Story Diagrams in Real-Time Software

Matthias Tichy, Holger Giese, and Andreas Seibel
Software Engineering Group
University of Paderborn, Germany
[mtt|hg|aseibel]@uni-paderborn.de

ABSTRACT

Software plays an increasingly important part in todays embedded systems. Development efforts for embedded software must consider the trade-off between fast development, maintainable code, correct as well as high-performance software. Graphical object-oriented languages can help in creating more maintainable code, while also providing better means to ensure correctness. Predictable real-time behavior w.r.t. the execution time of operations is additionally of particularly importance in the embedded domain. We present in this paper a graphical language employing graph transformations as a formal foundation. The language is especially geared for event-driven transformations of data structures. Additionally, we present our approach for worst case execution times estimation to predict software behavior in the time domain. We evaluate our approach using an example from the railway domain.

1. INTRODUCTION

Software plays an increasingly important part in todays embedded systems. As many embedded systems are used in a safety-critical context (e.g. cars, medical systems, trains), they must adhere to strict quality requirements. The software must not only work correctly, but also be predictable w.r.t. the execution time of operations. Additionally, product cycles are becoming shorter. Thus, software development has to finish in even shorter time frames.

Many approaches try to tackle the aforementioned problems. Block diagrams are employed for a graphical specification of control algorithms in tools like Matlab/Simulink. Object-oriented and graphical languages are used to specify event-driven software with complex object structures. They can in principal address the aforementioned problems but are seldom used in embedded software development. This stems partly from problems in ensuring worst case execution times (WCET) which is essential for correctness w.r.t. real-time specifications.

Graph transformations are used in many variants as a language for expressing structural transformations [17, 18, 7, 21]. Several approaches exist [16, 1] for ensuring the functional correctness of graph transformations. As mentioned above, WCET computation is required for ensuring correctness w.r.t. real-time specifications but current approaches for WCET estimation like [20, 5] are not applicable for software which executes arbitrary data structure changes.

In previous works, we have shown (1) how to estimate and optimize WCET for a graph transformation variant called Story Pattern [4], and (2) how to formally verify inductive invariants on sets of Story Patterns [1]. Both approaches help to develop software which satisfies strict quality requirements. In this paper, we draw on these previous works and present a graphical language for event-driven structural transformations whose usage is feasible in embedded and safety critical software development.

In the next section, we present the railcab research project[1] and the running example of this paper which stems partly from the railcab project. We describe the different parts of our language in Section 3. In Section 4 we introduce our approach for WCET estimation of programs which are specified in our language. Section 5 employs an evaluation which compares our estimated WCET with actually measured worst case execution times. In Section 6, we review some related approaches. We conclude the paper in Section 7 and present research directions for future work.

2. EXAMPLE

A new transport system called Railcab is developed in Paderborn. The transport system utilizes autonomous vehicles which drive on a nearly standard 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.

In previous works [9, 4], we addressed 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. This simple scenario considers only two shuttles approaching a single switch.

Train stations typically employ several switches and numerous shuttles which concurrently drive in the station area. Consequently, we need to consider a scenario with multiple switches and multiple shuttles. Additionally, shuttles have to communicate and coordinate not only with shuttles which they meet at the next switch but also with shuttles which they may meet at later switches. Figure 2 shows the extended scenario based on the above stated requirements.

The embedded software searches an internal data model for shuttles which it may meet two tracks ahead in order to initiate a communication and coordination between the shuttles. The communication, required to keep this data model up to date, is out of the scope of this paper. Details can be found in [9, 10].

[1] www.railcab.de/en

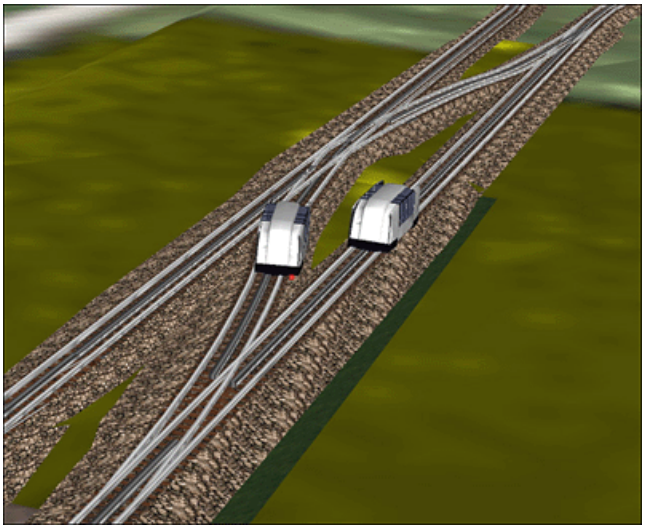

Figure 1: A possible collision of two shuttles at a switch

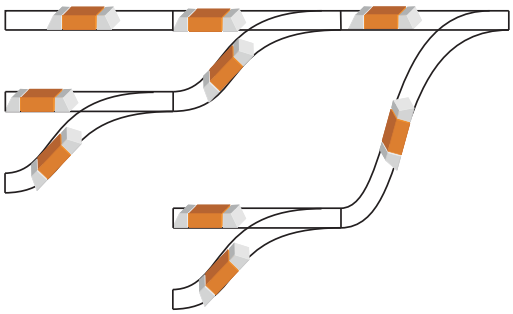

Figure 2: Extended scenario

In the following section, we present the language which can be used to specify the structure and behavior of the software for this task.

