Simulative and Analytical Evaluation for ASD-Based Embedded Software

Ramin Sadre¹, Anne Remke¹, Sjors Hettinga², and Boudewijn Haverkort^{1,3}

¹ Design and Analysis of Communication Systems, University of Twente, The Netherlands
{r.sadre, a.k.i.remke, b.r.h.m.haverkort}@utwente.nl

² s.a.hettinga@gmail.com

³ Embedded Systems Institute, Eindhoven, The Netherlands

Abstract. The Analytical Software Design (ASD) method of the company Verum has been designed to reduce the number of errors in embedded software. However, it does not take performance issues into account, which can also have a major impact on the duration of software development. This paper presents a *discrete-event simulator* for the performance evaluation of ASD-structured software as well as a *compositional numerical analysis method* using fixed-point iteration and phase-type distribution fitting. Whereas the numerical analysis is highly accurate for non-interfering tasks, its accuracy degrades when tasks run in opposite directions through interdependent software blocks and the utilization increases. A thorough validation identifies the underlying problems when analyzing the performance of embedded software.

1 Introduction

Due to the increasing complexity of embedded software it becomes more difficult to find and fix all errors made in early development phases. The company Verum [19] developed a structured design method with built-in model-checking [2] that they claim to reduce the number of errors made by programmers. Several companies, amongst others Philips Healthcare (PHC), are investigating the use of Verum’s Analytical Software Design (ASD) method. However, the ASD method does not take performance into account, even though performance evaluation is necessary in an early stage of a project to predict, e.g., the expected response time of tasks in the system. The to some extent simplified ASD architecture, as discussed in this paper, organizes a software system into a tree of blocks, allowing for top-down synchronous calls and for asynchronous calls running in the opposite direction from block to block (details will follow). Asynchronous calls have non-preemptive priority over synchronous calls and synchronous calls that issue further requests cause blocking at the current block until these requests are returned to the issuing block. Related work, like Layered Queueing Networks [5] and the Methods of Layers [15], however, only covers systems where calls just run in one direction. Also Modular Performance Analysis [3, 20] cannot deal with cyclic dependencies due to synchronous and asynchronous calls running in opposite directions. Note that, even though this work directly resulted from a cooperation with PHC and Verum, its applicability is not limited to ASD structures, since similar problems arise in all areas of software analysis where calls in opposite directions are allowed.

This paper’s main contribution is a discrete-event steady-state simulator for the performance evaluation of arbitrary ASD structures and the discussion of the intrinsic difficulties of an analytical solution. The simulator computes several measures of interest, e.g., the mean response time of the system for each task, the utilization of the blocks, and the mean waiting times of the calls at each block. Due to the cyclic dependencies and the fact that an ASD block is a one-server priority system with an open and a closed queue, the ASD tree structure cannot be represented as a standard queueing system. This makes analytical performance evaluation a challenging task and also, to the best of our knowledge, there is no simulator available off the shelf for the targeted tree structure. Following [9], this paper presents a first step towards a compositional numerical analysis method for a restricted class of tasks. A single ASD block is analyzed by solving the underlying CTMC of a queueing station model. This approach can, however, not easily be extended to multiple blocks, since it results in a global (non-compositional) CTMC that is potentially infinite in multiple dimensions. Therefore, we propose a decomposition approach, based on the single block analysis, using fixed-point iteration.

We provide a comparison of simulation and analysis results and discuss possible sources of inaccuracies. As can be expected, the simulator takes considerably longer to compute results than the very quick numerical analysis. The analysis is still highly accurate for non-interfering tasks that are spread over several blocks. However, its accuracy degrades when tasks interfere with each other, especially for higher utilizations. This paper helps to identify the intrinsic difficulties when analyzing the performance of embedded software, so that future work can directly address the identified weaknesses.

The paper is further organized as follows: In Section 2 the simplified version of the ASD structure is explained. The discrete-event simulator which captures ASD structures with several blocks is presented in Section 3. Section 4 discusses the analysis of a single ASD block and presents a compositional algorithm for the analysis of ASD structures with multiple blocks. Section 5 discusses analysis and simulation results for several test cases. Finally, Section 6 presents the conclusion and pointers to future work.

