Software Engineering

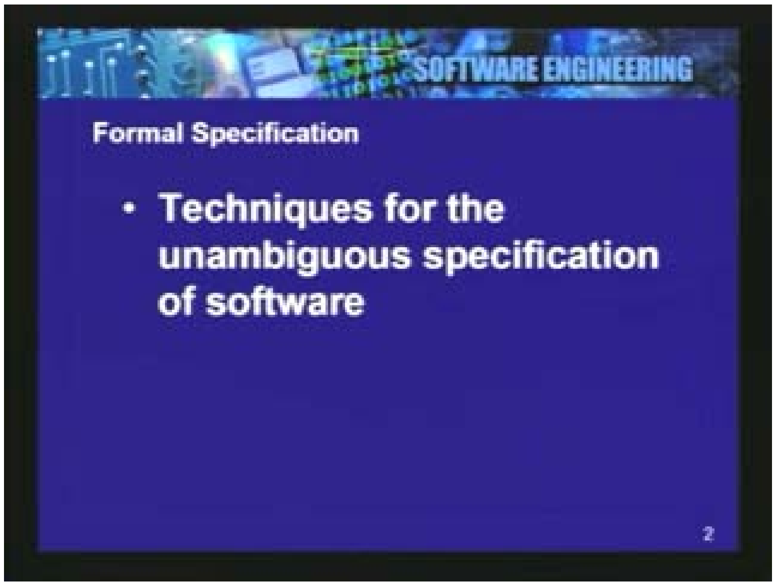
Prof. Umesh Bellur
Computer Science & Engineering
Indian Institute of Technology, Bombay

Lecture - 6

Formal Specification

Last class we took a look at what the requirement specification engineering process was like. And we saw that one of the ways by which the requirements can be specified by using natural language or English like syntax. But there are also problems to that approach, we saw some alternative approaches such as graphical notations, templates and forms that can be used in a structured approach and lastly we briefly got introduced to mathematical forms of specifications. In this lecture we are going to go deeper into formal methods or using mathematical notations in order to specify requirements of a system, and how they can be used for building the rest of the system. What exactly is a formal specification?

(Refer Slide Time: 1:46)



SOFTWARE ENGINEERING

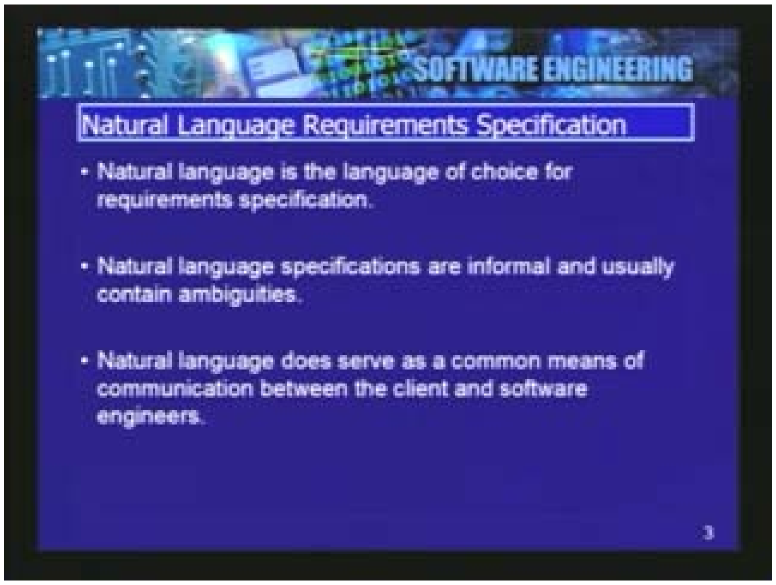
Formal Specification

- Techniques for the unambiguous specification of software

2

A formal specification as the name implies is a technique for unambiguous specification of software to be build. One of the problems that we saw by using the natural language representation was that even though it is the language of choice largely for requirements of today, it has several problems associated with it. One of the problems is that it gives rise to several ambiguities when the person who reads the document typically ends up interpreting the document in his or her own words.

(Refer Slide Time: 1:58)


SOFTWARE ENGINEERING

Natural Language Requirements Specification

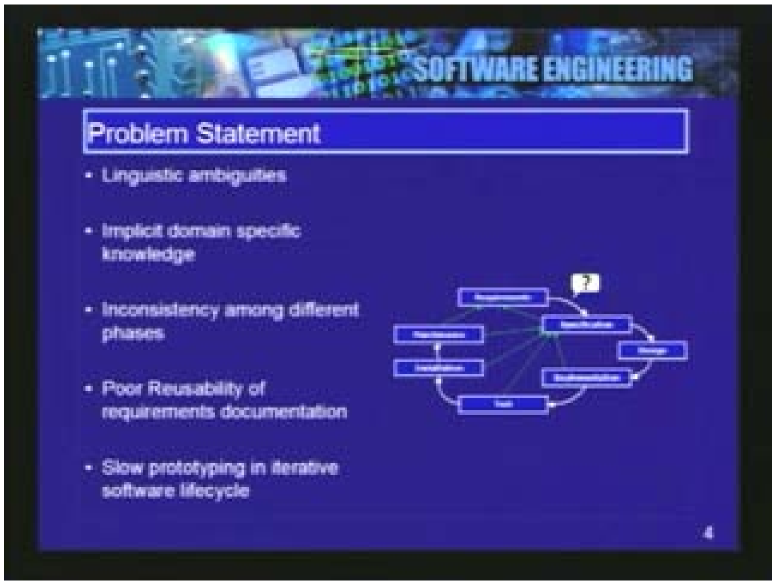
- Natural language is the language of choice for requirements specification.
- Natural language specifications are informal and usually contain ambiguities.
- Natural language does serve as a common means of communication between the client and software engineers.

3

These ambiguities is going to vary from person to person as a result of which two different people reading the specification can get two different ideas of what the system is to perform. The natural language does not serve as a good means for modularizing the software. So if a bunch of English statements are written it could be that, it cut across the many different modules, they are not related statements for example, there is no way of ensuring that all the statements are at the same level of consistency or at same level of detail. So with all these problems, it is kind of like hooves us to look for methods that give us a more structure or a formal means of writing down the specifications of software and that is what we are going to look into.

Just to summarize, in the problems that can occur with natural language specifications, linguistic ambiguities is one of them. The implicit domain specific knowledge which is often not represented, because the context is often assumed in the case of English specification. Let us take the insulin pumps example. The context of the medical world in which these terms are going to be used is not necessarily well defined. And there can be certain assumptions that are made which will cause a person down the line to read the specification to not necessarily understand it in the right way. There can be a lot of inconsistency between different phrases. There can be grammatical errors in the document itself which can lend itself for different interpretation.

(Refer Slide Time: 3:08)


SOFTWARE ENGINEERING
?

