further algorithms were reported in [16]–[18]. In [19], a distributed incremental graph query layer was adapted and deployed to a cloud infrastructure.

For local search, monitors compute matches of a graph query $\varphi(v_1, \ldots, v_n)$ along a *search plan* by assigning model objects to variables $v_1, \ldots, v_n$ and evaluating the predicate of the query. A search plan is an ordered list of search operations (e.g., checking type of objects, navigating along references) that traverses the runtime graph model in order to find all complete variable bindings satisfying the query condition.

Example 3: For the example query presented in Listing 1, a single search operation is created for each predicate in the `CloseTrains` query formula. Such an operation *either* binds variables to model objects *or* checks if the current variable binding satisfies the corresponding predicate. The operation ordering is shown at the beginning of each line. Assuming that none of the query parameters are assigned an initial value, i.e., all matches over the complete (runtime) model are requested, a possible search plan is the following: (1) substitute variable *start* with a model object of type `Segment`, (2) navigate along the reference of type `CurrentTrain` from *start* and find a potential substitution for $train_1$, (3) then navigate from *start* along a reference of type `ConnectedSegment` and find a potential substitution for *middle*, (4) repeat navigation along a `ConnectedSegment` link from *middle* to assign a model object to variable *end*, (5) find a possible substitution



Performing high-level WCET analysis
Distributed system?
Synchronous system?
yes
yes
Hierarchical Timing Language (HTL) breaks down the problem to WCET analysis of individual components
no
no
WCET estimation for asynchronous reactive programs is an open challenge
Data-dependent control flow?
Data with varying size?
yes
WCET of programs with dynamic memory allocation is an open challenge
yes
no
no
Parametric WCET estimation (symbolic execution, IPET)
Fixed execution path, low-level analysis is sufficient

Fig. 2. Summary of high-level (platform independent) static analysis techniques for computing WCET

for $train_2$ by navigating from *end* via `CurrentTrain`, and finally, (6) ensure that the model objects substituted to $train_1$ and $train_2$ are not the same. Once search step 6 is successfully completed, a match is found and registered, and the execution continues until all matches in the model are discovered.

III. Related Work and Open Challenges

We overview the applicability of existing *static and high-level timing analysis* approaches for queries@run.time.

a) Timing analysis techniques: Methods for efficiently analyzing timing properties and computing precise WCET bounds of programs have been actively researched since real-time systems appeared. Detailed surveys on such methods are available, for example, in [20] and [21]. Static WCET analysis techniques have two major categories [21].

- *High-level analysis* works with the abstract flow of a program, mainly using the control flow graph or the control flow automata obtained from the source or machine code. Such algorithms are presented in [6], [22], [23].
- *Low-level analysis* techniques [24]–[26] focus on platform-specific details (e.g., memory, caches, pipelines, and branch prediction) when assessing timing properties.

Various high-level, static WCET analysis techniques have been developed to provide safe timing estimates for the execution of various types of programs. Figure 2 summarizes the most popular methods used for WCET analysis. The implicit path enumeration technique (IPET) [27] is commonly used for this purpose. This technique analyzes the program paths (control flow) to determine what sequence of instructions will execute in the extreme case. The path analysis in IPET is based on solving and integer linear programming problem.

Moreover, initial WCET analysis support has been provided for distributed systems in [28] by breaking down the problem into WCET analysis of communicating components using the *Hierarchical Timing Language* (HTL). HTL organizes tasks into a hierarchical, tree-like structure that has the complete program as its root. All tasks are executed periodically, and dedicated communicator processes coordinate the exchange of information between the ones running on different hosts.

b) Runtime monitoring in resource-constrained environments: The tool polyLarva [29] provides means to adjust the possible overhead imposed by the runtime checks performed during monitoring. The Brace framework [30] supports monitoring in distributed resource-constrained environments by incorporating dedicated units in the system to support global evaluation of monitoring goals. LTL formulae are evaluated in a fully distributed manner in [31] for components communicating on a synchronous bus in a real-time system. Additionally, machine learning-based solution for scalable fault detection and diagnosis system is presented in [32] that builds on correlation between observable system properties.

c) Real-time database systems: Real-time database systems (RTDBS) [33] provide database support for applications where time-constrained data access and temporal data validity are required. A survey on such systems is available in [34].

Major limitations. We have identified the following major limitations in the state-of-the-art in the context of queries-@run.time (highlighted also in Figure 2).