2 The ASD architecture

The ASD suite can check for deadlocks and life-locks in the code. The developer splits the state diagram of the complete software system into smaller parts, each implemented as a single ASD block. Blocks have a clearly defined interface and other blocks only see the interface and consider the block itself as a black box. The work in this paper is based on a simplified version of the ASD state diagram which is necessary for formal verification by Verum’s ASD suite. The communicating ASD blocks are organized in a tree structure which determines that every master block (parent) can have multiple slave blocks (children), but every slave has only one master. Communication between blocks is done either via *synchronous* (S) or via *asynchronous* (AS) calls. The S-calls can only go top down in the structure, while the AS-calls only go bottom up. S-calls return to their caller when they have finished, AS-calls do not send returns.

A call that is processed by an ASD block can issue new S-calls or AS-calls as part of the response. When an S-call is sent to a slave block, the caller remains blocked until it has received a return on the call, just like a function call in a program. This effect is

Fig. 1: ASD architecture

Fig. 2: Timeline of nested synchronous calls

illustrated in Figure 2. In this example a master receives an S-call, does some processing (P) and then issues an S-call to one of its slaves, which does the same. While a slave is processing, the master stays blocked (B). The blocking is removed by the synchronous return. Issuing an AS-call however, is non-blocking. The caller does not have to wait until the call has finished, but just continues with other work. Within an ASD block, only one call can be handled at a time and calls in progress are not preempted. Incoming AS-calls have priority over S-calls. They are queued upon arrival and served according to First In First Out (FIFO). The tree structure together with the blocking ensures that there can only be one S-call per block, because there is only one master node that can issue an S-call to its slave.

The ASD tree structure forms a complete program that has to respond to one or multiple *tasks*. A task is a sequence of calls that flow through the tree (not necessarily through all levels) according to a predefined path. For example, a task could start with an S-call sent to the root block of the tree, followed by several S-calls to slave blocks, and finally closing with an AS-call. Such a task could be the reaction to a button pressed by the user. As seen in this example, tasks can mix S-calls and AS-calls. However, tasks starting with an S-call always enter the tree at its root, and tasks starting with an AS-call can only enter the tree at one of its leaves.

3 Simulation

We have implemented a discrete-event steady-state simulator for the performance evaluation of the ASD architecture. The simulator has been written in Java and allows to study ASD trees of arbitrary size. In addition to the tree structure, the user has to define tasks and to provide performance-oriented parameters. The supported model class is detailed below, followed by a brief description of the measures computed by the simulator and its performance.

A model definition for the simulator essentially consists of two parts. First, the user has to provide the ASD tree structure, i.e., a list of blocks and their master/slave relationships. The user then describes the tasks to execute. A task is defined as a sequence of calls that form a path through the tree. For each call in a task, the user has to specify (i) whether it is an S-call or an AS-call; (ii) to which block the call is sent. The simulator checks whether the constraints given by the tree structure are held. (iii) The mean service time of the call in the block. Service times are assumed to be negative-exponentially distributed.

All tasks are repeatedly executed. For tasks starting with an AS-call, the user has to provide the mean time between the generation of two instances of the task. These inter-arrival times are negative-exponentially distributed. However, if a task begins with an S-call, the simulator has to wait until the S-call has returned before it can create a new instance of that task due to the blocking nature of the S-call. In that case, the user defines the mean *think* time, i.e., the time between the end of one instance of the task and the begin of its next instance. Again, think times are negative-exponentially distributed. Note that the requirement that the involved distributions are negative-exponential has merely been a design decision than caused by technical limitations. Since the simulator is discrete-event based it can be easily extended to other distributions.

The behavior inside a block, i.e., how calls are processed, is simulated according to the description given in Section 2. If a call currently processed by a block X sends a new call to a master or slave block Y , the simulator assumes that the new call is sent to Y only after the local processing in X has finished, as illustrated in Figure 2. The real ASD-generated software will run on hardware with a limited number of processing units. In the simulator, the user can choose between (a) a fully parallelized model where each block has its own processing unit or (b) a model where all calls share one CPU. For the latter case, the simulator offers round-robin (RR) scheduling with adjustable time slice length and processor-sharing (PS) scheduling, which is equivalent to RR scheduling with infinitely small time slices.

The simulator computes the means of measures, together with confidence intervals, by running independent replications. Measures of interest include the mean response time of the system for each task, the utilization of the blocks, and the mean waiting times of the calls at each block. The current implementation of the simulator focuses more on functionality than on efficiency. The length of the simulation of course depends on the model size and on the desired width of the confidence intervals. The results shown for the rather simple example in Section 5.2 have been obtained on low-end hardware (dual core notebook @ 2.0 Ghz) after 40 seconds. The simulator has generated around $0.46 \cdot 10^6$ tasks per second. A more efficient implementation initiated recently, also written in Java, simulates around $1.1 \cdot 10^6$ tasks per second on the same hardware.

