ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/295402912

Performance Evaluation for Collision Prevention Based on a Domain Specific Language

Conference Paper · September 2013

DOI: 10.1007/978-3-642-40725-3_21

CITATIONS
9

READS
249

4 authors, including:

Freek van den Berg
University of Twente
11 PUBLICATIONS **76** CITATIONS
SEE PROFILE

Anne Remke
University of Münster
98 PUBLICATIONS **800** CITATIONS
SEE PROFILE

All content following this page was uploaded by Freek van den Berg on 27 September 2017.

The user has requested enhancement of the downloaded file.

Early Performance Evaluation for Collision Prevention based on a Domain Specific Language

Freek van den Berg[1], Anne Remke[1], Arjan Mooij[2] and Boudewijn Haverkort[1]

[1] DACS, University of Twente, Enschede, The Netherlands
[2] Embedded Systems Innovation by TNO, Eindhoven, The Netherlands

Abstract. The increasing complexity of embedded systems makes early-in-design performance evaluation important. We have constructed a performance model for this in the Parallel Object-Oriented Specification Language (POOSL). In a case study, we have contributed to an industrial project aimed at redesigning the movement control loop for collision prevention of interventional X-ray machines. Distance computations were identified as performance critical. We have made (i) a transformation from a domain specific language with system specification to POOSL; (ii) profiles of Proximity Query Package execution-times, our chosen distance query package. PQP varies for object complexities and relative positions of objects and therefore requires extensive profiling. The model has been successfully validated by comparing its simulations with movement control loop executions.

1 Introduction

Embedded systems are ICT systems designed to operate as part of larger systems and are commonplace in dependable, critical sectors, like aerospace, automotive, health-care and robotics. They have faced a significant complexity increase over time, caused by an ongoing exponential growth of hardware and software. Furthermore, they have faced increased interaction in a dynamic, interactive environment. External interactions of embedded systems are often implemented using control loops consisting of three (indefinitely running) functional steps: sensing (extracting environment information), thinking (from information to decision making) and acting (from decisions into observable behavior).

Control need to meet performance requirements (throughput and latency) that should be an integral part of the system requirements [1]. Model-based and model-driven design methods have been proposed to improve the complex design process for embedded systems [7] [18] [13], but addressing the performance aspects remains difficult. Particularly, predicting performance in the early design stages is hard, since the system does not exist yet [19] [4]. This paper addresses early-in-design performance evaluation using a DSL.

[1] This research was supported as part off the Dutch national program COMMIT, and carried out as part of the Allegio project under the responsibility of the Embedded Systems Institute with Philips Medical Systems B.V. as the carrying industrial partner.

We report about our contributions to an industrial study project at Philips Healthcare aimed at redesigning some of the collision prevention components, used in their interventional X-ray (iXR) machines[1]. Collision prevention strategies vary across product configurations and medical applications. In the Allegio project, we strive for reuse-ability across product configurations and hence started with specifying the safe functionality using a DSL. In the study, a prototype domain specific language (DSL, [23] [17]) for collision prevention was developed. A DSL instance is a formal system specification, from which executable code is generated; see Figure 1 (left). Early in the study, distance computations (implemented by different package) were identified as performance critical. In the context of robotics, various contributions [3, 16] about the use of the Proximity Query Package (PQP, [15]) for distance computations exist. We have decided to use PQP to illustrate our approach.

In this paper, we add early (design phase) DSL-performance evaluation to the basic DSL approach, before the embedded system even exists; see Figure 1 (right). In addition to the basic approach, the DSL is reused to create a performance model to generate up-to-date performance metrics dynamically. Hence, the performance of DSL-instances can be compared automatically. Formal specification mechanisms that allow reasoning about product families and enable design space exploration exist [20], [24]. DSL-performance evaluation requires an automatic transformation from DSL-instances to performance models, applicable to any valid DSL-instance. We used the Parallel Object-Oriented Specification Language (POOSL, [6, 22]) as modeling language and derive functional-flows of executable functions from DSL-instances. To obtain insight in the execution-times, we added PQP-profiles and use cases as extra model parameters.

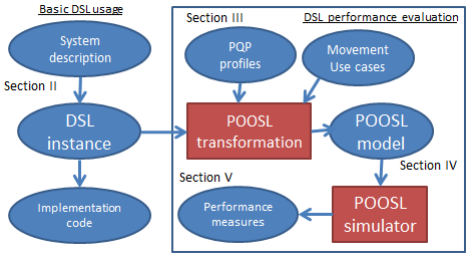

Fig. 1. Basic DSL usage (left), extended with performance evaluation (right)

