DOWNLOAD PDF FORMAL DEVELOPMENT OF PROGRAMS AND PROOFS

Chapter 1 : CiteSeerX â€" Citation Query Formal Development of Programs and Proofs

Specifically, this book addresses the derivation of programs and their proofs of correctness. Its underlying theme is that in programming, as in mathematics, the elegance of derivations is an essential goal.

The simplest form of this technique consists of feeding various inputs to the tested program and verifying the correctness of the output. In some cases exhaustive testing is possible, but often it is not. More sophisticated versions of this technique try to choose the inputs so that all, or at least the majority of the possible execution paths are examined. Independent of their degree of sophistication, these empirical methods do not actually that the respective program is correct. The only thing we can actually prove with the empirical approach is that the program is not correct - for this purpose, a single example of incorrect behavior suffices. Absent an observation of incorrect behavior, however, we can not know - in general - whether the program is correct, or whether we have just not tested it with an input that would trigger an error. As we all know, incorrect program behavior is pervasive. Some program errors are only irritating, but some can endanger life and limb. Would you like the manufacturer to "think" that the program is correct based on a number of empirical tests, or would you prefer an unambiguous and definitive proof? Once we establish the importance of program correctness as both an engineering and theoretical problem, we can turn our attention to actually solving it. It can be proven, for example, that there are no general algorithms that check or prove that a program correctly implements a given specification. We can not give general solutions for much simpler problems either. For example, there is no algorithm that would be able to decide whether an arbitrary program will terminate execution for a given input. Since we can solve almost no general problem in the area of program correctness we must set more modest goals. Luckily, the lack of general solutions does not mean that program correctness can not be proven in certain particular cases, or in a context that is restricted in some sense. Today we are going to discuss two program correctness proofs that use the substitution model and induction. Induction Proofs Induction is a technique we can use to prove that certain properties hold for each element of a suitably chosen infinite set. The most common form of induction is that of mathematical induction, whereby we establish the truth of statement for all natural numbers, or - more generally - for all elements of a sequence of numbers. Induction can also be performed on more complicated sets, like pairs of non-negative integers, or binary trees see below. An inductive argument can be thought of as being not a proof per se, but a recipe for generating proofs. To reduce the possibility of error, we will structure all our induction proofs rigidly, always highlighting the following four parts: We prefer that you use precise mathematical notation for all the statements that you make see our examples below. If you are not familiar enough with mathematical notation, however, we also accept semi-informal statements in plain English, assuming that they are correct, unambiguous, and complete. Of course, we can infer some things about n - it can not be a general real number, or a negative number, because then the phrase "the first n numbers" by the way, what kind of numbers? As mentioned in passing above, induction is feasible on sets that are more general than the set of natural numbers. Technically, induction can be performed on all well-founded sets. Aside A well-founded set is a set endowed with a partial order such that the set contains no infinite descending chains. Equivalently, and perhaps more intuitively, a well-founded set is a partially ordered set in which every non-empty subset has a minimal element. A partial order on a set S is a reflexive, transitive, and antisymmetric binary relation on S x S the Cartesian product of S with itself. It is possible for a subset of a partially ordered set to have more than one minimal element. Fix an element s 0 of S, and consider elements s 1 , s 2 , The property of having no infinite descending chains is equivalent to saying that no matter what our choice for s 0 is, and no matter how we choose s 1 , s 2 , The fact that all descending chains are finite guarantees that sooner or later we will reach a problem that can not be reduced further, a base case. The minimal elements of S form the base cases to be considered in the induction proof. Consider, for example, the set of pairs of natural numbers i. With this definition, any two pairs p, q and r, s can be compared i. Such a set is a totally ordered set; totally ordered set are also partially ordered sets. Quite clearly, 0, 0 is the smallest element of this set. Let us change the definition above slightly. Can you say which are the minimal elements of S? This form of induction is called strong induction. We write down the proof by following