Fig. 3: Queueing model of a single ASD block

4 Numerical analysis

In this section, we present our first steps toward an analysis method for ASD tree models. Our method is based on the decomposition of the tree on block level and targets, in its current stage, a restricted class of ASD models. Compared to the model class presented in Section 3, it requires that (i) each block has its own processing unit, (ii) a task either sends only S-calls or only AS-calls and (iii) each block only receives calls from at most two tasks: One task sends S-calls to the block, the other task sends AS-calls. Among other things, this implies that there can be only one task with S-calls. We first introduce a queueing station model for single ASD blocks in Section 4.1. The computation of the measures of interest for that model is presented in Section 4.2. Then, we explain how the single block analysis can be used to perform a decomposition-based analysis of ASD trees in Section 4.3.

4.1 Single ASD block analysis

In the restricted model class introduced above, a single ASD block is completely described by the following four rates: the arrival rate λ_{as} and the service rate μ_{as} of the AS-calls and the think rate z_s and the service rate μ_s of the S-calls. Please note that in the following we have left out the discussion of the trivial case where the block only receives calls from one task, i.e., either only AS-calls or only S-calls. We model the behavior of the block by a special queueing system with two queues and one service station, as shown in Figure 3. In this system, both types of calls, synchronous and asynchronous, have their own queue. AS-calls arrive at the top queue, that is modeled to be infinite, while S-calls arrive at the lower queue. As explained in Section 3, AS-calls have priority over S-calls and are served according to FIFO. Calls in progress are not preempted. The fact that only one S-call can be present in the block is modeled by a closed loop. An S-call is either (i) waiting for service in the corresponding queue, (ii) in service in the service station, or (iii) experiencing a think time outside the block. The think time represents the time between the moment an S-call returns from the block and the moment a new S-call enters the block. The presented model of a single block is a quasi-birth-death process (QBD). Figure 4 shows the resulting underlying CTMC. Apart from state $0, 0, E$ representing the empty system, the states are organized in two groups. The system is in state $0, a, S$ if an S-call is being served and a AS-calls are queued. Alternatively, the system can be in state s, a, AS , indicating that an AS-call is being served and that s synchronous and a asynchronous calls are queued. Remember that s can only be either 0 or 1.

Fig. 4: Underlying CTMC of the single-block QBD process

By solving the above CTMC, we can compute the desired measures of interest for a single block, such as the mean waiting time of the calls. However, we will see in Section 4.3 that the assumption of Poisson arrival, think and service processes is too restricting. Fortunately, our model can be easily generalized to phase-type (PH) distributions. We will only keep the requirement that the inter-arrival times of AS-calls are negative-exponentially distributed. We follow the common notation for PH-distributions [13] and denote $(\mathbf{a}, \mathbf{T})$ for the PH-distribution with initial probabilities $\mathbf{a}$, rate matrix $\mathbf{T}$ of the transient states, and rate vector $\mathbf{T}_0$ to the absorbing state. Let $(\mathbf{a}_{\mu as}, \mathbf{T}_{\mu as})$ be the distribution of the service time S_{as} of AS-calls, $(\mathbf{a}_{zs}, \mathbf{T}_{zs})$ be the distribution of the think time Z_s of S-calls, and $(\mathbf{a}_{\mu s}, \mathbf{T}_{\mu s})$ be the distribution of the service time S_s of S-calls. For reasons of conformity, we also write $(\mathbf{a}_{\lambda as}, \mathbf{T}_{\lambda as})$ for the distribution of the inter-arrival time A_{as} of AS-calls, although it will always be a negative-exponential distribution. The block-banded generator matrix $\mathbf{Q}$ of the resulting CTMC follows directly from the above descriptions, however the derivation can be found in [9]. The steady-state state probability vector $\mathbf{p}$ with $\mathbf{p}\mathbf{Q} = \mathbf{0}$ and $\sum_{i=1}^{\infty} p_i = 1$ can be written as $\mathbf{p} = [\mathbf{z}_0 \ \mathbf{z}_1 \ \mathbf{z}_2 \ \dots]$ with the sub-vectors $\mathbf{z}_0$ and $\mathbf{z}_i$ containing the state probabilities of the states at level 0, respectively level $i \geq 1$ of the QBD. The sub-vectors can be computed by a matrix geometric method, for example by the LR method [12], which yields $\mathbf{z}_0$ and $\mathbf{z}_1$ as well as the matrix $\mathbf{R}$ with $\mathbf{z}_i = \mathbf{z}_1 \mathbf{R}^{i-1}$ for $i \geq 1$.