- Existing resource constraint (CPU, memory) guarantees for runtime monitoring are only provided when the *expressiveness of the property language is limited*, i.e., are available for monitoring techniques relying on some form of finite automata or temporal logic expressions. Modern graph query languages can be more expressive than that.
- Currently, static WCET analysis requires that the program is *free from dynamic memory allocation*, which sets a theoretical bound on the available memory for program data during design time. This is a known limitation of WCET calculation as also pointed out by [35].
- Real-time databases support a relational data model, but not graph data. They are *not suited for deployment over a distributed, decentralized platform* where platform units have *resource limitations*. Furthermore, they do not support hard real-time applications.
- Estimates of query execution time for join operations based on computational complexity [36] do not provide tight WCET bounds, thus they are impractical.

Altogether, these limitations prevent the direct use of existing techniques for many CPS or IoT applications, and lead to several open challenges.

d) Open challenges: To support queries@run.time in CPSs, we need to address at least the following questions:

Q1: How to predict WCET of queries over dynamically evolving graph data deployed on a single platform unit?
Q2: How to predict WCET for distributed queries executed over a distributed runtime platform?
Q3: How to predict other non-functional properties of queries@run.time such as memory use, performability, mean execution time for real-time systems?

This paper provides a research agenda to address challenges Q1 and Q2 and some initial ideas to solve challenge Q1. As such, Q2 and Q3 remain open long-term challenges.

IV. Overview of Research Agenda and Key Ideas

We outline the following research agenda for Q1 and Q2:

a) Query WCET problem (Q-WCET): is to determine WCET bounds of a given query over evolving graph models deployed on a single platform unit. The input parameters of this problem are the graph model and the graph query itself. *Idea*: Limitations of existing timing analysis approaches indicate that calculating a single WCET for dynamically evolving graph data may not be feasible. Instead, we propose a parametric WCET calculation where WCET bounds for queries may depend on the model (i.e., WCET will likely increase as the model grows). At design time, we rely on information originating from the query search plan, and we reuse existing WCET techniques and tools to compute a parametric WCET formula [37] using the control flow graph derived after code generation. Then, at runtime, this parametric WCET formula is substituted with relevant parameters obtained from runtime model statistics [13]. An example is given in Section V.

b) Distributed query WCET problem (DQ-WCET): generalizes the Q-WCET problem by allowing the graph model and query to be distributed over a heterogeneous platform. Here an extra input parameter (wrt. the Q-WCET problem) is the actual allocation of the model fragments to the different units of the platform, which results in significant increase in evaluation time caused by network latency when different platform units need to communicate. Furthermore, asynchronous, reactive programs are performing the monitoring task, for which WCET estimation is still an open problem.
Idea: On the one hand, as a partial solution to this problem, we envision a query-specific refinement of the parametric WCET computation that assigns different cost values for local and remote invocations. An initial cost estimator can consider the quality of service guarantees for message delivery deadlines provided by the communication middleware and the maximum number of links in the graph model that connect objects allocated to different platform units for each reference type to get a safe upper bound for the number of required interactions over the network. On the other hand, further investigation is needed to enable WCET calculation for asynchronously communicating programs, which is a grand challenge in itself.

c) Distributed runtime-defined query WCET problem (DRQ-WCET): enables to add query definitions at runtime. In current queries@run.time solutions [2], this is not yet possible since sources are generated and compiled at design time.
Idea: One way to address this limitation is to create a flexible (interpreter-based or on-the-fly code generation) query framework, where queries can be defined and added runtime. In this case, one needs to ensure that the instructions in the query plan are assigned safe WCET estimates, thus both a parametric WCET formula and its value can be determined at runtime.

V. An Approach for Estimating WCET for Queries@Run.time

We introduce a general workflow (Figure 3) for computing WCET for queries@run.time executed *on a single platform unit* that involves both design time and runtime tasks.

a) From graph queries to source code: The process takes the definition of a *graph query* as input. Then, a query-specific *search plan is computed*, which serves as the input for *code generation*. The code generator produces *C source code* that is ready to be compiled for a target architecture and platform. Tools like Viatra [14] and eMoflon [38] provide required modeling and code generation features from query definitions.