precisely the four steps specified above: We perform induction on the set of natural numbers N. We perform the proof by using the substitution model. To reduce the size of the proof we will skip all non-essential steps in the application of the substitution model. In keeping with our previous discussion, we should imagine that there is a let statement wrapping the definition of fact, and that fact 0 is the part of the let between in and end. We start applying the substitution model after the function expression defining fact has already been substituted in fact 0. We will do this in other proofs as well, primarily to reduce the length of the proof. We should point out that you should not confuse the two uses of n: If you are confused by this, you can rewrite the definition of fact to use another identifier for its argument, say, k. By taking a few shortcuts in the substitution model, we establish that this expression represents in fact fact n , which we know from the induction hypothesis to be equal to n!. This proves the induction step and completes the induction proof. Induction on the Length of Lists Consider the function definition below: To simplify our proofs, we will assume that functions null, hd, and tl are or act like predefined unary operators. One might have considered writing mapsquares using a case expression. The style guide actually advises you against that, since a case with two alternatives is really equivalent to an if expression. Even if we ignore the style guide, we have not defined the substitution model for case expressions, so we could not use the alternative form of mapsquares in our proof. The description above, while reasonable, allows for some elements of ambiguity. For example, we have not indicated explicitly that the length of the output list is identical to the length of the input list. To avoid these problems, we will use a more formal notation in our proof. We use this form here as a notational convenience. Here are the four parts of the induction proof: We instantiate P n to get P 0 as follows: Assume that P n holds for an arbitrary natural number n. Note that since y has length n, the induction hypothesis applies to it. We have the following relations: In other words, mapsquares y returns a list containing the squares of the last n elements of x, in their original order. By rewriting these expressions we get:

Chapter 2 : Formal development of programs and proofs - CORE

In , C.A.R. Hoare gave the proof of correctness and termination of a rather complex algorithm, in a paper entitled Proof of a program: calendrierdelascience.com is a handmade proof, where the program is given together with its formal specification and where each step is fully justified by mathematical reasoning.

DOI Call for Papers Certified Programs and Proofs CPP is an international forum on theoretical and practical topics in all areas, including computer science, mathematics, and education, that consider certification as an essential paradigm for their work. Certification here means formal, mechanized verification of some sort, preferably with production of independently checkable certificates. Fri 6 Oct Full paper submission deadline: Wed 11 Oct Notification: Tue 14 Nov Camera-ready deadline: Sun 26 Nov Conference dates: Mon 8 - Tue 9 Jan Topics of interest We welcome submissions in research areas related to formal certification of programs and proofs. The following is a suggested list of topics of interests to CPP. This is a non-exhaustive list and should be read as a guideline rather than a requirement. Shorter papers are welcome and will be given equal consideration. Abstracts must be submitted by October 6, AOE. The deadline for full papers is October 11, AOE , and authors have the option to withdraw their papers during the window between the two. Submissions must be written in English and provide sufficient detail to allow the program committee to assess the merits of the paper. They should begin with a succinct statement of the issues, a summary of the main results, and a brief explanation of their significance and relevance to the conference, all phrased for the non-specialist. Technical and formal developments directed to the specialist should follow. References and comparisons with related work should be included. Papers not conforming to the above requirements concerning format and length may be rejected without further consideration. Whenever appropriate, the submission should come along with a formal development, using whatever prover, e. Such formal developments must be submitted together with the paper as auxiliary material, and will be taken into account during the reviewing process. Please do so by including a link to your files in the text of your paper, or by sending a zip or tar file to the PC chairs at cpp easychair. The results must be unpublished and not submitted for publication elsewhere, including the proceedings of other published conferences or workshops. The PC chairs should be informed of closely related work submitted to a conference or journal in advance of submission. Original formal proofs of known results in mathematics or computer science are welcome. One author of each accepted paper is expected to present it at the conference. For any questions about the formatting or submission of papers, please consult the PC chairs.

Chapter 3 : Mathematical proof - Wikipedia

Note: Citations are based on reference standards. However, formatting rules can vary widely between applications and fields of interest or study. The specific requirements or preferences of your reviewing publisher, classroom teacher, institution or organization should be applied.