4.2 Measures of interest for single blocks

An important measure of the system performance is the waiting time of the calls. Typically, one is interested in the mean waiting time but we will see in Section 4.3 that it is useful to know higher moments of the S-call waiting time distribution, as well.

We begin with the computation of the S-call waiting time. As in the previous section, we first explain our approach for the simple case where all involved processes are Poisson. An incoming S-call finds the block either empty with probability e_0 or it finds

Fig. 5: CTMC describing the waiting time of an S-call

$i + 1$ AS-calls with probability a_i , $i \geq 0$. In the first case, the waiting time is zero; in the second case, the system first has to process the $i + 1$ AS-calls and any other incoming AS-call. Figure 5 shows the absorbing CTMC representing the situation of a non-zero waiting time. The waiting time is the time that elapses between entering the state $1, i, AS$ (with probability a_i) and reaching the absorbing state $0, 0, S$. We introduce the matrix $\mathbf{T}$, containing the transition rates for the transient states $1, i, AS$:

$$\mathbf{T} = \begin{bmatrix} -\lambda_{as} - \mu_{as} & \lambda_{as} & 0 & & \\ \mu_{as} & -\lambda_{as} - \mu_{as} & \lambda_{as} & 0 & \\ 0 & \mu_{as} & -\lambda_{as} - \mu_{as} & \lambda_{as} & 0 \\ & 0 & & \ddots & \ddots & \ddots \end{bmatrix}. \quad (1)$$

The k th moment of the waiting time distributions is given by $E[W_s^k] = (-1)^k k! \mathbf{a} \mathbf{T}^{-k} \mathbf{1}$ with $\mathbf{a} = [a_0 \ a_1 \ \dots]$. In the more general case of PH-distributions, the entries in matrix $\mathbf{T}$ are replaced by block matrices based on the distributions $(\mathbf{a}_{\lambda_{as}}, \mathbf{T}_{\lambda_{as}})$ and $(\mathbf{a}_{\mu_{as}}, \mathbf{T}_{\mu_{as}})$. Analogously, the scalar probabilities a_i are replaced by vectors of the form $\mathbf{a}_i = \frac{1}{p_{S0}} \mathbf{z}_1 \mathbf{R}^i \mathbf{c}$, where p_{S0} is the probability to have no S-call in the block and $\mathbf{c}$ is a matrix. The definitions of p_{S0} and $\mathbf{c}$ can be derived from the CTMC, as explained in [9]. In order to obtain finite expressions, we approximate the waiting time moments by truncating the absorbing CTMC at level t . In most of our experiments, truncation levels $t \geq 50$ have not provided any improvements of the results. The exact number depends on the system load and is iteratively determined in our implementation.

The resulting expressions for the waiting time follow directly from the structure of the state space. Note that, in the special case of negative-exponentially distributed AS-call inter-arrival times *and* service times, the times to absorption can be directly derived from the first passage time analysis of birth-death processes [10]. Knowing the waiting time, we can calculate the mean inter-arrival time $E[A_s]$ of S-calls. Since the inter-arrival time of S-calls is the sum of the think time, the waiting time and the service time, it holds: $E[A_s] = E[Z_s] + E[W_s] + E[S_s]$. The restriction to Poisson distributed AS-call arrivals allows us to get a simple expression for their mean waiting time $E[W_{as}]$. Using the well-known results for non-preemptive priority scheduling with Poisson arrivals [4], we obtain:

$$E[W_{as}] = \frac{1}{2 \left(1 - \frac{E[S_{as}]}{E[A_{as}]}\right)} \left(\frac{E[S_{as}^2]}{E[A_{as}]} + \frac{E[S_s^2]}{E[A_s]} \right). \quad (2)$$

4.3 Analysis of multiple blocks

In the previous section, we have analyzed a single ASD block by solving the underlying CTMC of a queueing station model. This approach cannot be directly extended to general tree structures consisting of multiple blocks: the result would be a CTMC that is potentially infinite in multiple dimensions because of the infinite AS queues in each block. In the following we propose a decomposition approach that is based on the single block analysis. The approach is based on the following two observations: (i) When an S-call processed by block X sends an S-call to block Y , X stays blocked while the second S-call is processed by block Y . As a result, the call at block X experiences an *effective* service time that is the sum of its specified service time at X , the S-call waiting time at Y and the effective S-call service time at Y . (ii) From the viewpoint of block Y , the S-calls sent by X arrive with a *perceived* think time that is the sum of the perceived think time at X , the S-call waiting time at X and the original S-call service time at X .

