The Forgotten Interfaces: A Critique of Component-based Models of Computing

Bran Selic[*†]
[*]Malina Software Corp., Canada.
[†]Monash University, Australia.

ABSTRACT Many theoretical methods for dealing with component-based software design have been proposed. Unfortunately, practically all of them ignore the inconvenient fact that software-based systems need an underlying hardware and software infrastructure to function. Consequently, all software applications are susceptible to idiosyncratic effects stemming from the hardware as well as supporting software layers.

This layering relationship between software and its supporting hardware and software platform is unprecedented in engineering practice, since it represents a unique blending of the conceptual and physical domains. If we are to build truly reliable component-based software applications, it seems critical that the nature of this relationship is clearly defined and understood. Ignoring it would be irresponsible and, more worryingly, potentially dangerous.

In this article, we first analyze the non-trivial nature of this fundamental and unique relationship and also argue what may be the reason why it has been neglected so often. This leads to the notion of *engineering correctness* as something that is fundamental to reliable software design and which, it is claimed, is inseparable from the traditional and established concept of *logical correctness*. It is argued further that this requires new engineering-oriented approaches to component-based software application design that must factor the influence of platforms. In the final part of the article, one possible approach for achieving this is described.

KEYWORDS Component-based Software Development, Verifying and Reasoning about Programs, Formal Languages, The Physics of Software.

"*All machinery is derived from nature.*"
Vitruvius, 1st century BC (Pollio 1914)

NOTE: It is the author's privilege and honour to dedicate this article to his good friend and esteemed colleague, Prof. Martin Gogolla, to mark his 65th birthday.

JOT reference format:
Bran Selic. *The Forgotten Interfaces: A Critique of Component-based Models of Computing*. Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2020.19.3.a2

1. Introduction: On traditional component-based models

Interest in the notions of software component and component-based design was originally inspired by the inherently modular essence behind the object paradigm of computing. As a consequence, many modern software applications are based on this approach. It has also led to the definition of a number of theoretical models in support of component-based design. These were inspired primarily by the potential of component-based design for much simpler formal treatment when compared to more traditional computational models, e.g., (Baumeister et al. 2006; Broy & Stølen 2001; Crnkovic et al. 2011; IBM 2010; ITU-T Z.100 2010; Ommering et al. 2000; Object Management Group 2006).

Unsurprisingly, practically all of these theoretical models share a common ontological framework consisting of a compact set of core ideas, such as those in the typical component metamodel shown in Fig. 1 (Baumeister et al. 2006). The key to its conceptual simplicity is that a component can be viewed simply as a "black box", which *fully encapsulates* its internals (the Assembly), and thus hides much of its complexity.[1] Each component can then be treated as a basic transformer of inputs to outputs (Fig. 2).[2]

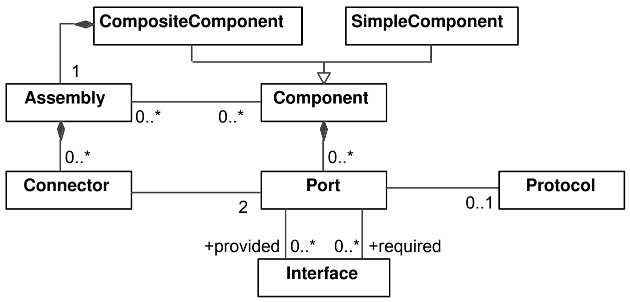

Figure 1 A typical theoretical component metamodel (Baumeister et al. 2006).

A component-based *system* is then realized by joining together specialized components by means of communication channels (Connectors) terminating on typed interface points of the components (Ports). The result is a design comprising a purpose-built network of components Fig. 3 (a). Moreover, the internal assemblies of components can themselves be realized recursively by networks of finer-grained components (i.e., CompositeComponent) as depicted in Fig. 3 (b).

What emerges from this is an aesthetically pleasing and conceptually simple model. Formal analysis of such systems can proceed either bottom-up or top-down using the exact same basic methods at every level, with the exception of the lowest primitive level. This incremental approach is certainly a much simpler problem compared to analyzing an equivalent traditional computer program.

