# Estimating latency and concurrency of Asynchronous Real-Time Interactive Systems using Model Checking

Stephan Rehfeld*
Beuth Hochschule für Technik Berlin, Germany

Marc Erich Latoschik†
Universität Würzburg, Germany

Henrik Tramberend‡
Beuth Hochschule für Technik Berlin, Germany

**ABSTRACT**

This article introduces model checking as an alternative method to estimate the latency and parallelism of asynchronous Realtime Interactive Systems (RISs). Five typical concurrency and synchronization schemes often found in concurrent Virtual Reality (VR) and computer game systems are identified as use-cases. These use-cases guide the development a) of software primitives necessary for the use-case implementation based on asynchronous RIS architectures and b) of a graphical editor for the specification of various concurrency and synchronization schemes (including the use-cases) based on these primitives. Several model-checking tools are evaluated against typical requirements in the RIS area. As a result, the formal model checking language Rebeca and its model checker RMC are applied to the specification of the use-cases to estimate latency and parallelism for each case. The estimations are compared to measured results achieved by classical profiling from a real-world application. The estimated results of the latencies by model checking approximated the measured results adequately with a minimal difference of 3.9% in the best case and -26.8% in the worst case. It also detected a problematic execution path not covered by the stochastic nature of the measured profiling samples. The estimated results of the degree of parallelization by model checking are approximated with an minimal difference of 9.3% and a maximal difference of -28.8%. Finally, the effort of model checking is compared to the effort of implementing and profiling a RIS.

**Index Terms:** D.2.4 [Software/Program Verification]: Model checking—; D.2.8 [Metrics]: Performance measures—; D.4.8 [Performance]: Modeling and prediction—;

## 1 INTRODUCTION

Many Virtual, Augmented, and Mixed Reality systems (VR, AR, MR), as well as several of today's immersive first-person computer games exhibit strong real-time requirements. Taking into account the severe psycho-physical artifacts caused by latency [16], specifically for high immersive setups based on head-tracking, these so-called Real-Time Interactive Systems should be considered at least firm, if not hard real-time [24]. If given timing deadlines are missed, the system's quality of service not only is degraded, but the results might be harmful.

To fulfill increased timing requirements of firm and hard real-time, developers have to control the timeliness of all of the underlying system(s). Modern computer systems use optimization techniques such as out-of-order-execution, pipelining, caching, and branch-prediction. The algorithms of these techniques are in general deterministic, but in practice, this deterministic behavior is not transparent any longer to developers due to the shear complexity of the interplay of all optimizations. Hence, the time a program needs to be executed can not easily be determined by counting every instruction and multiplying each instruction with the processor clock-ticks necessary for their execution. Additionally, consumer operating systems are built around multi-tasking and multi-user capabilities with fair scheduling as well as multiple service features (including networking), all of the latter were never meant to provide firm or hard real-time features in the first place.

Finally, RIS-architectures always had to deal with the non-determinism caused by user as well as by non-deterministic algorithms. Prominent examples include changes of the view frustum due to user-defined camera movements (e.g., caused by head-tracking), which may dramatically change the number of graphics primitives to be processed from one simulation step to the next, or heuristic search algorithms often used in artificial intelligence (AI) modules. Hence, timeliness is now affected by multiple hard- and software factors, many of these are out of control for a white-box analysis by developers, which is a big obstacle for the required latency optimizations and real-time capabilities.

In the past, latency was often reduced by increasing frequency, achieved by higher clock-speeds of processors. For several years now, processing speed is no longer achieved by higher clock-speeds, but by an increased parallelism of computations. Distributing the various tasks of a RIS on multiple nodes, CPUs, or cores has been a central aspect of RIS frameworks and tools for years. Ideally, a concurrent architecture would utilize as many computing units, e.g., cores, as available without having these units to wait for each other. The latter requires a thoughtful incorporation of non-blocking behavior without compromising the overall consistency of the simulated environment.

Non-blocking behavior makes a system even more stochastic from a programmer's point of view. A non-blocking call to a client, e.g., by internally branching the thread of control, by triggering an event, or by sending a message now may not cause immediate processing of the client task. Not only does such a behavior have an impact on the adherence to real-time constraints, it also has an impact on the overall consistency of a simulated scene. By relaxing almost every sequential ordering, common pitfalls of concurrent systems such as dead locks and race conditions emerge.

As of today, most RIS applications use coarse-grained concurrency schemes that isolate dedicated tasks like input processing, physics simulation, AI, application logic, or rendering as concurrently running tasks. Some advanced approaches also provide a much finer-grained concurrency, but in any case, all of these approaches will have to include some synchronization primitives to explicitly assure consistency across all simulation tasks while still supporting asynchronous behavior where possible.

So far, the only viable approach to control timeliness, to reduce latency, and to fulfill increased timing requirements are runtime tests and extensive profiling for a later optimization. Profiling depends on either direct source code access for white-box instrumentation or the availability of profiling tools tailored for the specific RIS platform. A prominent example is the now retired SGI Performer platform, which was developed to support concurrency on multi-CPU platforms [42]. Today, profiling support is commonly provided for dedicated tasks like graphics rendering, to complete frameworks like Unity3D or the Unreal Engine.

*e-mail:rehfeld@beuth-hochschule.de
†e-mail:latoschik@uni-wuerzburg.de
‡e-mail:tramberend@beuth-hochschule.de

Still, profiling and testing a RIS applications is an extremely time-consuming task for two reasons: First, because the number of possible execution paths grows about exponentially due to all the sources of non-determinism. Second, in order to profile a system one must first develop these systems with all the identified aspects which may influence the target measurements.

An alternative approach to profiling is *model checking*. Model checking uses a formal model of the target system. This model is then checked by a computer program for various software quality properties. The model also has to capture all identified aspects which may influence the target measures but this often is much less work than building a complete system. Hence it is a good method to detect potential problems of an architecture beforehand.

Model checking is already common in the development of software with zero-fault tolerance, such as the software of the mars rover *Curiosity* [22] and the development of large clustered systems like *Amazon Web Service* [38]. While it is a promising method to tackle the problems of timeliness control and concurrency of RIS applications, to our knowledge, model checking has never been used in the development of RISs so far. This motivates the following questions:

Q1: Can we apply model checking to predict the behavior of a concurrent RIS with respect to latency and degree of parallelism?

Q2: In the positive case of Q1, how does model checking perform in absolute quantities in comparison with profiling, the state-of-the-art in RIS engineering?