Once we know the effective S-call service time distribution and the perceived think time distribution of a block, we can analyze it by the method presented in Section 4.1. However, the effective service time and perceived think time are recursively defined since blocks depend on each other. We propose the iterative algorithm show in Figure 6 to solve the dependencies. The algorithm first initializes the distributions of each block with the distributions provided by the task specifications. Then, it employs a fixed-point iteration where in each iteration it analyzes the single blocks and updates the estimations of their effective service time, perceived think time and waiting time until a desired accuracy is reached. We assume that AS-call arrivals always are Poisson distributed. An important operation in estimating the effective service time distributions and perceived think time distributions (step 1 and 2 of the iteration loop) is the sequential composition function (seqcomp). For two PH-distributions $(\mathbf{a}_1, \mathbf{T}_1)$ and $(\mathbf{a}_2, \mathbf{T}_2)$, the *sequential composition* $(\mathbf{a}, \mathbf{T})$ is given by

$$\mathbf{a} = [\mathbf{a}_1 \quad (1 - \mathbf{a}_1 \mathbf{1})\mathbf{a}_2], \mathbf{T} = \begin{bmatrix} \mathbf{T}_1 - \mathbf{T}_1 \mathbf{1} & \mathbf{a}_2 \\ \mathbf{0} & \mathbf{T}_2 \end{bmatrix}. \quad (3)$$

However, a finite PH representation of the S-call waiting time distribution is not directly available from the block analysis due to the underlying dependencies. In Section 4.2, we have only computed the moments $E[W_s^k]$. Hence, we have to fit a PH-distribution to those moments (step 4 of the iteration loop). Our experiments have shown that two-moments fitting, as employed in [7, 17], and three-moments fitting from [14] yield equivalent results. Finally, it should be noted that we have not proved that the algorithm reaches the desired fixed point or, at least, terminates for all possible ASD models. However, in the experiments performed in Section 5, the algorithm has terminated after less than 20 iterations, even if a low relative threshold of 0.1% was chosen.

5 Validation

In this section, we study the performance of the analysis and the simulation. We begin with a simple test case with two blocks and three tasks in Section 5.1. In the second test case in Section 5.2, we evaluate two blocks executing two interfering tasks. For those

Initialization

For each block B executing a task with S-calls do:

1. Effective S-call service time distribution $(\mathbf{a}_{\mu s, eff}^B, \mathbf{T}_{\mu s, eff}^B) := (\mathbf{a}_{\mu s}^B, \mathbf{T}_{\mu s}^B)$.
2. Perceived think time distribution $(\mathbf{a}_{zs, pcv}^B, \mathbf{T}_{zs, pcv}^B) :=$ think time distribution of the task.
3. S-call waiting time distribution $(\mathbf{a}_{Ws}^B, \mathbf{T}_{Ws}^B) :=$ distribution with mean 0.

Repeat

For each block B do:

1. If the block sends S-calls to block Y :
 $(\mathbf{a}_{\mu s, eff}^B, \mathbf{T}_{\mu s, eff}^B) := \text{seqcomp}((\mathbf{a}_{\mu s}^B, \mathbf{T}_{\mu s}^B), (\mathbf{a}_{Ws}^Y, \mathbf{T}_{Ws}^Y), (\mathbf{a}_{\mu s, eff}^Y, \mathbf{T}_{\mu s, eff}^Y))$.
2. If the block receives S-calls from block X :
 $(\mathbf{a}_{zs, pcv}^B, \mathbf{T}_{zs, pcv}^B) := \text{seqcomp}((\mathbf{a}_{zs, pcv}^X, \mathbf{T}_{zs, pcv}^X), (\mathbf{a}_{Ws}^X, \mathbf{T}_{Ws}^X), (\mathbf{a}_{\mu s}^X, \mathbf{T}_{\mu s}^X))$.
3. Perform the single block analysis for this block with the specified λ_{as}^B and $(\mathbf{a}_{\mu as}^B, \mathbf{T}_{\mu as}^B)$ and the estimated $(\mathbf{a}_{zs, pcv}^B, \mathbf{T}_{zs, pcv}^B)$, and $(\mathbf{a}_{\mu s, eff}^B, \mathbf{T}_{\mu s, eff}^B)$.
 Compute new estimates of $E^B[W_{as}]$ and of $E^B[W_s^k]$.
4. $(\mathbf{a}_{Ws}^B, \mathbf{T}_{Ws}^B) := \text{fit}(E^B[W_s^k])$.
5. Calculate the changes in mean waiting times relative to the previous iteration.