Problem Statement

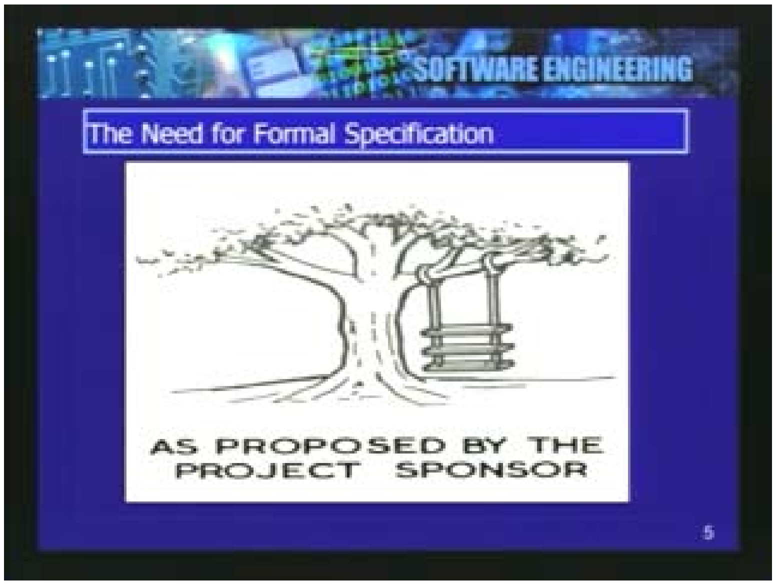
- Linguistic ambiguities
- Implicit domain specific knowledge
- Inconsistency among different phases
- Poor Reusability of requirements documentation
- Slow prototyping in iterative software lifecycle

4

It is very hard to reuse natural language specifications. For example if I wanted to reuse the specification of a GUI screen for another system which was actually being built for a different purpose, it would not be very easy to do so. The prototyping cycle is quite long in the case of natural language specifications. Simply because, a human has to read the specification and has to interpret and understand the specification and then work on the prototyping, whereas certain alternative techniques can be made available such as code generation for example, which is one of the goals of formal specifications. So just to give an interesting example, here is a case study, not a formal case study but a situation which depicts the need for formal specifications. The slide actually shows the project that is proposed by the project sponsor. Basically, this is a client who wants to build a piece of hardware in this particular case.

What we will see in the next successive set of slides is really how this can get twisted out of shape when you are using natural language and what comes out as an end product really has not much of relation to what is really being input to the whole process. So the second step is that somebody has to specify this in the project request and write it in their natural language that is being used, you can see that the initial specification of what the project sponsor wrote out, is kind of been malformed a little bit.

(Refer Slide Time: 4:51)

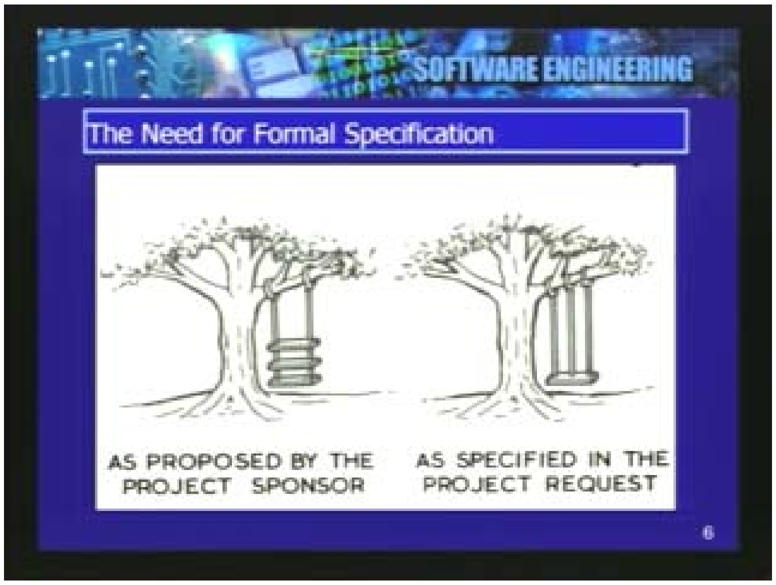


SOFTWARE ENGINEERING

The Need for Formal Specification

5

(Refer Slide Time: 5:32)



SOFTWARE ENGINEERING

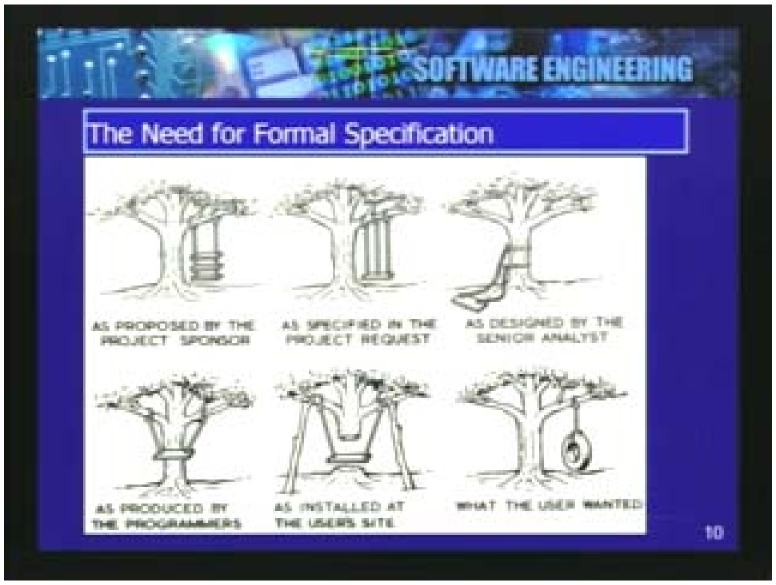
The Need for Formal Specification

AS PROPOSED BY THE PROJECT SPONSOR
AS SPECIFIED IN THE PROJECT REQUEST

6

As we go along we are going to see that the senior analyst who read this specification and interpreted it completely wrongly in this case. For example he may have looked at the flexibility of the rope and then designed a really flexible system but it is not a swing which is what wanted in the first place.

(Refer Slide Time: 6:31)


SOFTWARE ENGINEERING

The Need for Formal Specification

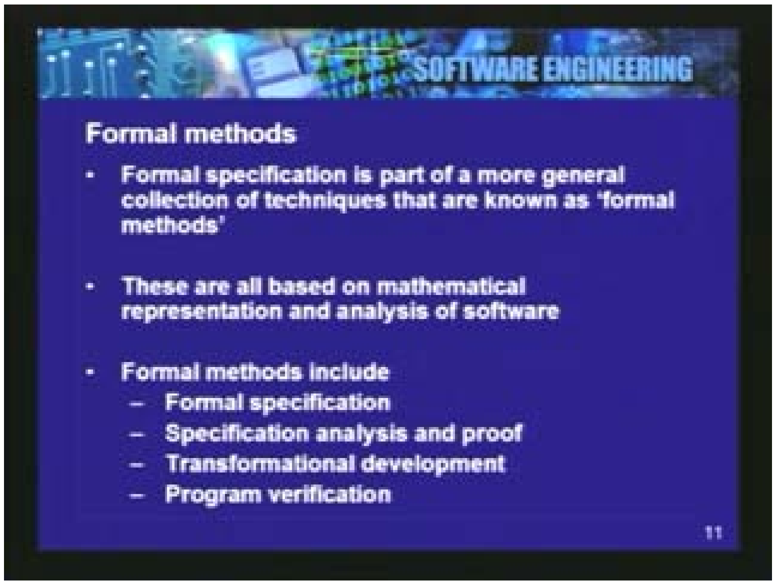
AS PROPOSED BY THE PROJECT SPONSOR
AS SPECIFIED IN THE PROJECT REQUEST
AS DESIGNED BY THE SENIOR ANALYST
AS PRODUCED BY THE PROGRAMMERS
AS INSTALLED AT THE USERS SITE
WHAT THE USER WANTED
10

As we go along, we will see that this thing is evolving in a different way because the programmers who are at the next step in the software development process are going to interpret the analyst statements yet very differently. Then finally they installed it at the user's site and this is how it looked. But what the person who was going to use this actually meant was, simply a tyre hanging from a piece of string off of a tree, so that his child could play a swing on this piece of tyre.

