Worst-Case Execution Time Optimization of Story Patterns for Hard Real-Time Systems*

Sven Burmester†, Holger Giese, Andreas Seibel, and Matthias Tichy
Software Engineering Group, University of Paderborn, Warburger Str. 100, Paderborn, Germany
[burmi|hg|aseibel|mtt]@uni-paderborn.de

ABSTRACT

In the future, technical systems are expected to operate more intelligent than today by taking their local context explored by means of sensors and network communication into account. To realize this vision, the systems must be able to represent and query as well as interact with a large number of possible situations not known a priori. Therefore, flexible means to store, query, and manipulate such context information are required. Known flexible and powerful representations are class diagrams or other graph-like notations. However, such dynamic data structures which are sources for unpredictable run-time timing behavior are traditionally not recommended for the development of hard real-time systems. In this paper, we describe our efforts to employ story patterns, which are used for the specification of query and update operations on dynamic data structures, in hard real-time systems.

Keywords

Real-Time, Story-Pattern, Worst-Case Execution Time Optimization

1. INTRODUCTION

Advanced technical systems of the future such as self-adaptive [18, 13, 14] or self-optimizing [3] technical systems will operate smarter than today's systems by adjusting their operation to the experienced context. Besides the information provided by sensors, the communication with other entities near by via wireless networks will increase the available information and its complexity.

The software of these systems must thus be able to represent and query a large number of possible not a priori known context situations. The means to store, query and manipulate such context situations must support model-based development and should not be restricted to fixed-sized arrays. UML and in particular class diagrams became the standard to describe the structure of the complex information. Story diagrams [19, 12] are an advanced technique to manipulate this information. However, class diagrams describe properties of dynamic data structures which result in unpredictable run-time timing behavior and are thus traditionally not considered as an option for the development of hard real-time systems.

In real-time systems, the provision of a service (e.g. reaction to an incoming message, a computation, ...) is associated with a certain deadline. If the deadline expires before the service is provided, the results in embedded systems are typically catastrophic due to damages to humans in case of automotive or railway systems. Those systems are named *hard real-time systems*.

In order to guarantee to meet the required deadlines, the worst-case execution times (WCETs) of methods or other implementation artifacts must be known. In addition, worst-case execution times are required for a schedulability analysis [5] which is used to check whether concurrent processes are executable on a given processor meeting the required deadlines.

Standard dynamic data structures are unbounded, i.e. they have no predetermined maximal amount of stored elements. Thus, no worst-case execution time can be given for operations on these data structures, since the execution time typically is dependant on the contained number of stored elements. Therefore, in order to determine a worst-case execution time, the maximal number of elements in those data structures must be fixed beforehand. Then, the worst-case execution time can be determined.

Additionally, algorithms in standard applications are optimized for the average case (e.g. the quicksort algorithm). Since the average case is only of low relevance in hard real-time systems, algorithms on those dynamic data structures should have an optimal (read: minimal) worst-case execution time. Therefore, we require (1) a model that allows the determination of WCETs of the generated code and (2) we should generate code so that the WCETs are optimized.

Current WCET analysis techniques are restricted to imperative programming languages. Dynamic, object-oriented programming languages are not addressed at all. Buttazzo even demands to avoid dynamic data structures in real-time systems [5].

The standard approach in WCET analysis is to analyze the longest executable path, to map each instruction of this path to *elementary operations*, and to determine the WCETs of these elementary operations. The elementary operations can be for example assembler instructions as in the [6] or Java Byte Code instructions as in [1].

In [6], the WCET of a fragment of generated C code is determined by summing up the number of processor cycles each C instruction's corresponding assembler instruc-

†Supported by the International Graduate School of Dynamic Intelligent Systems. University of Paderborn.
*This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

tions require. For loops, the *worst-case number of iterations (WCNIs)* is derived from a statechart model to obtain the maximum number of executions of the loop-bodies.

[15] describes multiple existing approaches that use different annotations to specify the WCNIs and thus the longest executable path. All described approaches are restricted to imperative programming languages, that do not provide or use dynamic data structures. Further, the authors explain that an execution time analysis on the hardware level, which considers techniques like caching or pipelining, is required to avoid a too pessimistic estimation.

We present in this paper how code generation from UML class diagrams can be improved such that the resulting source code can be employed in hard real-time systems as representation for complex content. We propose to specify query and update tasks on this content as story patterns [7, 19]. Therefore, we present how source code with a minimal worst-case execution time can be synthesized from story patterns. We generate code for C++. We assume that worst-case execution times are known for all calls to external functions. In conformance to standard approaches for real-time systems, we assume that all memory is allocated at the start of execution.

In the next section, we present the example which is used in the remainder of the paper. Section 3 contains foundations which are required for predictable real-time behavior. Based on this foundations, we present in Section 4 how worst-case execution times are determined. In Section 5, the approach for computation of optimal worst-case execution times is presented. We conclude in Section 6 and present possible future work.