Statistical proof The expression "statistical proof" may be used technically or colloquially in areas of pure mathematics , such as involving cryptography , chaotic series , and probabilistic or analytic number theory. See also " Statistical proof using data " section below. Computer-assisted proof Until the twentieth century it was assumed that any proof could, in principle, be checked by a competent mathematician to confirm its validity. Some mathematicians are concerned that the possibility of an error in a computer program or a run-time error in its calculations calls the validity of such computer-assisted proofs into question. In practice, the chances of an error invalidating a computer-assisted proof can be reduced by incorporating redundancy and self-checks into calculations, and by developing multiple independent approaches and programs. Errors can never be completely ruled out in case of verification of a proof by humans either, especially if the proof contains natural language and requires deep mathematical insight. Undecidable statements[edit] A statement that is neither provable nor disprovable from a set of axioms is called undecidable from those axioms. One example is the parallel postulate , which is neither provable nor refutable from the remaining axioms of Euclidean geometry. Mathematicians have shown there are many statements that are neither provable nor disprovable in Zermelo-Fraenkel set theory with the axiom of choice ZFC , the standard system of set theory in mathematics assuming that ZFC is consistent ; see list of statements undecidable in ZFC. Heuristic mathematics and experimental mathematics[edit] Main article: Experimental mathematics While early mathematicians such as Eudoxus of Cnidus did not use proofs, from Euclid to the foundational mathematics developments of the late 19th and 20th centuries, proofs were an essential part of mathematics. Early pioneers of these methods intended the work ultimately to be embedded in a classical proof-theorem framework, e. Visual proof[edit] Although not a formal proof, a visual demonstration of a mathematical theorem is sometimes called a " proof without words ". The left-hand picture below is an example of a historic visual proof of the Pythagorean theorem in the case of the 3,4,5 triangle. Animated visual proof for the Pythagorean theorem by rearrangement. A second animated proof of the Pythagorean theorem. Some illusory visual proofs, such as the missing square puzzle , can be constructed in a way which appear to prove a supposed mathematical fact but only do so under the presence of tiny errors for example, supposedly straight lines which actually bend slightly which are unnoticeable until the entire picture is closely examined, with lengths and angles precisely measured or calculated. Elementary proof An elementary proof is a proof which only uses basic techniques. More specifically, the term is used in number theory to refer to proofs that make no use of complex analysis. For some time it was thought that certain theorems, like the prime number theorem , could only be proved using "higher" mathematics. However, over time, many of these results have been reproved using only elementary techniques. Two-column proof[edit] A two-column proof published in A particular way of organising a proof using two parallel columns is often used in elementary geometry classes in the United States. In each line, the left-hand column contains a proposition, while the right-hand column contains a brief explanation of how the corresponding proposition in the left-hand column is either an axiom, a hypothesis, or can be logically derived from previous propositions. The left-hand column is typically headed "Statements" and the right-hand column is typically headed "Reasons". It is sometimes also used to mean a "statistical proof" below , especially when used to argue from data. Statistical proof using data[edit] Main article: Statistical proof "Statistical proof" from data refers to the application of statistics , data analysis , or Bayesian analysis to infer propositions regarding the probability of data. While using mathematical proof to establish theorems in statistics, it is usually not a mathematical proof in that the assumptions from which probability statements are derived require empirical evidence from outside mathematics to verify. In physics , in addition to statistical methods, "statistical proof" can refer to the specialized mathematical methods of physics applied to analyze data in a particle physics experiment or observational study in physical cosmology.

Inductive logic proofs and Bayesian analysis[edit] Main articles: Inductive logic and Bayesian analysis Proofs using inductive logic , while considered mathematical in nature, seek to establish propositions with a degree of certainty, which acts in a similar manner to probability , and may be less than full certainty. Inductive logic should not be confused with mathematical induction. Proofs as mental objects[edit] Main articles: Psychologism and Language of thought Psychologism views mathematical proofs as psychological or mental objects. Mathematician philosophers , such as Leibniz , Frege , and Carnap have variously criticized this view and attempted to develop a semantics for what they considered to be the language of thought , whereby standards of mathematical proof might be applied to empirical science. Ending a proof[edit] Main article: Sometimes, the abbreviation "Q. This abbreviation stands for "Quod Erat Demonstrandum", which is Latin for "that which was to be demonstrated".