3. LANGUAGE PRESENTATION

We present the language constructs for structural specification as well as behavioral specification for real-time systems. We start with the specification of the structure in Section 3.1. Then we continue describing the behavioral parts of the language in Section 3.2.

3.1 Structural Specification

We employ UML class diagrams [14] as a standard notation for the specification of structures. Because our approach must be applicable in the embedded system domain, we have to deal with restricted computation capabilities, especially less memory.

Consequently, we refine UML class diagrams in order to employ them in this embedded domain. In embedded systems, it is required to know the exact amount of required memory beforehand to avoid unsuccessful memory allocation. The number of class instances and the multiplicities of associations are the sources for unknown memory requirements in UML class diagrams.

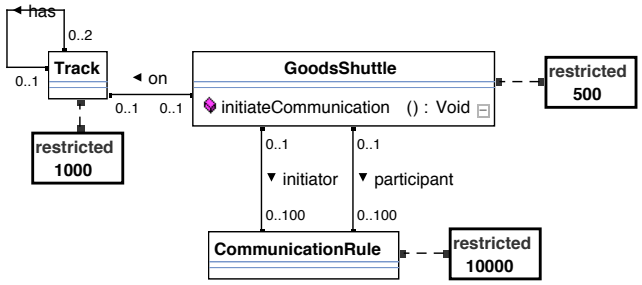

Figure 3: Extended class diagram of the depot system

It is obvious that we need to require the specification of an upper bound for the number of class instances. In the generated code, this upper bound is enforced by an implementation of the factory design pattern [8]. During initialization of a factory of a specific class, all instances of the specific class are allocated. This avoids the allocation of heap memory at runtime. The factory is then employed to *create* new class instances and reuse *deleted* class instances.

Figure 3 shows the UML class diagram refinements considering our shuttle system case study. Note that the defined maximal number of instances can also be described by OCL 2.0 [15] constraints. The following OCL 2.0 constraint specifies the maximal number of instances for the class Track.

```
context Track inv:
   Track.allInstances().size() <= 1000
```

Secondly, unbounded association multiplicities lead to unknown memory requirements for the data structures which implement the association. Thus, we prohibited the specification of unbounded association multiplicities and require fixed ones. In addition, the generated code for the association uses the upper bound to provide a static size of the data structure at the code level.

3.2 Behavioral Specification

Based on the structural model, we use the Story Diagram [7, 22] formalism for the specification of structural transformations and the control flow between the transformations. Story Diagrams are an extension of UML activity diagrams. In addition to standard UML activity diagrams, activities in Story Diagrams can be Story Patterns which specify structural transformations. Story Patterns are based on the graph transformation formalism [17]. The class diagrams presented above resemble a type graph. Story Patterns are the formalism which are used to transform instances of this type graph.

3.2.1 Real-Time Story Diagrams

We have shown how Story Patterns can be used in a real-time environment in [4]. In the following, we present a number of extensions in order to use Story Diagrams in a real-time context.

3.2.1.1 Transformation Feasibility Check.

The basic requirements of Real-Time Story Patterns are fixed maximum multiplicities for association roles and class instances in the class diagram. These requirements are

needed for both guaranteeing a fixed WCET and ensuring memory requirements.

Creations of links and objects are specified in the right hand side of a Story Pattern. We want to support the standard way of specifying instance and link creations in Story Patterns but must cope with creation failures when the above mentioned constraints would be violated. A creation failure leads to the situation where the left hand side has been successfully matched but the matched subgraph is not or only partly transformed to the right hand side.

We propose a syntax element which is used for the specification of checks whether an object or an link can indeed be created. This check is done during matching the left hand side. Only if all creation checks succeed in addition to the normal binding, the left hand side of the Story Pattern is considered to be matched. Thus, it is guaranteed that a subsequent creation of an appropriate object or link will be successful.

We add a new transition type Creation Failure to Story Diagrams to give the developer the possibility to react on a failed creation check. That transition is only taken when the left hand side has been matched, but at least one creation check has failed (see Figure 4).