2. EXAMPLE

In the new transport system developed in Paderborn[1] autonomous vehicles drive on a railway system. Communication is required between the shuttles for coordination purposes, for example for building convoys to reduce the air resistance and thus the general power consumption.

The railway system is divided into multiple sections, each coordinated by a so-called *Registry*. Before entering a track section, a shuttle has to register at the corresponding registry. The registry collects information about the shuttle's current position and velocity and broadcasts this information to the other shuttles via wireless communication.

In this paper, we regard a situation as shown in Figure 1 when two shuttles move towards the same joining switch. The shuttles need to coordinate how to pass the switch in order to avoid a possible collision. Obviously, this has to be finished before they reach the switch. Thus, this coordination problem is subject to hard real-time requirements.

In order to recognize and to handle such situations, we use an ontology based topology for every shuttle to store environmental information in a discretized manner. On the one hand, each shuttle recognizes changes in its environment by sensors, on the other hand, it periodically receives updates of this environmental information from the registry as described in [10]. Figure 2 shows the UML class diagram of a shuttle's topology.

Every shuttle knows a registry that is liable for the set of tracks on which the shuttle is located. A CommunicationRule is like an instruction to handle a certain problem. Therefore,

[1]http://www-nbp.upb.de/en

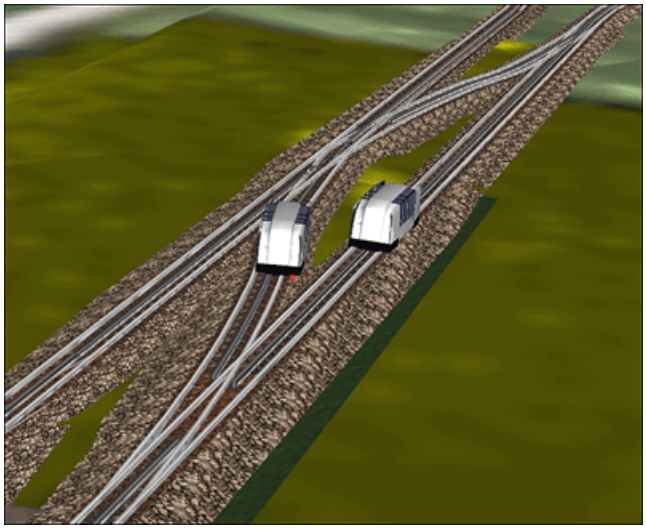

Figure 1: A possible collision of two shuttles at a switch

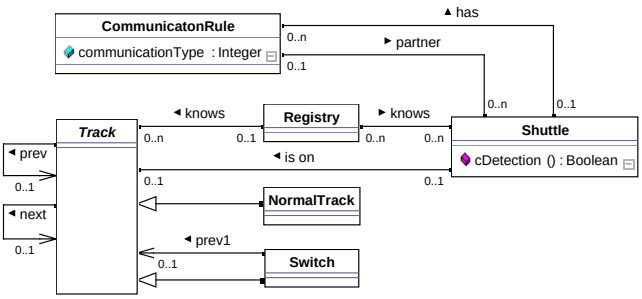

Figure 2: UML class diagram for the shuttle ontology

it has a partner association that describes which shuttles are involved in the respective problem. It also provides a type to classify the required coordination and thus the problem. Every shuttle provides a cDetection method that must be initiated if a shuttle is heading towards a switch. This method checks whether a situation might occur that causes a collision as illustrated in Figure 1. If a possible collision is detected, a CommunicationRule for avoiding the collision is created. This rule describes which shuttle has to slow down to avoid the collision. Of course, the shuttles may initiate a coordination, e.g. to buy or sell respectively the right of way. Figure 3 shows a story diagram that consists of one story pattern and specifies the method cDetection.

The behavior, specified by the story pattern of the story diagram, consists of two parts: First, an object matching searches for the situation that might cause a collision. This situation occurs when two shuttles are located on the tracks, that lead to a joining switch. If such an instance situation is matched, the second part of the behavior creates a CommunicationRule object with type = RIGHT_OF_WAY where RIGHT_OF_WAY is a constant for right of way. It also creates two links between the involved shuttles. When a matching is found and the CommunicationRule is created, the story