Example 4: For the closeTrains graph query along with its search plan shown in Listing 1, a pseudocode of the generated source code implementing the search plan is presented in Algorithm 1 as a combination of loop (`for`) and branch (`if`) statements. In line 1 (shortly, L1) the set *matches* serves as the container of the results and it is initialized as an empty set. The search algorithm is shown in L2-L11. It starts with a loop that binds objects of type Segment to variable *start*. L3 and L4 show navigation along the currentTrain reference from *start*. If an object of type Train is present on *start*, the search continues by navigating twice along the connectedSegment reference in each possible way, again, from *start* and binding variable *end* (L5-L7). In L8-L9, the presence of a train is checked on *end*. Finally, if $train_1$ and $train_2$ are bound to different objects (L10), a match $\langle start, end \rangle$ is added to *matches* in L11.

Algorithm 1 Compute results of `closeTrains`

```
1:  $matches \leftarrow \emptyset$
2:  for $start$ in {all Segment instances} do
3:      $train_1 \leftarrow$ getCurrentTrain($start$)
4:      if $train_1 \neq$ NULL then
5:          for $middle$ in getConnectedSegments($start$) do
6:              for $end$ in getConnectedSegments($middle$) do
7:                  if $start \neq end$ then
8:                      $train_2 \leftarrow$ getCurrentTrain($end$)
9:                      if $train_2 \neq$ NULL then
10:                         if $train_1 \neq train_2$ then
11:                             $matches$.add($\langle start, end \rangle$)
```

b) Program analysis: Next, a *control flow graph (CFG) is built* from the source code that serves as the input for high-level *flow analysis* using existing techniques like IPET [27].

Such a CFG consists of nodes representing variable assignments, if-checks, and for loops, which are embedded into each other, yielding a structured program composition that facilitates analysis [39]. Edges in the CFG represent potential continuations in execution (e.g. true/false branches for an if-statement, internal or exit paths for a loop).

In addition, the source code also serves as the input for certified or WCET-aware *compilers* (like CompCert or WCC for the C language) that produce an *executable* ready for *low-level WCET analysis*. This low-level WCET analysis is out of scope for the current paper and left for future investigations.

WCET analysis on the source or machine code can be done by using e.g. SWEET [40], OTAWA [41], or aiT [42].

Example 5: A CFG derived from Algorithm 1 is depicted in Figure 4a where lines of the matching algorithm (lines L2-L11) are represented by a CFG node. Loop nodes are created

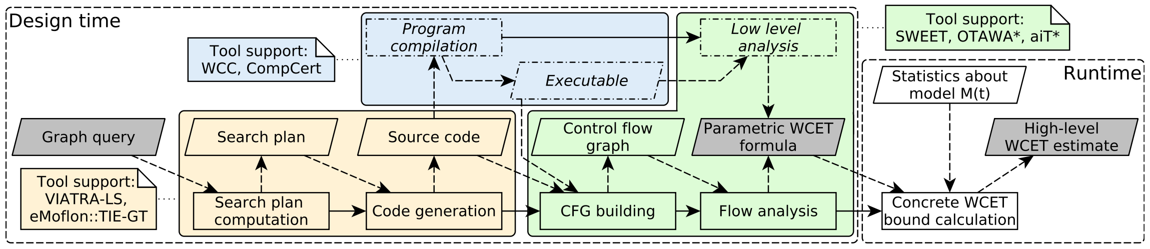


Fig. 3. Overview of high-level WCET computation for queries@run.time

from for loops in L2, L5 and L6 (depicted as hexagons), if-statements in L4, L7, L9, and L10 are added as decision nodes (diamonds), and variable assignments in L3, L8, and L11 are represented with basic blocks (rectangles). Furthermore, edges in the graph represent potential control flow.

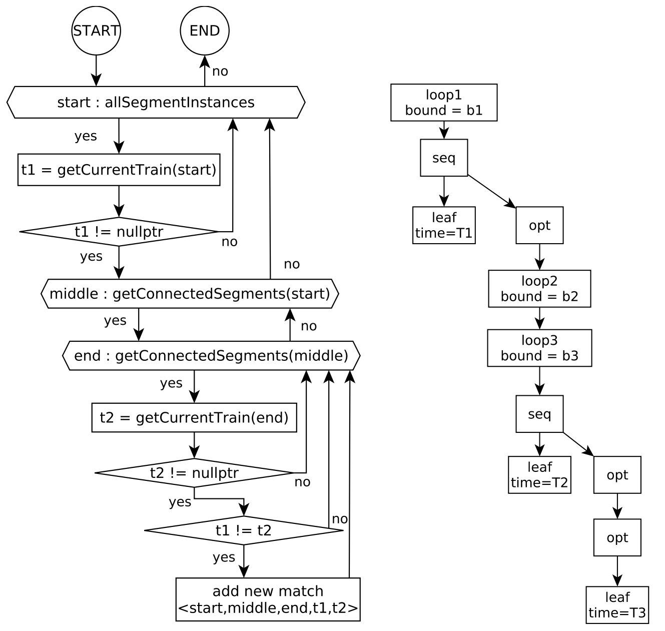
c) Parametric WCET computation: As the next step, the result of CFG analysis needs to be combined with the platform-specific instruction timing properties obtained from low-level WCET analysis to create a parametric WCET formula where the parameters are the data-dependent loop bounds which are unknown at design time. Up to our best knowledge, parametric WCET computation has only been prototyped in SWEET, which highlights the lack of mature tool support.