Today, many VR- and AR-related research questions refrain from low-level technical details. The impression may be given, all low-level technical aspects are solved, considering the widespread availability of sophisticated ready-to-use software packages like Unity or the Unreal Engine etc., systems with a strong background in the related area of computer games. But the late promises of consumer VR have initiated an increased interest in solving the still persistent technical deficits. Existing RIS software packages rely on classical profiling, if at all, which is error-prone due to the stochastic nature of the problem. Hence, alternatives would help to improve these tools. Maybe not just coincidentally, some game studios have currently decided to stop using these ready-made packages and to start own developments to be able to explore alternative solutions, e.g., Deck13 with the *FLEDGE Engine*, GameDuell with its own development based on HAXE or even small companies such as Black Pants Game Studio (*Scape Engine*).

The article proceeds with a review of the related work followed by an analysis and identification of potential model checking tools for the given task. This section is followed by an identification of five typical coarse-grained RIS concurrency and synchronization schemes as use-cases. These schemes guide the development of necessary implementation primitives to be applied to the chosen target RIS platform. A graphical editor is introduced which supports the specification of various concurrency and synchronization schemes. This editor is used to specify the required formal models of the use-cases which are then model-checked. The accompanied implementations of the use-cases are profiled and both results from model-checking and profiling are compared and discussed. Finally, the effort of model checking is compared to the effort of implementing and profiling a RIS.

## 2 Related Work

### 2.1 Real-Time Interactive Systems

#### 2.1.1 Concurrency and Parallelism

Many computations inside of a RIS are *data-parallel* problems, e.g., rendering and physics simulation. Therefore, they are often performed on the GPUs, which are usually tailored for data-parallel concurrency. In contrast, multi-core CPUs target *task-parallel* concurrency. Hence, tasks that can run in parallel need to be identified, scheduled to available cores, and the results need to be merged into a consistent world state, as is often the case for the tasks taking place in the application stage.

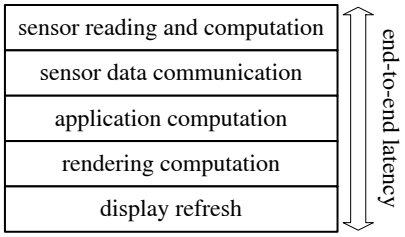



Figure 1: Sources that contribute to the end-to-end latency [44].

In the past, the VR community focused on coarse-grained task-parallel concurrency schemes for the application stage, like in Lightning [7], OpenMASK [34], ViSTA [3], DLoVe [12], Avango [45], and many state-of-the-art game engines like Unity3d or the Unreal Engine. Developers could always introduce a finer-grained concurrency in these systems, e.g., using the standard methods of concurrent programming provided by the underlying operating system, but then they had to cope directly with all the challenges as described by Lee [32]. Recent RIS platforms explicitly support fine-grained concurrency to better utilize the computational power of modern multi-core CPUs. For example, FlowVR [33] introduces a concurrent data-flow network where each node of the graph runs concurrently and Simulator X [31] exploits Hewitt's Actor Model [19] and message passing.

#### 2.1.2 Latency

Latency in general is the delay between cause and effect. In context of a RIS the most important latency from the user's point of view is the *end-to-end latency*: "the end-to-end latency is the time taken from an input device changing state to a consequent change on the screen" [44]. A high end-to-end latency can cause the following consequences [16, 14]:

- Induce simulator sickness.
- Change the user's behavior and lower his/her ability to perform tasks like reaching, grasping, or object tracking.
- Change the way multisensory information is combined into a percept.

Upon Mine [36], Steed [44] identified sources for latency depicted in Figure 1. Here, prominent approaches tackle the latency at the very end of the overall processing pipeline. *Post-Rendering 3D Warping* [35] warps the rendered image w.r.t. the most current position and orientation of the user's head at the end of the rendering stage to reduce the effect of latency causing a false (slightly outdated) perspective. Similarly, *frameless rendering* [11] also targets the same problem but uses a per-pixel calculation to include the most recent changes to the scene and the perspective.

Both techniques have shown to be effective. However, they cope as much as possible with resulting latencies as caused either between sensor read and final rendering or by the application stage computations (see Figure 1), they don't target the latency problem at the potential sources in the application stage.

#### 2.1.3 Concurrency and Latency Control

Usually, the performance of a RIS is determined by benchmarking or profiling. The platform code of the RIS is instrumented to collect

data for calculating specific metrics like parallelism and latency. Applications are then executed for various execution paths using the instrumented platform code. Furthermore, some micro benchmarks are common to only measure specific aspects. In most cases, the platform is instrumented manually by the person that wants to perform the benchmark. Most commercial platforms, e.g., the retired SGI Performer [42], Unity3d, or the Unreal Engine 3, and some research platforms, e.g., Simulator X [40], have dedicated profiling tools, that instrument the platform and calculate metrics. Furthermore, GPU manufacturer do offer frame-profiling tools to analyze the rendering of single frames, such as AMD GPU PerfStudio [2] and NVIDIA Nsight [39].

Results from benchmarking and profiling are usually used in short development cycles to optimize an application. The measured results are used to identify performance bottlenecks. The identified spots are tuned and the measurement starts again. The problem of this approach is that concurrent systems usually do have thousands or millions paths of execution and the measurements do only cover a small fraction of them. The non-deterministic sources dramatically increase this problem. Hence, a system may have execution paths that totally differ from what has been measured, and this is highly context-dependent. The stochastic approach of performing benchmarks or profiling for a longer time does not solve this problem, because there never is a guarantee that problematic paths really occur. A good example is a system described by Lee [32] that deadlocked after four years in production use, despite careful source code reviews and tests.

### 2.2 Model Checking

Starting in the 1980's, model checking became the primary method to reason about correctness [10]. Most programs can be described as a finite graph $M$, that consists of states, additional properties for each state, and transitions between states. The graph $M$ is also called *state space*. Model checking uses a formal language to specify an algorithm or system and to then generate the state space $M$ out of the formal specification. Afterwards, it is checked if the state space holds certain constraints.

Two families of formal languages exist, (a) imperative language and (b) declarative languages. Imperative languages usually are close to programming languages, such as C or Java, while declarative ones do have a more mathematical style based, e.g., on first order logic and temporal operators. Both approaches proved their value in real world scenarios. The imperative language PROMELA was used during the software development for the mars rover *Curiosity* [22], while the declarative language TLA+ was used for the development of E3, DynamoDB, EBS, and the internal distributed lock manager of *Amazon Web Services* [38]. *Curiosity* successfully landed and *Amazon Web Services* work stably and reliably for thousands of customers. Other formal languages and model checkers have ben used in specific target areas, e.g., UPPAAL [28] is often used in industrial projects [6], especially for embedded systems [4, 5].