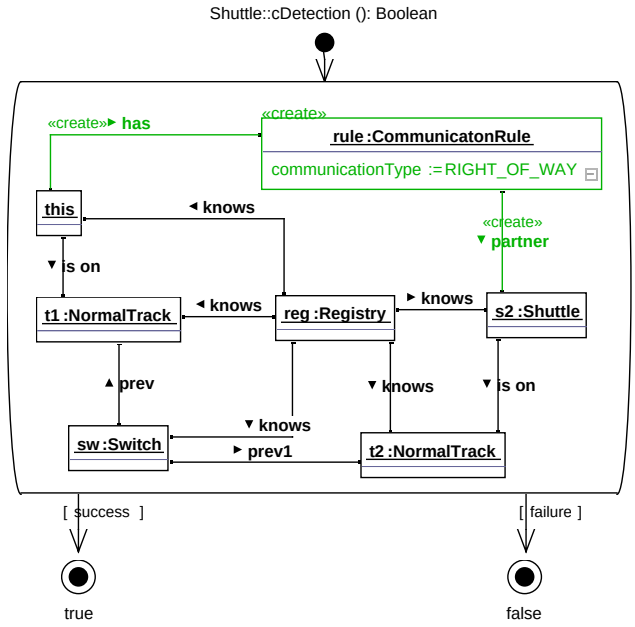

Figure 3: Story Diagram for collision recognition

diagram returns a true value, false otherwise. Both shuttles execute complementing story pattern which guarantee that one shuttle has right of way and the other one has to wait.

The mentioned coordination can be specified by another story diagram or by a Real-Time Statechart [4, 9, 2] that uses the return value of the cDetection method as transition trigger. Further aspects of real-time systems, like for example the communication, is out of the scope of story patterns and is handled for example in [11].

3. PREDICTABLE REAL-TIME BEHAVIOR

As indicated in the introduction, unbounded data structures lead to unpredictable real-time behavior. As class diagrams describe unbounded data structures and thus unbounded data structures are used to implement class diagrams, like the one from Figure 2, story patterns that operate on these data structures do not show predictable real-time behavior. Therefore, a WCET of a story pattern cannot be derived automatically from the model.

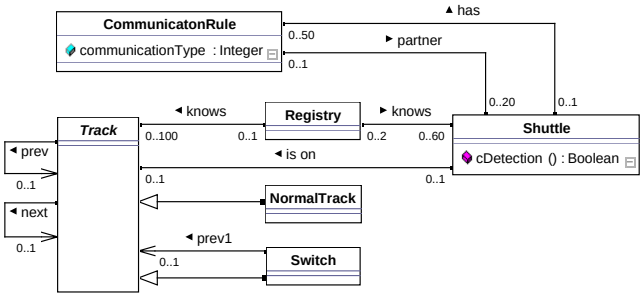

Figure 4: Class diagram with fixed maximum multiplicities

To overcome this limitation, we define so-called *fixed maximum multiplicities* in a class diagram, as shown in Figure 4. Note that the multiplicities, labeled with n in Figure 2, are replaced by concrete values in Figure 4. This enables an implementation using data structures with upper bounds. These upper bounds determine a *worst-case number of iterations (WCNIs)* when searching in these data structures which leads to predictable real-time behavior. This model-based development approach, combined with automatic code-generation leads to a well-structured implementation with analyzable nested loops and loops with fixed termination conditions.

Further, we make use of the factory pattern [8] to avoid dynamic resource allocation and deallocation after initialization time. As we know the implementation scheme of the access methods of the factory pattern and the implementation scheme of the code fragments that realize the story pattern, we derive their WCETs simply by adding the WCETs of the corresponding elementary operations.

There are several elementary operations on dynamic data structures in order to execute a story pattern. Elementary operations are creation and deletion of objects, adding and removing objects from different data structures, writing and reading attributes, and comparing objects. For each of these elementary operations, we use a runtime measurement tool executing a worst-case scenario running on the selected hardware platform. From this runtime measurement tool, we get the required WCETs. As different types of data structures are used (e.g. TreeSet, HashSet, LinkedList, ...), we compute the WCETs for the different data structures using different worst-case scenarios. The data structures used in the worst-case scenario have the maximum size as specified by the maximum multiplicities in the class diagram. As we know the code of the data structures, we also know the worst-case path when operating on them. In the future, the worst-case scenarios will be extended to capture degenerated data structures for a more precise WCET estimation.

The WCET of a story pattern does not only depend on the WCETs of its single code fragments and on the WCNIs when searching in data structures. The problem of WCET determination for story patterns is more complicated, because the order in which the elements of a story pattern are matched has significant impact on the resulting WCET as (partly) nested iterations can occur:

Multiple different matching sequences that lead to different WCETs exist because story patterns can contain bidirectional cycles. In the example story pattern of Figure 3, there exist several uni- and bidirectional cycles. For example, if the only bound object is this, this → t1 → reg → this is a bidirectional cycle because we also have the possibility to choose this → reg → t1 → this to match this part of the story pattern. For example, a unidirectional cycle is reg → sw → t1 → reg because the association between Switch and NormalTrack (which is a Track) is unidirectional.

