On the Art of the Evaluation and Presentation of RIS-Engineering

Marc Erich Latoschik*
University of Würzburg, Germany

Wolfgang Stuerzlinger°
Simon Fraser University, Toronto, Canada

ABSTRACT

This article analyses the tasks of presenting and evaluating relevant scientific research in the field of Real-time Interactive Systems (RIS), i.e., in areas such as Virtual, Mixed, and Augmented Reality (VR, MR, and AR) and advanced Human-Computer Interaction. It identifies different methods for a structured approach to the description and evaluation of systems and their properties, including commonly found best practices as well as dos and don'ts. The article is targeted at authors as well as reviewers to guide both groups in the presentation as well as the appraisal of system engineering work.

Keywords: System Evaluation, Publication, System Engineering, Real-time Interactive Systems (RIS), Engines, Frameworks, Toolkits

Index Terms: A.1 [Introductory and Survey]; D.2.11 [Software Engineering]: Software Architectures; D.2.1 [Software Engineering]: Requirements/Specification; D.2.5 [Software Engineering]: Testing and Debugging

1 INTRODUCTION

Real-time Interactive Systems (RIS) are an increasingly important field of research. Application areas range from Virtual, Mixed, and Augmented Reality (VR, MR, and AR) to advanced Human-Computer Interaction, real-time simulation, and computer games. Several RIS aspects are equally relevant to ambient and pervasive computing as well as to robotics. These different fields have crucial commonalities with respect to the software engineering problems and solutions involved. All RIS-applications depend on a set of three overall functional requirements:

1. Close coupling of user and system
2. Multimodal input and output (I/O)
3. Interactive 3D content and representation

Close coupling describes the tight integration of the user into a (possibly partly) computer- and technology-controlled environment where inputs are analyzed and stimuli are generated continuously, taking into account human cognitive and perceptive constraints in terms of restricting factors, such as temporal and spatial resolution, latencies, and continuity.

Multimodal input/output describes the property that systems process a variety of different media interactively. That is, possible input methods range from classic WIMP-oriented (Windows, Icons, Menus, Pointer) interfaces using 3D-adopted devices (3D mice, Spaceballs™, stylus, ...), over generic devices (gloves, 3D-tracker, exo-skeletons, ...), specialized input methods and devices (props, Shape Tapes, CubicMouse™, ...), classic desktop devices, touch devices for more direct interaction, to natural communication styles using, e.g., gesture and speech.

- *marc.latoschik@uni-wuerzburg.de
- °http://ws.iat.sfu.ca

Output is generated on the basis of physics for multiple human senses, such as vision, touch, and hearing. Historically, the first RISs were largely vision-oriented. Hence image generation played – and in part it still plays – a driving role in RIS development. Most importantly, 3D computer games and their market massively influenced graphics hardware improvements during the last two decades. A variety of spatial graphics displays like head-mounted displays (HMDs) and projection-based large screen systems (workbenches, walls, CAVEs™, ...) try to cover as much of a user's field of view (FOW) as possible to provide spatial cues.

Interactive 3D content and representation describes a central aspect of the depicted applications. The environments are dynamic and animated, and interactivity couples the input side to the output generation. In one way or the other they process spatial information, which is a central aspect of the embodiment of users in the real world. In VR, for example, presence and immersion likely depend on an unobtrusive embodiment that closely couples users into the environment to dissolve the real-to-virtual borders.

These three functional requirements entail several critical non-functional requirements, such as latency, speed, reusability, portability, and scalability. This often leads to an increased complexity of RIS-applications, which makes the task of engineering new RIS challenging.

1.1 Motivation

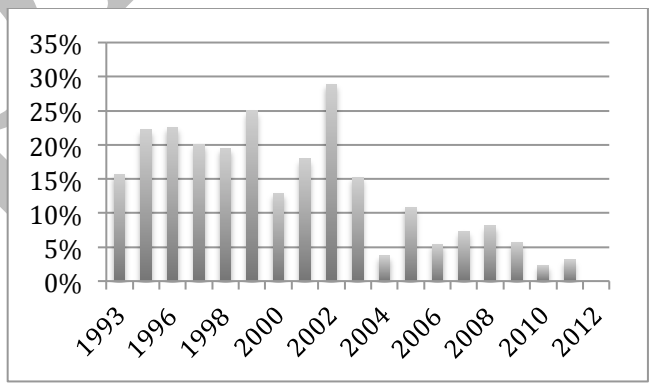

Figure 1: Ratio of system papers at IEEE VR and its predecessor. Percentage includes sketches.

Too often, there is ongoing re-invention of well-known architectural ideas in RIS-developments. As a result the rate of progress in the field of RIS architectures and software has been diminished, if not – in some areas – almost stopped. As evidence see Figure 1, which was initially presented at the IEEE VR 2012 panel *"Systems engineering science: Obsolete or Essential?"*[15].

So-called systems papers are generally regarded as hard to publish and often receive higher reject ratios in peer-review processes. This is in harsh contrast to the overall engineering costs and efforts. In contrast to smaller-scope RIS-engineering challenges, we have identified a diminishing concern on the architectural and large scale software aspects in recent years.

Many projects nowadays rely on commercial simulation and game engines. Often this might be the right solution from a cost-benefit analysis viewpoint. While this is not a problem in general, there are some longer-term concerns that arise. First, experience

shows that successful projects often live longer than initially planned for. They will be modified and extended to points hardly foreseeable at the beginning. Even the underlying requirements might change drastically. Hence, one cannot rely on a given tools' capability to cover all potential future requirements. Second, the scientific community has a different focus from the gaming community. Both will certainly benefit from an optimized content development tool chain and ever-improving simulation quality. But when it comes to the causality, observability, and controllability of the simulation in terms of objective, scientific measures, such as frame rate or latency, the requirements of both groups differ. What is tolerable in terms of enjoyment and playability often does not fulfill the requirements for a strict experimental setting. Possible other reasons for the decreased interest in challenges and results of RIS-development are manifold and are partly out of scope here. Lastly, the scientific community is also (largely) responsible for the education of the next generation of experts. This points to a need to maintain knowledge for the development and improvement of our scientific tools, to enable the next generation of experts to be taught the state-of-the-art for building such tools. To ensure this, the Software Engineering and Architectures for Realtime Interactive Systems (SEARIS) working group is dedicated to build a body of knowledge for RIS-technology. The group also organizes special events, such as workshops and panels, at the annual IEEE VR conference.