Related work For the performance evaluation of embedded software, a variety of techniques has been used recently, both from the more traditional field of queueing theory, as from the field of embedded system design. A good overview on techniques for software performance evaluation using a model-based approach is provided in [1]. We address a few recent case below, without attempting to be exhaustive. Process algebra models, in particular PEPA, have been used for the evaluation of an industrial production cell [8], whereas [2] uses PEPA to specify active badge models to comute so-called passage times. Three different approaches, i.e., timed Petri nets, data flow graphs (SDF) and timed automata

(UPPAAL) were compared for the evaluation of an image processing pipeline [9]. The Petri net approach provided the most expressive modelling framework, UPPAAL was most adequate in finding schedules and SDF appeared to be most scalable. [10] describes the evaluation of a printer datapath, where UPPAAL is used to compute worst-case completion times. Using PRISM, a CTMC model of an embedded system is evaluated and shutdown probabilities are obtained [14]. POOSL [6] supports a UML based modelling approach; two recent case studies papers address a production cell model [12] and an in-car navigation system model [21]. In the current case study, a soft real-time system in which consecutive distance query functions need to be executed is modeled. A worst-case analysis seems less appropriate here, as this assumes the extremely unlikely case that all functionality under-performs coincidently. We therefore turn our attention to discrete-event simulations, supported by the tool POOSL, which provide us with useful statistical results.

Paper outline This paper is further organised in five sections matching the components of Figure 1 (right). Section 2 describes the case study system. Section 3 specifies the PQP-query profiling, followed by Section 4 that presents the full DSL-based POOSL model. Section 5 validates the POOSL performance-model by comparing it to the real system. Section 6 concludes the paper.

2 System description

iXR systems are used for acquiring patient images using x-ray technology. They consist of large and heavy objects (Figure 2) that translate and rotate based on user input. Collision prevention is vital for safety of the patient and implemented by a movement control loop. We focus on one collision prevention technique based on 3D-models with computing the shortest distance between two 3D-objects as a key operation.

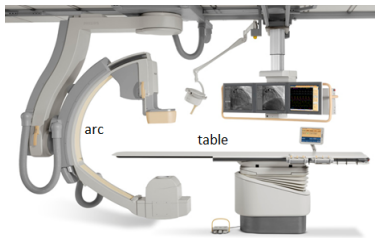

Fig. 2. Overview of an iXR system; the "table" on which the patient lies can be translated in 3 dimensions; the "C-arm" (with radiation and image capturing equipment) can rotate around the patient in various ways and at a relatively high speed.

The movement control loop tracks object-distances frequently. It intervenes in system operation when two objects are too close to each other by overriding user speed-requests by lower ones and demands stable and low response times to ensure timely and reasonable acting. The movement control comprises functions that execute using a FIFO scheduling policy in a single-threaded, sequential and non-preemptive manner. At its highest abstraction level, the movement control loop is decomposed into three succeeding functions, named `Sense`, `Think` and `Act`

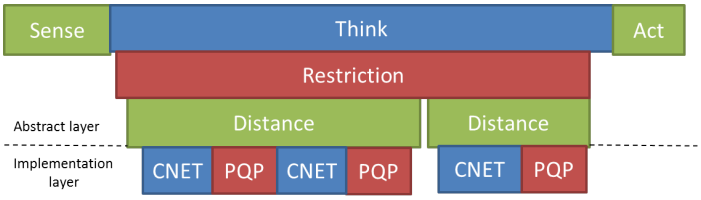

Fig. 3. The structure of the movement control's functions

(Figure 3). The functions are, respectively, responsible for reading geometric sensor positions, reasoning about current geometric positions and decision taking, and sending object speed-requests. `Sense` and `Act` are atomic functions by design. The more complex function `Think` has recently been redesigned using a DSL of which instances are automatically transformed to executable code and that are much smaller than traditional executable code. We would like to know whether for a given system configuration and in a variety of scenarios, `Think` executes quickly enough to ensure safety at all times.

`Think` is partitioned several so-called `restrictions`, each constituting of a conditional collision-danger clause and speed statement. The collision-danger clause yields true, if and only if it applies in the currently sensed situation, e.g., a small distance between objects. In this case only, the corresponding speed statement assigns a positive (translational or rotational) speed limit to a geometric object. In more detail, restrictions are implemented by a number of distance queries depending on the DSL-instance at hand. For `Think`, distance queries are the only way to sense, and speed limits the only way to act. Distance queries are performed for two geometrical objects on a specified geometric model.

At the lowest abstraction level, distance queries are executed. For our performance evaluation approach, they have been implemented using the Proximity Query Package (PQP [15]). Distance queries comprise one or more PQP calls (due to objects decomposition), each preceded by one preperatory CNET operation. The CNET operation is a relatively short and constant-time operation, relative to the PQP calls; for that reason, we do not further address it here. PQP provides a 3D-Eucledian distance between two objects, but shows variable execution-times that that are hard-to-determine a priori. In the next section, PQP is profiled to gain an understanding of its performance characteristics, so that we can use that information when building the model.