The reason why different matching sequences usually lead to different WCETs is because different matching sequences can have different WCNIs. When, for example, a link between a Registry and a Shuttle instance is specified, starting at the Registry object and binding the Shuttle object requires a search in a data structure with 60 as upper bound. Binding the Registry object from the Shuttle object requires just a search in a data structure consisting maximal of 2 instances. In this case the algorithm, which determines the matching

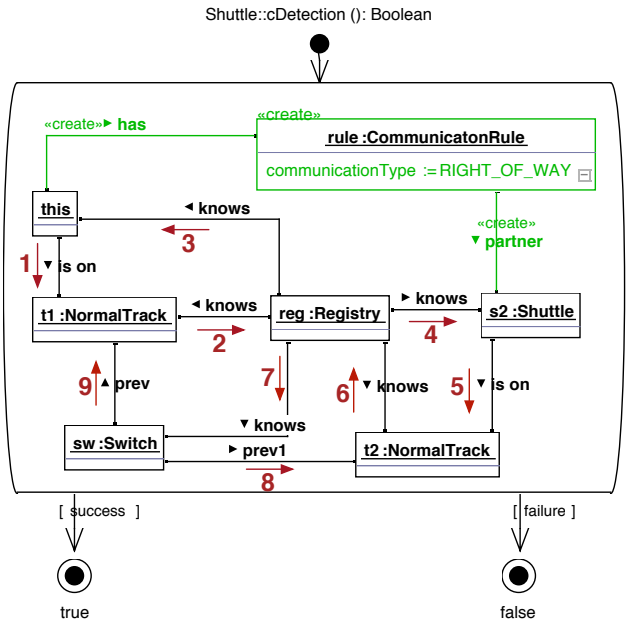

Figure 5: Story diagram and any matching sequence

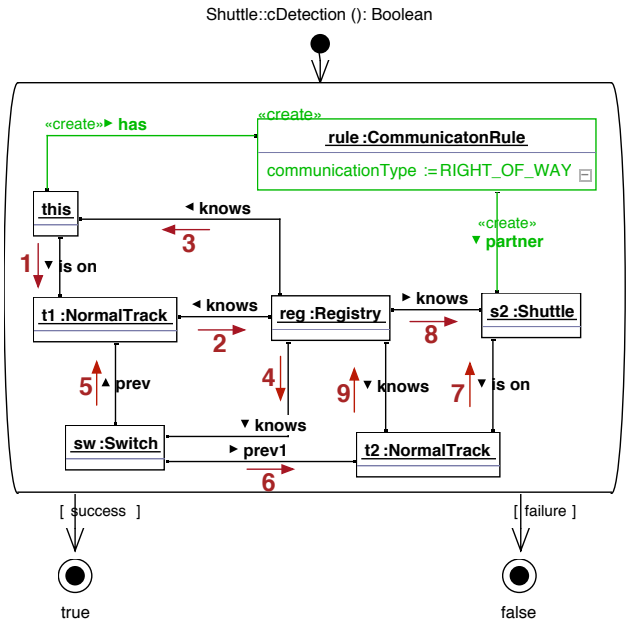

Figure 6: Story diagram with a better matching sequence (optimal)

sequence, has two possibilities that lead to the same instance matching but use different sequences.

Another reason why different matching sequences usually lead to different WCETs is that the matching process explores in the worst-case a path for each existing instance when binding an instance that is connected to a bound instance via a to-many association. To obtain an optimal WCET, the number of such paths has to be minimized. This is achieved by first respecting the path via associations with low multiplicities.

In Figures 5 and 6, the arrows with associated numbers represent different matching sequences for the shown story pattern. The two different strategies to perform the matching lead to different WCETs due to the different maximal sizes of the data structures as described two paragraphs before. As there exist multiple possible strategies to perform the matching, we should choose a strategy, that leads to an *optimal WCET*, i.e. a WCET that is as small as possible.

As we specified fixed maximum multiplicities and thus know the upper bounds of the corresponding data structures, we can determine a matching sequence so that the matching will use a minimum of comparisons when searching data structures and thus leads to the optimal WCET. In the next section is described how to determine the WCET for a matching sequence of a story diagram. Section 5 describes how to find the optimal matching sequence.

4. WCET DETERMINATION

In order to calculate and optimize the WCET of a story pattern, we introduce the so-called *story graph* [17]. This graph consists of different types of edges respecting that there are different kinds of checks to be performed during matching: For example, starting at the this object binding object t1 (step 1 in Figure 6) and then binding binding reg (step 2) is simple as the corresponding associations are to-one associations. For step 3, it is just a simple check for existence of a link is required, as the source and the target objects are already bound. Binding sw from reg in step 4 requires the search in a data structure, as it is not a to-one but a to-many association.

Before explaining further details like *story graph* creation, edge selection mechanism, timing constraints and WCET calculation with a *story graph*, a formal definition of the *story graph* is given in Definition 1.