The objective is to allow the ability to formally prove or disprove, by mathematically-based analysis, the logical correctness (or lack thereof) of a proposed design. This is based on the desire to harness the power of formal mathematical methods in support of software engineering, something that has proven essential to the success in all traditional engineering disciplines.

[1] This is an important difference from the type of encapsulation of objects found in most traditional object-oriented programming languages. Namely, in these languages any elements designated as "private" cannot be accessed from outside, but, these same internal elements can still freely access any publicly visible external element. In other words, encapsulation is unidirectional; it only works from the outside in, but not the other way around. As a result, classes of such objects are implicitly bound to any external entities that are accessed by their internal elements, which greatly diminishes their potential for reuse in different contexts.

[2] Note that, in principle, it is possible for a given service type to be associated with more than one input (output) port. For example, in cases of transaction-type services whose access involves a multi-stage protocol, each client of a component may require its own dedicated port, since each service access has its own specific state. (Not all component-based systems support this capability.)

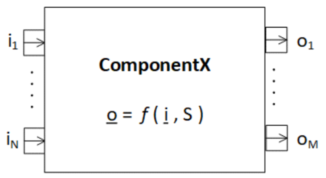

Figure 2 Prototypical component model: component as a services provider and an input-to-output transformer.

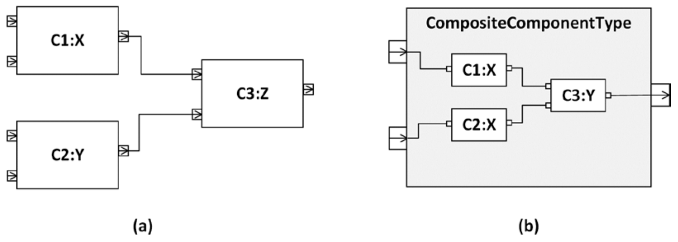

Figure 3 Examples of component-based design by means of (a) binding, (b) containment.

Needless to say, the ability to harness the immense power of mathematics —- which is key to the proven success of all traditional engineering disciplines —- is a noble and highly desirable objective, especially given that the design and construction of reliable software has proven to be very difficult and fraught with risk. Unfortunately, there is an inherent and often overlooked flaw in practically all of these theoretical component-based models. As discussed in the following sections, the source of the flaw is the unique and idiosyncratic nature of software. However, the reason it is commonly overlooked has historical roots in the culture of the formal methods community.

2. A historical perspective on the evolution of software and its impact on component models

Aristotle noted that: "In order to understand something, it is necessary to understand how it came to be." We apply this dictum here to help clarify why most current theoretical component models came to be flawed.

The very term "computing" exposes its origins. The first computers and most early applications were designed to deal with numerical problems. This line of thinking eventually led to a view that programming should be treated primarily as an exercise in applied logic.[3] Under the influence of early giants of software theory, such as Edsger Dijkstra and C.A.R. Hoare, this viewpoint has emerged as the dominant one — the one that is still being taught in most software engineering courses. This is readily reflected in the essential nature of the vast majority of traditional programming languages and programming theory. For example, despite the fact that much modern software involves some form of interaction with the physical world, *quantity* in most mainstream programming languages is still being treated

[3] For example, Dijkstra wrote: "I see no meaningful difference between programming methodology and mathematical methodology" (Dijkstra 1995b).

as a purely numeric concept, (e.g., through as integer and real, which are based on the "pure" mathematical notion of number). Yet, the ability to deal with physical quantities is at the core of all engineering. In fact, attempts to address software as an engineering problem have occasionally been deemed harmful by some notable theoreticians of software science.[4] This attitude has led to some astonishing and expensive disasters in practice, such as the case of the Mars Climate Orbiter spacecraft, whose very expensive failure was traced to a simple type mismatch of physical quantities (Stephenson et al. 1999). Detecting type mismatches is, of course, a primary responsibility of compilers; yet, because the quantities involved were stripped of their physical dimension, the mismatch went undetected.