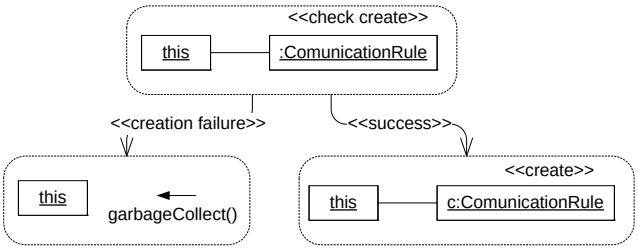

Figure 4: Creation Check Extension

Another solution is to implicitly execute the creation check before each object or link creation. We decided against that solution in order to give the developer explicit control about the situation. We are currently evaluating whether we need different transitions for different failed link and object creations. Then, the developer can specify different behavior for each possible creation check failure. We are currently in the process of adding the proposed syntax element into Story Diagrams and its accompanying code generation as well as integration into the worst case estimation.

3.2.1.2 Loops.

UML activity diagrams allow the specification of loops in the control flow. As Story Diagrams share the control flow concepts of UML activity diagrams, loops can also be specified in Story Diagrams. It is often required that one or a set of graph transformations should be applied to all possible bindings of a left hand side of a graph transformation. For this special case, the special loop concept for-each has been introduced in Story Diagrams. The for-each loop is executed for all possible bindings of the left hand side (see Figure 5).

Loops impose a challenge for WCET estimation as the WCET of a Story Diagram obviously depends on the maximal number of loop iterations. In the general case, the maximal number of iterations need to be obtained from the loop code. Fortunately, the for-each-loop construct allows us to compute the maximal number of iterations as this corresponds to the

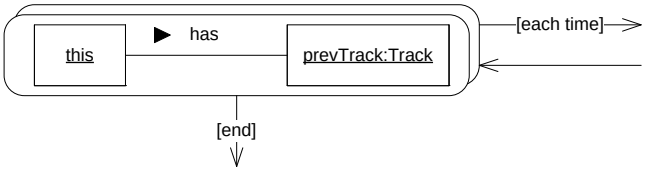

Figure 5: Example of a for-each-loop

maximal number of bindings for the left hand side of the for-each Story Pattern. The maximal number of bindings is computable from the specified maximal multiplicities of associations and classes which we already required due to memory constraints (see Section 3.1). Considering Figure 5, we compute that at most two bindings can exist for the left hand side based on the association's multiplicity specified in the class diagram. Consequently, the activities which are reached via the each time transition can only be executed two times.

In [22], a *fresh matches* semantics is employed for for-each loops. The *fresh matches* semantics means that if during execution of the for-each loop new bindings are created, those bindings are additionally matched in the loop condition. Thus, the number of loop iterations cannot be deduced from the class diagram and the loop may even not terminate at all. This behavior is obviously not suitable in the context of real-time systems. Thus, we currently prohibit that the activities inside the for each loop do create additional bindings. Additionally, we propose to employ the *pre-select* semantics [22] to avoid the inherent problems of the *fresh matches* semantics which are presented in detail in [19].

We allow other loops in the control flow, but we rely then on input from the developer about the maximal number of iterations.

3.2.1.3 Example.

The Story Diagram specifies how to find possible hazardous situations which have to be avoided. Possible collisions of two shuttles are hazardous situations. A situation needs to be detected where still enough time is left to initiate avoidance mechanisms. We simplify this by declaring that enough time is left, if two shuttles may meet at one of the next two switches.

To avoid a possible collision, we require that a shuttle which is detecting the possible hazardous situation, creates a communication rule. This communication rule guarantees, that no collision happens. The complete collision avoidance is not part of this paper, but details can be found in [11].

3.2.2 Real-Time Story Charts

Story Charts [13, 22] are a formalism which blends the specification of event-based behavior (e.g. Statecharts [12], UML state machines [14]) with the specification of structural transformations based on graph transformations. The structural transformations are executed in order to react to incoming events.

In previous works, we refined UML state machines for the specification of event based behavior in the hard real-time domain [2, 3] into the Real-Time Statechart formalism. In order to blend Real-Time Statecharts with graph transformations similar to Story Charts, we can simply use the above described Story Diagrams as side effects of transitions