Definition 1 *Let $G = (V, E)$ be a story graph with V the nodes, $E \subseteq (V \times V \times \mathbb{N}^3 \times \mathbb{N} \times E_s \times T)$ the edges and let $G_s = (V_s, E_s)$ be a story pattern. Each node $v_s \in V_s$ is mapped to a node $v \in V$ and each edge $e_s \in E_s$ is mapped to one or multiple edges $e \in E$ (see below). Thus, it holds $|V| = |V_s|$ and $|E| \geq |E_s|$. An edge $e = (s, t, w, c, L_e, t_e) \in E$ consists of the following elements: $s \in V$ is the source node of the edge e. $t \in V$ is the target node of the edge e. $w = A_{tt} = (w_f, w_d, c_a)$ is defined as AnalysedTypeTime which includes all timing information required to compute the WCET when choosing the edge e. $c \in \mathbb{N}$ is the maximum number of iterations that is required for binding the edge e. $L_e = \{e_{s1}, ..., e_{sn}\} \subseteq E_s$ is a set of Link/MultiLink-references of the story pattern G_s associated with the story graph edge e. $t_e \in T = \{$ BindNormal, BindOptional, CheckIsomorphism, CheckLink, CheckAttribute, CheckConstraint, CheckNegativeLink, CheckNegativeNode $\}$ is the type of the edge $e \in E$.*

V_s and E_s define the nodes and edges of a story pattern. See [19] for a detailed formalization of story patterns. An AnalysedTypeTime $w = A_{tt} = (w_f, w_d, c_a)$ contains runtime information. w_f is a fixed execution time that occurs due to a code fragment before starting a possible loop. w_d is the execution time for a single loop iteration. The number

of iterations is stored in c_a that results from the defined multiplicity of the related association in the related class diagram introduced in section 3. c_a should not be confound with c. In most cases they are equal, but there is an exception when these values differ. If there is an edge $e_i \in E$ with $t_i = \{\text{CheckNegativeNode}\}$ and the AnalysedTypeTime w_i then $c_i = 1$, but c_{ai} is the number of iterations that is derived from the defined exact multiplicity of the related association. The generated code that is necessary for checking negative nodes never starts a further nested loop, but for WCET computation of the code fragment for the negative node check, the fixed multiplicity is necessary and stored in c_{ai}. c is only used when calculating the WCET $K(L)$ of a matching sequence L (cf. Definition 2).

Every *story graph* edge $e \in E$ can optionally have one or more associated Link/MultiLink-references $L_e \subseteq E_s$. L_e does not influence WCET computation, but provides information required for implementation issues.

$t_e \in T$ describes the classification of a *story graph* edge $e \in E$. As defined in Definition 1 the set T includes eight classification types. These classification types describe groups of link types of a story pattern. For example, $t_e = \{\text{BindNormal}\}$ defines a link which describes a normal matching of an instance (except the links with optional condition). Every classification type describes indirectly a pre selection criterion and a post selection effect used while finding an optimal matching sequences described in the next section. The pre selection criterion describes which *story graph* edge $e \in E$ is available for selection. In every computation step, the algorithm for optimization has to choose a *story graph* edge $e \in E$ which was not considered before. The selection of a *story graph* edge $e \in E$ affects the *story graph* in a way that is implicitly encoded in the classification type $t_e \in T$ of the selected edge $e \in E$ (e.g. after binding a node via a link, this link does not need to be checked any more) what is called the post selection effect.

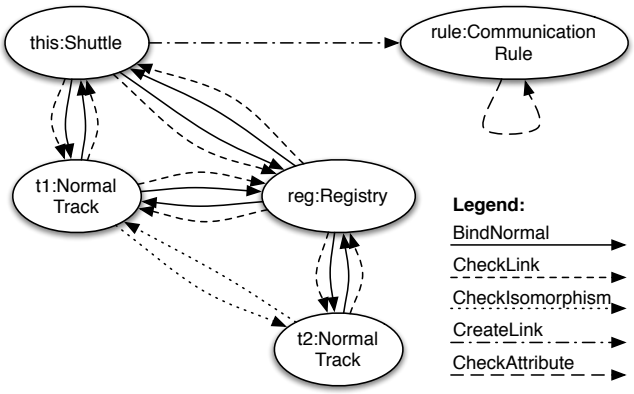

Figure 7: Cut-out of the resulting *story graph* from the story pattern example

The *story graph*, resulting from our example story pattern inside the story diagram introduced in Figure 3, consists of 7 nodes and 39 edges. Due to lack of space, we present just a cut-out of the *story graph* consisting of 5 nodes and 20 edges illustrated in Figure 7. This is adequate to explain the importance of the *story graph*.

Note that the *story graph* edges $e \in E$ are not inevitably related to story pattern edges $e_s \in E_s$. For example, there is no Link/MultiLink-reference $e_s \in E_s$ for attribute checks and assignments, but the *story graph* contains an edge $e_i \in E$ with $t_i = \{$ CheckAttribute $\}$ with the rule node as source and as target. For every story pattern Link/MultiLink-reference $e_s \in E_s$, several *story graph* edges $e \in E$ are created. For example, for every $e_i \in E$ and $t_i = \{$ BindNormal $\}$ a corresponding *story graph* edge $e_j \in E$ with $t_j = \{$ CheckLink $\}$ exists also. As mentionend above, either the BindNormal or the CheckLink link is taken for the matching sequence. The *story graph* edges $e_i \in E$ and $t_i = \{$ CheckIsomorphism $\}$ exists between all objects which have the same class diagram type.