In this context, this article addresses the need for good communication of RIS-research by using a closer analysis of the scientific process of RIS-evaluation and presentation. It assists authors and reviewers by presenting guidelines for the authoring and evaluation of RIS publications:

- **Authors**: To assist in writing good papers. In our experience, we regularly see a substantial subset of systems papers, which often contain (the core of) good idea(s), but which get rejected because the ideas are badly evaluated or badly presented.
- **Reviewers**: To assist in consistent assessments of system papers in the field. Our intent is explicitly *not* to come up with quick-and-dirty rules of what is acceptable. That is for every field to establish by itself. However, we give general guidelines on how to present results, which meet the current standard in the field or go beyond it.

We will analyze typical pitfalls in the evaluation and presentation of RIS-engineering approaches. Our goal is to provide and stimulate a structured and organized methodology for the evaluation and publication of RIS-engineering as well as for the assessment of such work.

The reflections are based on our in-depth experience in system design: The authors have participated in many system projects as chief software architects and maintainers of several large systems in both academia and in industry and are now guiding young researchers in similar tasks. They have taught related courses on VR and software engineering for many years. As frequent reviewers, many times program committee members and conference chairs, and organizers of system tracks and workshops, they have a combined comprehensive experience in the evaluation and assessment of system engineering research.

However, others have also looked at papers that are targeted at the description of systems research [5][6][11][12][16][19][20][23]. We have incorporated some of the most important and relevant lessons from this previous work here. Even though parts of this are fairly generic content, they are particularly applicable to our context. Thus we decided to include them here for completeness.

We also point out that we are not advocating a single, particular style of presentation, similar to how there is not a single "perfect" writing style. Yet, we discuss necessary *elements* for any good RIS paper.

We will often use latency as an exemplary measure throughout this paper. Yet, the advice here applies equally to all other applicable technical measurements, such as frame rate, memory consumption, or networking bandwidth.

This article is structured as follows: We begin with a discussion on how to motivate the main problems and questions tackled by an article or presentation in section 2. Section 3 sheds some light on stylistic issues, from terminology to illustrations followed by a closer look at the description of engineering content in section 4. Section 5 discusses current evaluation methods applicable for RIS-developments. Section 6 comments on typical RIS-related evaluation targets. The article concludes with a summary and an outlook on future endeavors to strengthen RIS-engineering as an important area of research.

2 The Core Issues: Value & Motivation

It is the obligation of authors to provide a clear value to readers. This is especially important for RIS architectures. After all, just knowing that others replicated an existing system or created a system that varies on some minor aspects from another is of low value to the average reader. Given that reviewers, program committee members, and program chairs have an obligation to provide an interesting program for scientific events, papers without a clear value are often easy candidates for rejection.

To bring out this value, an article initially has to first clearly identify the core questions or problems the presented work will attack or solve. Also, the document needs to spell out the motivation for solving the problem(s) and review all relevant and important previous work in this area. Then, a publication should state its main claim(s) in one or two sentences, i.e. spell out what the novel ideas are, before going into the core explanation of the main contribution(s). The following evaluation then backs up the claims with results. A discussion section then interprets the results for the reader and identifies the value for the field. As reviewers, we have seen substantial numbers of papers that leave out essential parts of this progression. We give more detail about most of these parts below in section 4.

In the progression of writing we have sketched out here, it is essential that readers not familiar with your project can follow your line of thought. Breaks in that the line of thought often lead to misunderstandings and misinterpretation. Also, authors should honestly accept the severity of the questions and problems they are addressing.

3 Stylistic and Formal Issues

3.1 Terminology

The discussion of engineering problem is not taking place in a void. Software Engineering is a major field of computer science and has established a very large body of knowledge as well as its own terminology. Writing about software engineering has to conform to the established terminology, as applicable. For example, if you have a specific solution to some engineering problem, make sure that the approach you are using is not already known under a different term.

An often-encountered pitfall is the invention of your own idiosyncratic terminology. The motivation frequently is to highlight a unique characteristic of the individual work, which separates it from – at closer look – similar approaches. For example, a requirement is a requirement and not a *demand*, *request* or *property*. These terms relate to each other to a certain

extent and could be used interchangeably in *rare* cases. E.g., a specific system's property may fulfill a requirement and hence these two terms could be used interchangeably in contexts where there is no dependency on the differences in the respective meanings. However, in most cases terminology is well enough defined through previous work or resources such as dictionaries. Consequently, terminology has to be used with diligence and care.

Similarly, reviewers will (and should) pose questions if the generic terms library, toolkit, framework, or engine are used interchangeably. After all, they denote different architectural approaches. If authors are unsure about the correct meaning of a term, the reader can at least expect a definition upfront.

Correct terminology is important. Readers appraise your writing based on their knowledge of the field and its concepts. Thus, successful communication between readers and authors requires an established vocabulary with a clear meaning. It is not unusual that a reviewer finishes reading an article, only to discover at the end that the authors talk about a well-known idea hidden behind their own idiosyncratic terminology. This often makes it (unnecessarily) hard, or even impossible for the reviewer to understand the core ideas presented. More often than not, this will lead to a bad assessment of the work, as the authors make it effectively difficult for the reader to understand the message. This also necessitates clear writing to ensure that the message is unambiguous and easy to understand. For RIS-engineering this is important to make it easier to understand the (often) complex solutions that are necessary to address a given problem.