Since those early days, the scope and range of computer applications has increased exponentially in practically all aspects of society, to the point where software has become central to its successful functioning. And, as noted, an increasing proportion of that software is intended to operate in and collaborate with physical and social environments. Thus, it must be capable of coping with all the complexity, physicality, and unpredictability of those environments. Yet, paradoxically, none of this is accommodated in the semantics of our most widely used programming languages.

It is, therefore, not surprising that this same "purist" bias is heavily present in modern theoretical component models. For instance, standard formal analysis approaches used with these kinds of models such as model checking and theorem proving, typically take a purely *qualitative* approach, which we shall refer to here as *checking for logical correctness*. This mostly means determining whether or not a design will conform to the desired specification, such as causality patterns and the like. On the other hand, quantitative characteristics, such as response time, availability, or security, are not covered by such analyses.[5] These are relegated to separate engineering-type analyses and *only after logical correctness has been established*. At first glance this might appear reasonable. After all, until the logic of a design is proven correct it seems superfluous to bother with "premature optimization"[6] of any quantitative *engineering* characteristics.

However, it is argued below that these engineering or "non-functional" characteristics — as they are, unfortunately, often called — can sometimes have a fundamental impact on the logic of a design. In other words, the design of component-based applications should not be merely a matter of applied (mathematical) logic, but, instead, it should be treated as a more general *engineering* problem, one that necessarily involves quantitative reasoning and trade-offs.

3. Why it is risky to separate functional and non-functional properties

It is widespread practice in both academia and industry to distinguish and keep separate so-called "functional" requirements from "non-functional" ones. This too is a consequence of the same historical bias towards a formalized mathematical view of software. The term "functional" is generally interpreted as to referring to the logical characteristics of a system, independently of its engineering characteristics. The very fact that these characteristics are referred to as "non-functional" suggests that they are a second-order concern. Otherwise, they would have been given a name that more accurately reflects what they are, as opposed to one that tells us what they are not.

This term reflects an assumption that the two facets can be dealt with separately. This is most unfortunate since experience has shown that the so-called "non-functional" (engineering) characteristics, such as security, performance, and availability, are *non-modular* and can rarely be localized within a design. Instead, they typically cut across many parts of a design and, therefore, can have a major impact on the design. This means that they are not easily modified or added post hoc. In fact, in many cases ensuring that these requirements are satisfied may take up the majority of the design effort (Selić 2016).

For example, the design for dealing with a "logical" requirement such as "messages must be delivered in the order they were sent" is likely to be quite different than for, say, an "engineering" type requirement such as "all messages must be delivered, in the order sent, within 0.5 seconds with a reliability of 99%". The latter imposes both temporal and reliability constraints. Compared to the former, it will undoubtedly require dedicated fault detection and recovery mechanisms as well as an implementation tuned towards efficiency. Moreover, if this application is intended to run over a physically distributed environment, *it may not even be feasible to come up with a design that fully satisfies such a requirement* due to constraints imposed by the physics of the situation (Fischer et al. 1985). On the other hand, such concerns are irrelevant if this same application is targeted to a single computing node.

In a very real sense, the software and hardware technology that underlies a software application can be considered as the *"raw material" out of which the software application is*

[4] Thus, Hoare noted in one interview: "I still feel glad to emphasize the duty, the defining characteristic of the pure scientist — probably to be found working in universities — who commit themselves absolutely to specialized goals, to seek the purest manifestation of any possible phenomenon that they are investigating, to create laboratories that are far more controlled than you would ever find in industry, and to ignore any constraints imposed by, as it were, realism. Further down the scale, people who understand and want to exploit results of basic science have to do a great deal more work to adapt and select the results, and combine the results from different sources, to produce something that is applicable, useful, and profitable on an acceptable time scale." (Frana 2002)

[5] A salient example of this can be seen in the various temporal logics that have been proposed. In these statements about time are abstracted down to "henceforth" or "eventually" (which, of course, could be a very long time) (Manna & Pnueli 1992). Many other formalisms take the same line, include many that adopt "zero time" assumptions, based on the view that computation is instantaneous (Lee 2002). While this may be valid in some cases, prudent engineering suggest that it is first necessary to validate such an assumption before proceeding to work adopt it.