What this is trying to depict is the need to be able to carry through a set of requirements or a set of statements, all the way from the requirements part of the software process, through the analysis part, to the building part, of the implementation part and installation part finally. And all of these need to reflect what the user originally asked for in the requirements document. This is why it is important to have a means of specification where you are going to be able to verify it formally, may be using tools. And then ensure that you can go from phase to phase such as from requirements to analysis phase, from the analysis to design phase, from the design to the implementation phase and so on, without loosing the track of what the original requirements were meant to be.

Formal methods which are a broader set of methods and simply doing the specifications themselves, specification is part of broader collection of techniques for formal methods. Most of these formal methods are based on some kind of rigorous mathematical specification of the software. The formal method includes three to four steps typically.

(Refer Slide Time: 7:36)


SOFTWARE ENGINEERING

Formal methods

- Formal specification is part of a more general collection of techniques that are known as 'formal methods'
- These are all based on mathematical representation and analysis of software
- Formal methods include
  - Formal specification
  - Specification analysis and proof
  - Transformational development
  - Program verification

11

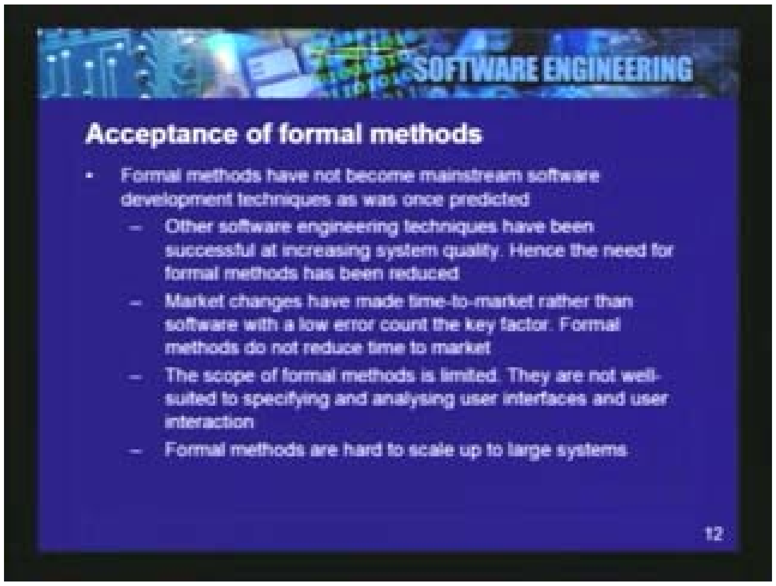
- The first one is the formal specification itself, where the software requirements are laid out and then specified by the program analyst and a formal method using algebraic steps as we shall see later in this presentation.
- The second step is that of a specification analysis and proof. Specification can be checked for consistency which you cannot do in the natural language kinds of specifications.
- The third step is one of transformational development. What this really means in this particular case is that, typically if you take a look at the software development life cycle process, you go from requirements to analysis and you go from requirement analysis to design and each one of these steps needs to have a certain degree of continuity in between them. For example, the output of the requirement phase is the requirements document, which is being fed into the analysis phase. The output of the analysis phase is an initial or a very high level design of the system, which is fed to the design phase. The output of the design phase is a detailed design of the system, which can then be taken and then implemented by the programmer or the developer.

This development process typically has as its interfaces between the various phases. The analysis document is created, and then the document is read by the designer. He produces a design document that is read by the developer. He then produces an implementation. What this lends itself to is, these people, developer/analysts end up trying to interpret what has been written down on a piece of paper or within a document and that interpretation can vary from person to person. If there were means to take the requirements verified and then transform the requirements automatically into code which is the end product of the software development process. That would essentially ensure that we retain the continuity between the various phases of the software development life cycle. And that is what transformational development means. What it is trying to focus on is the idea of taking a requirement specification which has been specified formally, verifying it for consistency then generating the design from it.

Allow a human to verify the design and then generating the code from that point onwards generating the test from the specification, testing the code against the specification. All of this can be automated or it can be controlled through machine tools which will essentially ensure that the human ambiguity that is brought into the picture because of persons reading the specification or reading the design document and then producing analysis, or producing implementations and that variability will get taken away. And every time there is consistent transformation that is taking place from a specification to a piece of code which is really what we are after, the end of the day.

Finally program verification refers to checking or testing to make sure that what we have build indeed conforms to the specification that was given to us in the first step. Program verification also can be automated when we use formal methods because test cases can be generated from the specifications themselves and the test cases can then be applied against the final end product of the process which is a code. This can help us verify whether it indeed conforms to the specification that was set out at the beginning of the process. One of the things or caveats that we have to keep in mind is as we go along this path, is that formal specifications or formal methods in general have not taken of the way that they were predicted to say way back in the late seventies and the early eighties.

(Refer Slide Time: 11:44)


SOFTWARE ENGINEERING

Acceptance of formal methods

- Formal methods have not become mainstream software development techniques as was once predicted
  - Other software engineering techniques have been successful at increasing system quality. Hence the need for formal methods has been reduced
  - Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market
  - The scope of formal methods is limited. They are not well-suited to specifying and analysing user interfaces and user interaction
  - Formal methods are hard to scale up to large systems