3.1.1 Languages

Besides software engineering terminology, there are three basic languages that are commonly used to communicate essential ideas: Mathematics, declarative and programming languages (aka code). A mathematical formula is unambiguous and provides a compact representation. The same is true for reasonable pseudo-code or a common declarative language, such as X3D. Both languages should be used to describe central concepts and design decisions, which will increase replicability. They can also serve as a basis for a formal evaluation. Ideally, authors should try to reuse existing notations as far as possible, again to make it easier for others to understand the work. As new RIS work typically involves new features, *minor* adaptations of notations are often necessary and appropriate.

For different cases, code snippets in a real programming or declarative language, instead of pseudo code, are helpful and important: For requirements, such as code elegance, simplicity, or compactness, real code examples are hard to beat as expressive media. For example, possible users of the system will appreciate and reviewers can appraise simplicity claims more easily, if these are backed-up by a few lines of example code emphasizing the actual idea and concept.

However, we caution authors that code or declarative examples have to be well chosen and should be as succinct as possible. Few reviewers are happy to wade through abundant Java module definitions or lots of irrelevant code in a RIS publication. Thus we recommend omitting all *nonessential* aspects in such examples. Also, we advise simplifying and abstracting routine aspects as far as possible. This will typically also lead to better code for the whole system. More complete code examples are best relegated to appendices or to on-line supplementary materials.

3.2 Illustrations and Diagrams

Good illustration of concepts and techniques are vital to any documentation and presentation of a system. Given the complexity of modern real-time interactive systems, such illustrations are often necessary for good communication. Often, the sheer complexity of processes and parts, modules, or components involved, calls for some general depictive overview to capture the essence of the specific topic discussed. If done right, this aids the reader greatly in the understanding of the work.

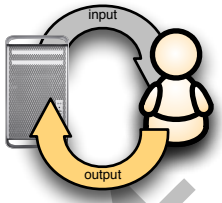

Figure 2: Illustration of our system establishing a closed loop between user and computer.
[This is not an example to follow, see text.]

Yes, illustrations must have a clear focus on the illustrated topic as well as a decent level of detail. Figure 2 is a typical example of an oversimplified and (almost) trivial illustration. This might be OK as an eye catcher, but it is often superfluous to repeat commonly agreed on facts in the respective area of research.

The polar opposite to Figure 2 is depicted in Figure 3. This figure compiles a set of commonly found mistakes. First, the illustration does not have a clear focus. It shows a collage of hardware and software items at various levels of detail, down to some object descriptions in the DB part. Even the interconnections are conceptually on several different levels, sometimes denoting hardware connections, sometimes protocols, and sometimes connection types.

A better version of this diagram would likely break it into several diagrams corresponding to different levels of detail. Also, the interconnections should be clearly labelled by type, e.g. using different line styles and legends, as well as having clear directions. In addition, resist the urge to overload illustrations, since this will likely make your labels small, which makes them (almost) unreadable in print. Figure captions are another noteworthy problem area. For example, the caption of Figure 3 is much too brief to explain the complexity of the content of the illustration. Here one solution is a brief caption with a reference to the continuous text (which itself refers to each part of the figure), or the caption must explain everything necessary to understand the illustration.

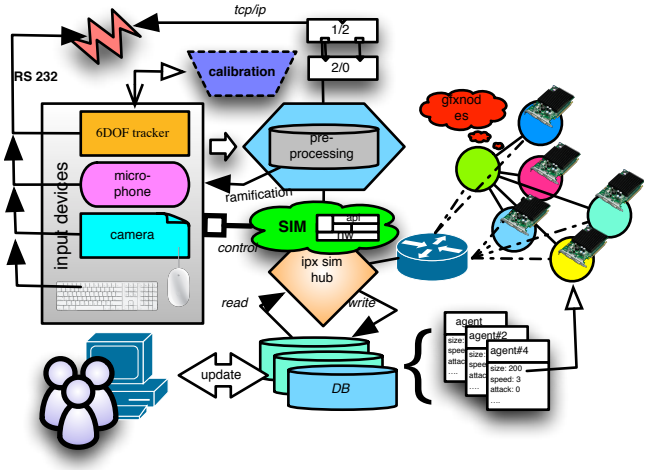

Figure 3: Overall architecture of a proposed simulation engine.
[This is not an example to follow, see text.]

Additionally, Figure 3 will likely confuse readers due to its layout and incoherent usage of stylistic elements. Stylistic elements, such as symbols, colours, and fonts, should be used consistently for the same type of information. Such consistency greatly helps the reader to understand the system, as visual similarity helps to identify commonalities in terms of meaning. Often the layout of diagrams can be changed to have the directionality of the flow of information be consistent with the semantic content that is being communicated.

3.2.1 Existing Illustration Schemas

Illustration programs nowadays come with a large variety of predefined and visually pleasing artwork, symbols, and illustration schemas. However, one should not rely on such predefined content without further thought, specifically if they do not conform to any given established standard or best practice.

There are several widely used and partly standardised illustration languages and schemas, from block diagrams, flow charts, to the Unified Modelling Language (UML). The latter nowadays is widely used and plays a prominent role, specifically due to its good support for describing object-oriented designs and architectures. These illustration schemas are well known and (to a large extent) unambiguous, which makes them an advantageous choice for illustrations.

The drawback of existing illustration schemas often is the level of detail, which may be too fine- or large-grained for any given case. We encourage authors to reflect in advance on the appropriateness of the level of detail used to communicate with the reader at a given point. Another problem lies in the development paradigms that such schemas are based on. For example, UML is heavily tied to the object-oriented approach and has drawbacks when it comes to declarative (logic) programming or functional languages. Consequently, we encourage authors to use the correct schema for any given topic, preferably based on existing schemas. We emphasize again in this context that any illustration should focus on the core issue(s) and eliminate superfluous, redundant, routine, or unessential aspects to clarify the communication of the core idea(s). In other words, a complete UML diagram is not appropriate and should likely only be part of supplementary on-line materials.