Using model checking to evaluate the architecture and concepts of a RIS requires additional efforts. First, the developers need to become familiar with model checking and need to learn a specification language. Second, specifications need to be written and checked. Newcombe et. al. [38] report about the efforts at AWS to learn PlusCal and TLA+. Developers typically learn these languages within 2 – 3 weeks. Writing a specification is done within "a couple of weeks" [38]. Lamport [26] explored model checking using PlusCal on an algorithm [13] that was known to contain a bug [15]. Lamport's effort to write a PlusCal specification of the algorithm was about 10 hours.

### 2.3 Discussion

A formal method promises to overcome some of the deficits caused by the stochastic nature of profiling and testing. The requirements for low latency and high computational power of RISs seem predestinated for model checking. Depending on the quality of the model, it may also weaken the influence of the many non-determinisms in RIS applications discussed before. Hence, we will apply model checking to typical RIS concurrency and synchronization schemes and will compare the results to a common profiling approach. We will perform this task using an actor-based platform. We chose Simulator X [31] since 1) it is freely available on GitHub and 2) it has been instrumented in prior work for the necessary profiling [40].

In the actor model, every concurrently running thread of control (called process in the model) can communicate asynchronously with every other. This asynchronous communication can be seen as a generalization of subroutine calls, now extended to concurrent architectures. This generalization will a) allow to implement various concurrency and synchronization schemes an it will b) render the results achieved on top of this model applicable to other systems as well. In addition, it will maximize parallelism and minimize latency due to increased performance. On the other hand, it will require explicit synchronization primitives to gain fine-grained control over frequencies and triggering order of tasks. All typical RIS architectures can be implemented on-top of the actor model and the underlying asynchronous message passing system. However, certain programming techniques–while possible in general–are considered deprecated to unleash the full potential of software quality aspects the model provides, e.g., direct unguarded shared memory access.

## 3 IDENTIFYING A SUITABLE MODEL CHECKER

We have compared 9 formal languages and model checkers to identify suitable candidates to reason about the parallelism and latency of a concurrent RIS. Two type of criteria are used: Specific criteria are important for our purpose to use model checking to estimate the latency and parallelism of a concurrent RIS on the target platform. General criteria cover aspects that make the formal language and the model checker easier to use. Both criteria were combined but weighted $2:1$ in the final result to increase the RIS-specific utility. A detailed discussion about every candidate and every criteria is beyond the scope of this article. Instead the results are presented in Table 1. The criteria are explained in more detail in the following sections 3.1 and 3.2.

*Rebeca* won this comparison and is the best candidate for the goal of the work presented here. Rebeca [43] is an imperative specification language for message-passing-based systems, designed to close the gap between formal methods an software engineering [41]. The syntax is similar to Java. Rebeca is documented by a handbook [20] and several publications. Its syntax is similar to Java, hence it is easy to learn for many developers. An extension to Rebeca, called *Timed Rebeca*, exists to model check specification of real-time systems. The Rebeca Model Checker [1] (RMC) is an open-source tool written in Java. RMC exports the state space in an XML file, hence it is easy to integrate to other tools.

### 3.1 General Criteria

GC1: Documentation A good documentation makes it easier to write specifications in a formal language and to use its model checker. A language will receive the grade ++ if an up to date book exists about it. It receives + if several scientific publications written by different people exist. The grade 0 is given if only publications of one original author of the language exists.

GC2: Widely used Wide usage of a language does have two advantages. First, it points that the language and the model checker reached a mature state usable for real life scenarios. Second, templates for standard problems can be found, that shortens the time

Table 1: Comparison of nine formal languages and its model checkers.

| Familiy | TLA+[25] dec. | PlusCal[27] dec. | cTLA[18] dec. | ReActor[8, 9] dec. | Alloy[23] dec. | Rebeca[43] imp. | PROMELA[21] imp. | UPPAAL[28] imp. | McErlang[17] dec. |
|---|---|---|---|---|---|---|---|---|---|
| GC1: Documentation | ++ | ++ | + | 0 | ++ | + | ++ | ++ | + |
| GC2: Widely used | ++ | ++ | −− | −− | ++ | + | ++ | ++ | + |
| GC3: Implementation | ++ | ++ | −− | −− | ++ | ++ | ++ | + | ++ |
| SC1: Real-time | + | + | ++ | −− | 0 | ++ | −− | ++ | ++ |
| SC2: Message passing | −− | −− | + | ++ | 0 | ++ | ++ | 0 | ++ |
| SC3: Integration | ++ | ++ | −− | −− | ++ | ++ | 0 | ++ | 0 |
| Points | 8 | 8 | -1 | -4 | 10 | 16 | 6 | 3 | 12 |

required to write a specification. The grade ++ is given if a language was used for real life applications. The grade + is given if the language is used by several research groups at different institutes. A −− is given if the tool was only used by the original author.

GC3: Implementation This criterion describes if an implementation of a model checker is available to the public. A language is rated ++ if a model checker and its source code is available. It receives a + if a closed source implementation is available and a −− if no implementation is available to the public.

### 3.2 Specific Criteria

SC1: Real-time RIS are real-time systems. Hence, it is useful if the language itself or the standard library contains elements for the specifications of real-time systems. A language is rated ++ if the language itself contains elements for real-time behavior. The grade + is given if not the language but a standard library contains elements for real-time systems. A 0 is given if real-time behavior needs to be specified from scratch.

SC2: Message passing The chosen target platform uses message passing. Hence, it is advantageous if the formal language already contains elements for message passing or if a reusable specification exists. A language receives a ++ if it contains elements for message passing and a + if message passing is not supported directly but can easily be specified. The grade 0 is given if message passing can be specified. The grade −− is given if there exists any known shortcoming for creating a reusable message passing specification.

SC3: Integration to other tools Our goal is to integrate model checking into the development process of a RIS application. Hence, it is important that the model checker can be integrated into the current tool chain. Simulator X is implemented in Scala [30], which produces byte-code for the Java Virtual Machine. Thus, if the model checker is implemented in Java, it receives a ++, and a 0 if it is written in another language.

## 4 Concurrency and Synchronization Schemes

This section identifies different prominent concurrency and synchronization schemes (CSSs) that deal with triggering order and frequency. Recent RIS architectures support various concurrency schemes, from coarse-grained architectures which encapsulate classical sub-systems for, e.g., input, graphics, physics, or AI, to fine-grained concurrency which provides inter-sub-system parallelization of various degrees. While the chosen RIS platform Simulator X belongs to the latter group, the examples use a coarse-grained concurrency to increase interpretation and utility of the results for the wide range and de-facto standard of existing systems. The identified CSSs are explained based on a typical VR scenario consisting of the following sub-systems:

1. A *tracker connection* that communicates with a tracking system and provides the transformation of the user's head.
2. An *application logic* that manipulates the scene upon user's interaction, such as spawning new objects after the user pushed a button.
3. A *physics simulation* that simulates the physical behavior of objects in the scene.
4. A *renderer* that renders the scene using a graphics port.