In this paper, we use a parametric WCET computation approach [43] that relies on the *expression tree* obtained from the CFG instead of executing a computationally expensive IPET [27]. An expression tree is built by using *loop*, *leaf*, *seq*, and *opt* nodes. A *loop* has a single instruction as its body and it is supplied with a symbolic loop bound b_i encoding the number of iterations that the loop is executed. A *leaf* node represents a set of simple instructions and has a parameter T_i that represents its WCET required for completion. A *seq* node has a sequence of nodes as its children, while *opt* nodes denote statements that are executed if certain conditions hold.

We assume that T_i intervals have a fixed upper bound, however, the method presented in [43] allows supplying context sensitive information (i.e., varying execution times) when modeling execution times of instruction blocks within loops.

Example 6: A sample expression tree is depicted in Figure 4b. Symbolic loop bounds $b_1 - b_3$ are supplied for *loop1* $-$ *loop3*, respectively, while other instructions are modeled as single basic blocks with assumed context independent, constant execution times T_i during this high-level WCET analysis. The parametric WCET formula computed from the expression tree is $\omega(b_1, b_2, b_3) = b_1 \cdot (T_1 + b_2 \cdot b_3 \cdot (T_2 + T_3))$.

d) Runtime WCET bounds: At runtime, we extract the required parameters from the runtime model and substitute the respective values into the parametric WCET formula (e.g., for loop bounds) in order to compute concrete WCET bounds. For that purpose, we adapt ideas from model-specific search plans [13], [44], [45] which rely on model statistics to derive efficient search plans for query execution. These runtime statistics include the number of instances of each object and reference type (denoted as O_i and R_j), and the average degrees of outgoing references ($\frac{R_j}{O_i}$) and are continuously maintained


START
END
no
start : allSegmentInstances
yes
t1 = getCurrentTrain(start)
t1 != nullptr
no
yes
no
middle : getConnectedSegments(start)
yes
no
end : getConnectedSegments(middle)
yes
t2 = getCurrentTrain(end)
t2 != nullptr
no
yes
t1 != t2
no
yes
add new match <start,middle,end,t1,t2>
loop1 bound = b1
seq
leaf time=T1
opt
loop2 bound = b2
loop3 bound = b3
seq
leaf time=T2
opt
opt
leaf time=T3

(a) Control flow graph (b) Expression tree

Fig. 4. Inputs of static WCET analysis for `CloseTrains` query program

with model changes. Thus, WCET bounds change as the underlying model evolves. Furthermore, a given set of updates may impact individual WCET of multiple queries differently.

Example 7: At runtime, the parametric WCET formula $\omega(b_1, b_2, b_3)$ is instantiated with the actual parameter values based on the runtime model presented in Figure 1. Namely, b_1 get the total number of Segment instances, i.e., 12, b_2 gets the maximum degree of outgoing currentTrain graph edges among any object of type Segment, i.e., 2, while b_3 is set to the same value as b_2, i.e., 2. This way the actual WCET using the statistics obtained from the snapshot of the runtime model is $\omega(12, 2, 2) = 12 \cdot (T_1 + 2 \cdot 2 \cdot (T_2 + T_3)) = 12 \cdot T_1 + 48 \cdot T_2 + 48 \cdot T_3$. Now, if segment s8 in the railway network is shut down, i.e., no longer monitored and used by trains, the model would have only 11 Segments in total. This way the actual WCET estimate would drop to $\omega(11, 2, 2) = 11 \cdot T_1 + 44 \cdot T_2 + 44 \cdot T_3$.