3 PQP profiling

In this section, a performance profile of PQP is created by measuring its execution times in a variety of representable circumstances, using a 3D model of a real iXR system. We create empirical cumulative distribution functions (CDFs) of the execution-times, which are sampled later for simulation purposes. PQP-queries require two sets of triangles as input and execute an heuristic algorithm

that selects one triangle per set, such that the distance between triangles is minimal. PQP returns exactly this distance. The key of PQP's performance lies in the way triangles are traversed. PQP uses bounded volumes [15] as an abstraction mechanism for objects. Therefore, PQP is faster than the $O(nm)$ theoretical worst-case (with n and m the number of triangles per object), but this comes at the price of variable execution-times that are hard-to-determine a priori.

Experimental profiling conveyed that both the complexity of the input objects and their relative geometric positions affect the performance of PQP significantly. Hence, PQP requires a differentiated way of profiling; there is not just one profile that fits all foreseen queries. In the following, we deal with object complexities first, after which the relative geometric positions are taken into account. We classify PQP-queries on the basis of input object-pairs to account for run-time variations resulting from different object complexities. In our study, we profiled PQP in isolation from the movement control loop in the 3D model. However, as a result we were not able to profile all PQP object-pairs needed for our model. To compensate for this, we estimated the unknown performance of object-pairs using a known measure for object-pair complexity, namely the product of triangles per object [11]. We then matched unknown queries with the nearest, but more difficult query to avoid underestimating profiles.

Additionally, the relative geometric positions of objects influence the execution-times of PQP. This stems from the unpredictable, heuristic way PQP searches for the two distance-defining triangles. PQP operates in a 3-dimensional geometric space in which objects can be translated and/or rotated using 12 dimensions from their starting position (leading to a total of more than 10^{35} positions). The immense geometric space demands the use of profiling methods that consider a restricted but representative sample of the space of all possible positions. We use four, named 2D, 3D, 9D grid, and 9D random. The 2D and 3D profiling methods cover geometric positions that match special test cases, which have been designed to test iXR systems in performance demanding geometric positions. Profiling using few dimensions allows for small step sizes, but is however restricted to a small part of the sample space. In contrast, the 9-dimensional methods profile more distant samples. The 9D grid and random vary in 3 of 4 positions per dimensions and select geometric positions with uniform probability, respectively. This results in a large part of the search space being covered, while potentially missing local maximums.

We took 13467 (2D), 137417 (3D), 88671 (9D-grid) and 90012 (9D-random) geometric positions on which we performed eleven PQP distance queries, one per distance pair. PQP-queries were classified for eleven object pairs and four profiling methods, resulting in 44 kinds of PQP-queries. We constructed 44 empirical CDFs for simulation, as follows:

$$\hat{F}_j(t) = \frac{1}{n_j} \sum_{i=1}^{n_j} \mathbf{1}\{x_i^j \leq t\}, \quad j = 1, \cdots, 44,$$

where n_j is the sample size, and x_i^j the execution-time of sample i for case j.

While sampling from these CDFs during simulation, we choose to not use interpolation. Instead, we "round-up" to the nearest next higher true sample and return values that have actually been observed during profiling.

To show the difference between different profiling methods, the results for PQP-query number 1, which is the most complex one in terms of object complexities and execution-time, is shown in Figure 4. As can been seen, the execution times of the 2D and 3D methods are higher than their 9D counterparts. This is a result of using test cases, which are designed to push the machine to its limits and therefore have a bias towards "difficult" geometric positions. In contrast, the 9D methods comprise unbiased sampling and take samples from the whole geometric space, which corresponds to the average case.

4 POOSL-performance model

In this section, a performance model of the `Think`-component is introduced, which is by far the most time-consuming part of the movement control loop. The model is specified in the UML-based POOSL language [6, 22], a system-level description and language that enables fast-simulation of the `Think`-component using the Rotalumis engine [5]. POOSL models comprise components that operate autonomously and concurrently, and communicate through lines that connect their interfaces synchronously.

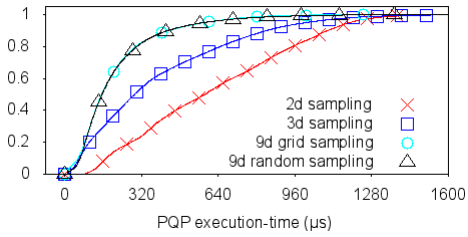

Fig. 4. Profiling PQP_1 using different sampling methods

4.1 POOSL model outline

The constructed POOSL-performance model comprises five components requiring three parameters (Figure 5). During model creation, we have introduced assumptions to keep the model simple; loops, conditional code executions and functions are assumed independent, which not necessarily reflect reality. Hence, conclusions drawn from the model need to be treated with caution.