Last, but not least, we point out that a scientific publication is not meant to serve as system documentation due to the large differences in both target audience and coverage of the work. However, a scientific publication can provide a valuable supplement to system documentation by providing a high-level overview.

4 Description Targets

In this section we cover several guidelines related to the description of the system and the novel contributions. We discuss how architectures, algorithms, and alternatives should be described and how the discussion section should also identify the lessons learned.

For any publication it is essential that there is a clear focus on the main new idea(s). Most systems replicate large amounts of existing work in the field, and that previous work is in general amply documented. Consequently, we advise authors to focus less or even omit as many parts around those idea(s) as possible, unless that work is directly relevant to your new idea(s).

4.1 Describing the Right Thing

Consider a new real-time interactive system whose innovative architecture significantly decreases some measure, say latency, relative to previous work as an example. Said system likely has many other modules and functionalities, and probably has taken years to create. Describing all those modules around it will obscure and distract the reader from your main innovation. Moreover, the other modules likely only replicate previous work, and thus their description does not contain much value to the reader. This is a frequent problem in RIS submissions. Conversely and if your claims are targeting an improved modularity or system architecture, do not describe individual modules in more detail than necessary. Instead focus on convincing the reader that your modular decomposition is significantly better than previous work and explain how the module structure is established, how the interplay is managed, or how the API is defined. For ways to describe such claims see below in the evaluation section.

4.2 Describing Irrelevant Aspects

If you describe too many aspects of your system that are irrelevant relative to your main claim(s), reviewers will see this as rightfully as unnecessary “padding”. Too much padding will often result either in rejection or in conditional acceptance. In the later case you will be forced or at least heavily encouraged to rewrite your paper to remove the padding. Some conferences enforce this through conversion into a “short” submission.

Given this, it is better for authors to a priori focus a paper on the aspects of the system that relate directly to their main claims and to concentrate their efforts on substantiating these claims. As examples, consider a paper that presents the benefits of well-established concepts like a scene graph, a field propagation graph, an event system, a scripting layer, or a component-based architecture in depth. In terms of scientific novelty and benefit this is information that can safely be assumed to be known to every reader familiar with the domain. Thus, such a section will almost certainly have to largely be reduced or even completely be removed from the paper. A notable exception would be work that presents a *new variation* of well-known concepts and solutions that has clear benefits for the RIS community. Potential examples for such cases are beneficial variations in terms of clever implementation details and/or the *improved* conformance with general non-functional software requirements important for RISs, e.g., performance, scalability, extensibility, configurability, controllability, maintainability, or reliability. To motivate and justify novelty or improvement aspects it is mandatory to extensively cover the state-of-the-art and to compare one’s approach(es) to existing solutions.

For each claim that you make, it is very important that the claim is stated clearly in a concise way, ideally in a sentence or two. Papers that do not explicitly state their claims are very prone to misunderstandings and (re-)interpretation by reviewers, which typically weighs against the work. It is in general better to be more specific here, rather than too general – a common reviewer complaint is that a given submission “overstates its claims”, while one rarely, if ever, sees the opposite. Around that you also need to state what the RIS problem being solved is (even if it may be obvious), what the new idea to solve it is, in which way the new idea improves previous work, what you are comparing against, how the new work stacks up, and what this means to the reader.

4.3 Describing too much

Authors should resist the urge to put too many new ideas into a paper. They will end up with a document that either explicitly or implicitly makes many claims, yet does not have enough space to substantiate them all. Moreover, they will also need to review previous work for each of the ideas adequately, present the evaluation of each one of them and discuss what the results mean. Likely, this will cause severe conflicts with the maximum length

typically allocated for conference papers. This is particularly relevant for RIS submissions, as complexity naturally increases the number of ideas in a system.

For previous work it is important to list an appropriate subset of it, typically by focusing on the first examples, highly cited work, and very recent work. It is good style to summarize previous work briefly and in a fair manner. Papers that cite only a few external sources and other work by the authors – neither in the previous work section nor in the discussion section – are likely to get rejected, as they do not adequately describe how the new work fits into the field. This is a valid reason for rejecting a paper, as the value to the reader is not presented in an appropriate way.

4.4 Describing Alternatives and Trade-offs

It is a good idea to mention noteworthy design trade-offs in the main design section. If a particular trade-off is not well known in the field or not covered by other publications, it is an excellent idea to illustrate this trade-off with hard data in the evaluation and then to discuss it later. If a particular decision was based on previous work, it is a good idea to spell out which of the underlying assumptions of said previous work still hold true today, and which ones do not anymore. Alternatively, and if pilot experiments were performed on (small-scale) prototypes, it is appropriate to briefly describe the pilot experiment(s) and results that led you to the decisions you made.

In the description of alternative implementations, it is very important not to use conditional wording indiscriminately. While this is a basic academic writing issue, it is particularly troublesome in the context of interactive systems. If authors have to use conditional working, they should make certain that the text cannot be understood as "optional unimplemented extensions" that you are speculating about. A bad, concrete example is the sentence: *The system can do A, B, C, or D*. This leaves the reviewer and reader wondering if the system actually does *any* of these four alternatives. A better phrasing is: *The system implements A, B, and C. D is an unimplemented option that we believe to fit well into the framework, because [give some arguments(s) here].*

4.5 The Discussion Section

The discussion section is likely one of the most important parts of your paper. It typically follows the evaluation section, which describes the main evaluation, its methodology and the raw results. In the discussion, you summarize the main results of your evaluation and explain to the reader what your results means to the RIS field. Even if it seems obvious to you, this needs to be spelled out in clear language. Otherwise you rely on the reviewers and the reader to interpret your results correctly! In our experience, this kind of ambiguity often backfires in unforeseen ways and leads to a lot of frustration on the side of the authors.