[6] The quote: "Premature optimization is the root of all evil" was originally formulated by D. Knuth (https://en.wikiquote.org/wiki/Donald_Knuth), and was later echoed by C.A.R. Hoare. However, Knuth was referring to time and effort spent on low-level optimizing of algorithms rather than to system-level design.

constructed. And, as in all engineering, the properties of the construction material will not only impact the characteristics of the end result, but can also influence the design itself. It is well known, for example, that one cannot guarantee that a high-security application will be sufficiently secure if it is to execute on top of an inherently unsecure operating system. Analogous to traditional engineering, *software applications impose a level of "stress" on their platforms* and it is therefore necessary to determine if the platform[7] (i.e., the "raw material") has the necessary load-bearing capacity.

This problem is exacerbated by the fact that most of these varied engineering characteristics (e.g., execution speed, memory capacity, communications bandwidth) are idiosyncratic and require different analysis and design strategies. In other words, there is no universal solution that covers all of them; each one has to be addressed individually. To make this even worse, some of them are in direct conflict with each other. For instance, responsiveness and performance versus security. Security typically involves careful and graduated certification steps, all of which hinder performance and responsiveness. Similarly, high availability generally requires some form of redundancy, which has negative consequences on both cost and performance. This type of design conundrum involves complex trade-off analysis and demands deep multi-domain expertise and experience.

All of this supports the argument that software design extends beyond mere logical correctness and that it constitutes a *fully-fledged engineering problem* in the classical sense. Nevertheless, there are some critical aspects of the relationship between software and its platform that demand software-specific handling that make it unique in the field of engineering.

4. On the nature of the relationship between software applications and their platforms

Practically all component models require an underlying *component framework*, which is a kind of virtual machine responsible for executing the specification of a component-based application (Fig. 4). This framework provides the resources and mechanisms required to realize the precise semantics of its specific component model. This includes responsibility for the creation and destruction of components and their interconnections, support for specific forms of inter-component communications, as well as various additional services, such as timing, scheduling, memory storage provisioning, connection to physical devices such as sensors and actuators, etc.

A component framework also serves another key purpose: it isolates the component model from the underlying software-hardware platform providing, thereby, a degree of *platform independence*. This is the property of a software application that allows it to be executed on different platforms with little or no adjustment. Needless to say, this is a very useful and desirable capability given today's rapid advances in technology and a immense platform diversity.

This relationship between a component application and its underlying platform is typically represented by a layered structure, such as the one shown in Fig. 4.

[7] The *platform* of a software application is defined here refers not only to the underlying hardware, but also the full complement of software that it requires to execute successfully (see Fig. 4).

Although this kind of layered representation is frequently used, its precise semantics are not always clear. What exactly do we mean when we place one layer on top of another, as in Fig. 4?

Whether the layering relationship is between software and hardware or software and software, the semantics are the same. In essence, *it means that the elements in the upper layer take advantage of the services and resources of the lower layer to fully realize their implementation*. A direct, and often overlooked consequence of this is that, despite their encapsulation, *software components can never fully encapsulate their implementation*. In a very concrete sense, some part of the implementation of a component invariably "spills over" into the layer below.

For example, when a component sends a message to another component, it does so by invoking the communications service provided by its component framework. In fact, despite the simple and aesthetically pleasing image suggested by typical theoretical component models whereby components interact directly with each other via connectors, *the only direct interactions that a component has are with its framework*. These are the "forgotten interfaces" alluded to in the title of this article: *the interfaces between a software component and its platform*. The culture of the formal methods community prefers to ignore this reality, but this is precisely where all the idealized component models fail.

The difficulty comes from the fact that supporting layers rarely dedicate exclusive access of their resources to application-level entities such as components and connectors. Instead, *lower-layer resources are typically shared* between multiple upper-layer elements. This is clearly evident at the hardware layer, where multiple concurrent applications share the same CPU, physical memory, hardware busses, etc. However, the same typically applies in all supporting software layers.