12

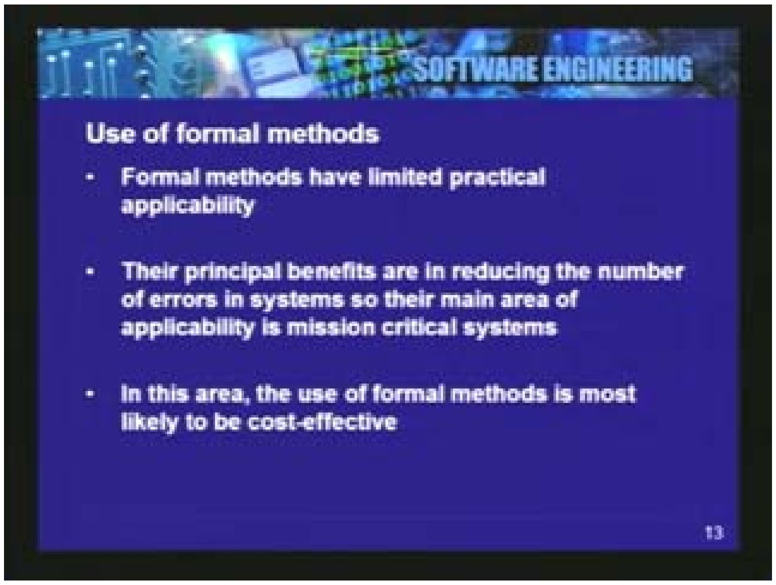
A lot of hope was being held out at that point of time that these would become main stream software development methodologies. But so far they have not happened and we should kind of take a look at some of the reasons why this is the case before we get into the techniques and how we are going to end up doing this. The first reason for this is that, other software engineering techniques have been catching up with respect to increasing the efficiency as well as decreasing the number of errors introduced in the software development process. One of the advantages of the formal specification is that humans cannot introduce errors into this process because they are not really as involved like in alternative software development methodologies practices.

A lot of other techniques have come up; especially process based techniques that can help control system quality and keep it under check. So the needs for formal methods have come down, but that is not really a primary reason for this. The more important reason is that formal methods have kind of failed to scale to really large systems. In order to formally specify medium sized enterprise class system like an ERP system, it would take a quite a bit of mathematical expertise which is largely absent from the industry, even today.

As a result of which formal methods have primarily been tried away from, although they have been applied in critical systems which is where we will see the main applicability has been. So the scope is kind of limited they are not very well suited for example also to analyzing or specifying a front end GUI based systems. They are primarily very good at back end system specification, but lots of systems today are GUI driven. Also the market factors and market changes is something that we cannot underestimate. As dictated the time to development or the time to market is really short because of the competition around us and the need for getting these systems into production very fast has kind of driven that.

Formal methods take a quite bit of time to develop because it is focused on doing the job right in the first go as supposed to an incremental method of evolution, where errors or bugs are found and fixed as the system evolves rather than a system being perfect from the word go. The use of formal methods because of all these reasons has a certain limited practical applicability at this point in time. Although the principle benefits are primarily in the areas of reducing number of errors and they are primarily applied in the area of machine critical systems.

(Refer Slide Time: 14:50)


SOFTWARE ENGINEERING

Use of formal methods

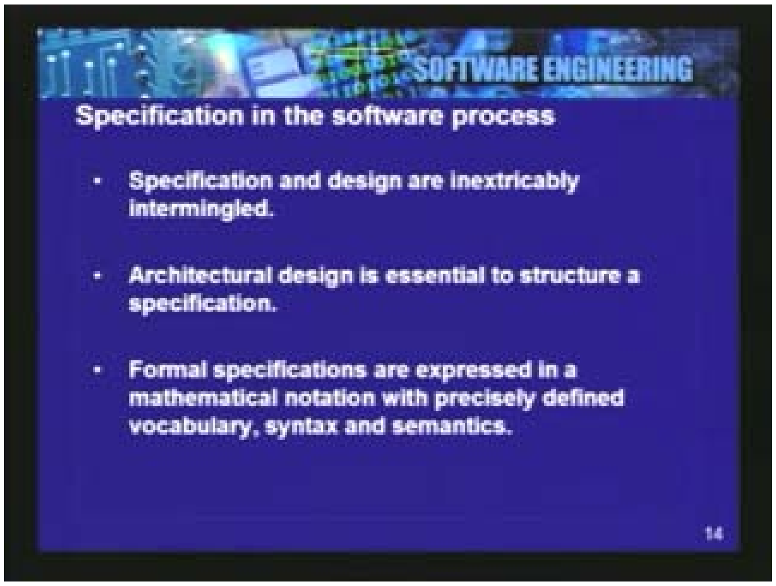
- Formal methods have limited practical applicability
- Their principal benefits are in reducing the number of errors in systems so their main area of applicability is mission critical systems
- In this area, the use of formal methods is most likely to be cost-effective

13

A good example would be software that runs avionics, software that runs large planes that you are flying today is all formally tested. Certain part of the software at least, the smaller parts which are very critical, something like the parts that controls the rudder or that controls the breaking and so on is formally tested. The critical pieces of software are essentially built using formal specifications and formal methods. The rest of the system is built using regular software engineering processes and practices and these are put together.

Where does formal specification fit into the software development life cycle will be the next question that we would tend to ask. The specification and design although mix with each other, formal specification falls somewhere between the requirements specification stage which may be done with natural language first just for the sake of convenience and between the design stages. It is actually an analysis methodology more than a requirement specification methodology as we shall see on this slide. The first step is really that of requirements definition that is done by the user that is typically still done in the natural language and that cannot be avoided. It is then translates into a requirement specification. Now it is possible to argue that the need for requirement specification step can be avoided and we can straight away jump into the formal specification step.

(Refer Slide Time: 15:39)



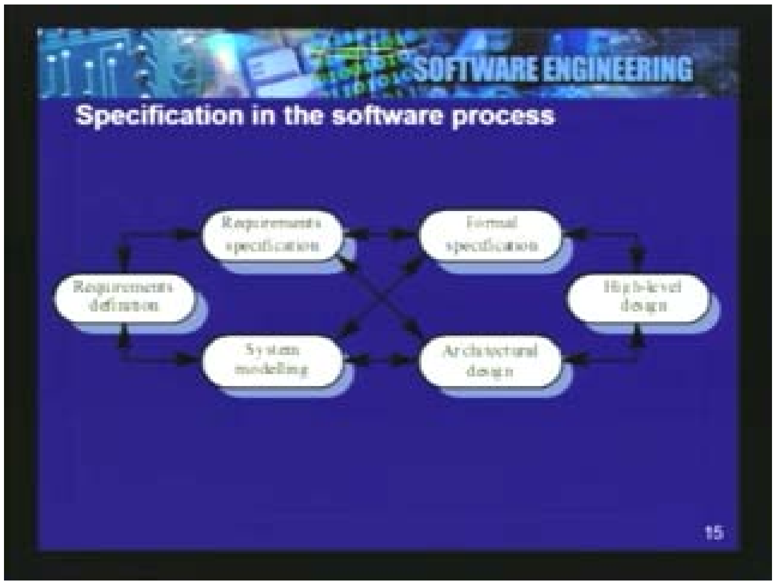
SOFTWARE ENGINEERING

Specification in the software process

- Specification and design are inextricably intermingled.
- Architectural design is essential to structure a specification.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