VI. Conclusion and Future Work

In this paper, we discussed several high-level challenges for estimating timing properties for queries@run.time used in CPS or IoT systems where graph queries are evaluated at runtime over graph models as data structures. In particular, we showed why existing WCET analysis techniques are not directly applicable for dynamically evolving data structures. Then we presented initial results for a high-level static WCET analysis technique for graph queries which combines design-time and run-time analysis for parametric WCET estimation. We illustrated these concepts for an open CPS case study.

Future work should investigate how tight the proposed WCET estimations are compared to empirical execution times on a given hardware. Moreover, adapting our WCET analysis method to a distributed setting with significant network latency is another future challenge. Finally, other extra-functional properties can also be assessed for queries@run.time.

Acknowledgment

This paper is partially supported by MTA-BME Lendület Cyber-Physical Systems Research Group, the NSERC RGPIN-04573-16 project, and the MEDA program. Furthermore, the authors would like to thank András Vörös, Kristóf Marussy, Gábor Szilágyi, Ákos Hajdu and Gyula Sallai their help.

References

[1] G. S. Blair, N. Bencomo, and R. B. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[2] M. Búr, G. Szilágyi, A. Vörös, and D. Varró, "Distributed graph queries for runtime monitoring of cyber-physical systems," in *Fundamental Approaches to Software Engineering*, 2018, pp. 111–128.

[3] T. Hartmann, F. Fouquet, A. Moawad, R. Rouvoy, and Y. Le Traon, "GREYCAT: Efficient what-if analytics for data in motion at scale," *Information Systems*, vol. 83, pp. 101–117, 2019.

[4] T. Vogel and H. Giese, "Model-driven engineering of self-adaptive software with EUREMA," *ACM Trans. Auton. Adapt. Syst.*, p. 18, 2014.

[5] B. H. C. Cheng *et al.*, "Using models at runtime to address assurance for self-adaptive systems," in *Models@run.time*, 2011, pp. 101–136.

[6] G. Logothetis and K. Schneider, "Exact high level WCET analysis of synchronous programs by symbolic state space exploration," *Proceedings -Design, Automation and Test in Europe, DATE*, pp. 196–203, 2003.

[7] K. Havelund, "Rule-based runtime verification revisited," *Int. J. Softw. Tools Technol. Transfer*, vol. 17, no. 2, pp. 143–170, 2015.

[8] I. Dávid, I. Ráth, and D. Varró, "Streaming model transformations by complex event processing," in *International Conference on Model Driven Engineering Languages and Systems*, 2014, pp. 68–83.

[9] L. Burgueño, J. Boubeta-Puig, and A. Vallecillo, "Formalizing complex event processing systems in maude," *IEEE Access*, vol. 6, 2018.

[10] W. Dou, D. Bianculli, and L. Briand, "Model-driven trace diagnostics for pattern-based temporal specifications," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 2018, pp. 278–288.

[11] M. Abril *et al.*, "An assessment of railway capacity," *Transportation Research Part E: Logistics and Transportation Review*, vol. 44, no. 5.

[12] D. Emery, "Headways on high speed lines," in *9th World Congress on Railway Research*, 2011, pp. 22–26.

[13] G. Varró, F. Deckwerth, M. Wieber, and A. Schürr, "An algorithm for generating model-sensitive search plans for pattern matching on EMF models," *Software and Systems Modeling*, no. 2, pp. 597–621, 2015.

[14] Z. Ujhelyi *et al.*, "EMF-IncQuery: An integrated development environment for live model queries," *Science of Computer Programming*, vol. 98, pp. 80–99, 2015.

[15] S. Ma, Y. Cao, J. Huai, and T. Wo, "Distributed graph pattern matching," in *Proceedings of the 21st international conference on World Wide Web*. ACM, 2012, pp. 949–958.

[16] R. Mitschke, S. Erdweg, M. Köhler, M. Mezini, and G. Salvaneschi, "i3QL: Language-integrated live data views," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 417–432, October 2014.

[17] M. Peters, C. Brink, S. Sachweh, and A. Zündorf, "Scaling parallel rule-based reasoning," in *ESWC*, 2014, pp. 270–285.

[18] C. Krause, M. Tichy, and H. Giese, "Implementing graph transformations in the bulk synchronous parallel model," in *Fundamental Approaches to Software Engineering*, 2014, pp. 325–339.

[19] G. Szárnyas *et al.*, "IncQuery-D: A distributed incremental model query framework in the cloud," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2014, pp. 653–669.