This resource-sharing characteristic of platforms has the following two major consequences for component-based applications as well as theoretical component models:

(*a*) As already noted, a component can never fully encapsulate its implementation, because some of that implementation responsibility is taken over by the platform.
(*b*) Since platform resources are shared, there is a definite possibility of *contention*. As a critical and highly undesirable consequence, *components sharing the same platform resources can interfere with each in unpredictable ways*.

For example, a scheduling delay in processing of an arriving message for a component could lead to a missed deadline. Or, a message may be lost due to temporary lack of buffer space, and so on. The severity of such incidents depends on the applications and circumstances, but there cannot be any doubt that they open up the possibility of defects that cannot be uncovered by logical analysis of just the top-level component model.

This problem is further complicated if a platform simultaneously supports multiple, independently-designed applications — a very common situation in computing. If such applications are indeed designed independently (i.e., without knowledge of each

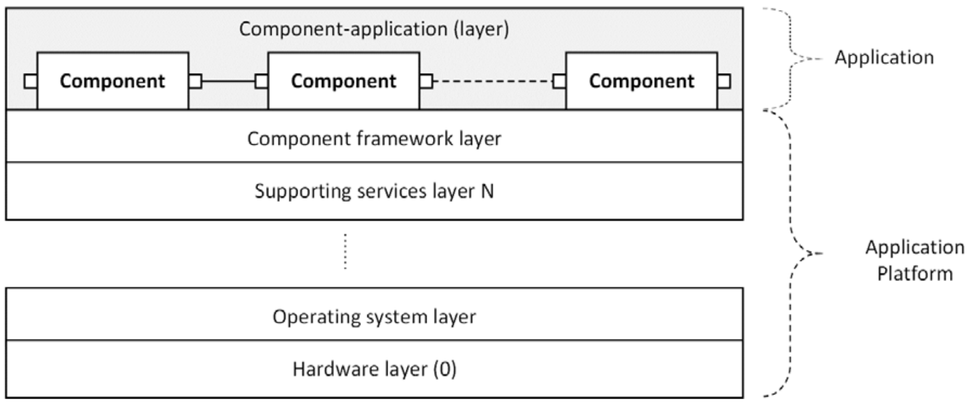

Figure 4 A component-based application, its supporting component framework, and its platform.

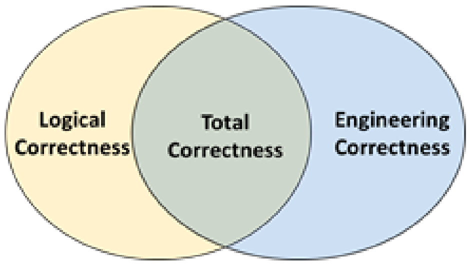

Figure 5 Total Correctness = Logical Correctness ∩ Engineering Correctness.

other), it may not even be feasible to predict in advance what kind of mutual interference to expect and what consequences that might bring about.

5. On design correctness

A design is deemed to be *correct* if it always performs according to its specification. This unquestionably includes meeting or exceeding all of its engineering *quality of service* requirements. We refer to the latter condition as *engineering correctness*. (Note that these characteristics are becoming of increasingly greater concern, as modern society becomes fundamentally dependent on software for its day-to-day functioning.[8])

From a pragmatic point of view, what is needed is *total correctness*, which means both logical and engineering correctness (Fig. 5). And, as discussed above, because of possible inter-dependencies between a system's logic and its engineering qualities, the only safe approach is to start from the intersection of the two, rather than focusing on and solving one them first and only then dealing with the other.

[8] The truth of this statement was dramatically illustrated by the recent coronavirus pandemic that enveloped the globe at the beginning of 2020. Without the support of the Internet and related telecommunications software, its consequences on society would undoubtedly have been orders of magnitude more devastating.

6. A potential engineering solution to platform independence

Because of the interdependence between logic and engineering concerns, it may seem at first that we may have to abandon the highly desirable goal of platform independent design. However, this is only if we interpret that term in an absolute sense, which, by the way, is how it is typically understood by many practitioners. Such an interpretation equates to expecting that a "platform independent" application will execute correctly *on any platform*, large or small, new or old, distributed or not. While there may be some simple software applications where such a property holds, they are likely to be trivial.