Tracking systems can support many frequencies, e.g., ranging today from 50Hz to 250Hz for optical systems and up to 1kHz for orientation sensors of head mounted displays. The actual frequency is of lesser importance for the following scenarios, but its application-independent frequency is. Hence, let us assume that the frequency of the tracking system is bound to 50Hz by the hardware and can not be manipulated by the application itself. Furthermore, let the frequency of the graphics display be 60Hz.

### CSS1: Sub-systems run in sequence

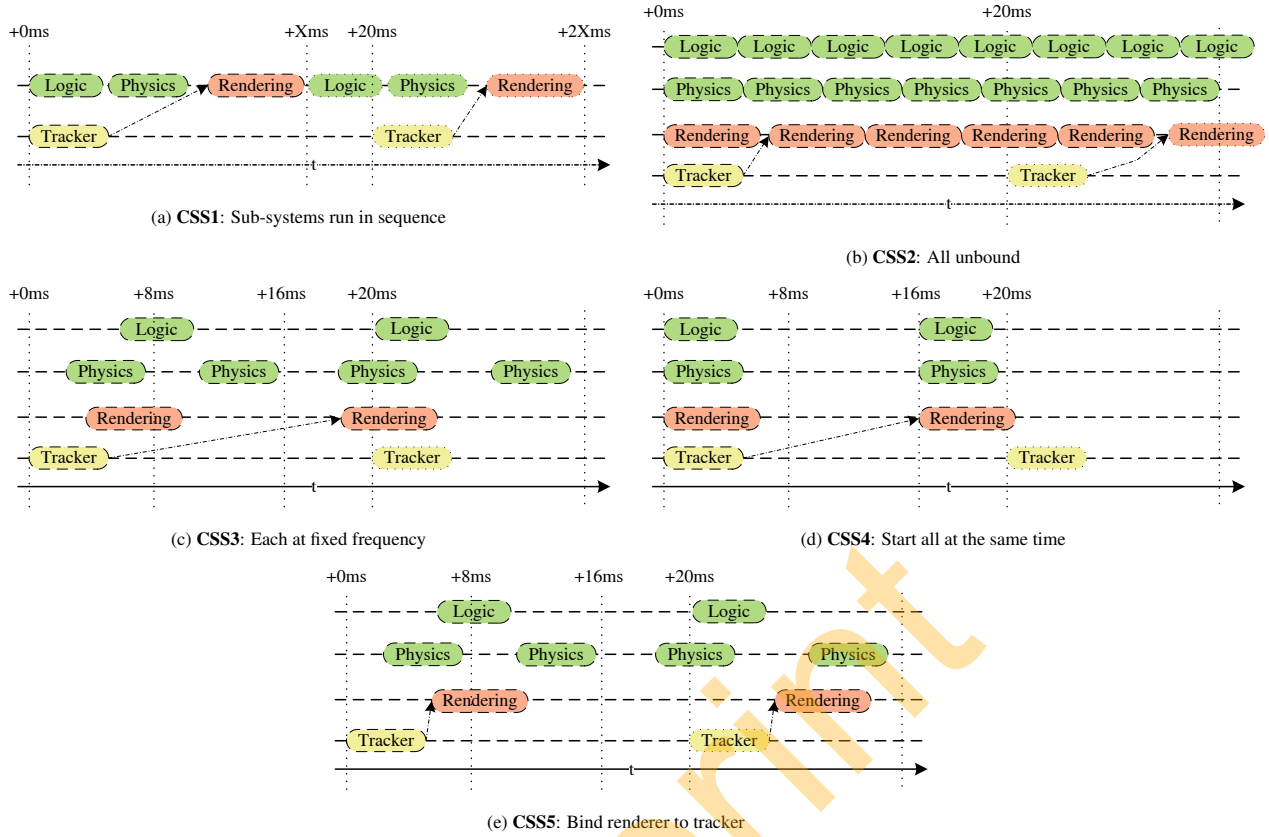
This is one of our two base-line schemes. All sub-systems run in sequence. First, the application logic performs its calculations. Afterwards, it triggers the physics simulation. Finally, the physics simulation triggers the renderer, that renders the new scene and the loop starts again. The frequency of the tracker is controlled by the tracking system and hence is not part of the loop. The scheme is illustrated in Figure 2a.

### CSS2: All unbound

The second base-line scheme is that all sub-systems run with the maximum possible frequency, as illustrated in Figure 2b. After performing a simulation step, the sub-system updates its internal world state upon the data it received and triggers itself. In this configuration, each subsystem runs with its maximum frequency. Usually, an unbound frequency is neither necessary nor desirable. An extremely high frequency consumes a lot of processing power and has no advantages for the simulation quality. We still provide this extreme test case to later check the limitations of our approach.

### CSS3: Each at fixed frequency

Limiting the frequency within the system is reasonable. For numeric stability, the physics simulation may run with a higher frequency, e.g. 120Hz. The renderer should have the same frequency as the output channel it is connected to. Because a user can not recognize changes faster than the frequency of the renderer, the application logic may have the same frequency. The scheme that is illustrated in Figure 2c, was suggested by Mönkkönen [37].




(a) **CSS1**: Sub-systems run in sequence

(b) **CSS2**: All unbound

(c) **CSS3**: Each at fixed frequency

(d) **CSS4**: Start all at the same time

(e) **CSS5**: Bind renderer to tracker

Figure 2: Five different concurrency and synchronization schemes as described in Section 4. Only the communication between the tracking and the rendering is visualized explicitly by a dashed-doted line, because it is of extraordinary importance for latency. Triggering and communication between other sub-systems are left out to increase clarity.



#### CSS4: Start all at the same time

To increase parallelism, renderer, physics, and application logic can start in parallel like shown in Figure 2d. This scheme usually adds latency between the simulation subsystems and the renderer.

#### CSS5: Bind renderer to tracker

In VR it is always desirable to have a low latency between input and output. Hence, its often more reasonable to couple the renderer to the tracking connection and not to the simulation subsystems (Figure 2e). A latency between the physics simulation and the renderer is hardly recognizable, while a latency between moving the head and seeing and updated scene, rendered with the new view frustum, is usually easily recognizable by the user. The other sub-systems run with their own frequencies, such as in **CSS3**.

## 5 Primitives

Five different synchronization and concurrency primitives are identified which are necessary for the specification of the schemes **CSS1–5** in the formal Rebeca language, and for the implementation in the asynchronous target RIS platform.

### 5.1 P1: Unmoved mover

The *unmoved mover* exists only one time and sends one message to a set of other processes. It is the starting point for an application and is required for all CSSs.