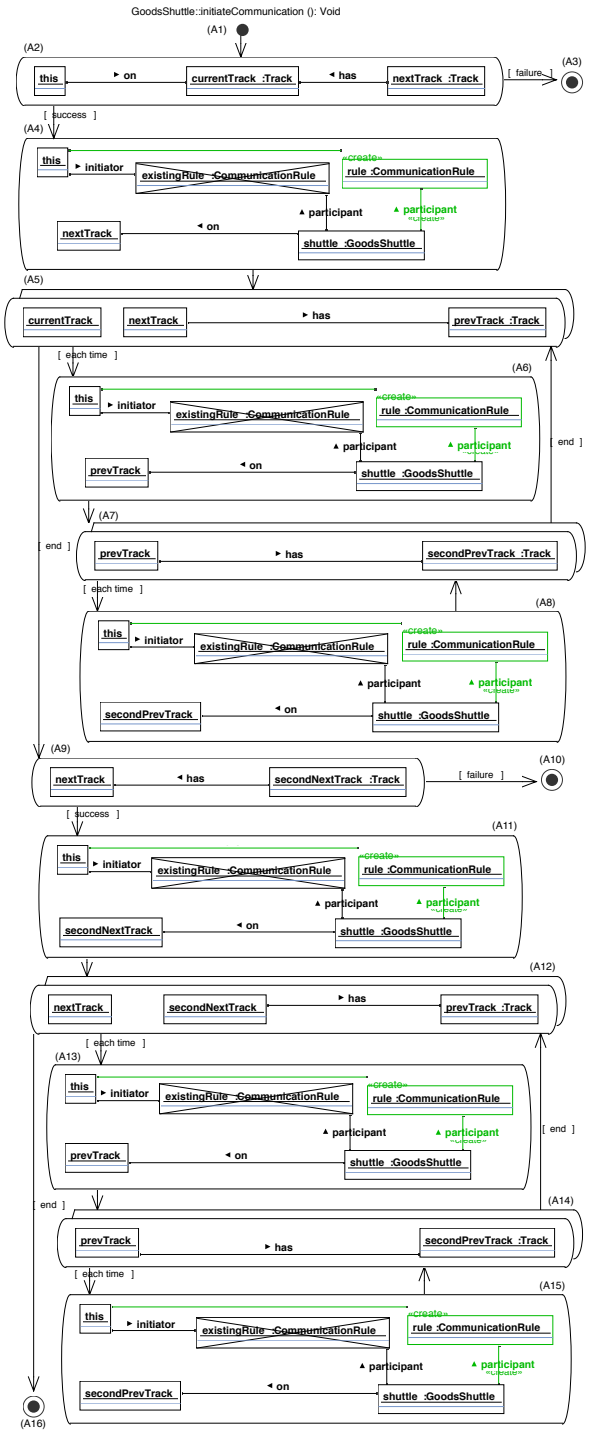

Figure 6: Story Diagram for collision avoidance

as well as entry and exit actions.

The Real-Time Statechart formalism requires that for each side effect the WCET is known. If this requirement is fulfilled, schedulability of the specified behavior can be checked and code can be subsequently generated. In the following, we present how WCET's can be estimated from Story Diagrams.

3.2.2.1 *Example.*

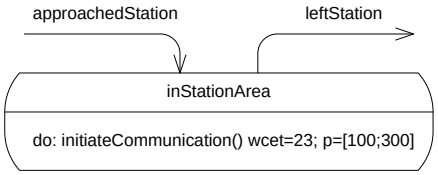

Figure 7: Part of the Real-Time Story Chart Diagram for the shuttle behavior

Figure 7 shows an extract of the event-driven behavior of a shuttle. The state inStation is via an incoming approachedStation event. This event stems from another component of the shuttle. The initiateCommunication story diagram of Figure 6 is executed periodically (every 100 to 300 ms) as long as this state is the current state. The state is left after the leftStation message is received.

4. WORST-CASE EXECUTION TIMES ESTIMATION

The WCET estimation of Story Diagrams is composed into two parts. First, the WCET of every Story Pattern is estimated. In previous work [4], we have presented the analysis steps which are required to estimate a WCET for Story Pattern. This approach is based on a platform specific *profile* which contains WCET's for a set of basic operations like checking whether an object is contained in a linked list or not. Our *profile tool* does a fully automated measurement for creating this profile before analyzation – at the moment it supports C++ and the phyCORE-MPC555 board. Another possibility to obtain these WCET's is to use a special tool like the aiT Worst-Case Execution Time Analyzer [20]. It may also be gathered from several tests on the target platform under certain circumstances. The WCET of the complete Story Diagram can then be computed based on the individual Story Pattern's WCET's.

But the WCET of every Story Pattern only, is not sufficient. We need the worst case number of possible bindings of loop condition Story Pattern in order to compute the number of iterations of the for-each loop. The worst case number of possible bindings is identical to the worst case number of iterations (WCNI) described in [4]. For other loops, the maximal number of loop iterations specified by the developer is used (see Section 3.2.1.2).

Figure 8 visualizes the entire WCET estimation process.