The POOSL-model corresponds to the `Think`-part of the movement control loop. It is triggered by an incoming message at its `start`-port, after which it delegates work, via the components `Cache` and `Distance_query`, to the PQP component, by triggering the respective `start`-ports. In return, messages go all way back through the `finished`-ports to confirm successful executions. This mechanism yields a single-threaded model, in accordance with the real `Think`-component. The model components have specific responsibilities: `Think` generates distance queries to be performed and forwards them to the `Cache`. The

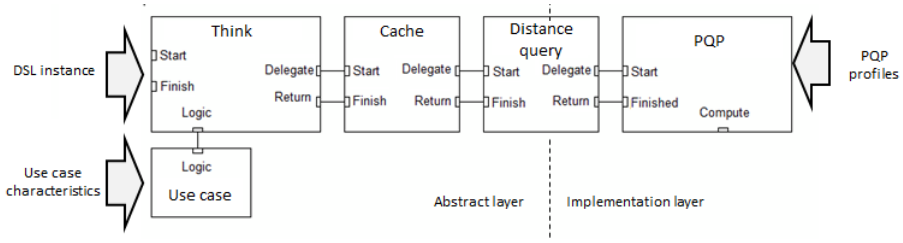

Fig. 5. POOSL performance-model of the movement control

`Use case` determines which conditional distance queries need execution. The `Cache` filters out redundant distance queries within one Think-loop to ensure intra-loop consistent distance values and enhanced performance. `PQP` simulates PQP-executions by performing delays, based on a profile.

To simulate the `Think`-component, it is required to know which functions execute and how long their executions take. Therefore, the model has been equipped with three parameters: DSL-instances, use cases and PQP profiles. We explain these parameters and their role in the performance model. First, a DSL-instance is transformed into a sequence of (conditional) distance queries that repeats indefinitely, constituting `Think`-component. For each distance query in `Think`, the `Use case`-component is accessed to decide whether it should be executed or not. The `Use case`-component is initialized with the use-case parameter, which comprises a set of use case characteristics. They refer to the conditional distance queries in the `Think`-component and represent the probabilities with which they should execute. Using random numbers, the `Use case` maps the probabilities to Boolean values, which are returned to `Think`.

Distance queries in the `Think`-component that have no cache-hit in `Cache` become PQP-queries in the `PQP`-component. The CDFs of PQP's execution-times (the third parameter) are used for sampling execution-times using random numbers. The object pair of the PQP-query determines which CDF is used, while the geometric position and thus the model are only implicitly considered by the CDF's variation.

4.2 The DSL instance

iXR machines need to make safe movements. Therefore, a prototypical DSL with a tailored grammar has been designed that specifies safe object movements by keeping track of object distances and speeds. In practice, DSL-instances are automatically transformed to executable code. They declare models and objects that form the basis of distance queries. Additionally, they consist of restrictions that impose a speed limit on an object when an activation condition is met. The condition generally contains distance queries between objects to check whether they are far enough apart. The speed limit enforces a specific object to remain below a speed, potentially overriding higher user requests. The limit can be a constant value, but can also dynamically depend on object distances. Besides the

```
1  supervisor
2
3  object TableBase, TableTop, Beam, Detector
4  model Now, Future, NowHyst, FutureHyst
5
6  restriction ApproachingTableTopBeam
7  activation
8     Distance[FutureHyst](TableTop, Beam) <18 + 125 &&
9     Distance[Future](TableTop, Beam) <Distance[Now](TableTop, Beam) - 0.3
10 effect
11    limit Beam[Rotation] at ((Distance[Future](TableTop,Beam) - 25) / 100))
12
13 restriction CloseTableTopBeam
14 activation
15    Distance[NowHyst](TableTop, Beam) <17.5 &&
16    Distance[Future](TableTop, Beam) <Distance[Now](TableTop, Beam) + 0.3
17 effect
18    limit TableTop[Translation] at 0
```

Table 1. A DSL-instance example with 2 restrictions

transformation producing implementation-code, we have constructed a second transformation from DSL-instances to POOSL-model `Think`-components that can be applied to all DSL-instances (Figure 1). It generates a POOSL `Think`-component with a fixed amount of distance queries. However, distance queries (may) execute conditionally for two reasons. First, boolean connectors are susceptible to lazy evaluation, which means that distance queries in their right hand might not execute, depending on the evaluated result of the left hand. Second, the activation may evaluate to true, which causes distance queries in the effect to not be executed. The transformation accounts for this by enclosing conditional POOSL-code fragments with if-statements. We call the conditions of if-statements use-case characteristics. Consequently, the execution of distance-queries is varies per loop.