The WCET for a story pattern and a specific matching sequence is determined as described in Definition 2.

Definition 2 *Let $K(L)$ be the WCET for a solution L of a story graph $G = (V, E)$ with $L = (e_1, ..., e_n)$, $e_i = (s_i, t_i, w_i, c_i, L_{ei}, t_{ei}) \in E$, $w_i = (w_{fi}, w_{di}, c_{ai})$ and $c_0 = 1$. Then* $K(L) = \sum_{i=1}^{n} \left[(w_{fi} + w_{di} \cdot c_{ai}) \cdot \left(\prod_{j=0}^{i-1} c_j \right) \right] + pC(G, L)$.

A solution L is an n-tuple of *story graph* edges $e \in E$. This n-tuple defines the matching sequence described in Section 3. As this matching sequence should be used for code generation of a story pattern, it can be translated into a regular matching sequence for the story pattern (the translation needs the Link/MultiLink-references stored in L_{ei} of every *story graph* edge $e_i \in E$). The function $pC(G, L)$ returns a runtime that is caused by code fragments that do not affect the runtime of the related matching sequence related to L. These code fragments are executed before or after a story pattern is matched successfully. For example, object and Link/MultiLink-reference deletion take place after successful matching. For $i = 1..n$ the execution time of the *story graph* edge e_i is multiplied with the number of how many times it will be checked in worst-case (WCNI).[2] This is described by the product inside $K(L)$ which is the number of iterations of the considered *story graph* edge e_i. By building the sum of these execution times and adding $pC(G, L)$, we get the WCET of the considered matching sequence L.

5. WCET OPTIMIZATION

At the end of Section 3, we stated that story patterns could have many valid matching sequences. This means that there exists at least one matching sequence in the set of all possible matching sequences that will take a minimum of runtime during its execution in the worst-case. So, for an optimal WCET, the determination of a solution L is required that minimizes $K(L)$.

In order to determine $min(K(L))$, we use a brute force back tracking search method, as listed in Figure 8. This algorithm determining the optimum requires exponential runtime in relation to the number of Link/MultiLink-references of a story pattern and the existing bidirectional cycles. It uses recursion to compute valid solutions L. A solution L is only valid when its WCET is less than the WCET of the best solution the algorithm found up to this point. Further, the solution has to contain all necessary *story graph* edges $e \in E$

[2]Due to a technical issue in the product function, we initially start with $c_0 = 1$.

```
 1: s ← Defined WCET of the engineer
 2: AL ← ApproximatedSolution(G)
 3: k ← K(AL) ∨ defined upper bound of the engineer
 4: minimum ← Ø
 5: L ← Ø ∧ L.valid = true
 6: function OptimalSolution(G, L)
 7:     Sort all edges from i = 1...n ascending by w_fi + (w_di · c_ai)
 8:     if (Not all edges e_i ∈ E in G marked) ∧ (Edges still reachable) ∧ (L.valid = true) then
 9:         for All reachable edges e_i ∈ E in G do
10:             G' = (V', E') ← G = (V, E)
11:             L' ← L ∘ e_i
12:             Process all necessary markings e'_i ∈ E' of G'
13:             if K(L') < k then
14:                 OptimalSolution(G', L')
15:             else
16:                 L'.valid ← false
17:             end if
18:         end for
19:         if still edges e_i available then
20:             L.valid ← false
21:         end if
22:     end if
23:     if L.valid = true then
24:         minimum ← L
25:         k ← K(minimum)
26:     end if
27:     if k ≤ s then
28:         Terminate
29:     end if
30: end function
```

Figure 8: Algorithm for $min(K(L))$ determination

to become valid. Thus, after termination of the algorithm, the invariant $minimum = min(K(L))$ is true.

As a method with exponential time might lead to problems in practice, we improved the algorithm as shown in Figure 8: We use a heuristics (line 2), we ensure a monotonic decreasing of the upper WCET bound (line 3, 13, 25), and we introduce a lower WCET bound (line 1, 27).

First of all, a heuristics [17] is applied that leads to acceptable values for the WCET in the average case shown in Figure 9. The function *OptimalSolution* uses this function *ApproximatedSolution* to determine a solution L so that its WCET $= K(L)$ can be used as first upper bound. This cuts down the search space of possible solutions L at the beginning of OptimalSolution. OptimalSolution will recognize solutions as infeasible as soon as the execution time is greater or equal the heuristics WCET. This way of using a first upper bound and then decreasing the upper bound monotonic reduces the computation time significantly.