Think about and describe how your work has been influenced by past work, confirms current thinking in the field, and can potentially influence future work by others. Ideally, this section also distills guidance for others facing the same problem On the other hand, it is important to also describe the context of your results and to spell out the assumptions behind your work. If you can characterize how sensitive your work is to smaller and larger changes of these assumptions, you can generally strengthen your claims significantly. All this highlights the value of your work to others in an appropriate manner.

4.5.1 Lessons Learned

Beyond the consequences of the main evaluation, the discussion section is also the right place to describe the "lessons learned". If the authors did not learn anything that they already knew based on previous work, it is likely not worthwhile to write a scientific publication about the endeavor. Consequently, these learned lessons should be communicated to the reader. We especially encourage RIS authors to describe in the discussion section how the choices made in the beginning turned out. It is acceptable to admit that a particular aspect of the work turned out far from optimal, especially if you have gained insights into what a better choice might be! If you have such insights, it is best to list them.

One interesting lesson that should be reported is new insights about hidden costs of architectural decisions. As a (somewhat) trivial and/or outdated example, consider that it may seem a good system engineering idea to take object oriented programming to its logical conclusion by creating a classic boundary representation for geometry with separate objects for every entity, including each vertex. However, it is today well known in RIS engineering that this has severe drawbacks. First, it has a high memory overhead. Second, the "killer aspect" of this idea is the increased number of cache misses. Especially when the data gets much larger than the secondary cache, this will lead to hundreds to thousands of CPU stalls. Consequently, it is important for the field to identify such hidden costs caused by various components of the system, be it by the memory subsystem, OS scheduling, network architectures, I/O interfaces, etc. This is one of the fundamental reasons why timings and benchmarking are so important for RIS engineering, as they can reveal deep insights into architectural decisions.

On the other hand, we advise authors to avoid the urge to generalize too far or to make unsustainable claims about scalability. One common problem with scalability claims is that scaling by more than one order of magnitude beyond established results very likely changes the nature of the main bottleneck and its location from one part into a very different (unforeseen) one.

Another kind of software engineering claim is that a specific approach is *more general* than previous work. Often this takes the form of a new software toolkit/framework that supports one or more specific features (say hardware abstraction or clustering) better than previous approaches. Another kind of approach presents a module, convention, or standard on top of other approaches to make a specific feature widely available. A third approach presents a new language for describing systems, with the aim to spur new abstractions and/or research. For all of these claims it is essential that the authors provide adequate evidence for the generalizability of their work. Simply arguing for generalizability is rarely sufficient, as in our experience reviewers are very good with identifying contexts where a particular approach will break or severely degrade, due to their different experience. Thus it is more appropriate if the authors identify the assumptions behind their approach a priori and state the limitations to generalizability. An even better form of support form of such claims implements a given approach in several contexts, such as different platforms, and reports on the experience.

Finally, we encourage authors to identify potential extensions either as the very last part of the discussion section or after the conclusions. In both cases it is important that the presentation of such future extensions clearly identifies them as not yet implemented.

5 Evaluation Methods

An evaluation of a system has to be concise and meaningful. There are several methods to evaluate claims regarding properties a system has or requirements it fulfills. The methods differ in information value and significance to the field of application. The

predominant evaluation methods applicable to system papers at the time of writing are:

1. Formal Verification, e.g., [2]
2. Black-box tests, e.g., [20]
3. White box tests, e.g., [20]
4. Software metrics, e.g., [8][9][10]
5. Usability tests and user studies, e.g., [7]

In addition, some evaluation methods can be performed dynamically or statically, depending on the proposition to be checked. These evaluation methods are now an integral part of state-of-the-art software engineering to assure the quality of the software at the diverse testing stages. We will discuss the applicability of each evaluation method to RIS-engineering in the following sections.

5.1 Formal Verification

A formal verification of a piece of software is a mathematical or logical proof of the correct behaviour of that software. That is, it verifies conformity to a given formal specification. This answers the question: "Are we building the system right?" Hence, verification is mainly concerned with the evaluation of non-functional requirements. This is in contrast to validation, which evaluates the functional requirements: "Are we building the right system?".

Formal verification has its value during the evaluation of a system's correctness. Its mathematical basis is sound from the ground up and provides the highest degree of informative value and significance.

A general drawback of this approach is the prerequisite of a formal specification of the system behaviour. Such a specification is still rarely found for complete systems. This is especially true for state-of-the-art agile development methods, which often perform multiple short circles of design-develop-evaluate with several prototypes in between.

A more specific drawback concerns the types of requirements targeted by a formal verification. These requirements must be accessible to a formal mathematical approach. For example, requirements like reusability, scalability, or parameterization, which are often found as claims in software architectural designs, are hardly accessible to formal methods. In addition, a formal verification is often voluminous even for small evaluation targets. As a result, formal verification currently is almost non-existing in RIS publications. However, it should be noted, that formal methods are promising approaches for certain problem areas, including hard real-time systems. Specifically, latency and performance measures may potentially be candidates for a formal verification, as appropriate methodologies become available.

5.2 Black-box testing

A black-box test is a valid approach to dynamically evaluate the correct behavior of software, i.e., the correct implementation of functional requirements, without assumptions about the internal workings of the system. This should be considered as a prerequisite for any software. A black-box test cannot reveal any insights into the architecture of a system; its test target is the executable of the software.

A black-box test can also be useful in the evaluation of non-functional requirements. Supported by a sophisticated test harness and appropriate test data, a black-box test can derive system properties associated with non-functional requirements, such as bandwidth, performance, and latency. For the latter cases, the results should be backed up by an analysis of how the test-harness interferes with the software itself and how the test data, here also known as benchmarks, are defined. Moreover, it is important to consider how the results map to cases that have not been tested, specifically how they apply to real-world scenarios.