Instead, we choose to interpret that term in a conditional sense. That is, we constrain platform independence by qualifying it with a *range of platforms* on which the application in question can perform successfully. We refer to these as the *acceptable platforms* of the platform-independent application. These platforms have the necessary resources and with the necessary engineering characteristics that are required by the application.

Clearly, it would not be practical to explicitly list all possible individual platforms that might fall into the acceptable platform category for a given application, especially since new platforms appear all the time. However, it is possible instead to define an acceptable platform for an application in a *generic* way as *a set of resource and quality of service constraints that an acceptable platform must satisfy*.

One way of realizing this is to define the relationship between an application and its platform in terms of explicit interfaces between the application and its platform. This is partly already in place: every platform has a defined application programming interface (API), which is what applications running on that platform use to access its services.[9] But this is not enough, since we are also interested in engineering characteristics such as performance, security, and the like. This means associating relevant quantitative characteristics with every API, for example, worst case execution times for time-critical API calls. In addition, the notion of interface should be extended to other properties, such

[9] In the well-known 7-layer model of Open Systems Interconnection (OSI), these interface points are referred to as Service-Acess-Points (SAPs) (ITU-T X.200 1994).

as, say, a "CPU interface" that specifies the necessary minimum speed and other critical CPU characteristics, or a "memory interface" that identifies the type and capacity of required hardware memory, etc.

Once the full set of such required "interfaces" and their characteristics is defined for an application, it becomes possible to do a very precise analysis to determine the suitability of any given platform relative to a particular application. Naturally, this also requires that the corresponding engineering parameters of a candidate platform are known and explicitly stated. Unfortunately, this type of information is rarely available in full, so it may have to be derived through experimentation or by other means. International standards such as the MARTE standard from the Object Management Group (OMG) provide a conceptual framework for this purpose (Object Management Group 2008). A detailed description of how this framework can be used to realize the "acceptable platform" design pattern is provided in Section 6.5.4 of reference (Selić & Gérard 2014).

7. Summary

The inherently modular nature of the object concept as defined within the object-oriented approach to computing provided the inspiration behind the emergence of a number of theoretical models of component-based systems. These were primarily motivated by the conceptual simplicity of the component concept, which stands in contrast to the unavoidable complexity of traditional code-based approaches to computer software design. Published empirical and theoretical research of these approaches, e.g., (Baumeister et al. 2006; Broy & Stølen 2001; Crnkovic et al. 2011; IBM 2010; ITU-T Z.100 2010; Ommering et al. 2000; Object Management Group 2006), has shown that they can indeed greatly simplify formal mathematical analysis.

However, motivated primarily by the desire for simplicity and conceptual purity, most of these theoretical models are imperfect because they abstract away key real-world phenomena that could invalidate the results of formal analyses. The phenomena in question stem from the limitations imposed by the physical constraints of the underlying platform (i.e., the "raw material" out of which an application is ultimately constructed) as well as the sharing of platform resources, neither of which is under the direct control of the application.

Consequently, any conclusions drawn through formal analyses about the logical correctness of a component-based design using these oversimplified models may not be reliable and may lead to a false sense of security. It is, therefore, argued here that (a) pure logical correctness is insufficient and must be complemented with engineering correctness, which takes into account the limitations of the underlying platform, and (b) that platform constraints can fundamentally alter the "logical" design of an application. Given that society is increasingly more dependent on highly interactive and complex software-based systems, we must extend our theoretical models of component-based systems to accommodate this reality, regardless of how much it may corrupt their conceptual mathematical simplicity (based on the famous saying attributed to Einstein that: "Everything should be made as simple as possible, but no simpler".)