### 5.2 P2: Sub-system

A *sub-system* is a process that performs the calculation, communicates the results to other processes, or communicates with I/O channels. This could be a renderer, physics simulation, artificial intelligence, or any other part from coarse to fine granularity. A sub-system can perform the following three tasks:

1. Update its internal world state upon incoming messages.
2. Modify its internal world state, or using input or output channels.
3. Communicate its internal world state to other sub-systems.

Hence, sub-systems need two types of messages:

MT: Triggers local (sub-system-specific) simulation step(s).

MS: Communicates (parts of) the new world state.

We assume that a new world state can be communicated by multiple messages of type MS. Besides receiving and communicating new world states, a sub-system may trigger other processes, especially other sub-systems. Here, the condition and point in time when a trigger message is sent is crucial. We call the mechanism of sending these messages *synchronization point*. A synchronization point is a condition that may eventually becomes true and upon which a sub-system sends a trigger. Two synchronization points are obvious for coarse grained sub-systems:

1. *OnStepBegin*, when the calculation of a new simulation step begins

2. *OnStepEnd*, when the calculation of a new simulation step finished

### 5.3 P3: Frequency Limiter

Because an extremely high frequency is neither desirable nor useful, a process is required to limit the frequency. In general, a loop is realized in a message passing based system by a process that sends a message to itself. A frequency limiter is added to such a loop and delays a message it has received. For example, if the frequency limiter does have an upper limit of 60Hz, and it receives two message within a time span less than 16666$\mu$s, the second message is forwarded with delay. The limiter is used in all schemes, but **CSS2**. If the other process in the loop needs too much time, the frequency simply drops.

### 5.4 P4: Frequency Trigger

External devices, such as the tracker, do have a fixed frequency, that can not be controlled by the application. Hence, an element is required that always sends a message in a fixed frequency. While the frequency limiter relies on the circumstance that the process triggers itself, the frequency trigger sends a message to another process in a fixed frequency. If the triggered process needs too much time, the trigger still sends messages and the triggered process will saturate.

### 5.5 P5: Barrier

In **CSS4** all subsystems are started at the same time. Hence, there needs to be an element that sends a trigger after is received a specified set of messages. For example, in **CSS4**, the condition is that it received the trigger from the renderer, physics, and logic.

## 6 Implementation

In this section, a prototypical implementation of a model checker for concurrency and synchronization schemes is described. The overall tool chain of the model checker is illustrated in Figure 3a. The remainder of this section is structured along the tool chain.

### 6.1 Graphical Editor

Rebeca does not have any abstraction mechanisms, such as inheritance or a template mechanism like in C++. Defined processes can only be parametrized by constructor parameters. Furthermore, the set of other processes to which a process is able to communicate with, must be known at definition time of the process. To overcome these limitations, we created a graphical editor to specify concurrency and synchronization schemes such as described in Section 4 by using the primitives described in Section 5.

The graphical editor is shown in Figure 4. It was implemented using NetBeans RCP. In Figure 4, primitive **P1** is part of every specification by default and hence cannot be removed. Primitives **P2** – **P5** can be added using a palette of primitives and are shown as nodes. Edges between the nodes represent communication between the primitives. Together they define the modeled processes and the communication topology. Additionally, durations of simulation steps and the processing of messages can be configured. Finally, a code generator generates a Rebeca specification out of the graphically defined models.

### 6.2 Rebeca Spec

The automatically generated Rebeca specification file consists of all required primitives, the communication topology, and timing information. It currently doesn't cover the full Rebeca specification language but only the relevant parts required here. Hence, we also developed a text editor for Rebeca specifications to further alleviate customization of the specification. This editor was developed by extending NetBeans's standard text editor.

### 6.3 RMC

The generated specification is checked using RMC. RMC itself can already check if the specification deadlocks or if processes do saturate. Furthermore, RMC exports the state space into an XML file.

### 6.4 State Space

This XML file contains states and transitions. Besides other information, a state contains the clock of each process at this state. Furthermore, the transitions contain the information which process processed which message. Care needs to be taken, because this XML file can become very large, depending on the granularity of the model. The state space is parsed by the graphical editor again to perform further analysis of the state space.

### 6.5 Analysis Algorithms

Before the average parallelism and latency can be calculated, every possible scenario needs to be generated out of the state space. A scenario is one path through the state space that represents a loop. Hence, there must be a transition between the first and the last state in the scenario. Because the state space tends to be large for non-trivial problems, generating the scenarios out of the state space is time consuming. Furthermore, care needs to be taken about memory. Hence, scenarios need to be created in a depth-first approach. Nevertheless, creating the scenarios can easily require about 3 GiB of RAM.

For each scenario, the average parallelism and the latency between two previously configured processes are calculated. We implemented the two metrics as defined in [40].

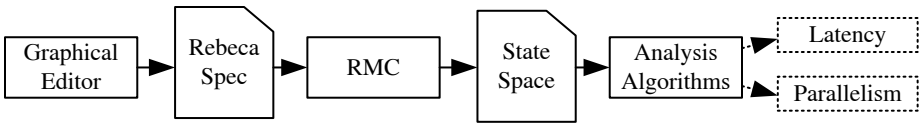
#### 6.5.1 Latency

Latencies can occur everywhere between the sub-systems. Our evaluation here analyzes the latency between the tracking system and the renderer. First, all transitions where the tracker sends a message to the renderer are searched. Afterwards, the next transition in the scenario where the tracker performs a simulation step is searched. Then the time span between both transitions is calculated. Usually, a scenario contains more of one transition that represents sending a message from the tracker to the renderer. Hence, the latency for each message can be calculated in parallel. Nevertheless, it remains very time-consuming.

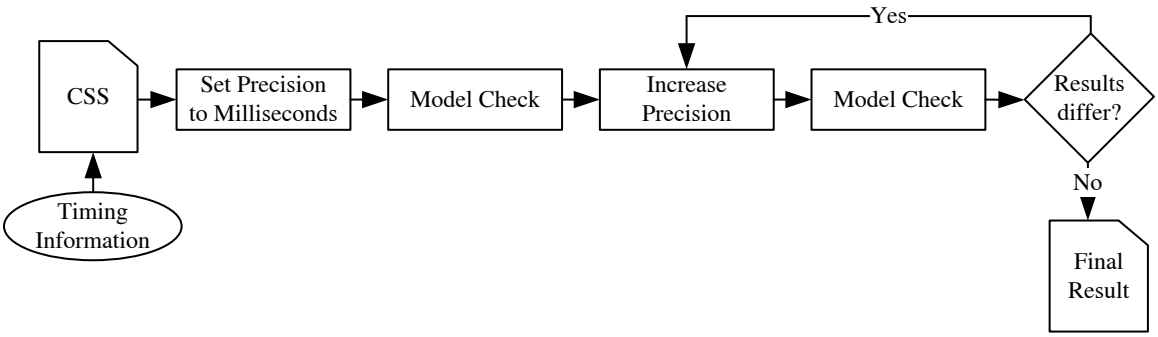
#### 6.5.2 Average Parallelism

The average parallelism reflects how many processes in average perform work at the same time. The average parallelism is measured as the Degree of Parallelism (DOP). The DOP states how many processes are performing work in parallel for any point in time $t$. This can easily be determined from the transitions between the states in a scenario. The clock of the process in the source state represents the begin and the clock of the process in the destination state represents the end of the processing. Thus, two tuples are generated for each transition, where the clock of the process is used as a time stamp. All tuples can then be sorted by their time stamp. Using this list, the average parallelism can be calculated as described in [40].