Until all changes in mean waiting times are smaller than a given threshold.

Finally, compute the mean response time for each task by summing the expected waiting times and service times of the blocks visited by the task.

Fig. 6: Iteration procedure for multi-block analysis

two test cases, we provide the results as obtained by the analysis and by the simulator. In the third test case (Section 5.3), we discuss an ASD structure that can be currently only evaluated by means of the simulator.

5.1 Test case 1: Two blocks

In our first test case, we consider a system consisting of two ASD blocks A and B , as depicted in Figure 7a. The blocks execute three tasks. Task 1 sends S-calls to block A which in turn generate S-calls to block B . The mean think time of the task is 10. The mean service time is 3 at block A , respectively 2 at block B . Task 2 sends AS-calls to block B with arrival rate $\lambda_{as,2}$ and mean service time 2. Task 3 sends AS-calls to block A with arrival rate 0.3 and mean service time 1. For this example we have omitted the right slave of the root node, assuming an infinitely high service rate. This results in a degenerated tree, which does not affect the applicability of the method.

In Figure 8, we show the results for the mean response times for various arrival rates $\lambda_{as,2}$, as computed by the numerical analysis and the simulation using the fully parallelized model (together with the 95% confidence intervals). In addition, the figure also shows the analytically calculated utilization of block A for the different arrival rates. It is the effective utilization of the block, i.e., it also includes the blocking times when A is waiting for B . We observe that the analytical results are almost overlapping with those obtained by simulation. Only for task 1, a small difference is visible at $\lambda_{as,2} = 0.4$, which corresponds to a relatively high utilization of 89%.

Fig. 7: ASD structures of the test cases

Fig. 8: Test case 1: Mean response times and utilization as function of $\lambda_{as,2}$

Fig. 9: Auto-correlation coefficients (AC) of the S-call waiting times for $\lambda_{as,2} = 0.4$

Fig. 10: Test case 1: Mean response times as function of the number of iterations for $\lambda_{as,2} = 0.4$

To better understand the source of the error, we give the mean waiting times of the two call types for $\lambda_{as,2} = 0.4$ in Table 1. The respective relative errors (RE) between analysis and simulation are shown in the third row. The analysis obviously underestimates the waiting time of the S-calls at block *A* (first column). A reason for this can be found by inspecting the auto-correlation of the S-call waiting times at block *B*. Figure 9a gives the corresponding lag-*k* auto-correlation coefficients computed by the simulation for $\lambda_{as,2} = 0.4$. Remarkably, the waiting times show a positive auto-correlation that does not fade at larger lags. It is known that such long-range dependencies can increase the response times of queueing stations [6]. Since our decomposition analysis fits a PH-distribution to the waiting time distribution of *B*, it cannot account for that effect when building the effective service time process of the S-calls at *A*. Finally, Figure 10, shows the estimated mean response times as a function of the number of iterations of the analysis algorithm. We observe that the estimated values quickly converge to the final results after a few iterations. In total, the analysis results are computed in less than five seconds on low-end hardware (dual core notebook @ 2.0 Ghz) with a relative threshold of less than 0.1% for the iteration loop. The computation of the simulation results takes substantially longer with 30 seconds to two minutes with our original implementation, depending on the desired width of the confidence intervals.

	$E^A[W_s]$	$E^B[W_s]$	$E^A[W_{as}]$	$E^B[W_{as}]$
Simulation	7.28	20.17	50.71	8.49
Analysis	7.00	19.86	50.26	8.49
RE	-3.8%	-1.5%	-0.9%	0.0%

Table 1: Test case 1: Mean waiting times for $\lambda_{as,2} = 0.4$

Fig. 11: Test case 2: Mean response times and utilization as function of $\lambda_{as,2}$

	$E^A[W_s]$	$E^B[W_s]$	$E^A[W_{as}]$	$E^B[W_{as}]$
Simulation	2.61	5.78	11.78	3.42
Analysis	2.27	5.69	9.63	3.44
RE	-13.0%	-1.6%	-18.3%	0.6%