Usually, it is just required to obtain an implementation with a WCET that fits in a specific timing interval or just a given WCET from a requirement specification needs to be fulfilled. Thus, the engineer may define a target value for the WCET (s in Figure 8). The algorithm terminates when it determined a matching sequence that leads to a WCET that is below this target value. Obviously, this WCET can be larger than the optimal WCET.

```
1: function ApproximatedSolution(G)
2:     L ← Ø
3:     Sort all edges from i = 1...n ascending by w_fi + (w_di · c_ai)
4:     for Not all edges e_i in G are marked do
5:         for Unmarked edges reachable do
6:             Choose possible edge e_i from S(G) with smallest w_fi + (w_di · c_ai)
7:             L ← L ∘ e_i
8:         end for
9:     end for
10:    return(L)
11: end function
```

Figure 9: Algorithm to determine a first matching sequence

Tests showed that due to the improvements, a solution for a common story pattern can be found in acceptable time. Figure 10 illustrates the distribution of the different WCETs of the example shown in Figure 3. The figure shows the WCET values and the number of matching sequences with the respective WCET. The WCETs unit is milliseconds and is listed logarithmic.

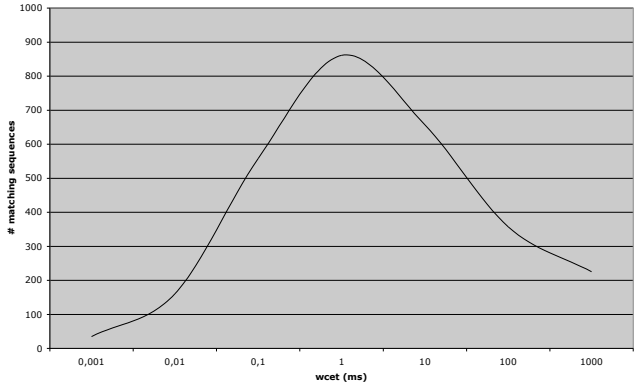

Figure 10: Frequency distribution chart of matching sequences and their WCETs

We see that most solutions have a WCET in the middle of minimum and maximum. Without the described improvements of the algorithm, the computation of the optimal matching sequence took about eight minutes on a 900 MHz PowerPC 750fx processor. Using the heuristics and monotonic decreasing of the upper bound it took about three seconds and only three possible matching sequences were found till optimum. Figure 11 shows the improvement of the example's WCET in relation to the time needed for optimization. Note that the x-axis is increasing exponentially.

The black line shows the temporal development of the best solution during the computation process. The grey line shows the optimal WCET that could be possible. This points out that using monotonic decrease of the upper bound is heavily decreasing the number of possible matching sequences.

For the evaluation shown in Figure 12, an abstract story pattern with twelve Link/MultiLink-references is used. The figure shows the improvement of the WCET in relation to

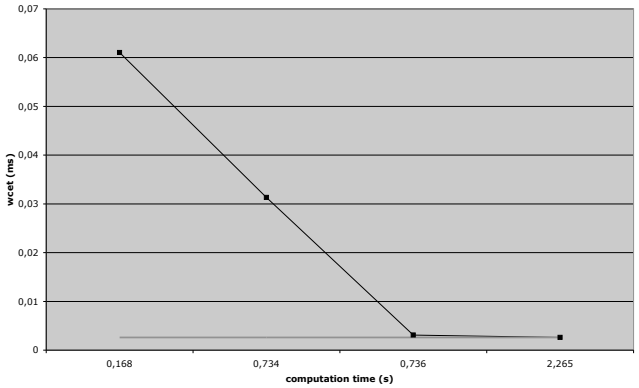

Figure 11: Improving the WCETs with the restricted algorithm

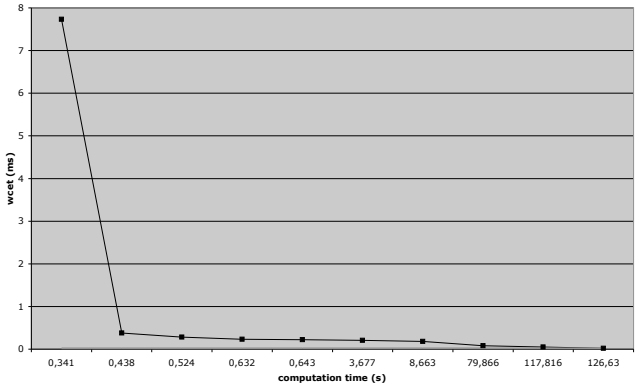

Figure 12: Improving the WCETs with the restricted OptimalSolution but with an abstract story pattern

the runtime of the optimization algorithm. Figure 13 shows a chart that results from computation time measurements of story diagrams with one abstract story pattern with a high number of bidirectional cycles. The x-axis describes the number of Link/MultiLink-references of the story pattern and the y-axis the OptimalSolution computation time in seconds. The diagrams show that story patterns with more then twelve Link/MultiLink-references require long computation times. Note that the computation time does not only depend on the number of Link/Multilink-references, but also on the number of bidirectional cycles which also increase the number of possible matching sequences.