[20] V. P. Kozyrev, "Estimation of the execution time in real-time systems," *Programming and Computer Software*, vol. 42, no. 1, pp. 41–48, 2016.

[21] R. Wilhelm *et al.*, "The Determination of Worst-Case Execution Times: Overview of the Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1—-36:53, 2008.

[22] C. Ferdinand *et al.*, "Combining a high-level design tool for safety-critical systems with a tool for wcet analysis of executables," in *Proc. of the 4th European Congress on Embedded Real Time Software (ERTS)*.

[23] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács, "Abc: algebraic bound computation for loops," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2010, pp. 103–118.

[24] F. Bodin and I. Puaut, "A wcet-oriented static branch prediction scheme for real time systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. IEEE, 2005, pp. 33–40.

[25] R. Sen and Y. N. Srikant, "WCET estimation for executables in the presence of data caches," in *Proceedings of the 7th ACM & IEEE EMSOFT '07*, 2007, p. 203.

[26] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," *Proceedings - Euromicro Conference on Real-Time Systems*, vol. 2006, pp. 217–226, 2006.

[27] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *ACM SIGPLAN Notices*, vol. 30, no. 11. ACM, 1995, pp. 88–98.

[28] A. Ghosal *et al.*, "A hierarchical coordination language for interacting real-time tasks," in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. ACM, 2006, pp. 132–141.

[29] C. Colombo *et al.*, "polyLarva: runtime verification with configurable resource-aware monitoring boundaries," in *International Conference on Software Engineering and Formal Methods*, 2012, pp. 218–232.

[30] X. Zheng *et al.*, "Efficient and Scalable Runtime Monitoring for Cyber-Physical System," *IEEE Systems Journal*, pp. 1–12, 2016.

[31] A. Bauer and Y. Falcone, "Decentralised LTL monitoring," *Formal Methods in System Design*, vol. 48, no. 1-2, pp. 46–93, 2016.

[32] C. Alippi *et al.*, "Model-Free Fault Detection and Isolation in Large-Scale Cyber-Physical Systems," *IEEE Trans. Emereg. Topics Comput. Intell.*, vol. 1, no. 1, pp. 61–71, 2017.

[33] K. Ramamritham, "Real-time databases," *Distributed and Parallel Databases*, vol. 1, no. 2, pp. 199–226, Apr 1993.

[34] G. Ozsoyoglu and R. T. Snodgrass, "Temporal and real-time databases: a survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 4, pp. 513–532, Aug 1995.

[35] J. Herter and J. Reineke, "Making dynamic memory allocation static to support WCET analyses," *Worst-Case Execution Time Analysis*, 2009.

[36] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *Journal of the ACM (JACM)*, vol. 65, no. 3, p. 16, 2018.

[37] B. Lisper, "Fully Automatic, Parametric Worst-Case Execution Time Analysis," *Proceedings of the Third International Workshop on Worst-Case Execution Time (WCET) Analysis*, pp. 99–102, 2003.

[38] A. Anjorin, M. Lauder, S. Patzina, and A. Schürr, "Emoflon: leveraging emf and professional case tools." in *GI-Jahrestagung*, 2011, p. 281.

[39] N. Wirth, "On the composition of well-structured programs," *ACM Computing Surveys*, vol. 6, no. 4, pp. 247–259, 1974.

[40] S. Bygde, A. Ermedahl, and B. Lisper, "An efficient algorithm for parametric WCET calculation," *Journal of Systems Architecture*, vol. 57, no. 6, pp. 614–624, 2011.

[41] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: an open toolbox for adaptive wcet analysis," pp. 35–46, 2010.

[42] C. Ferdinand and R. Heckmann, "ait: Worst-case execution time prediction by static program analysis," in *Building the Information Society*, R. Jacquart, Ed. Boston, MA: Springer US, 2004, pp. 377–383.

[43] C. Ballabriga, J. Forget, and G. Lipari, "Context-sensitive parametric wcet analysis," in *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, 2015.

[44] G. Varró, K. Friedl, and D. Varró, "Adaptive graph pattern matching for model transformations using model-sensitive search plans," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 191 – 205, 2006.

[45] R. Geis, G. Veit Batz, D. Grund, S. Hack, and A. Szalkowski, "Grgen: A fast spo-based graph rewriting tool, icgt 2006, a. corradini et al," 2006.