In our case study, we used a DSL-instance with four pairs of restrictions of which two are listed in Table 1. The restrictions prevent collisions between the TableTop and Beam object. The remaining pairs are similar, but apply to other object pairs. We explain the first two restrictions. The `ApproachingTableTopBeam` (lines 6-11) activates when the objects are within a certain distance (line 8) and approaching each other (line 9). If so, the speed limit is lowered, using a monotone break pattern that maps lower speed limits to lower object distances (line 11). The `CloseTableTopBeam` (lines 5-18) activates for extremely small distances and results in a speed limit of 0, an emergency stop (line 18).

The DSL-instance is an important model parameter. Table 2 contains pseudo-code of the `Think`-component derived from the DSL-instance (Table 1) and contains seven distance queries. Both restrictions start with an unconditional distance query (pseudo code, lines 1 and 9), since the left operands of the &&-

operators execute indefinitely (DSL instance, lines 8 and 15). Next, two queries per restriction are conditional(pseudo code, lines 3,4,11 and 12), since the right operands of the &&-operators are lazy-evaluation susceptible (DSL instance, lines 9 and 16). Finally, the first restriction has a fourth distance query (pseudo code line 6) that only executes when both operands of the &&-operators to yield true, because it is in the activation clause (DSL instance, 11).

4.3 Use cases

The system's required functionality depends on its usage. Hence, we construct a use-case dependent model. We define a use-case as a set of characteristics, which consist of a label that refers to a fragment of code, and a execution probability each. Table 3 displays, for four use cases, the probability values for each label (with "other" the probability for all other labels). A label is decomposed into a fixed prefix "r", restriction number (1 to 8) and restriction part (activation or limit) in line with the pseudo code (Table 1). The corresponding value represents the probability that correspond fragment of conditional code gets executed. For instance, the value 0.29 of r2a for use case 1, indicates that in restriction 2 distance query 2 and 3 (pseudo code, lines 11 and 12) get executed in 29% of the cases. This is implemented in the POOSL-model by performing two-way communication. First, `Think` provides a label, after which `Use case` looks up the probability value and computes a Boolean value using a random number, which it returns.

Restriction 1 ApproachingTableTopBeam
1 $Distance[FutureHyst]$
2 if (r1a){
3 $Distance[Future]$
4 $Distance_[Now]$
5 if (r1lim) {
6 $Distance[Future]$
7 }
8 }

Restriction 2 CloseTableTopBeam
9 $Distance[NowHyst]$
10 if (r2a) {
11 $Distance[Future]$
12 $Distance[Now]$
13 }

Table 2. Pseudo code of the POOSL-Think component for two restrictions

4.4 PQP profiles

Section 3 discussed the creation of PQP-profiles to enable the sampling of execution-times for simulation. For this purpose, the profiles are injected into the `PQP`-component of the POOSL-model, which simulates the execution of a CNET-operation, for converting object to triangles, followed by a PQP-query based on a distance pair. The execution of CNET took 517 μs on average with a 58 μs standard deviation. We model this as an uniform distribution U(459,575) distant-pair invariant. The `PQP`-component receives distance queries (with an object pair parameter) from `Think`. `PQP` selects the right CDF (belonging to the object pair), draws a sample from this CDF and simulates a CNET-operation followed by a PQP-query (using a POOSL delay). An acknowledgement is then

sent back to `Distance Query`. We have contributed to the POOSL-language by constructing a new distribution subclass for empirical CDFs.

5 Validating the movement-control performance model

In this section, we asses the validity of the performance model. We will refer to `Think` as `Think`-prototype, as it is based on a DSL-prototype.

Label	r2a	r2lim	r3a	r4a	r4lim	other
UC1	0.29	0.99	0.02	0.11	0.71	0
UC2	0.07	0.04	0	0	0	0
UC3	0.01	0.79	0.01	0.12	0.62	0
UCΩ	1	1	1	1	1	1

Table 3. Four use case 1,2,3 and Ω: characteristics (label and probability value)

Experimental set-up To reach an experimental set-up, we select values for the three parameters of the POOSL-model. First, we have used the shown DSL-instance (Table 1) as DSL-instance parameter, for all experiments. Second, we have used four use cases (Table 3) as use case parameter. Use cases can be seen as a sequence of geometric positions representing dynamic machine positions. These positions affect the performance of PQP-queries by affecting the relative object positions, an effect neglected by the model. Additionally, the model does take the variable execution of functionality into account. For this purpose, use cases are mapped to a numbers of characteristics (Tabel 3), which each represent the execution probability of a certain code fragment. Third, we have used simulations based on four different profiling methods, which are 2D, 3D, 9D-grip and 9D-random (Section 3). Additionally, we executed the `Think`-prototype for three use cases.