Table 2: Test case 2: Mean waiting times for $\lambda_{as,2} = 0.4$

5.2 Test case 2: Interfering tasks

For the second test case, we again consider two blocks A and B . In contrast to the previous test case, the blocks have to execute two interfering tasks, as shown in Figure 7b. Again, task 1 sends S-calls to block A which in turn generate S-calls to block B . The mean think time is 10. The mean service time is 3 at block A , respectively 2 at block B . The AS-calls generated by task 2 run into the opposite direction: every AS-call sent to B also generates an AS-call to A . The arrival rate of AS-calls at B is $\lambda_{as,2}$. The mean service time at B is 2, respectively 1 at block A . Figure 11 shows the mean response times for various arrival rates $\lambda_{as,2}$, as computed by the numerical analysis and the simulation using the fully parallelized model (together with the 95% confidence intervals), as well as the analytically calculated utilization of block A . This time we can see a much clearer difference between simulation and analysis for utilizations higher than 80%, i.e., for $\lambda_{as,2} \geq 0.35$. We give the mean waiting times at each block for $\lambda_{as,2} = 0.4$ in Table 2, together with the relative errors (RE) between analysis and simulation (third row). We notice a significant error in the waiting time analysis at block A (column 1 and 3).

Interestingly, the waiting times of the S-calls at block A and B do not exhibit any significant autocorrelation in this case. Figure 9b shows the corresponding autocorrelation coefficients for both blocks, as obtained by simulation for $\lambda_{as,2} = 0.4$. The

coefficients are always close to zero, even for a lag of 1. Of course, this does not exclude other kinds of dependencies (see below). We have also checked two other possible sources of errors. First, there is the assumption in our algorithm that AS-call arrivals are Poisson. We have verified the simulated arrival times of the AS-calls at block *A*, and, indeed, they are independent and nearly Poisson distributed. Second, we have verified whether the error could be caused by the three-moments fitting procedure itself. Therefore, we have also computed the fourth moment of the S-call waiting time distribution. The relative error of the fourth moment of the fitted distribution to the unfitted waiting time distribution (see Section 4.2) is -7.0% at block *A* and -10.8% at block *B*. Although these values look large, other effects have, from our experience, a much higher impact on the system performance than the fourth, and often even the third, moment [16].

We believe that the large analysis errors for high utilizations are caused by the fact that in this test case the two blocks are made dependent from each others in "two directions": by the S-calls from *A* to *B* and by the AS-calls from *B* to *A*. When an S-call is queued for service at block *A*, it implies that no S-call is served by block *B*. This means that *B* can serve AS-calls while the S-call is waiting. Since each AS-call served by *B* sends a new AS-call to *A*, the waiting S-call at *A* has to wait even longer (remember that AS-calls have higher priority). A similar effect increases the waiting times of AS-calls when *A* is processing an S-call. This behavior of inter-dependent queueing stations is not modeled by our decomposition algorithm.

5.3 Test case 3: Complex structure

For our last experiment, we consider an ASD tree structure consisting of 5 blocks as shown in Figure 7c. We define three tasks. Task 1 sends S-calls to *A* with a mean think time of 5, followed by S-calls to *B*. Task 2 sends AS-calls to *D* with an arrival rate of 0.2, followed by AS-calls to *B*. Task 3 sends calls in the following order: AS-calls to *E* with arrival rate $\lambda_{as,3}$, AS-calls to *B*, AS-calls to *A*, and S-calls to *C*. All mean service times are 1.0. Figure 12 shows the resulting mean response times for the three tasks for different $\lambda_{as,3}$, as obtained by simulation using the fully parallelized model. Relative confidence intervals are smaller than 2% for all results (not shown). We observe that the response times of task 1 and task 3 quickly increase with increasing arrival rate since both tasks compete for the resources of block *A*. Note that *A* is often blocked because of S-calls to *B* and *C*. The response time of task 1 increases even faster than the response time of task 3 because the AS-calls of the latter have priority over task 1's S-calls.

We also show in Figure 12 the mean response times obtained by simulation when using the processor sharing (PS) model. As expected, the response times are much higher than in the parallelized model and the system reaches its maximum utilization already when $\lambda_{as,3}$ approaches 0.15. In addition, the increasing arrival rate affects *all* tasks in the PS model, while task 2 is rather independent from task 1 and 3 in the parallelized model.

6 Conclusions

This paper presented a performance evaluation approach for embedded software, along the lines of the the ASD suite of Verum. The simulator, as presented in this paper,

Fig. 12: Test case 3: Mean response times as function of $\lambda_{a,s,3}$

derives accurate results for general models, however, is time-consuming. The numerical analysis is very accurate for non-interfering tasks, but suffers from correlation effects when tasks interfere and the utilization increases.