5.3 White-box testing

White-box testing assumes knowledge of the "inside" of the box, i.e., the particulars of the system to be investigated. White-box tests include static inspection-based evaluations, performed, e.g., as an expert review where professionals study the test target, i.e., the code and/or any related engineering documents and formal descriptions. But purely static tests hardly reveal any insights into the behavior of the running system or the fulfillment of functional requirements, hence white-box testing also includes dynamic run-time tests which, e.g., test control and data flow coverage based on test data [13]. Other approaches to white-box testing investigate bottlenecks inside a system by specifically biasing the tests in various directions. One well-known example in computer graphics is vertex-heavy vs. pixel-fill-heavy render tests.

White-box testing is a method that communicates insights from one professional to the other one. This is a frequent task during system engineering [18]. It is capable of deriving deep insights into the architecture of a system. Here, terminology and a clear objective are crucial during the analysis. Used as a method which backs up claims made beforehand, a review has to pinpoint the specific problems a chosen design or algorithms solves. For such tests, evaluators should make sure to discuss alternatives and rate the chosen approach with respect to the alternatives. This method fosters replicability to a large extent, which makes it a valid approach for RIS-engineering.

Simulation is another approach to white-box testing of a RIS prototype. Typically, this replaces specific components of a system, such as networking, with a simulator or a set of predefined data sets. Then the performance of the system is assessed under such conditions, in a repeatable fashion, which can help to optimize the system. Yet, we caution that this approach is always dependent on the explicit and implicit limitations of the simulator and/or data sets. Thus, the generalizability of the results of such simulation-based tests is always limited.

5.4 Software Metrics

Software metrics are trying to back up the engineering part of computer science with quantitative measures towards objective and reproducible data and proof of concepts. Software metrics are supported by various development environments, which include modules to automatically capture certain metrics.

The debate of the usefulness of software metrics has continued for decades. In the words of Tom DeMarco: "*I can only think of one metric that is worth collecting now and forever: defect count*" [4]. A central problem is the complicated relationship between requirements, system properties, and measures. Most properties cannot be measured by just one metric but are based on the combination of several metrics, as property surrogates hiding the mutual dependencies between metrics and properties [3]. The shortcomings of lines of code as a measure highlight this to an extent. This problem is severe in the case of RIS-architectures, which exhibit certain properties not easily captured by available metrics. For example, module coupling in RIS-architectures often is sensitive to the current application context and hence changes dynamically. This semantic module coupling is poorly reflected by a syntactic analysis of mutual function calls [1][14] and hence is hardly accessible to automatic methods.

The call for objective measures is understandable, and software metrics can support the evaluation of certain properties for simple cases. Still, this approach is far from a bulletproof method that is always applicable. After all, RIS-development and system design

is essentially a part of the engineering science aspect of computer science. Yet, in contrast to hard real-time systems, which always have to meet timing guarantees, RIS systems have somewhat softer constraints. Typically, the rendering subsystem of a RIS prototype is considered to be good enough if it works stably at 60 or 120 Hz (depending on the use of stereo) and drops only "rarely" below this speed. Good RIS papers will thus quantify how often such slowdown episodes occur, how long they last, and what the most frequent causes for slowdowns are. Conversely, RIS engineering is far from empirical approaches that are important and necessary in other scientific fields.

5.5 Usability Tests and User Studies

Usability tests and user studies are the primary evaluation method of human-computer interaction. These evaluation methods rate the subjective properties of a system as perceived by users. Hence, they are a type of black-box test for user-centred properties and requirements, usability being the premier requirement here.

As an evaluation method for RIS-systems, usability can be tested for at least three different groups of users:

1. **Core developers** perform development tasks at the core system, e.g., maintenance, porting, or low-level extensions, that is, extensions that are not provided by a supported plugin API-concept.
2. **Application/Content developers** perform application building tasks, e.g., module configuration, content design, or development of high-level extensions on top of a supported plugin API.
3. **Application users** are the end-users of the system. Their goal is to perform an application specific task. They should ultimately not be aware of the underlying system operation.

While the third group naturally is a target group for usability tests and user studies, similar system requirements are hard to evaluate with these methods for the first two groups. The main problem of usability evaluations with these groups is usually a lack of sufficiently large enough numbers of people of said groups for sound statistical analysis. This is especially true for novel systems often still in a prototype stage.

The second problem is a lack of sophisticated measures applicable to the programming and development tasks. Consider groups 1 and 2: One would have to start with an evaluation of the usability of the underling programming language and paradigm and then evaluate the concepts and approaches built on top of it – after all the usability of the whole will also depend on these parts.

Usability tests and user studies are a valid and well-known method. However, they are applicable only in very specific cases of RIS-evaluations, i.e., for end users or for a sufficiently large group of developers, say more than 100. They are often useless for the analysis of architectures and system designs.

5.5.1 API-Evaluation in a Research Context

For authors that aim to evaluate the usability of APIs with group 2, we encourage them to consider using their system for assignments or projects targeting one or two new plugins with a sufficiently large number of students (say 20 or more) in a teaching context. Then report what parts of your system were used correctly, where students ran into problems and analyze in detail what these problems were.

Similarly and for systems that are in the public domain, an empirical study of the experience of outside plugin developers can similarly identify parts that work well and parts that do not. If insights that are valuable for the field are identified through this process, it makes sense to report them in a publication. With such evaluations it is also important to identify the cost of learning explicitly. After all, a seemingly simpler API may be harder to learn, due to conceptual difficulties. Or there may be some unfortunate interaction with some difficult programming language semantics.