We have performed experiments uniformly (Figure 6) for three use cases, as follows: we performed a real-time execution on `Think`-prototype machine by providing (use case-driven) user inputs, for 278, 284 and 301 seconds, respectively. Next, we performed four simulations (one for each profiling method) using the use case characteristics from the corresponding execution, covering 125488, 144178 and 140618 cycles, respectively. Finally, we plotted the distributions of the simulations and execution in one graph per use case. We used one PC (i5-2400, QuadCPU@3.10Ghzm 3Gb RAM) to execute `Think`-prototype and another PC (AMD A6-3400m, QuadCPU@1.4Ghz, 6Gb RAM) to simulate the POOSL-model. In the following we discuss the results per use case.

Use case 1: Arc movements The `Arc`-object has been rotated in various ways around and towards the `Table`-object, while the `Table` remained motionless. iXR systems make pictures from various angles this way. Figure 7 (left-top) displays the experiment outcomes. Simulations based on 2D and 3D profiling show pessimistic results, whereas the ones based on 9D-profiling match `Think`-prototype better. We attribute the difference to the most complex PQP_1 query (Figure 4)

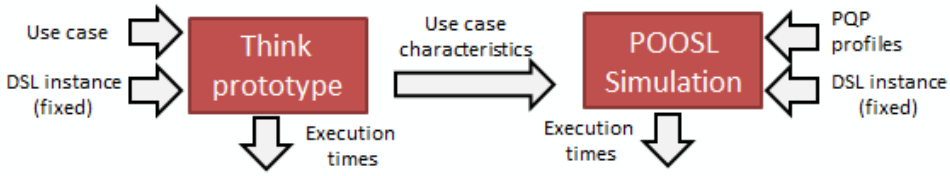

Fig. 6. The experimental setup; comparing simulations with real executions

that conveys an overestimation for 2D and 3D-based profiling. Underestimations occur for 9D-simulations, but only for low execution-times.

Use case 2: Table movements The `Table`-object has been moved towards and away from the `Arc` mimicking situations in which patients enter and leave iXR systems. The `Arc` moved very occasionally to avoid repetition. Figure 7 (right-top) shows that 9D-simulations match the `Think`-prototype execution well. Therefore, replacing unknown PQP-queries with similar ones and adding independences to the model did not affect the results much. Simulations based on 2D and 3D-profiling methods overestimate execution-times, which we again contribute to the PQP_1 profile. Again, 9D-simulations underestimate for low execution-times.

Use case 3: Stationary objects Stationary behaviour that is common for iXR machines, has been performed. Nevertheless, computations took place due to the particular DSL-instance, particularly when objects are close to each other. All stationary positions had geometric positions that resemble those in previous use cases. Figure 7 (left-bottom) displays the results for use case three. The curve of the `Think`-prototype contains three points of infliction. They are presumably caused by spending time in two clusters of geometrical positions and yield execution-times of around 12 and 16 milliseconds, respectively. On average, the simulation outcomes match the `Think`-prototype well, but is not able to deal with the points of inflictions which require use case characteristics to be dependent. In line with the previous two use cases, simulations based on 2D and 3D profiles appear again more pessimistic than their 9D-counterparts. In this case, this makes them match the `Think`-prototype well for the most part. However, they convey optimistic results around the points of infliction. Therefore, point of inflictions are a phenomenon that requires more attention.

Use case Ω: Universal machine This use case uses 1-values for all its characteristics, so that all conditional code is always executed. Consequently, machines that support use case Ω, can handle all use cases. Use case Ω has not been executed on the `Think`-prototype. Figure 7 (right-bottom) shows the simulation results for use case Ω with the highest execution-times as a result of executing more code.

We draw a two-fold conclusion from the experiments. First, applying use cases saves dramatically on hardware resources and determining the use case charac-

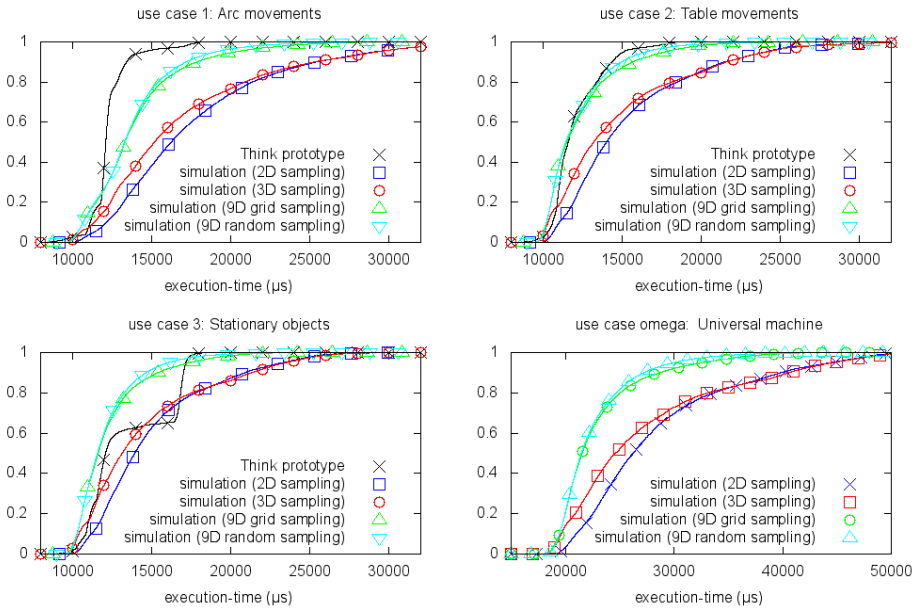

Fig. 7. The execution-time CDFs for use case 1 and 2 (top), use case 3 and Ω (bottom)