14

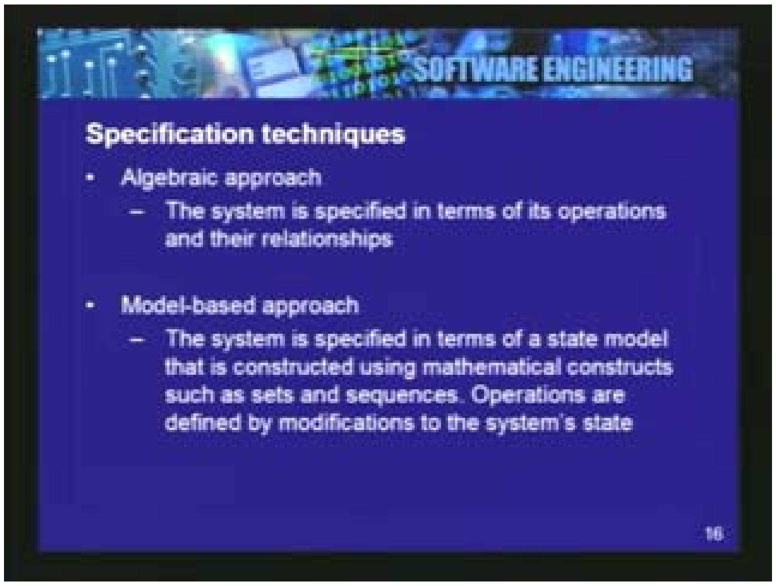
(Refer Slide Time: 16:41)


SOFTWARE ENGINEERING

Specification in the software process

But it is generally not recommended. Once we get into the formal specification step then the high level design can actually be generated out to the formal specification and different system models can be driven off of the formal specification. For example UML models can be generated of the formal specification, other test models for example can also be generated out of the formal specification.

(Refer Slide Time: 17:04)


SOFTWARE ENGINEERING

Specification techniques

- Algebraic approach
  - The system is specified in terms of its operations and their relationships
- Model-based approach
  - The system is specified in terms of a state model that is constructed using mathematical constructs such as sets and sequences. Operations are defined by modifications to the system's state

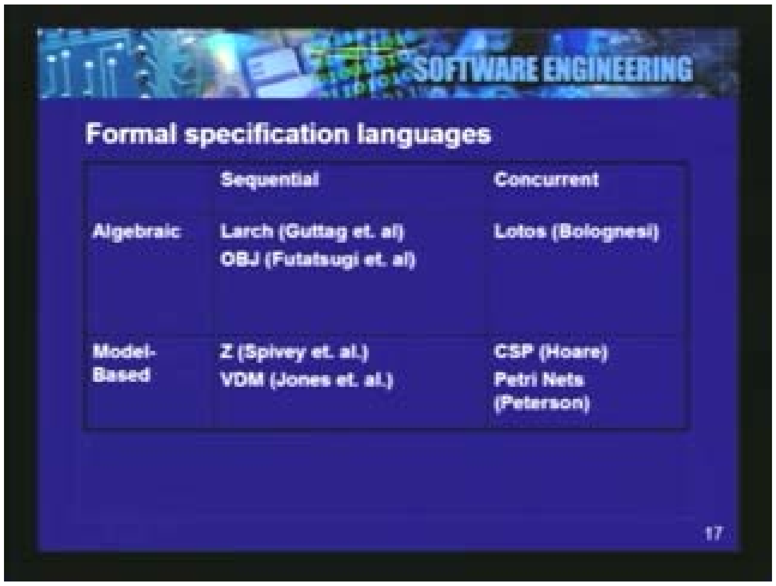
16

Specification techniques for formal specifications largely fall in two categories or two classes. The first one is called an algebraic approach and the second one is called model based approach.

There are different languages techniques and tools which fall in to each one of these categories. The algebraic approach essentially focuses on specifying the system in terms of its operations or its interfaces and the relationships between these various interfaces. Whereas the model based approach concentrates more on specifying state models and specifying the system using mathematical concepts such as sets and sequences.

So the operations are mainly being defined as modifications to the state and the state is central to the model based approach whereas the operations themselves are interface specifications and ex-humatic specifications those that are sent to the algebraic approach. Of course both of them do have the node. The formality really is introduced by the notion of axioms in the case of algebraic approach and by the notion of invariants in the case of model based approach. There is a degree of similarity exists between these two approaches. But the difference mainly is that the model based approach is primarily state based, whereas the algebraic specifications approach is primarily based on the operational signatures of the various functions that are going to make up the interface of the module that is being specified. For formal specification there are several languages evolved over a course of time as can be seen from this slide and we have kind of categorized the different languages into four different categories.

(Refer Slide Time: 18:50)


SOFTWARE ENGINEERING

Formal specification languages

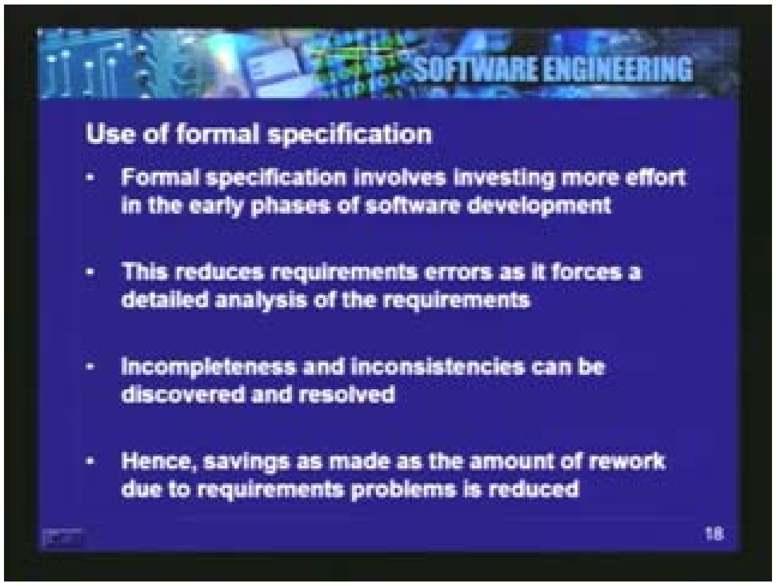
	Sequential	Concurrent
Algebraic	Larch (Guttag et. al) OBJ (Futatsugi et. al)	Lotos (Bolognesi)
Model-Based	Z (Spivey et. al.) VDM (Jones et. al.)	CSP (Hoare) Petri Nets (Peterson)

17

The first category is that of the specification languages those focuses on sequential programs and is algebraic in nature. The two primary languages in this category are ‘Larch’ and ‘OBJ’. When you move to concurrent languages there are fewer languages in this case using algebraic specifications and concurrency essentially introduces the notion of multiple modules working together at the same time. Whereas in model based approaches, the quite popular ones are ‘Z’ and ‘VDM’ that you might have heard of before and, ‘CSP’ and ‘Petri Nets’ in the case of concurrent approaches.

One of the things that we have to be aware of before we dive in to the details here is that, formal specification involves investing a lot more effort upfront, because you are taking time to mathematically model the constructs. It takes a lot more time in the early phase of the software development. Whereas it will ease out in the later phases because transformational development is the focus of formal methods, whereas hand based manual development is really the focus of other software engineering practices. So it also ends up reducing a lot more errors over the life cycle of the software, because of the verification steps that are done upfront and the consistency checks that are performed upfront and the inconsistencies or the incompleteness that may exist within the specification, for example the lack of a context defining a certain term can be removed upfront early in the lifecycle or can be restored or resolved.