4.1 WCET Estimation for Story Diagrams

After the first step of estimating the WCET and the WCNI of every Story Pattern, the analysis can proceed estimating the WCET of the Story Diagram. Our static approach for estimating the WCET of Story Diagrams is path based on the control flow of the Story Diagrams. Story Diagrams are an extension of UML activity diagrams. As UML activity diagrams, every Story Diagram consists of exactly one start activity and several stop activities. Between these activities there are story activities, which could be Story Patterns or branches (NOP Activities), connected by transitions which

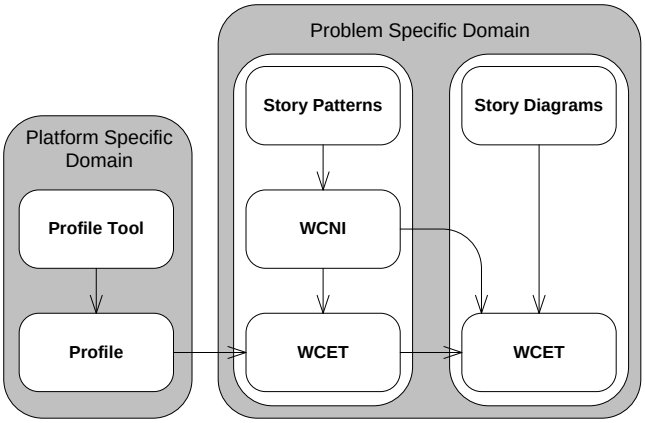

Figure 8: Analysis process

describe a control flow (see Figure 6). Other approaches using low level path based analysis have to extract a control flow graph (CFG) from the executable assembler code, for example [20]. Our model already implies a CFG with all necessary informations for WCET estimation.

To estimate the WCET of every stop activity, we use a recursive algorithm which estimates the WCET's for the longest path from the start activity to each stop activity by summing the WCET of every Story Pattern on any path. The path with the maximum WCET then determines the WCET which is associated with the stop activity.

Loops are recognized by the algorithm. We use the number of possible bindings of the loop condition Story Pattern (the for-each Story Pattern) in order to determine the number of loop iterations. This information is given by the WCNI of a for-each Story Pattern which is the initiator of a loop. A for-each Story Pattern has two outgoing transitions: (1) every time a binding has been found for the for-each Story Pattern, the each-time transition is taken, (2) if all bindings have been processed, the end transition is taken.

For a non-nested loop the WCET of the for-each Story Pattern and all contained Story Patterns is simply multiplied by the number of loop iterations.

Loops can be nested. Thus, we additionally need to recursively increase the WCNI of the inner loop by multiplication with the WCNI of the outer loop. This principle is called for-each compensation and is depicted in Figure 9.

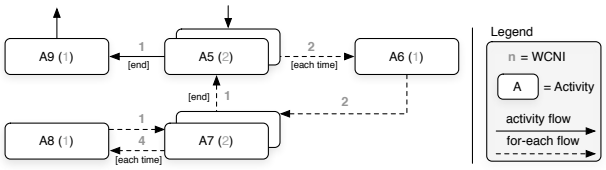

Figure 9: Example of the for-each compensation process

This figure shows a simplified version of the Story Diagram of Figure 6. The activities are numbered from the top to the bottom with A1 to A16. The for-each compensation process maps the WCNI's of the for-each initiator Story Patterns to the transitions of the according for-each flows. Initially, every transition has a WCNI of 1. The for-each compensation multiplies the WCNI of the for-each Story Pattern with every transition on the for-each flow except three situations. First, the end-transition is ignored. Second, transitions which lead back to the initiating for-each Story Pattern are ignored. At last, transitions that go out of another for-each Story Pattern are ignored. The WCNI is multiplied with the WCET of a story activity which is part of the for-each loop.

Because in some cases the user does not want to iterate over all matched instances on the left hand side of a for-each Story Pattern, the user may define an object as already bound as done in our example in Story Pattern A5 and A12 with currentTrack and nextTrack. This has, for example, the effect that the matchable objects for nextTrack contain an object that is equal to the currentTrack in Story Pattern A5. Due to the isomorphic binding of Story Patterns, this object will not considered in further matchings of the Story Pattern again. This reduces the possible number of matchable objects which implies a reduction of the number of iterations of the Story Pattern. Our analysis recognizes such patterns and corrects the WCNI of the affected Story Pattern automatically.

5. EVALUATION

Because we do not consider the issues of multi-threading, process scheduling and processor caching in our approach, we needed an appropriate hardware and software platform. Consequently, we used a phyCORE-MPC555 with integrated 40MHz 32-Bit PowerPC micro-controller from Motorola and the embedded operating system DREAMS[2][6] for the runtime measurements of our example of Section 2. We use the Story Diagram of Figure 6 as the scenario for our evaluation.

5.1 Worst-Case Instance Situation

To show the quality of our WCET estimation of Story Diagrams, a worst-case instance scenario has to be created in order to measure the WCET of the executed example application. Therefore, an initial instance situation has to be created in a way that every Story Pattern of the Story Diagram of Figure 6 executes as most as possible operations. It is not enough that the left hand side is successfully matched and the right hand side successfully executed. Also, the employed data structures have to be initialized in a way that searching an element results in the data structure operation's WCET. Figure 10 shows a snapshot of the initial instance situation of our generated example application.