teristics is a delicate procedure that can lead to the underestimation of execution-times. Second, the execution-time ratio between use case Ω and others is DSL-dependant, since it is determined by the proportional amount of conditional code.

Kolmogorov distances We computed Kolmogorov distances (a CDF distance measure) between the `Think`-prototype and corresponding simulations to compare their results. The lower the distance, the better the simulation performed. Table 4 shows the distances between the simulations and the `Think` prototype, per use case and profiling method, computed as follows:

$K = \sup_x |F(x) - G(x)|$ with K the distance, and F and G the CDFs

Generally, the higher the number of dimensions the profiling method covers, the lower the distances are. Both 9D-simulations show Kolmogorov distances that match each other, which is confirmed by them having similar CDFs for all use cases. Since a large amount of sampling points has been used, we conclude that for the current geometrical domain it does not matter whether the grid or random method has been used.`Think`-prototype conveyed an irregular CDF with infliction points. Although the model was not able to reproduce this pattern, the Kolmogorov distances turned out low. Additionally, we determined how quick simulations converged, which means that simulating longer does not change the

results much. We computed the Kolmogorov distance between short simulations runs (0.25, 1, 15 and 75 seconds) and the longer runs of the experiments. 2D and 3D-simulations converged within a second, while the 9D-ones needed 15 seconds.

Execution-time ratios Finally, we have compared the ratios of execution-times between the simulations and the `Think`-prototype (Figure 7) and picked the maximum for each CDF, computed as follows:

$$\text{R}= \sup \{x/y | F(x) = G(y) \wedge y > 0\}$$

with R the execution-time ratio, and F and G the CDFs

	2D	**3D**	**9D grid**	**9D random**
UC1	0.67	0.58	0.41	0.41
UC2	0.42	0.30	0.16	0.16
UC3	0.29	0.18	0.27	0.30

Table 4. Kolmogorov distances between simulations and Think-prototype, per use case and profiling method

2D and 3D simulations show the biggest ratios greater than 2, which were recorded for high-execution times. On the other hand, lower ratios occurred for lower execution-times. This means that the simulations execution-times have a higher variance than `Think`-prototype.

	2D	**3D**	**9D grid**	**9D random**
UC1	2.02	2.03	1.53	1.68
UC2	1.79	1.79	1.54	1.72
UC3	1.81	1.81	1.55	1.58

Table 5. Maximum execution-time ratio between simulations and Think-prototype, per use case and profiling method.

Discussion The validity of the model has been evaluated by comparing POOSL-simulations with `Think`-prototype executions. Executions led to three use cases with their characteristics, which were used as simulation inputs. We address the concerns the reuse of execution information might raise, but have been very open about what these use cases are. Simulations yielded higher execution-times than the `Think`-prototype, resulting from the profiling methods (both the sample space selection and overestimation of unknown queries). On top of this, 2D and 3D sampling showed higher execution-times then 9D-ones, as a result of test case-based profiling. Overestimation is wanted to some extend, since underestimation might lead to non-safe machines in the very end. However, over-dimensioned hardware literally has its price. Table 5 shows the execution-time ratios. 2D and 3D-simulations lead to ratios greater than 2, while the 9D-ones remained below 1.6. Finally, simulations of various lengths have been performed and conveyed that the simulations performed were of sufficient length.

6 Conclusion

Embedded systems have increased in complexity over time, making early-in-design performance evaluation indispensable. For this purpose, we have constructed a DSL-based POOSL-performance model for `Think`-prototype of iXR

medical systems. Addtionally, we profiled distance queries and introduced use cases to make the model more situation specific. In our model, we presumed distance queries, execution of conditional code fragments and subsequent loops independent. We evaluated the validity of the model by comparing POOSL-simulations with `Think`-prototype executions. In general, the model overestimated the execution-times of `Think`-prototype with a ratio of 1.6 overestimation. Simulations converged within 15 seconds, enabling quick feedback.