Chapter 4 : math - Why can't programs be proven? - Stack Overflow

Reviewer: Haim I. Kilov "Good programming is the art and science of keeping things simple" (p. viii). The Year of Programming () at the University of Texas at Austin, which originated from this conviction, resulted in a series of somewhat unusual proc more.

February 23, Brendan Cournoyer Informal vs. In his book, Informal Learning: With the formal learning bus, "the driver decides where the bus is going; the passengers are along for the ride. What Is Formal Learning? Formal learning programs are typically synonymous with full-scale learning management systems LMS , with courses and curricula mapped in a very structured way. As a result, content is generally created by a specified group of instructional designers and trainers. The content development process usually lasts longer with formal learning programs, as those involved are often tasked with creating long, thorough presentations and publishing via a potentially complex LMS tool set. Content and learning materials can be delivered via a traditional class room training model, complete with lectures, required reading and scheduled testing. Live webinars and screen-sharing technology can also be used so that remote learners can attend the required sessions. Formal learning is a popular choice for companies that wish to have more control over the learning experience of their employees. There are a variety of LMS options to choose from, each with varying levels of price, support, functionality and flexibility. B2B Sales Has Changed: Informal learning programs , on the other hand, provide a lot more flexibility in the way content is both created and consumed. By removing the formality of a full-scale LMS, companies are usually able to create more content quickly and deliver it to their audiences in the way that makes the most sense. The idea here is that rather than limiting the responsibilities of content development to a few instructional designers, subject matter experts from across the organization can now become part of the process. Since anyone can create learning resources quickly and easily, more content can be developed by those who best understand the needs of the learners. Informal learning allows companies to save time on tedious live training sessions that are associated with many though not all formal programs and tend to cut into employee productivity. In other words, a well-developed informal learning program provides information to learners anytime, anywhere. How to Decide Between Informal vs. Is it difficult to leverage the expertise of others in the company? Do you have trouble getting timely messages out quickly? How often does certain content need to be updated? Audience â€” Are live formal training sessions wasting too much time? Are the sessions having the desired effect? Do learners have trouble finding follow-up information when they need it? Tracking and Reporting â€” How many people need to generate reports? Do you need more or less control over required courses? What exactly do you need to track? Hopefully that helps clear up the basic differences between formal and informal learning. Learn more about training best practices:

Chapter 5 : Formal Development of Programs and Proofs by Edsger W. Dijkstra

Specification[edit] Formal methods may be used to give a description of the system to be developed, at whatever level s of detail desired. This formal description can be used to guide further development activities see following sections ; additionally, it can be used to verify that the requirements for the system being developed have been completely and accurately specified. The need for formal specification systems has been noted for years. Development[edit] Once a formal specification has been produced, the specification may be used as a guide while the concrete system is developed during the design process i. If the formal specification is in an operational semantics, the observed behavior of the concrete system can be compared with the behavior of the specification which itself should be executable or simulateable. Additionally, the operational commands of the specification may be amenable to direct translation into executable code. If the formal specification is in an axiomatic semantics, the preconditions and postconditions of the specification may become assertions in the executable code. Verification[edit] Once a formal specification has been developed, the specification may be used as the basis for proving properties of the specification and hopefully by inference the developed system. Human-directed proof[edit] Sometimes, the motivation for proving the correctness of a system is not the obvious need for reassurance of the correctness of the system, but a desire to understand the system better. Consequently, some proofs of correctness are produced in the style of mathematical proof: A "good" proof is one which is readable and understandable by other human readers. Critics of such approaches point out that the ambiguity inherent in natural language allows errors to be undetected in such proofs; often, subtle errors can be present in the low-level details typically overlooked by such proofs. Additionally, the work involved in producing such a good proof requires a high level of mathematical sophistication and expertise. Automated proof[edit] In contrast, there is increasing interest in producing proofs of correctness of such systems by automated means. Automated techniques fall into three general categories: Automated theorem proving , in which a system attempts to produce a formal proof from scratch, given a description of the system, a set of logical axioms, and a set of inference rules. Model checking , in which a system verifies certain properties by means of an exhaustive search of all possible states that a system could enter during its execution. Abstract interpretation , in which a system verifies an over-approximation of a behavioural property of the program, using a fixpoint computation over a possibly complete lattice representing it. Some automated theorem provers require guidance as to which properties are "interesting" enough to pursue, while others work without human intervention. Model checkers can quickly get bogged down in checking millions of uninteresting states if not given a sufficiently abstract model. Proponents of such systems argue that the results have greater mathematical certainty than human-produced proofs, since all the tedious details have been algorithmically verified. The training required to use such systems is also less than that required to produce good mathematical proofs by hand, making the techniques accessible to a wider variety of practitioners. Critics note that some of those systems are like oracles: There is also the problem of " verifying the verifier "; if the program which aids in the verification is itself unproven, there may be reason to doubt the soundness of the produced results. Some modern model checking tools produce a "proof log" detailing each step in their proof, making it possible to perform, given suitable tools, independent verification. The main feature of the abstract interpretation approach is that it provides a sound analysis, i. Moreover, it is efficiently scalable, by tuning the abstract domain representing the property to be analyzed, and by applying widening operators [11] to get fast convergence. Applications[edit] Formal methods are applied in different areas of hardware and software, including routers, Ethernet switches, routing protocols, and security applications. There are several examples in which they have been used to verify the functionality of the hardware and software used in DCs[clarification needed]. Intel uses such methods to verify its hardware and firmware permanent software programmed into a read-only memory [citation needed]. Dansk Datamatik Center used formal methods in the s to develop a compiler system for the Ada

programming language that went on to become a long-lived commercial product. There are many areas of hardware, where Intel have used FMs to verify the working of the products, such as parameterized verification of cache coherent protocol, [17] Intel Core i7 processor execution engine validation [18] using theorem proving, BDDs , and symbolic evaluation , optimization for Intel IA architecture using HOL light theorem prover, [19] and verification of high performance dual-port gigabit Ethernet controller with a support for PCI express protocol and Intel advance management technology using Cadence. Formal methods are most likely to be applied to safety-critical or security-critical software and systems, such as avionics software. For sequential software, examples of formal methods include the B-Method , the specification languages used in automated theorem proving , RAISE , and the Z notation. In functional programming , property-based testing has allowed the mathematical specification and testing if not exhaustive testing of the expected behaviour of individual functions. The Object Constraint Language and specializations such as Java Modeling Language has allowed object-oriented systems to be formally specified, if not necessarily formally verified. For concurrent software and systems, Petri nets , process algebra , and finite state machines which are based on automata theory - see also virtual finite state machine or event driven finite state machine allow executable software specification and can be used to build up and validate application behavior. Another approach to formal methods in software development is to write a specification in some form of logicâ€”usually a variation of first-order logic FOL â€”and then to directly execute the logic as though it were a program. There is also work on mapping some version of English or another natural language automatically to and from logic, and executing the logic directly. Examples are Attempto Controlled English , and Internet Business Logic, which do not seek to control the vocabulary or syntax. A feature of systems that support bidirectional English-logic mapping and direct execution of the logic is that they can be made to explain their results, in English, at the business or scientific level. You can help by converting this section to prose, if appropriate. Editing help is available.

Chapter 6 : Formal methods - Wikipedia

THE STRUCTURE OF THE PROGRAM DERIVATION PROCESS The ideal we seek in program derivation is to begin with a formal specification of a problem, and then to proceed by a well-defined process of transformation and refinement until eventually a program implementation is derived.