Using the profiling tool of the target platform on some demo applications, we observed that many applications idle most of the time. Hence, the average parallelism would be less than 1.0 in this case. Therefore, we decided to ignore idle times, where no process performs work. This is also provided by the profiling tool.




(a) Tool chain of the model checking tool.




(b) The workflow how to use the toolchain of Figure 3a.

Figure 3: The tool chain of the model checking environment and the workflow how to use it efficiently.

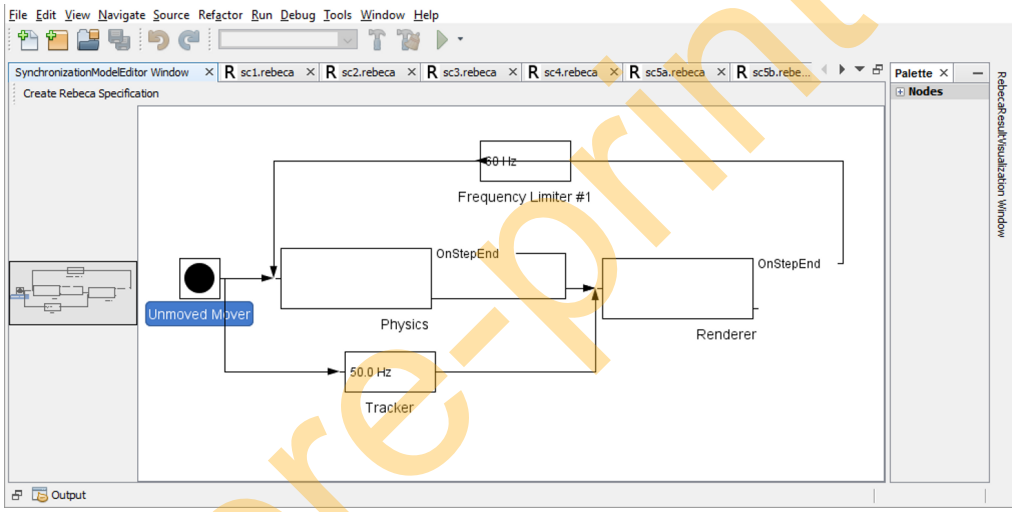



Figure 4: A screenshot of the graphical editor for synchronization and concurrency schemes.



## 7 Results

We model-checked the use-cases **CSS1-5** based on the implementation described in this article and compared the results to measurements achieved by profling of the same use-cases.

### 7.1 Example Application and Preparation

Figure 3b illustrates the workflow of our toolchain described in Figure 3a. The model checker is fed with the formal specification of the system including timing information. Absolute values for the latter is dependent of the application and the hardware it is executed on and hence is ideally measured first, e.g., using basic prototype implementations. For our measurement, Simulator X's barrel stack benchmark application was used (Figure 6). In this application, a large barrel is created at start-up. Ten seconds later, a stack of barrels is spawned in front of the large barrel. Another 35 seconds later, the large barrel is pushed into the stack. A user inspects the whole scene in an immersive setup with a head-tracked camera. The application is terminated after 60 seconds overall run-time. The profiling tool of Simulator X was used to determine parallelism and latency.

The first check of the model is done with a temporal precision of $10^{-3}$ (milliseconds). Afterward, the precision is increased to $10^{-4}$ and the model check is performed again. If the results of two consecutive checks are equal, the process is terminated and the final results are determined. Otherwise, the temporal precision is increased again for the next iteration. This iterative approach prevents an early state space explosion.

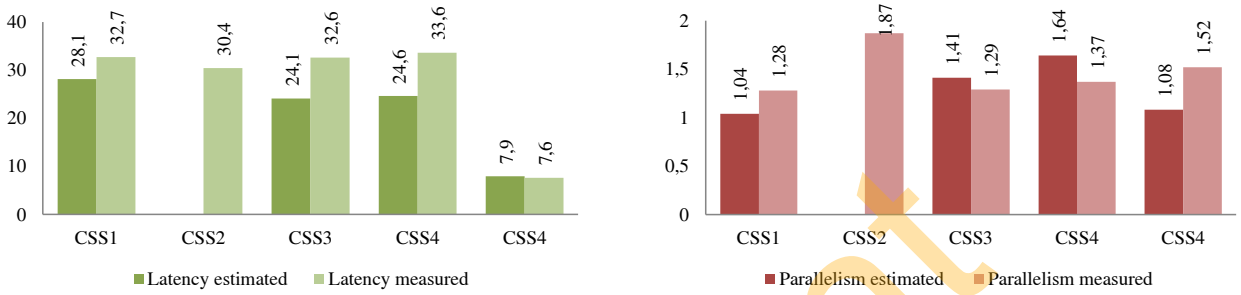
In our test case, the results stabilized at a temporal granularity of $10^{-5}$. Hence, increasing the granularity to $10^{-6}$ (microseconds) did not change the results for latency and parallelism, but produced a very large state space. As was expected, the time consumed to generate the state space depends heavily on the temporal granularity and grows about exponentially with the state space. While the state space at $10^{-3}$ is generated in less than a second, generating it at $10^{-6}$ can take more than an hour.

### 7.2 Discussion

The measured and estimated results are presented in Table 2 and visualized in figures 5a and 5b. For the latency the *median* is used for

Table 2: Estimated and measured results for latency and parallelism for the five concurrency and synchronization schemes. The difference is calculated to the respective measured values. A positive value means that the model checker overestimated the value, while a negative one means that the value was underestimated.

| | Latency | | | Parallelism | | |
|---|---|---|---|---|---|---|
| | estimated | measured | difference | estimated | measured | difference |
| CSS1 | 28.1ms | 32.7ms | -14.1% | 1.04 | 1.28 | -18.8% |
| CSS2 | – | 30.4ms | – | – | 1.87 | – |
| CSS3 | 24.1ms | 32.6ms | -26.1% | 1.41 | 1.29 | 9.3% |
| CSS4 | 24.6ms | 33.6ms | -26.8% | 1.64 | 1.37 | 19.7% |
| CSS5 | 7.9ms | 7.6ms | 3.9% | 1.08 | 1.52 | -28.9% |




(a) Estimated and measured latency.