(Refer Slide Time: 20:20)


SOFTWARE ENGINEERING

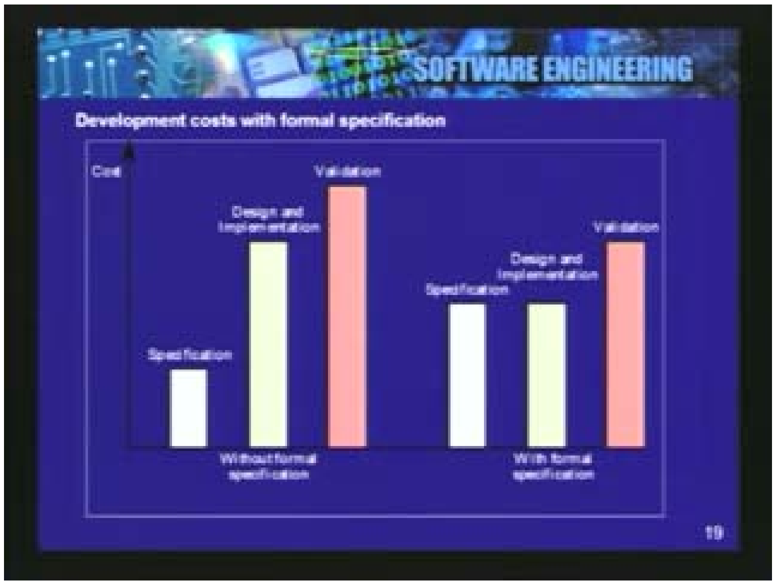
Use of formal specification

- Formal specification involves investing more effort in the early phases of software development
- This reduces requirements errors as it forces a detailed analysis of the requirements
- Incompleteness and inconsistencies can be discovered and resolved
- Hence, savings as made as the amount of rework due to requirements problems is reduced

18

In terms of the overall life cycle if we take a look at how the cost gets spread out, here is the slide which shows the comparative differences of time that is spent on specification versus design implementation, validation with and without formal specifications.

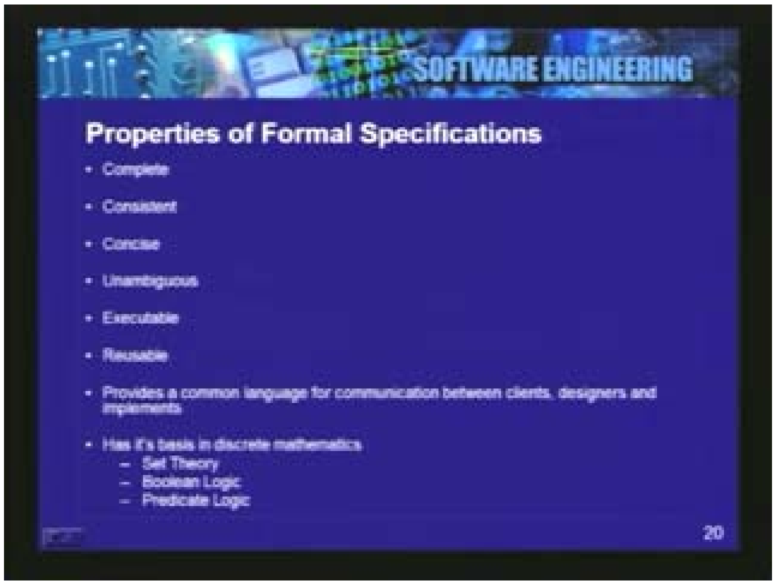
(Refer Slide Time: 20:55)



It can be clearly seen that in the case of formal specifications a lot of time is spent in the specification itself, and not as much time is spent in the design and implementation because it is transformational in nature. And validation is also fairly simple to do. In the case of building software without the use of formal methods, 'specification' does not take a whole lot of time, because it is done using natural language, whereas the design implementation, the validation phases are the ones ends up taking most time. One other thing that is not shown here is the fact that the numbers of errors that get introduced in a software development life cycle without formal specification are quite high. The cost of error as you go down in the process, it gets more and more expensive to fix.

Errors are best handled at the requirement stage as it is suppose to at the implementation stage. Where the design might have to be changed, requirements might have to be re read and these specifications might have to be rewritten and so on. So it is better to try and catch the errors upfront within the life cycle and that ends up costing a lot less during the overall development process as supposed to other methods. What are the some of the properties of formal specifications? We have talked a lot about what they are at a very high level.

(Refer Slide Time: 22:44)


SOFTWARE ENGINEERING

Properties of Formal Specifications

- Complete
- Consistent
- Concise
- Unambiguous
- Executable
- Reusable
- Provides a common language for communication between clients, designers and implements
- Has it's basis in discrete mathematics
  - Set Theory
  - Boolean Logic
  - Predicate Logic

20

The first thing is, the specification has to be complete. In a sense that everything needs to be described within the user document definition will have to be carried over appropriately. The context that is required, that the user might have assumed in the case of writing the user requirements documents or the requirements definition document will now have to be actually spelt out. That is what completeness is all about. It has to be consistent. For example there cannot be ambiguities of sort that we saw with respect to the number of chips and the power that has to be consumed. The example that we saw in the last class can be carried over here. In that case there was an inconsistency with the two requirements. One requirement being that fewer chips have to be used, which might be more chips have to be put on to one chip that would consume a lot more power.

The second requirement is that the amount of power consumed by any one chip has to be lower. So there was like a fundamental inconsistency between those two requirements and that is what consistency really means. Conciseness is one of the problems with natural language specification. The requirements documents for even medium sized systems can run into hundreds of pages. It is not something that software developers going to take kindly in order to read, interpret and then build the system using. So conciseness of the specification will come into play automatically when you are using formal methods or mathematical notations. In fact that is one of the by products using these kinds of notations.

Unambiguous: This is kind of related to the consistent property. Executable is an important property and this forms the basis of transformational development that we talked about few minutes ago. Essentially a lot of the specifications written in the languages that we saw, ‘Larch’, ‘Z’, VDM and so on are directly executable. What does it means that is, designs can be generated from these specification documents. The designs can then be turned in to the code automatically if the designer gives the green signal. The code can then be directly executed.

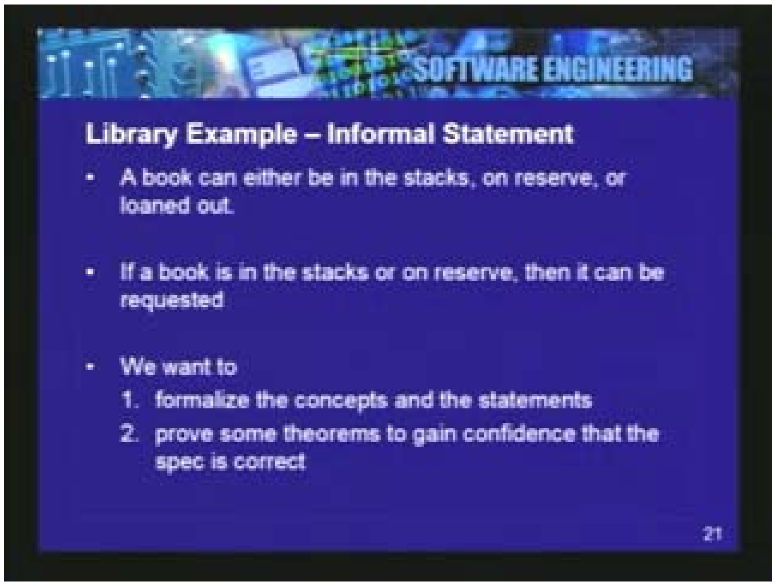
If the steps going from a requirement specification all the way down to executable code, are now completely handled by a machine as supposed to humans getting involved in the process. And that is the holy grail of software development in any case.