Introduction Formal methods are system design techniques that use rigorously specified mathematical models to build software and hardware systems. In contrast to other design systems, formal methods use mathematical proof as a complement to system testing in order to ensure correct behavior. As systems become more complicated, and safety becomes a more important issue, the formal approach to system design offers another level of insurance. Formal methods differ from other design systems through the use of formal verification schemes, the basic principles of the system must be proven correct before they are accepted [Bowen93]. Traditional system design has used extensive testing to verify behavior, but testing is capable of only finite conclusions. In contrast, once a theorem is proven true it remains true. It is very important to note that formal verification does not obviate the need for testing [Bowen95]. Formal verification cannot fix bad assumptions in the design, but it can help identify errors in reasoning which would otherwise be left unverified. In several cases, engineers have reported finding flaws in systems once they reviewed their designs formally [Kling95]. Roughly speaking, formal design can be seen as a three step process, following the outline given here: During the formal specification phase, the engineer rigorously defines a system using a modeling language. Modeling languages are fixed grammars which allow users to model complex structures out of predefined types. This process of formal specification is similar to the process of converting a word problem into algebraic notation. In many ways, this step of the formal design process is similar to the formal software engineering technique developed by Rumbaugh, Booch and others. At the minimum, both techniques help engineers to clearly define their problems, goals and solutions. However, formal modeling languages are more rigorously defined: Several engineers who have used formal specifications say that the clarity that this stage produces is a benefit in itself [Kling95]. As stated above, formal methods differ from other specification systems by their heavy emphasis on provability and correctness. By building a system using a formal specification, the designer is actually developing a set of theorems about his system. By proving these theorems correct, the formal Verification is a difficult process, largely because even the simplest system has several dozen theorems, each of which has to be proven. These tools can prove simple theorems, verify the semantics of theorems, and provide assistance for verifying more complicated proofs. Once the model has been specified and verified, it is implemented by converting the specification into code. As the difference between software and hardware design grows narrower, formal methods for developing embedded systems have been developed. An alternative to this approach is the lightweight approach to formal design. In a lightweight design, formal methods are applied sparingly to a system. This approach offers the benefits of formal specification, but also avoids some of the difficulties. Formal methods are viewed with a certain degree of suspicion. There are several reasons for this, but most of the problems seem to be a result of misapplication. Most formal systems are extremely descriptive and all-encompassing, modeling languages have generally been judged by their capacity to model anything. Unfortunately, these same qualities make formal methods very difficult to use, especially for engineers untrained in the type theory needed for most formal systems. In addition,the mathematics required for formal methods is becoming a more prominent fixture of engineering curricula, engineering schools in Europe are already requiring courses in VDM, Z and similar formal specifications. Ultimately, formal methods will acquire some form of acceptance, but compromises will be made in both directions: Provability And Automated Verification Formal methods are distinguished from other specification systems by their emphasis on correctness and proof, which is ultimately another measure of system integrity. Proof is a complement, not a substitute, for testing. Testing cannot demonstrate that a system operates properly; it can only demonstrate that the system works for the tested cases. Because testing cannot demonstrate that the system should work outside the tested cases, formal proof is necessary. Formally proving computer systems is not a new idea. Knuth and Dijkstra have written extensively on the topic, although their