(b) Estimated and measured parallelism.

Figure 5: Visualized results from Table 2.

Figure 6: The barrel stack benchmark application of Simulator X.

measured and estimated values. Using the median in the measured data is necessary, because the measured data also contains the initial bootstrapping of the application where assets were loaded which resulted in a few frames with extremely long rendering times. This initialization phase was excluded from model checking since it is in general not a critical phase for the usage of the application and it is prone to many non-determinisms, e.g., file access times, from the underlying system. Figure 5a reports the estimated latency to underestimate the measured latency with only one exception: the smallest difference of 3.9% occurred for **CSS5**, which results only to 0.3ms in absolute quantity. The largest difference of -26.8% occurred for **CSS4**. The goal to successfully report a reduced latency for **CSS5** was predicted well by the estimation of the model checking and was confirmed by the measurement of the profiling.

While model checking **CSS2**, a scenario with an overflowing queue at the renderer was found. Therefore, latency and parallelism was not calculated out of the state space. The renderer did not saturate while measuring. This problematic execution path did not occur during the measured profiling samples but was successfully detected using model checking. However, the problem can simply be reproduced, by using an overly complex scene, that drops the frequency of the renderer in all cases below the frequency of the tracker. In this case, the number of messages from the tracker in the in-queue of the renderer will increase. This will first result in an increased latency, and later into an out-of-memory process termination. However, the detected problem is a special case and concurrency problems usually are much harder to reproduce intentionally.

The estimated and measured parallelism is visualized in Figure 5b. The smallest difference is in **CSS3** with an overestimation of 9.3%, while the largest difference is found for **CSS5** with an underestimation of -28.9%. Unlike the results for the latency estimation, the model checker underestimated the parallelism in **CSS1** and **CSS5**, while overestimating it in **CSS3** and **CSS4**. Hence, there is not such a clean tendency of the direction of the difference as for the latency estimation.

## 8 Model Checking vs. Implementing and Profiling

To estimate the costs of model checking in terms of effort, we reconstructed relevant data from sources such as commits to a git repository, calendar, and e-mail. Hence, this is a very rough estimate but it can give a first idea of the relative costs. The total days of all participants are summarized in Table 3. The synchronization layer to implement **CSS1–CSS5** in the target framework Simulator X took roughly 2 weeks. To learn the concepts of model checking took 3 weeks and to learn Rebeca took 2 more weeks (here per person). Implementing the graphical editor including the code generator for Rebeca required 3 weeks. Creating the specification out of the designed system required 1 week. Checking and refining the model required 2 weeks as did implementation of the algorithms to analyze the state space.

Hence, specific implementation, specification, and check took 30 days. The check alone only took 15 days. In contrast to the ef-

Table 3: Costs in terms of efforts to learn about model checking, to develop tools, and to apply the method initially.

| Phase | Task | Effort (days) | Σ (days) |
|---|---|---|---|
| *Learning Model Checking* | In General | 15 pp | |
| | Rebeca | 10 pp | 25 pp |
| *Specific Implementation* | Sync Layer | 15 | 15 |
| *Tool Development* | Graphical Editor and Code Generator | 15 | |
| | Analysis Algorithms | 10 | 25 |
| *Application of Model Checking* | Creating Specification | 5 | |
| | Checking and Refining | 10 | 15 |

fort of building a complete framework or to change core aspects of an existing system these costs can be considered highly acceptable. For example, the initial development of the overall base framework used took at least 20 times as long as a very conservative approximation. Development times of many years, including ongoing service, bug-fixes, and maintenance, are not unusual in this domain.

## 9 Conclusion and Future Work

This article explored the usefulness and applicability of model checking for estimating latency and degree of parallelism in asynchronous RIS applications. Nine model checking languages and tools were analyzed to find appropriate candidates for their application in the target area. Out of these nine candidates, Rebeca was chosen based on a set of target-specific criteria. Simulator X, a RIS framework based on message passing was identified as the target platform because it supports various concurrency schemes due to its asynchronous nature and its actor model.

Five concurrency and synchronization schemes were then identified as use-cases typically found in the native or slightly modified version in many asynchronous RISs with similar sub-system requirements. These use-cases guided the implementation of concurrency and synchronization primitives for the target platform, and a graphical editor to specify the formal Rebeca model of the use-cases based on the developed primitives.

We then compared the model-checking-based estimations of latency and degree of parallelism of all five use-cases against measured results achieved by profiling of a real-world application. In one scenario, the latency was predicated nearly correct, with the small difference of 0.3ms. In the other cases, the model checker always underestimated the latency with differences between -14.1% to -26.8%. The estimated values for the average parallelism spanned a larger interval between the maximum difference of -28.9% and the minimum difference of 9.3%.

Both results are very encouraging and could be further optimized in future work. General tendencies like the underestimation of the latency prediction could be adjusted by a scaling factor. In general, the better the model captures the final system, the better the results will become. For example, our assumption about the slightly worse prediction of the degree of parallelism might be due to the increased non-determinisms affecting concurrency, from the scheduling of the underlying message-passing implementation and operating system, to cache-misses in combination with the enabled hyper-threading we used. To summarize these results given the initial research questions, we conclude:

Q1: **Answer:** Yes, model checking can be applied to predict the behavior of a concurrent RIS with respect to the target properties of latency and degree of parallelism.

Q2.1: **Answer:** Results by model checking vary in absolute quantity for the two target properties, still, this variance is sufficiently close to the measured results to be useful for post- as well as pre-optimizations of RIS architectures.

Q2.2: **Answer:** Model Checking also identified a problematic scheme, that did not show-up during profiling. Due to the stochastic nature of a RIS and the underlying hardware platform, disadvantageous execution paths did not occur during measuring but here were found by the model checker.

As a result, we propose that model checking provides a promising alternative to the state-of-the-art profiling of RIS architectures, especially with respect to the important RIS properties of latency and degree of parallelism. While the process of model checking is not free and will take time for itself, it has the potential to shorten the later development work and fine-tuning, which often is highly unpredictable and much more time-consuming than the model checking, and to reduce the non-determinisms of todays complex interplay of optimizations from all the different system layers.