6. CONCLUSIONS & FUTURE WORK

Graph like structures are required for storing context and local knowledge in future complex intelligent and adaptive technical systems. Story patterns are an appropriate modeling language for modifying graph like structures. In order to satisfy safety and hard real-time requirements, worst-case execution times for the execution of story patterns are required. We presented in this paper an approach which (1) determines these worst-case execution times on a given hardware and (2) computes an optimal worst-case execution time based on an optimal search order of the story pattern elements. The computation of the optimal order consists of two steps. In the first step, a heuristics is used in order to find an optimal search order for the average case. Thereafter, in step two, a brute force algorithm is employed to find a better solution than provided by the heuristics.

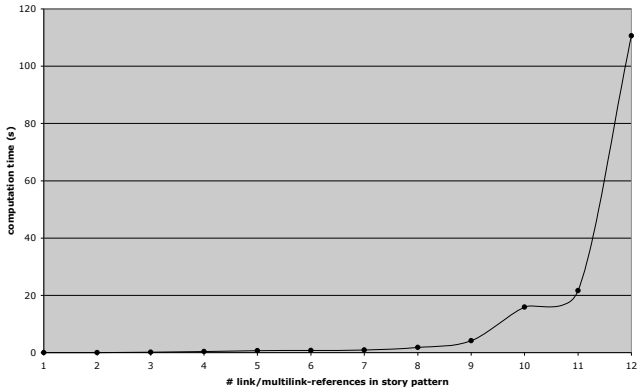

Figure 13: Computation times in relation to the number of Link/MultiLink-references of a abstract story pattern

To improve our WCET optimization algorithm, we plan to respect knowledge about the minimal remaining costs in the algorithm in a branch-and-bound manner. Further, we plan to support the WCET determination of story diagrams, consisting of multiple story patterns as well. Therefore, we plan to integrate our approach with the MAXT approach [16] that requires the specification of the WCNIs for every activity in the story diagram and their WCETs. We will use our algorithm for single story patterns to determine the single activities' WCETs. In [16], the WCET for the cyclic flow graph is then computed with integer linear programming (ILP).

Story charts [12] are an extension of standard UML state machines by story pattern. The states are enriched by story patterns as do methods. Story charts lack appropriate notions for time. Real-Time Statecharts [4, 9, 2] are an appropriate state based modeling notation for the specification of real-time behavior. The presented WCET determination and optimization approach will be used in order to integrate story patterns into Real-Time Statecharts. In addition to the usage of story patterns in story charts, we will not only support story patterns as behavior specification for do methods, but for all kind of actions (entry and exit methods as well as transition actions).

REFERENCES

[1] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000)*, 2000.

[2] S. Burmester and H. Giese. The Fujaba Real-Time Statechart PlugIn. In H. Giese and A. Zündorf, editors, *Proc. of the first International Fujaba Days 2003, Kassel, Germany*, volume tr-ri-04-247 of *Technical Report*, pages 1–8. University of Paderborn, 2003.

[3] S. Burmester, H. Giese, and O. Oberschelp. Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In *Informatics in Control, Automation and Robotics*. Kluwer Academic Publishers, 2005. to appear.

[4] S. Burmester, H. Giese, and W. Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, pages 1–15, November 2005.

[5] G. C. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer international series in engineering and computer science : Real-time systems. 1997.

[6] E. Erpenbach. *Compilation, Worst-Case Execution Times and Scheduability Analysis of Statechart Models*. Ph.D.-thesis, University of Paderborn, Department of Mathematics and Computer Science, 2000.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. LNCS 1764, pages 296–309, November 1998.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA, 1995.

[9] H. Giese and S. Burmester. Real-Time Statechart Semantics. Technical report, 2003.

[10] H. Giese, S. Burmester, F. Klein, D. Schilling, and M. Tichy. Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In B. Henderson-Sellers and J. Debenham, editors, *OOPSLA 2003 - Second International Workshop on Agent-Oriented Methodologies*, pages 21–32, Anaheim, CA, USA, Center for Object Technology Applications and Research (COTAR), University of Technology, Sydney, Australia, Oct. 2003.

[11] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[12] H. J. Köhler, U. A. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. pages 241–251. ACM Press, 2000.

[13] D. J. Musliner, R. P. Goldman, M. J. Pelican, and K. D. Krebsbach. Self-Adaptive Software for Hard Real-Time Environments. *IEEE Inteligent Systems*, 14(4), July/Aug. 1999.

[14] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

[15] P. P. Puschner and A. Burns. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, May 2000.

[16] P. P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times - A Graph-Based Approach. In *Real Time Systems, 13*, Technical Report, pages 67–91. Springer Link, Kluwer Academic Publishers, July 1997.

[17] A. Seibel. Story Diagramme für Eingebettete Echtzeitsysteme. Bachelor Thesis at University of Paderborn, Department of Computer Science, Paderborn, Germany, February 2005.

[18] J. Sztipanovits, G. Karsai, and T. Bapty. Self-adaptive software for signal processing. *Commun. ACM*, 41(5):66–73, 1998.

[19] A. Zündorf. Rigorous Object Oriented Software Development. Habilitation Thesis at University of Paderborn, Department of Computer Science, Paderborn, Germany, 2001.