Several assumptions have been made about the ASD-generated software to make modeling and analysis possible. The structure of the ASD-generated software as described in this paper does not include all constructions allowed by the ASD suite. Even though deviations from the presented structure are possible from a modeling point of view in ASD, it is often not possible to formally verify them with ASD. Furthermore, service time, inter-arrival time and think time distributions were assumed to be negative exponential. To analyze real systems with different distributions, these need to be approximated by phase type distributions. This results in a larger state-space, however it does not pose any principal problem for the presented numerical analysis. Clearly, the simulator can be easily adapted to deal with different distributions.

Due to the increasing relative error for interfering tasks, the analysis as presented in this paper cannot directly be used for larger embedded software designs. However, we consider it a first important step towards more precise methods. Research will be continued in the recently started COMMIT project Allegio [1] in collaboration with Philips Healthcare. Future work will include a comparison with empirical data and simulations with two and more processing units. Furthermore, we will take the correlation of the waiting time processes at higher lags into account and better explore the iteration behavior of the algorithm, as done in [18]. However, note that not taking into account certain dependencies between blocks is a general weakness of compositional algorithms [21, 17, 8]. Hence, we also plan to use abstraction techniques on the complete underlying state space, which is infinite in as many dimensions as ASD blocks are present, like proposed in [11].

Acknowledgements Anne Remke is funded through 3TU.CeDiCT and a NWO Veni grant.

References

1. Allegio: <http://www.esi.nl/research/applied-research/current-projects/allegio/> (2011)
2. Broadfoot, G.H., Broadfoot, P.J.: Academia and industry meet: Some experiences of formal methods in practice. In: 10th Asia-Pacific Software Engineering Conference (APSEC 2003). pp. 49–58 (2003)
3. Chakraborty, S., Künzli, S., Thiele, L.: A general framework for analysing system properties in platform-based embedded system designs. In: In DATE (2003)
4. Cobham, A.: Priority assignment in waiting line problems. *Operations Research* 2(1), 70–76 (1954)
5. Franks, G., Al-Omari, T., Woodside, M., Das, O., Derisavi, S.: Enhanced Modeling and Solution of Layered Queueing Networks. *Transactions on Software Engineering* 35(2), 148–161 (2009)
6. Grossglauser, M., Bolot, J.C.: On the relevance of long-range dependence in network traffic. *IEEE/ACM Transactions on Networking* 7(5), 629–640 (October 1999)
7. Haverkort, B.: Approximate analysis of networks of PH|PH|1|K queues: Theory & tool support. In: MMB. LNCS, vol. 977, pp. 239–253. Springer (1995)
8. Heindl, A.: Decomposition of general queueing networks with MMPP inputs and customer losses. *Performance Evaluation* 51(2-4), 117–136 (2003)
9. Hettinga, S.: Performance Analysis for Embedded Software Design. Master’s thesis, University of Twente (2010)
10. Jouini, O., Dallery, Y.: Moments of first passage times in general birthdeath processes. *Mathematical Methods of Operations Research* 68, 49–76 (2008)
11. Klink, D., Remke, A., Haverkort, B., Katoen, J.P.: Time-bounded reachability in tree-structured QBDs by abstraction. *Performance Evaluation* 68, 105–125 (2011)
12. Latouche, G., Ramaswami, V.: A logarithmic reduction algorithm for quasi birth and death processes. *Journal of Applied Probability* 30, 650–674 (1993)
13. Neuts, M.: *Matrix-Geometric Solutions in Stochastic Models — An Algorithmic Approach*. Dover Publications, Inc. (1981)
14. Osogami, T., Harchol-Balter, M.: A Closed-Form Solution for Mapping General Distributions to Minimal PH Distributions. In: TOOLS. LNCS, vol. 2794, pp. 200–217. Springer (2003)
15. Rolia, J., Sevcik, K.: The Method of Layers. *Transactions on Software Engineering* 21(8), 689–700 (1995)
16. Sadre, R.: Decomposition-Based Analysis of Queueing Networks. Ph.D. thesis, University of Twente (2006)
17. Sadre, R., Haverkort, B.: FiFiQueues: Fixed-Point Analysis of Queueing Networks with Finite-Buffer Stations. In: TOOLS. LNCS, vol. 1786, pp. 324–327. Springer (2000)
18. Sadre, R., Haverkort, B.: Decomposition-based queueing network analysis with FiFiQueues. In: *Queueing Networks: A Fundamental Approach*, International Series in Operations Research & Management Science, vol. 154, pp. 643–699. Springer Verlag (2011)
19. Verum: <http://www.verum.com> (2010)
20. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer* 8(6), 649–667 (2006)
21. Whitt, W.: The Queueing Network Analyzer. *The Bell System Technical Journal* 62(9), 2779–2815 (1983)