We compare the approach in which a transformation from DSL to a performance model (e.g. POOSL) is used with one in which the performance model is generated manually. The former approach separates the roles of the domain expert and performance analysis expert, allowing both parties to solely focus on their areas of concern. Additionally, the DSL-transformation makes it possible to generate multiple performance model instances (e.g. to support different product families or compare design alternatives) based on different DSL-instances. A manual low-level approach does not restrict the definition of similar components in a variety of ways, making high-level changes labor-some. DSL-approaches counteract this by defining high-level concepts by default. Compared to a manual performance model, DSLs make the switch to other performance techniques easier. Similarly, a DSL-instance can, using multiple transformations, be the source of different artifacts, such as code, models and documentation.

In future work, analytical methods are applicable, such as data flow diagrams (with geometric position inputs and speed limit outputs), process algebra (summing distributions) and queuing networks (with queuing stations as execution delays). We foresee using DSLs in some cases as a possibility.

References

1. S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *Transactions on Software Engineering*, 30(5):295–310, 2004.
2. J. Bradley, N. Dingle, S. Gilmore, and W. Knottenbelt. Extracting passage times from PEPA models with the HYDRA tool: A case study. In *Proc. of the* 19^{th} *annual UK Perf. Engineering Workshop*, pages 79–90, 2003.
3. S. Carpin, C. Mirolo, and E. Pagello. A performance comparison of three algorithms for proximity queries relative to convex polyhedra. In *Proc. of Int. Conference on Robotics and Automation*, pages 3023–3028. IEEE, 2006.
4. T. de Gooijer, A. Jansen, H. Koziolek, and A. Koziolek. An industrial case study of performance and cost design space exploration. In *Proc. of the* 3^{rd} *Int. conference on Perf. Engineering*, pages 205–216. WOSP/SIPEW, 2012.
5. Eindhoven University of Technology. Software/Hardware Engineering - High-Speed Simulation of POOSL Models with Rotalumis. http://www.es.ele.tue.nl/she/index.php?select=42.
6. Eindhoven University of Technology. Software/Hardware Engineering - Parallel Object-Oriented Specification Language (POOSL). http://www.es.ele.tue.nl/poosl/.
7. T. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM 2006: Formal Methods*, volume 4085, pages 1–15. Springer, 2006.

8. D. Holton. A PEPA specification of an industrial production cell. *The Computer Journal*, 38(7):542–551, 1995.
9. G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. Vaandrager, M. Voorhoeve, S. de Smet, and L. Somers. Formal modeling and scheduling of datapaths of digital document printers. In *Formal Modeling and Analysis of Timed Systems*, pages 170–187. Springer, 2008.
10. G. Igna and F. Vaandrager. Verification of printer datapaths using timed automata. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 412–423. Springer, 2010.
11. P. Jiménez, F. Thomas, and C. Torras. 3d collision detection: a survey. *Computers and graphics*, 25(2):269 – 285, 2001.
12. H. Jinfeng, J. Voeten, M. Groothuis, J. Broenink, and H. Corporaal. A model-driven design approach for mechatronic systems. In 7^{th} *Int. Conference on Application of Concurrency to System Design*, pages 127–136. IEEE, 2007.
13. G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proc. of the IEEE*, 91(1):145–164, 2003.
14. M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. *Control Engineering Practice*, 15(11):1427–1434, 2007.
15. E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Fast distance queries with rectangular swept sphere volumes. In *Proc. of Int. Conference on Robotics and Automation*, volume 4, pages 3719–3726. IEEE, 2000.
16. F. Lingelbach, D. Aarno, and D. Kragic. Constrained path planning for mobile manipulators. In *Proc. of the* 3^{rd} *Swedish Workshop on Autonomous Robotics*, 2005.
17. M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *Computing Surveys*, 37(4):316–344, 2005.
18. B. Schatz, A. Pretschner, F. Huber, and J. Philipps. Model-based development of embedded systems. In *Advances in Object-Oriented Information Systems*, volume 2426, pages 298–311. Springer, 2002.
19. V. Sharma and K. Trivedi. Architecture based analysis of performance, reliability and security of software systems. In *Proc. of the* 5^{th} *Int. workshop on Software and perf.*, pages 217–227. ACM, 2005.
20. R. Tawhid and D. Petriu. User-friendly approach for handling performance parameters during predictive software performance engineering. In *Proc. of the* 3^{rd} *Int. conference on Perf. Engineering*, pages 109–120. WOSP/SIPEW, 2012.
21. B. Theelen, J. Voeten, and R. Kramer. Performance modelling of a network processor using poosl. *Computer Networks*, 41(5):667–684, 2003.
22. B.D. Theelen, O. Florescu, M. Geilen, J. Huang, P.H.A. van der Putten, and J. Voeten. Software/hardware engineering with the parallel object-oriented specification language. In *Proc. of MEMOCODE*, pages 139–148. IEEE, 2007.
23. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.
24. S. Wang and K. Shin. Early-stage performance modeling and its application for integrated embedded control software design. *Software Engineering Notes*, 29(1):110–114, 2004.

View publication stats