To decrease the deviation between measured and estimated results, we plan to extend the model with more details. We also consider to apply model checking to additionally relevant software quality aspects of RISs. We think this will be especially complex but also fruitful for effects such as non-deterministic memory reads and branch prediction. One promising technology to integrate these effects into the model is probabilistic model checking like it is supported by Probabilistic Timed Rebeca.

Another future work is to extend model checking on cases where existing game engines such as Unity or Unreal are used. One idea is to simply abstract these systems as a sub-system in our approach to be checked as part of a larger system. Alternatively, we would like to model internal processes of these engine. Of course, the latter potentially would require information provided by the developers of such engines.



## Acknowledgements

The authors wish to thank Ehsan Khamespanah and Pedram Merrikhi.

## References

[1] https://github.com/ekhamespanah.

[2] AMD. Gpu perfstudio http://developer.amd.com/tools-and-sdks/graphics-development/gpu-perfstudio/. Internet, 2015.

[3] I. Assenmacher and T. Kuhlen. The vista virtual reality toolkit. In Latoschik et al. [29], pages 23–26.

[4] T. Bourke and A. Sowmya. Automatically transforming and relating uppaal models of embedded systems. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 59–68, New York, NY, USA, 2008. ACM.

[5] T. Bourke and A. Sowmya. Analyzing an embedded sensor with timed automata in uppaal. *ACM Trans. Embed. Comput. Syst.*, 13(3):44:1–44:26, Dec. 2013.

[6] N. C. W. M. Braspenning, E. M. Bortnik, J. M. van de Mortel-Fronczak, and J. E. Rooda. Model-based system analysis using chi

and uppaal: An industrial case study. *Comput. Ind.*, 59(1):41–54, Jan. 2008.

[7] M. Bues, T. Gleue, and R. Blach. Lightning: Dataflow in motion. In Latoschik et al. [29], pages 7–11.

[8] R. Burmeister. Reactor: A notation for the specifications of actor systems and its semantics. In G. für Informatik, editor, *Software Engineering 2013 - Fachtagung des GI-Fachbereichs Softwaretechnik*, pages 127–142. Köllen Druck+Verlag GmbH, 2013.

[9] R. Burmeister and S. Helke. The observer pattern applied to actor systems: A tla/tlc-based implementation analysis. In *Proc. International Conference on Theoretical Aspects of Software Engineering (TASE 2012)*. IEEE Computer Society, 2012.

[10] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, Nov. 2009.

[11] A. Dayal, C. Woolley, B. Watson, and D. Luebke. Adaptive frameless rendering. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[12] L. Deligiannidis. *Dlove: a specification paradigm for designing distributed vr applications for single or multiple users*. PhD thesis, Tufts University, Medford, MA, USA, 2000. AAI9955979.

[13] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele, Jr. Even better dcas-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 59–73, London, UK, UK, 2000. Springer-Verlag.

[14] M. Di Luca. New Method to Measure End-to-End Delay of Virtual Reality. *PRESENCE Teleoperators and Virtual Environments*, 19(6):560–584, Dec 2010.

[15] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele, Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 216–224, New York, NY, USA, 2004. ACM.

[16] L. H. Frank, J. G. Casali, and W. W. Wierwille. Effects of visual display and motion system delays on operator performance and uneasiness in a driving simulator. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 30(2):201–217, 1988.

[17] L.-A. Fredlund and H. Svensson. Mcerlang: A model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 125–136, New York, NY, USA, 2007. ACM.

[18] G. Graw, P. Herrmann, and H. Krumm. Verification of uml-based real-time system designs by means of ctla. In *Object-Oriented Real-Time Distributed Computing, 2000. (ISORC 2000) Proceedings. Third IEEE International Symposium on*, pages 86–95, 2000.

[19] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[20] H. Hojiat and Z. S. Kaviani. *Rebeca2 Reference Manual*, Feb 2012.

[21] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[22] G. J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, Feb. 2014.

[23] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, Apr. 2002.

[24] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[25] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[26] L. Lamport. Checking a multithreaded algorithm with +cal. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 151–163, Berlin, Heidelberg, 2006. Springer-Verlag.

[27] L. Lamport. The pluscal algorithm language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, pages 36–60, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[29] M. E. Latoschik, D. Reiners, R. Blach, P. Figueroa, and R. Dachselt, editors. *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2008 workshop*. Shaker Verlag, 2008.

[30] M. E. Latoschik and H. Tramberend. A scala-based actor-entity architecture for intelligent interactive simulations. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2011 workshop*, 2011.

[31] M. E. Latoschik and H. Tramberend. Simulator X: A Scalable and Concurrent Software Platform for Intelligent Realtime Interactive Systems. In *Proceedings of the IEEE VR 2011*, 2011.

[32] E. A. Lee. The problem with threads. *Computer*, 39:33–42, May 2006.

[33] J.-D. Lesage and B. Raffin. High performance interactive computing with flowvr. In Latoschik et al. [29], pages 13–16.

[34] D. Margery, B. Arnaldi, A. Chauffaut, S. Donikian, and T. Duval. Openmask: Multi-Threaded — Modular animation and simulation Kernel — Kit : a general introduction. In S. Richir, P. Richard, and B. Taravel, editors, *VRIC 2002 Proceedings*, pages 101–110, 2002.

[35] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 7–ff., New York, NY, USA, 1997. ACM.

[36] M. Mine. Characterization of end-to-end delays in head-mounted display systems. *The University of North Carolina at Chapel Hill, TR93-001*, 1993.

[37] V. Mönkkönen. Multithreaded game engine architectures. WWW, Sep 2006.

[38] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, Mar. 2015.

[39] NVIDIA. Nvidia Nsight `http://www.nvidia.com/object/nsight.html`. Online, 2015.

[40] S. Rehfeld, H. Tramberend, and M. E. Latoschik. Profiling and benchmarking event- and message-passing-based asynchronous realtime interactive systems. In *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology*, VRST '14, pages 151–159, New York, NY, USA, 2014. ACM.

[41] A. H. Reynisson, M. Sirjani, L. Aceto, M. Cimini, A. Jafari, A. Ingólfsdóttir, and S. H. Sigurdarson. Modelling and simulation of asynchronous real-time systems using timed rebeca. *Sci. Comput. Program.*, 89:41–68, 2014.

[42] J. Rohlf and J. Helman. Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 381–394, New York, NY, USA, 1994. ACM.

[43] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and verification of reactive systems using rebeca. *Fundam. Inform.*, 63(4):385–410, 2004.

[44] A. Steed. A simple method for estimating the latency of interactive, real-time graphics simulations. In *Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology*, VRST '08, pages 123–129, New York, NY, USA, 2008. ACM.

[45] H. Tramberend. *Avocado : a Distributed Virtual Environment framework*. PhD thesis, Bielefeld University, 2003.