One possible approach for addressing this issue is outlined in Section 6, based on the "acceptable platform" design pattern. It proposes to make explicit the interfaces between that an application and its platform (i.e., the "hidden interfaces"). This includes more than just the set of all API signatures of the platform used by the application, but also explicit declaration of their individual *required quality of service* (i.e., "engineering") properties. In other words, these interfaces capture explicitly the type and characteristics of the platform for which the application is designed. These can then be directly matched against the corresponding interfaces (including their declared *provided qualities of service*) of a potential platform to reliably determine its ability to support the application. In essence, the issue is reduced to a type-compatibility analysis.

In summary, what is called for is a new approach to component-based models that goes beyond just mathematical logic. The design of software applications is, in fact, an engineering problem requiring engineering methods.

Appendix: Is Software Fundamentally Mathematical?

In his book, *Mathematics: The Loss of Certainty*, the mathematician Morris Kline describes the evolution of mankind's relationship with mathematics (Kline 1982) An early view is exemplified by the Pythagorean school of Ancient Greece, where it was theorized that the essence of the Universe was rooted in the concept of natural number. To their chagrin, this early "theory of everything" was fatally contradicted when they uncovered irrational numbers. (In fact, they were so distressed by this discovery, that they kept it secret from the rest of the world.) Kline then describes a series of subsequent mathematical inventions (or discoveries?), such as non-Euclidean geometries, which suggested that the relationship between the physical world and mathematics was not necessarily homomorphic, since mathematics was capable of constructs that had no equivalent in physical experience. The culmination of this progression came with Gödel's incompleteness theorems (Gödel 1986), which exposed a fundamental limitation of formal axiomatic (i.e., mathematical) systems to fully describe physical reality (Hawking 2002).

Nevertheless, the notion that mathematics was somehow at the very core of physical reality persisted in the scientific community and is still a mainstay with many. This line of thinking was further supported by the accuracy of predictions made using early mathematical models of physical phenomena in astronomy and physics. Eugene Wigner pointed out (Wigner 1960) that even if mathematics is not a direct reflection of reality, it is surprisingly accurate at modeling some important aspects of it — the only difficulty being is that we can never be sure at what point it ceases being trustworthy. The trustworthiness of mathematical analyses of physical systems is at the core of the issue discussed in this article.

This persistent belief in the power of mathematics has found its way into the domain of computer programming. The fact that computers are at their core a mechanization of Boolean logic, naturally suggests that mathematical methods can and do play a central role in the field. One of the most prominent proponents of this view, Edsger Dijkstra, a true giant of software science, argued that software design should be a matter of applied mathematics claimed and that, consequently, program design should align with mathematical methodology (see footnote 3 on page 4). Consequently, he bemoaned the introduction of the interrupts into computers:

> "*[The interrupt] was a great invention, but also a Pandora's Box ... essentially, for the sake of efficiency, concurrency [became] visible ... and then, all Hell broke loose.*" (Dijkstra 1995a)

Specifically, Dijkstra was lamenting the loss of determinism and predictability that would result from the interference caused by unpredictable interruptions of computations performed on an otherwise predictable machine. But, what he seems to have missed is that the purpose of interrupts goes far beyond mere efficiency (why "mere" efficiency — is efficiency not important?). In fact, interrupts are a fundamental means by which computers interact with the external (physical) world in real time. This may not have been so obvious in the early days of computing, but a large proportion of modern software applications involve some degree of asynchronous interactions with that world. And, experience has shown that these are probably the most difficult applications to get right, since we know that the physical world is highly diverse, concurrent, and unpredictable. Yet, we are demanding that software not only to recognize this complexity but also cope with it safely and reliably.

As argued in this article, it should be clear by now that computers and computing and software applications are not separable from the physical world. Therefore, software design cannot and should not be reduced to "mere" mathematical methodology.

Acknowledgments

The author expresses his gratitude to Prof. Antonio Vallecillo for his helpful and encouraging feedback as well as his invaluable assistance in the preparation of the LaTeX version.