An even better approach is to have two separate, roughly equivalent, student populations that use two different systems (or versions of the same system) that differ only in a few well-chosen aspects. A cross-comparison can then reveal the relative value of the differences. We rarely see such comparisons to evaluate system API design choices.

6 Evaluation Targets

We give here several examples how to describe an evaluation. We mention examples for approaches that do not meet the current standard, those that meet the current standard, and approaches that may form an appropriate standard in the future.

6.1 Evaluating the "Right" Measure "Right"

In system evaluation it is important to pick appropriate measure(s) and to evaluate them in an appropriate way. Here it is important to realize that the standard in any given scientific field changes over the years. For example, and while it may have been acceptable to provide a single frame-rate measurement 20 years ago, it is today necessary to test with multiple, sufficiently different geometric data sets and to give a range of frame-rate values. We expect that soon a characterization of frame-rates in the form of average and standard deviation will be the norm, or better yet a confidence interval or another equivalent form of characterization of the distribution of frame-rates. Similar for other RIS measures, such as latency, tracking accuracy, tracking precision, etc.

6.1.1 Benchmarks

In general, it is appropriate to use existing data sets, test methodologies, and/or benchmarks to evaluate a RIS system. Such data sets, test methodologies and benchmarks enable comparisons across the work of different groups and systems. This is an integral part of the scientific approach and helps the overall RIS field to progress.

To illustrate the importance of this topic, we point out that authors using non-standard data sets always have to be prepared to answer the question if their specific data set is appropriate to test their system and can adequately demonstrate the generality of their RIS approach.

6.2 Evaluating Performance

Any publication that claims superior performance in a given aspect needs to back this claim up through experimental measurements. For example, if the presented system architecture is associated with a claim that latency is reduced through design decisions, a latency measurement has to be performed and presented. Surprisingly, we still see submissions that do not meet this basic requirement.

Yet, a single measurement is not enough by today's standards to adequately support a claim. Computing an average over different scenarios is more reasonable, but often does not provide the whole picture. A better alternative is to provide a combination of average and standard deviation, a 95% confidence interval, or some other characterization of the distribution of values.

In comparisons two (or more) solutions are compared through some metric. Many RIS-papers showed improved performance of one approach over others by simply comparing averages. While such a comparison can yield some insights, it ignores the issue that distributions may overlap to a degree that makes them indistinguishable. Consequently, we encourage the field of RIS-

engineering to use well-established and robust statistical comparison mechanisms, such as t-tests and analyses of variance (ANOVAs).

6.2.1 Frame rate vs. Latency

One particular topic that is important for the design of VR systems is the trade-off between frame rate and latency. While the two are coupled to some degree, there is clearly no one-to-one correspondence. Yet, we regularly see statements by authors that high-frame-rate systems have low latency. This is only true if there is no pipelining involved. For example, it is fairly easy to develop a 120Hz system, which has 200ms of latency – all one needs to do is to send the data through a long multi-hop network link. That system will likely have also high variability in latency, which is disastrous for human performance see e.g., [17]. This problem is also well known in gaming circles, where "laggy" systems and systems where the frame rate varies substantially are universally criticized. Based on this and other reasons, modern game systems aim for a (reasonably) constant frame rate.

Consequently, we encourage authors to evaluate both for frame-rate as well as the latency. Both measures should ideally not be expressed just as an average, but also compute and list a standard deviation, a 95% confidence interval, or equivalent information.

6.3 Evaluating Usability/Immersion/Fun/…

There are few systems papers that evaluate the usability, immersion, simulator sickness, the fun factor, or other "soft" requirements or qualities, of an application running in the system. These types of measures are usually assessed through (subjective) questionnaires. The influences of architecture or software engineering aspects on such measures are indirect and causalities between both areas are often hidden.

In our opinion it is much better in this case to also measure (intermediate) "hard" quantities, such as frame-rate and latency and to report the outcome of both evaluations. Then interested, more specialized, readers can take such results further to investigate, e.g., the link between latency and simulator sickness.

We still point out that the design of the user interface API for a system has interesting effects on the usability of applications based on said system. For example, a system where the API does not provide special methods for objects in contact is likely suboptimal when used for applications where (almost) every object is in contact with another object – like in the real world. In such an application context, naïve users will expect objects to stay in contact with others and not to have objects interpenetrate without explicit user actions. Consequently, the overall usability of this system may suffer see e.g., [22].

However, the only way to adequately prove this is to do a cross-comparison between different user groups faced with different conditions or with a repeated measures experimental design. This kind of experiment will by nature only focus on the usability issue and thus will be best reported as a stand-alone publication and not as part of a system paper.

7 Conclusion

During RIS-development, engineering tasks, i.e., the design and implementation of novel approaches targeting a specific shortcoming or problem, are essential. In contrast to the increasing complexity of systems and the high development and maintenance costs of systems, research results in this area are increasingly hard to publish. The reasons for this are manifold and are partly out of scope for this article. However, one common issue is the lack of a proper and structured approach to the description and evaluation of systems and their properties.

This article identified, collected, and summarized different methods for such a structured approach. We included best practices and dos and don'ts whenever appropriate. Similar to software engineering, there is no "*silver bullet*" which one can follow step-by-step. Yet, authors are strongly advised to present and evaluate their main contributions using at least some of the methods and concepts presented here. Reviewers and readers may use this article as a collection of templates and best practices, which they can utilize to assess the value a contribution.

As evaluation methods, white- and black-box tests were identified as applicable to typical RIS-requirements. Formal verification and software metrics are considered to be of lesser importance. They do have their niche, but one has to plan beforehand when and how to apply them. Usability tests and user studies are a primary human-computer interaction evaluation method, but are mainly applicable during evaluations with end-users. Usability studies rarely yield strong value for assessment of developers, unless they are used with sufficiently large developer groups.