methods of proof are based on the traditional mathematical methods. In pure sciences, proofs are verified through extensive peer review before publication. Given the cost and time requirements of systems engineering, traditional proving techniques are not really applicable. Because of the costs of hand verification, most formal methods use automated theorem proving systems to verify their designs. Automated theorem provers are best described as mathematical CAD tools: Benefits Of Formal Models Formal methods offer additional benefits outside of provability, and these benefits do deserve some mention. However, most of these benefits are available from other systems, and usually without the steep learning curve that formal methods require. By virtue of their rigor, formal systems require an engineer to think out his design in a more thorough fashion. In particular, a formal proof of correctness is going to require a rigorous specification of goals, not just operation. This thorough approach can help identify faulty reasoning far earlier than in traditional design. Engineers using the PVS system, for example, reported identifying several microcode errors in one of their microprocessor designs. Traditionally, disciplines have moved into jargons and formal notation as the weaknesses of natural language descriptions become more glaringly obvious. There is no reason that systems engineering should differ, and there are several formal methods which are used almost exclusively for notation. Unlike many other design approaches, the formal verification requires very clearly defined goals and approaches. In a safety critical system, ambiguity can be extremely dangerous, and one of the primary benefits of the formal approach is the elimination of ambiguity. Weaknesses Of Formal Methods: Bowen points out that formal methods are generally viewed with suspicion by the professional engineering community, and the propensity of tentative case studies and advocacy papers for the formal approach would seem to support his thesis [Bowen93]. There are several reasons why formal methods are not used as much as they might be, most stemming from overreaching on the part of formal methods advocates. Because of the rigor involved, formal methods are always going to be more expensive than traditional approaches to engineering. However, given that software cost estimation is more of an art than a science, it is debatable exactly how much more expensive formal verification is. In general, formal methods involve a large initial cost followed by less consumption as the project progresses; this is a reverse from the normal cost model for software development. While not a universal problem, most formal methods introduce some form of computational model, usually hamstringing the operations allowed in order to make the notation elegant and the system provable. An excellent example comes from SML. Statements of proof in SML depend on a purely functional programming model: Traditionally, formal methods have been judged on the richness of their descriptive model. While an all-encompassing formal description is attractive from a theoretical perspective, it invariably involved developing an incredibly complex and nuanced description language, which returns to the difficulties of natural language. Case studies of full formal methods often acknowledge the need for a less all-encompassing approach. This reasoning has led to the lightweight approach to formal specification. The Lightweight Approach The flaws in formal specifications have been heavily focused on in the past few years, leading to several alternate approaches. The traditional view of formal methods as all-encompassing highly abstracted schemes has led to formal methods being all-encompassing, extremely rigorous, and very expensive. While theoretically appealing, formal methods have generally been ignored by engineers in the field. The lightweight approach to formal design recognizes that formal methods are not a panacea: In a lightweight design, formal methods are used in specific locations, and different formal methods may be used in different subsystems, ideally playing to the strengths of each method [Easterbrook 98]. In such a system, Petri Nets might be used to describe the communications protocol, and a LARCH system might be used to model the data storage. For other parts of the system, formal specifications might be avoided entirely: The lightweight approach is a traditional engineering compromise, and there is a tradeoff. As formal methods become more common, engineers will have to learn type theory, modern algebra and proof techniques. Ultimately, engineers will have to think more like mathematicians. Available tools, techniques, and metrics Larch: A general high-level modeling language, and a collection of implementation dialects designed to work with specific programming languages. Standard Meta-Language is a strongly typed functional programming language originally designed for exploring ideas in type theory. SML has become the formal methods workhorse because of its strong typing and provability features. As with most automated theorem proving systems, HOL is a computer-aided

proof tool: Originally designed for modeling communications, Petri Nets are a graphically simple model for asynchronous processes. Relationship to other topics Requirements Specifications. The goal of the formal approach is the same as testing: While formalists have sometimes claimed that formal methods can replace testing, a more realistic approach is to say that formal verification complements testing. Ultimately, formal methods are another verification tool. Formal verification is a way of ensuring the correctness of the theory behind the design. Conclusions While formal systems are attractive in theory, their practical implementations are somewhat wanting. By attempting to describe all of any system, formal methods have overreached, and generally failed. The lightweight approach, which provides a compromise between the necessities of engineering and the goals of formal design, provides a good compromise and is the most productive route for future research.

Chapter 7 : Lecture 9: Proofs of Program Correctness

Download PDF: Sorry, we are unable to provide the full text but you may find it at the following location(s): calendrierdelascience.com (external link).

Chapter 8 : Informal vs. Formal Learning: What's the Difference? | Brainshark

How is Formal Development of Programs and Proofs abbreviated? FDPP stands for Formal Development of Programs and Proofs. FDPP is defined as Formal Development of Programs and Proofs very rarely.