References

Baumeister, H., Hacklinger, F., Hennicker, R., Knapp, A., & Wirsing, M. (2006). A component model for architectural programming. *Electron. Notes Theor. Comput. Sci.*, *160*, 75–96. doi: 10.1016/j.entcs.2006.05.015

Broy, M., & Stølen, K. (2001). *Specification and development of interactive systems*. Springer. doi: 10.1007/978-1-4613-0091-5

Crnkovic, I., Sentilles, S., Vulgarakis, A., & Chaudron, M. R. V. (2011). A classification framework for software component models. *IEEE Trans. Software Eng.*, *37*(5), 593–615. doi: 10.1109/TSE.2010.83

Dijkstra, E. W. (1995a). *My recollections of operating system design.* Retrieved from https://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF (ED 1303)

Dijkstra, E. W. (1995b). *Why american computer science seems incurable.* Retrieved from https://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1209.PDF (ED 1209)

Fischer, M. J., Lynch, N. A., & Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. *J. ACM*, *32*(2), 374–382. doi: 10.1145/3149.214121

Frana, P. (2002, July). *An interview with charles antony richard hoare. oral history 357.* Retrieved from https://conservancy.umn.edu/bitstream/handle/11299/107362/oh357th.pdf

Gödel, K. (1986). Some basic theorems on the foundations of mathematics and their implications. In *Kurt gödel collected works* (Vol. I, p. 144-195). Oxford University Press.

Hawking, S. (2002). *Gödel and the end of the universe.* Retrieved from http://www.hawking.org.uk/godel-and-the-end-of-physics.html (Stephen Hawking Public Lectures)

IBM. (2010). *Service Component Architecture (SCA), Document Version 1.0.*

ITU-T X.200. (1994). *Data Networks and Open System Communications: Open Systems Interconnection — Model and Notation – Basic Reference Model: The Basic Model (version 07/94).*

ITU-T Z.100. (2010). *Specification and Description Language -– Overview of SDL-2010.*

Kline, M. (1982). *Mathematics: The loss of certainty*. Oxford University Press.

Lee, E. A. (2002). Embedded software. *Advances in Computers*, *56*, 55–95. doi: 10.1016/S0065-2458(02)80004-3

Manna, Z., & Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems - specification*. Springer. doi: 10.1007/978-1-4612-0931-7

Object Management Group. (2006, April). *CORBA Component Model, Version 4.0.* (OMG Document formal/06-04-01)

Object Management Group. (2008, June). *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Beta2.* (OMG Document ptc/2008-06-09)

Ommering, R., van der Linden, F., Kramer, J., & Magee, J. (2000). The koala component model for consumer electronics software. *IEEE Computer*, *33*(3), 78–85. doi: 10.1109/2.825699

Pollio, V. (1914). *The ten books on architecture*. Dover Publications Inc. (Morgan, M.H. translator)

Selić, B. (2016). Programming $\subset$ modeling $\subset$ engineering. In *Proc. of isola 2016* (Vol. 9953, pp. 11–26). doi: 10.1007/978-3-319-47169-3_2

Selić, B., & Gérard, S. (2014). *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*. Morgan Kaufmann.

Stephenson, A. G., LaPiana, L. S., Mulville, D. R., Rutledge, P. J., Bauer, F. H., Folta, D., . . . Norvig, P. (1999, November). *Mars Climate Orbiter Mishap Investigation Board Phase 1 Report.* Retrieved from https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf

Wigner, E. (1960). The unreasonable effectiveness of mathematics in natural sciences. *Communications on Pure and Applied Mathematics*, *13*(1), 1–14. doi: 10.1002/cpa.3160130102

About the author

Branislav (Bran) Selić is President and Founder of Malina Software Corp., a Canadian company providing IT consulting and training services to industry. He is also Director of Advanced Technology at Zeligsoft (2009) Limited in Canada. On the academic side, he is currently adjunct professor at Monash University (Australia) and a visiting researcher at the University of Sydney (Australia). In the course of 45+ years of direct professional experience in industry, he was involved in the design and implementation of a variety of large-scale software systems primarily in the cyber-physical systems domain. Bran is the principal author of two technical textbooks, editor of seven others, and has over 100 publications in various technical journals and conferences. He holds a Mag. Ing. degree in Systems Theory (1974) and a Dipl. Ing degree and Electrical Engineering (1972), both from the University of Belgrade.