The other important property is the reusability of the specification. If a specification, for example has been well written for a performance requirement; “the response time of a particular call to a software module shall not exceed fifty milliseconds”. That is a specification of non functional requirement and that specification might be applicable in a variety of cases. It can be applicable in the construction of a website, it can be applicable in a construction of a control system that is driving some kind of process industry let us say the chemical industry and so on. In each one of these cases, this piece is specification that was written out for some other system can be reused as-is without having to change much.

In that case, you cannot however reuse natural language specifications. You can simply have to cut and paste into another document. But in this case the entire artifact that has been created as a specification can be reused as code, and even the code that is generated out at the end of the day can be reused. What formal specifications tend to do is to provide a language of communication between the various people in the world who are in the production of the software. It provides a common language whereas the analysts, the developers, the testers, and the implementers, all of them can understand exactly what is being asked of them and go about the task without any clarifications being required.

It has its basics in discrete mathematics, set notation, set theory, Boolean logic or predicate logic are typically the techniques that are commonly used in order to specify formally some of the systems.

(Refer Slide Time: 27:11)


SOFTWARE ENGINEERING

Library Example – Informal Statement

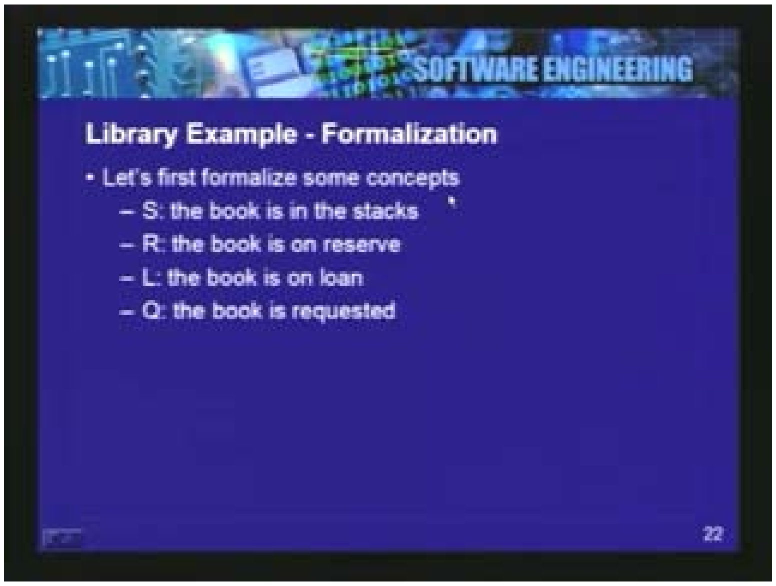
- A book can either be in the stacks, on reserve, or loaned out.
- If a book is in the stacks or on reserve, then it can be requested
- We want to
  1. formalize the concepts and the statements
  2. prove some theorems to gain confidence that the spec is correct

21

Having kind of seen what formal specifications are, what are some of advantages of using formal specifications, let us try to get into a small example that will try and illustrate some of these techniques here. The example is that of a library system that is to be constructed. The library has two sets of requirements that the library system is going to use. The first requirement is the book can either be in the stacks, it can be on reserve or it can be loaned out. If it is on reserve then it is still in the library but somebody has asked for it. If the book is in the stacks or on reserve, then it can be requested by a user to be checked out.

Just for the sake of simplicity we have ignored other requirements that are commonly pertaining to that of a library system. What we want to do is, formalize the concepts and statements here. And prove some theorems to kind of help us get us confidence that what we have written now in terms of formal specifications, is indeed satisfying all the properties that we saw in the couple of slides ago, which is in the notion of unambiguity, in the notion of completeness, in the notion of consistency and so on and so forth. Let us first start formalizing some of these concepts. The first set of concepts that we want to take a look at is the four conditions: where can the book be? We say here that S stands for the fact that the book is in the stacks.

(Refer Slide Time: 29:03)


SOFTWARE ENGINEERING

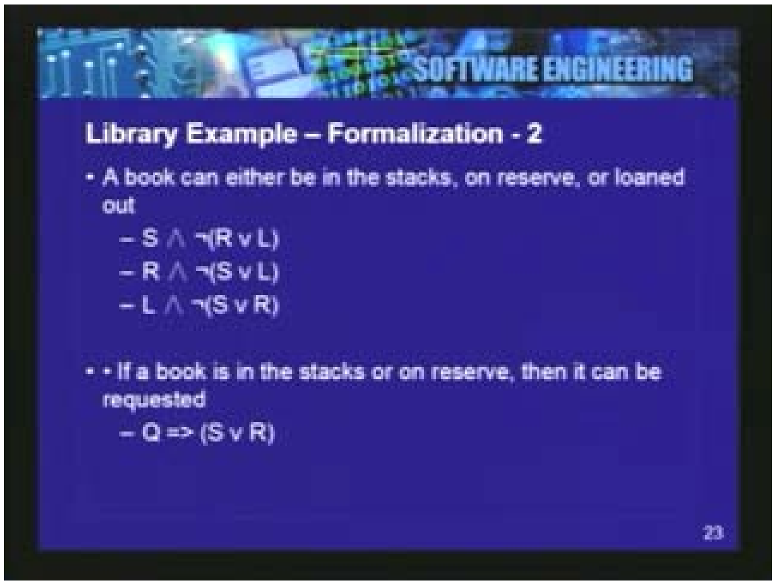
Library Example - Formalization

- Let's first formalize some concepts
  - S: the book is in the stacks
  - R: the book is on reserve
  - L: the book is on loan
  - Q: the book is requested

22

This is again conciseness of notation instead of having to say the book is in the stacks each time around, just the notation of using a single letter for this namely S is going to help us understand this. R stands for the fact that the book is on reserve. Similarly L stands for that the book is on loan which means the book has been already given out. And Q is the book has been requested.