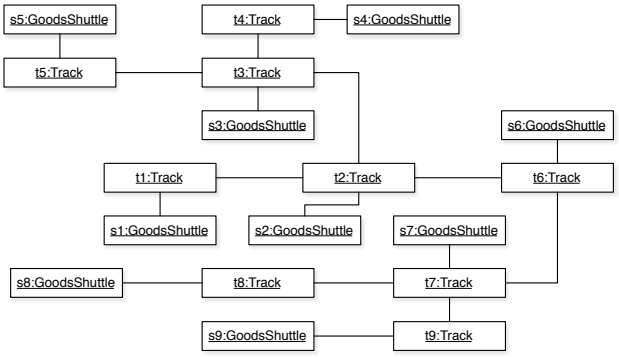

Figure 10: Initial worst-case instance situation

[2]Distributed Extensible Application Management System

For a clearer presentation, Figure 10 does not include objects of the type CommunicationRule. GoodsShuttle s1 is the initiator of 92 other CommunicationRule objects. This leads to the WCET for the creation of a link between s1 and another CommunicationRule object since link creation implies searching for a possible duplicate in the data structure. We cannot initially create 100 CommunicationRule objects because the Story Diagram is going to add 8 more CommunicationRule objects to the associated data structure of the initiator association of s1 during successful execution.

Normally, we should leave 14 slots free of the initiator's associated data structure in s1, but we use the optimization concerning isomorphic checks (see Section 4.1). The Story Pattern A5 and A12 use isomorphic checks to prevent further infeasible matches like currentTrack in Story Pattern A5 which is already bound in Story Pattern A2 and checked for a possible collision in Story Pattern A4. Therefore, nextTrack would be matched in Story Pattern A5 and then checked for a possible collision in Story Pattern A6 again. The for-each loops initiated by Story Pattern A5 and A12 therefore only match one object of the type Track instead of two possible matches. The analysis recognize the isomorphic matches and reduce the for-each iteration number by the number of isomorphic situations in the initiator for-each Story Pattern. Thus, only 8 CommunicationRule objects can be created and added to the initiator associated data structure of s1 during execution of the Story Diagram.

Also every GoodsShuttle object except s1 is a participant of 99 other CommunicationRule objects. Those CommunicationRule objects are not connected by an initiator link with s1. This results in the WCET for creating a participant link between a GoodsShuttle object and a CommunicationRule object and creating a link between this CommunicationRule object and s1. Additionally, this implies the worst case for the negative application condition (NAC) check between a GoodsShuttle and the existingRule object of the type CommunicationRule.

The Story Patterns A4, A6, A8, A11, A13 and A15 of Figure 6 require a situation where another GoodsShuttle object on a neighboring Track object is located which has a participant link to another CommunicationRule object. Thus, most of the code of the NAC is being executed, but not finished with true. This means that the left hand side can be matched with most of the checks being executed. Thus, every GoodsShuttle object except s1 is a participant of another CommunicationRule object.

5.2 Analysis and Measurement

The evaluation has to show the quality of our WCET estimation from our analysis in direct comparison to the measured WCET of the example application. We present three of our experiments with the Story Diagram of Figure 6.

In the first experiment, we looked at the WCET of both the left hand side and the right hand side (LHS + RHS). The second experiment does only consider the left hand side of the Story Diagram without the transformation concerned by the right hand side (LHS). The difference between these experiments is the WCET of the right hand side only (RHS).

Every experiment has been executed 10 times. In every step, we increased the multiplicity's of the associations participant and initiator at the CommunicationRule side by 10. We started with multiplicity 1 and finished with the multiplicity 100. The experiments with multiplicity 1 must be distinguished from the ones with multiplicity > 1, because one-to-one associations and different matching sequences as well as other operations are used inside the NAC's binding objects and during the execution of the right hand side (cf. [4]).

We only increase the multiplicity's of the association initiator and participant. This has the effect that the WCET is increasing linear with increasing multiplicity, because we do not increase the number of nested loop iterations. Thus, we can interpolate the analyzed and measured data as linear functions of the form $f(x) = a \cdot x + b$ with $x \in \mathbb{N} \backslash \{0, 1\}$. We introduce three linear functions which describe the WCET of our analysis. $a(x) = 0.2321 \cdot x + 0.5035$ which is the analysis of the complete example, $a_l(x) = 0.2045 \cdot x + 0.2577$ describes the analysis of the left hand side only and $a_r(x) = 0.0276 \cdot x + 0.2458$ describes the right hand side only.

Also the measurement data of every experiment can be interpolated. Thus, we present the experimental results as a linear function. $m(x) = 0.1968 \cdot x + 0.4447$ which is the WCET of the complete example, $m_l(x) = 0.1695 \cdot x + 0,2355$ is the WCET of the left hand side only, and $m_r(x) = 0.0273 \cdot x + 0.2092$ is the WCET of the right hand side only.

Figure 11 plots these functions in a cartesian coordinate system. The x-axis is the multiplicity of both associations initiator and participant. The y-axis is the WCET in ms.