7.1.1 Future Work

The area of RIS-engineering could greatly benefit from a more focused body of knowledge directly associated to this field. This body of knowledge should summarize well-known approaches and their rationales and applicability, similar to a set of blueprints developers can utilize when appropriate. This would greatly reduce the potential for constant re-invention and would help researchers, specifically when new to the field. This idea can and should be used for description and evaluation tasks as well, to establish a more concise structure to follow. As an example, consider how the field of human-computer interaction has benefitted from the standardization of pointing tests via Fitts' law studies.

A similar path is the development of agreed-on benchmarks. RIS-applications will certainly vary to a large degree concerning their functional properties. Still, we envision a set of benchmarks of increasing complexity and scope, which can be applied to several evaluation methods and which will boost the significance of black-box tests for non-functional requirements. Such benchmarks could be structured based on their functional requirements, e.g., if they require graphics, physics, audio, haptics, or what kind of interaction they are based on. The benchmarks can then serve as test data within a unified test harness. Current developments to use large gaming platforms, such as Unity or Unreal, for RIS development are very promising for this, as they implicitly unify platform usage and thus provide ideal preconditions for such benchmarking.

However, relying completely on platforms borrowed from a different application area is risky as well. Given the multi-billion dollar industry of gaming has different objectives compared to VR, MR and AR, borrowing technologies can also import some undesired properties. E.g. generation of stereo images inside graphics drivers may be appropriate for games, but usually does not provide enough control for VR, MR and AR, type stereo systems. Hence, this cannot be a complete substitute for novel RIS-developments, even given the close relationship between the fields. After all VR, MR, and AR are constantly pushing the limits of novel human-computer interactions.

Acknowledgments

We thank the reviewers for their feedback, which led to substantial improvements of this paper.

REFERENCES

[1] E. B. Allen and T. M. Khoshgoftaar, "Measuring Coupling and Cohesion: An Information-Theory Approach," in *METRICS '99: Proceedings of the 6th International Symposium on Software Metrics*, ed. Washington, DC, USA: IEEE Computer Society, 1999, p. 119.

[2] P. Bjesse, "What is formal verification?," *SIGDA Newsl.*, vol. 35, p. 1, 2005.

[3] C. K. a. W. P. Bond, "Software Engineering Metrics: What Do They Measure and How Do We Know?," presented at the METRICS 2004, 2004.

[4] T. DeMarco, "Why Does Software Cost So Much?," *IEEE Software*, vol. 10, 1993.

[5] O. Goldreich. (1996, 13.9.1012). "*Ho NOT to write a paper*." Available: http://www.wisdom.weizmann.ac.il/~oded/PS/writing.ps

[6] O. Goldreich. (2004, 13.9.2012). "*How to write a paper*." Available: http://www.wisdom.weizmann.ac.il/~oded/PS/re-writing.ps

[7] ISO, *ISO 9000:2005 Quality management systems - Fundamentals and vocabulary*. Geneva, Switzerland: ISO International Organization for Standardization, 2005.

[8] ISO/IEC, ISO/IEC TR 9126-2:2003: Software engineering - Product quality - Part 2: External metrics. Geneva, Switzerland: ISO International Organization for Standardization, 2003.

[9] ISO/IEC, *ISO/IEC TR 9126-3:2003: Software engineering - Product quality - Part 3: Internal metrics*. Geneva, Switzerland: ISO International Organization for Standardization, 2003.

[10] ISO/IEC, *ISO/IEC TR 9126-4:2004: Software engineering - Product quality - Part 4: Quality in use metrics*. Geneva, Switzerland: ISO International Organization for Standardization, 2004.

[11] R. E. Johnson, K. Beck, G. Booch, W. Cook, R. P. Gabriel, and R. Wirfs-Brock, "How to Get a Paper Accepted at OOPSLA," presented at the OOPSLA 1993.

[12] J. Kajiya, "*How to get your SIGGRAPH paper rejected*". Available: http://www.siggraph.org/publications/kajiya.pdf, 1993.

[13] M. E. Khan, "Different Approaches to White Box Testing Technique for Finding Errors" in *International Journal of Software Engineering and Its Applications* 5: 1–6, 2011.

[14] M. E. Latoschik and H. Tramberend, "Engineering Realtime Interactive Systems: Coupling & Cohesion of Architecture Mechanisms," in *Proceedings of the Joint Virtual Reality Conference of Euro VR - EGVE - VEC*, ed, 2010, pp. 25-28.

[15] M. E. Latoschik, S. Feiner, D. Schmalstieg, and C. Cruz-Neira, "Systems Engineering Science: Obsolete or Essential?,", IEEE VR 2012 panel, 2012.

[16] R. Levin, D. D. Redell, "An Evaluation of the Ninth SOSP Submissions" or "How (and How Not) to Write a Good Systems Paper", ACM SIGOPS Operating Systems Review, 17(3), 35-40, July 1983.

[17] A. Pavlovych and W. Stuerzlinger, "Target Following Performance in the Presence of Latency, Jitter, and Signal Dropouts," Graphics Interface, 2011.

[18] M. Ryschkewitsch, D. Schaible, and W. Larson, "The Art and Science of System Engineering," ed: NASA, 2009.

[19] H. Schulzrinne. (2012). "*Writing Technical Articles*". Available: http://www.cs.columbia.edu/~hgs/etc/writing-style.html

[20] M, Shaw. (2003). "Writing good software engineering research papers: minitutorial". In *Proceedings of the 25th International Conference on Software Engineering* (ICSE '03), 726-736.

[21] A. Spillner, T. Linz, and H. Schaefer, Software Testing Foundations: Rocky Nook inc., 2011.

[22] W. Stuerzlinger, C. Wingrave, "The Value of Constraints for 3D User Interfaces", Virtual Realities: Dagstuhl Seminar 2008, Springer Verlag, 203-224, Jan. 2011.

[23] J. Wilkes, "How to write a good [systems] paper," in *2006 EuroSys authoring workshop*, 2006.