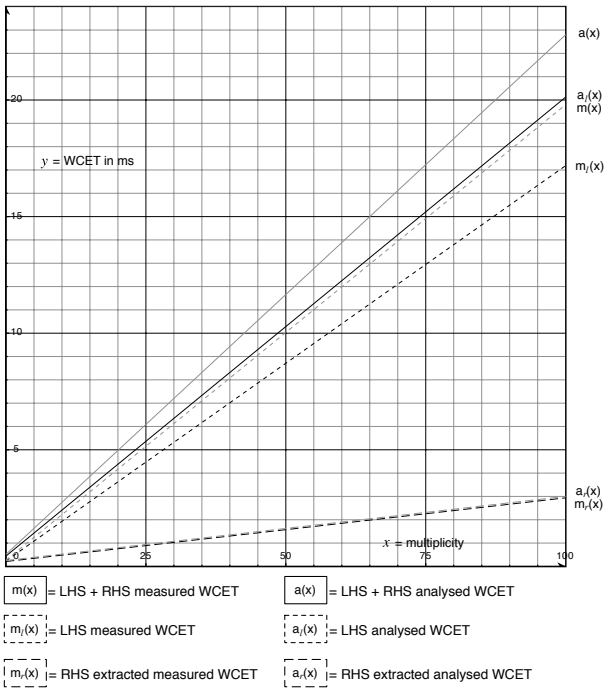

Figure 11: Evaluation chart

To get a better overview about the cornerstones of the analyzed and measured data, Figure 12 shows the WCET of every experiment at the first- and last step only.

5.3 Evaluation Conclusion

The difference between the experiments with $x = 1$ and $x > 1$ is a result of different operations which are used for matching and transformation as described in Section 5.2. For the $x = 1$ case we see a remarkable smaller relative difference between the measured and analyzed RHS exper-

	Measurement	Analysis	Difference (A-M)
LHS + RHS (x = 1)	0.643 ms	0.686 ms	0,043 ms (6.6%)
LHS (x = 1)	0.323 ms	0.324 ms	0.001 ms (0.03%)
RHS (x = 1)	0.320 ms	0.362 ms	0.042 ms (13.1%)
LHS + RHS (x = 100)	20.124 ms	22.788 ms	2.664 ms (13.2%)
LHS (x = 100)	17.171 ms	19.772 ms	2.601 ms (15.1%)
RHS (x = 100)	2.953 ms	3.016 ms	0.063 ms (2.1 %)

Figure 12: Evaluation table

iment as the $x > 1$ case shows. This may occur because at the moment we do not have optimized every code fragment of the profile measurement. We lose 5.25 μs per Story Pattern which contains a right hand side. This overestimation is caused by three operations: one object creation and two link creations. The object creation operation is already optimized in the profile tool, so that the overestimation per create link operation should be 2.625 μ. The LHS experiment of the $x = 1$ case does only contain optimized operations. The $x > 1$ case does contain more unoptimized operations than the $x = 1$ case. Several code fragments contain too much measured code, at the moment. This could be a reason for the overestimation of the LHS experiment in the $x > 1$ case. Different to the RHS of the $x = 1$ case, the $x > 1$ case does only use optimized code fragments which should be the reason for a lower aberration between measurement and analytical results. The existing aberration for every experiment of the $x > 1$ case could be the time which is caused by the instance situation not being the worst-case (cf. Section 5.1).

In general, it has to be said that the clock of the system is not as precise as needed. A tick with has a execution time of 800 ns. Some code fragments oscillate between one and two ticks which cause roundoff errors. The code fragment WCET estimation optimization is reducing this error by executing the code δ times. With increasing δ, we are able to get a refined timing with decreasing roundoff errors. Our evaluation used $\delta = 100$ for the code fragments.

6. RELATED WORK

Several approaches and tools for WCET estimation, like aiT from AbsInt[3], already exist. Most of them handle low-level analysis to estimate the WCET of executable code [20]. These approaches have to obtain flow-graph analysis from the executable code in order to appropriate cycles and recursions in the code flow. The user can obtain upper bounds for determined cycles. Also the code language is often restricted to procedural languages like C in aiT, for example. Using executable code offers a finer granularity in WCET estimation because parts of the flow-graph can be described more detailed. Our approach obtains the WCET for whole code fragments which is a well defined sequence of code. The main aspect of our approach is to obtain an high-level analysis to appropriate WCET's. Therefore obtaining basic WCET's from the code fragments is satisfying our necessities. The input of our analysis is the model which is an equivalent to the flow-graph and a target profile containing the code fragments WCET's.

Curatelli and Mangeruca present in [5] a method to compute the number of iterations in data dependent loops. They present a formal model for the loop condition as well as the loop body. The formal model of the loop body is very expressive due to a set of loop index counters and transformations of the loop index counters which include other loop index counters on the right side of the assignment.

The body of the loops of Story Diagrams as well as the generated loop bodies for binding the left hand side of Story Pattern can be mapped to this formal model. For example, the binding-loops of Story Pattern can be mapped to a set of loop index counters for each to-many association. In the loop body this index counters are then increased. Unfortunately, the formal model of the loop condition is not expressive enough for our task as the loop condition does not allow to set individual maximum values for each index counter but only a linear constraint containing all index counters.

7. CONCLUSIONS

Embedded software must satisfy strict quality requirements. i.e the software must work correctly w.r.t. functional requirements as well as real-time requirements. Many different textual and graphical approaches exist to help in developing those software. We presented an approach for the special case of graph transformations in embedded real-time systems.

Our approach contains means for structural specification in form of refined UML class diagrams. Structural transformations are specified using Story Diagrams. Additionally, we propose a syntactic extension which allows to check whether creation of elements is indeed possible despite tight memory constrains.

We present how to estimate worst case execution times for Story Diagrams in order to ensure temporal correctness. The WCET estimation of Story Patterns is based on [4]. The WCET of a Story Diagram is then computed based on the Story Pattern's WCET's. Loops in Story Diagrams are specifically treated based on known maximum number of iterations of the loop. Thus, Story Diagrams can be used as side effect in Real-Time Statecharts.

The provided evaluation shows that our approach is a safe (pessimistic) approximation of the WCET's of the presented example scenario.

The presented approach is part of the Fujaba Real-Time Tool Suite. We are currently implementing the mentioned language extension concerning the checking of create operations. We offer a C++ code generation which was used to generate the code for our example scenario.

7.1 Future Work

We are currently looking into reducing the difference between the measured execution time and the estimated one. One aspect is the refinement of the profile which is used to compute the WCET of a Story Pattern.

The estimation can be improved by an improved analysis of the behavior of the Story Pattern. For example, if an object is created in the Story Pattern, there are two cases to consider: (1) if the object creation of a class A succeeds, only maxInstances-1 instances of A can be matched in the left hand side and (2) if the object creation fails, the right hand side would not be executed at all. Another example is a series of object creations in different Story Patterns of a

[3]http://www.absint.com/ait

Story Diagram. It remains to be seen whether an improved analysis scales for bigger Story Diagrams.

The used data structures (linked lists) could be replaced by sophisticated ones which have a better execution time in the worst case. Additionally, we will continue to evaluate the approach in other scenarios in order to improve the quality of the estimation.

Acknowledgment

We thank Anatol Derksen who helped in working with the phyCORE-MPC555 running the DREAMS operating system and for his great assistance during the practical evaluation step.

8. REFERENCES

[1] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of the* 28^{th} *International Conference on Software Engineering (ICSE), Shanghai, China*. ACM Press, 2006. (accepted).

[2] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, pages 670–671. ACM Press, May 2005.

[3] 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*, Lecture Notes in Computer Science, pages 25–40. Springer Verlag, November 2005.

[4] S. Burmester, H. Giese, A. Seibel, and M. Tichy. Worst-Case Execution Time Optimization of Story Patterns for Hard Real-Time Systems. In *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany*, pages 71–78, September 2005.

[5] F. Curatelli and L. Mangeruca. A method for computing the number of iterations in data dependent loops. *Real-Time Systems*, 32(1-2):73 – 104, February 2006.

[6] C. Ditze. *Towards Operating System Synthesis*. Number 157 in HNI-Verlagsschriftenreihe. University of Paderborn, Germany, 2000.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G.Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] 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, October 2003.

[10] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pages 179–188. ACM Press, November 2004.

[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 9th European Software Engineering Conference*, pages 38–47. ACM Press, September 2003.

[12] D. Harel. STATECHARTS: A Visual Formalism for complex systems. *Science of Computer Programming*, 3(8):231–274, 1987.

[13] H. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, pages 241–251. ACM Press, 2000.

[14] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004. Document: ptc/04-10-02 (convenience document).

[15] Object Management Group. *Object Constraint Language Specification 2.0*, May 2006.

[16] A. Rensink. Towards Model Checking Graph Grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

[17] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation : Foundations*. World Scientific Pub Co, February 1997. Volume 1.

[18] G. Täntzer. AGG: A Tool Enviroment for Algebraic Graph Transformation. In *Proc. of Applications of Graph Transformation with Industrial Relevance (AGTIVE2000), Kerkrade, The Netherlands*, LNCS. Springer Verlag, 2000.

[19] M. Tichy, H. Giese, and M. Meyer. On Semantic Issues in Story Diagrams. In *Proc. of the 4th International Fujaba Days 2006, Paderborn, Germany*, September 2006. submitted.

[20] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 606–611, Washington, DC, USA, 2005. IEEE Computer Society.

[21] A. Zündorf. Graph Pattern Matching in PROGRES. In *Proc. of the* 5^{th} *International Workshop on Graph-Grammars and their Application to Computer Science*, LNCS 1073. Springer Verlag, 1996.

[22] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001.