(Refer Slide Time: 29:34)


SOFTWARE ENGINEERING

Library Example – Formalization - 2

- A book can either be in the stacks, on reserve, or loaned out
  - $S \wedge \neg(R \vee L)$
  - $R \wedge \neg(S \vee L)$
  - $L \wedge \neg(S \vee R)$
- • If a book is in the stacks or on reserve, then it can be requested
  - $Q \Rightarrow (S \vee R)$

23

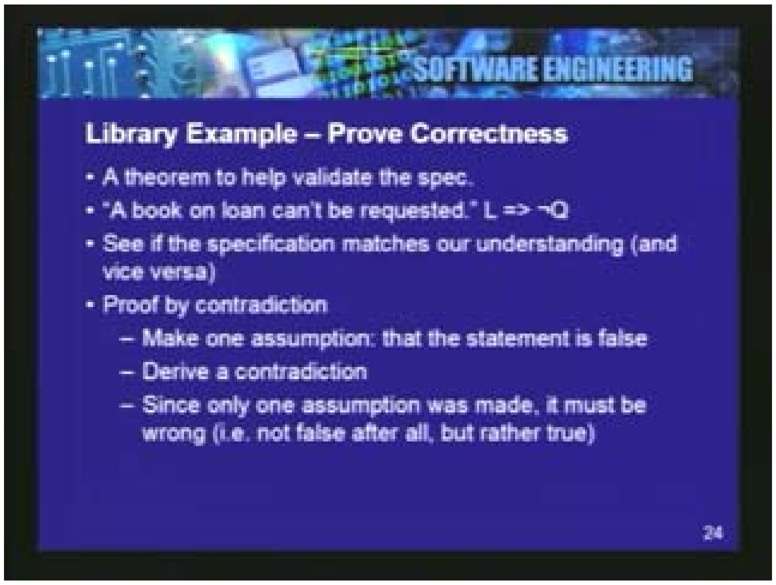
The first formalization that we want to do is, that the book can either in the stacks on reserve or loaned out. It cannot be in two places at the same time is what we are trying to say. Let us see how we can formally denote some of this. The first thing that we are trying to say here is that the book is in the stacks which means it is S and this '^' notation stands for AND. If it is in the stacks then it cannot be on reserve or loan at that point of time. That is how you read this particular notation: [$\mathbf{S \wedge \neg (R \vee L)}$], S **'AND' 'NOT of'** R **'OR'** L.

Similarly, the next set of specifications tries to say that it is on reserve [$\mathbf{R \wedge \neg (S \vee L)}$], which implies that, it is not on the stacks neither has it been loaned out. This means it has been brought over some library counter somewhere and kept ready for the person who had requested it. Similarly the last specification [$\mathbf{L \wedge \neg (S \vee R)}$] essentially stands for L and NOT of S OR R. This means, if it is loaned out then it cannot be on the stacks neither it can be on reserve. This is the set of specifications or formal notation for indicating that the book can be either in the stacks or on reserve or loaned out, but not any two at the same time.

The next bullet here is essentially trying to talk about how the book can be requested.
The book can be requested if it is either on the stacks or it is being held in reserve for the person who is requesting the book at that point in time [$Q \Rightarrow (S \vee R)$]. Remember a notation Q stands for the book being requested and L stands for the book being loaned and there is a differentiation between the two states. Q implies S OR R in this particular case. So if the book is either on the stacks or it is on reserve, then the book can be requested. That is the implication of this particular notation. What we are seeing here, is a way of concisely putting together a mathematically notation around certain concepts that are pretty familiar to us.

If you take a look at the English language specification of this, they are fairly familiar concepts; the notion of libraries, the notion of books being on stacks, on reserve, the notion of loaning books or checking out books and so on. What we have gone and done to the set of natural language specifications or English specifications is, just converted them into a formal notation which we can then use and manipulate for some program proving concepts or some theorem proving concepts. Now what we are interested in doing going forward is to basically write down a theorem to validate this specification.

(Refer Slide Time: 33:20)



SOFTWARE ENGINEERING

Library Example – Prove Correctness

- A theorem to help validate the spec.
- "A book on loan can't be requested." L => ¬Q
- See if the specification matches our understanding (and vice versa)
- Proof by contradiction
  - Make one assumption: that the statement is false
  - Derive a contradiction
  - Since only one assumption was made, it must be wrong (i.e. not false after all, but rather true)

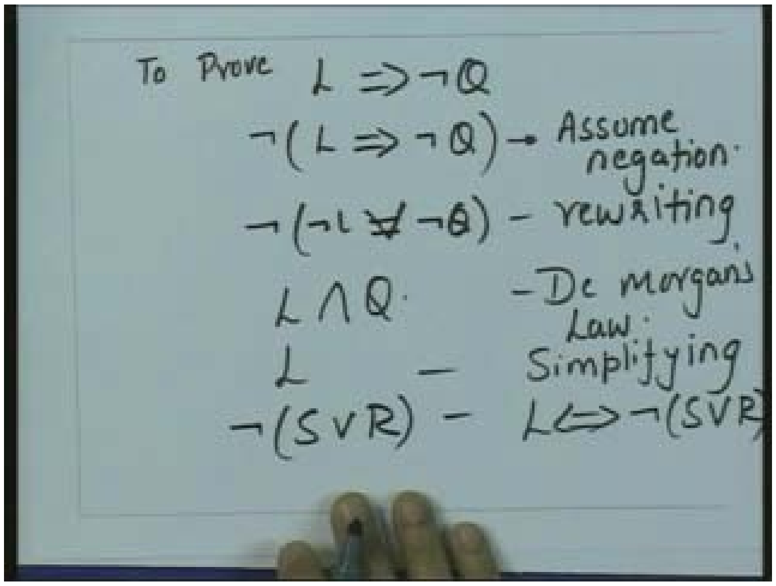
24

The theorem is going to be a pretty simple one. The first thing is that the book that is on loan cannot be requested. The book is not even in the library at that point in time and therefore the book that is on loan cannot be requested. To put it down formally what we are interested in proving is that, if it is on loan that is **L**, then it cannot be requested which is **NOT of Q [**$L \Rightarrow \neg Q$**].** And this will help us check whether the specification matches our understanding of the software and vice versa. The approach that we will take in this particular case is proof by contradiction approach. The proof by contradiction is that we start out by making one assumption, that this statement is false is the assumption that we will make in this particular case. And then will derive a contradiction through successive refinement and steps that we shall see shortly.

Having derived contradiction since it is pretty clear that only one assumption was made the assumption must have been false. because this come up among contradiction therefore the statement must be true in the first place and it cant be false at all so that is the approach that will take and Let us go down the set of steps that would now require for this. If we have to prove that L implies NOT Q. So we will start of by assuming the negation. We have to assume NOT of L implies NOT Q. $\neg(L \Rightarrow \neg Q)$ → 'Assume Negation' - This is the first step. As soon as this is done, we will start refining what we have just written out.

By rewriting this implication we can say that this automatically leads to NOT of NOT L OR NOT Q instead of 'NOT of' L implies NOT Q. $\neg(\neg L \vee \neg Q)$ → 'Rewriting' By De Morgan's law this can simply be written as L AND Q. $L \wedge Q$ → De Morgan's Law By simplifying L AND Q we simply end up in L in this case. L → Simplifying Remember, the first set of notations that we had L AND NOT of S OR R. Because of that, by conditional, L will simply end up implying NOT of S OR R. $\neg(S \vee R)$ By conditional that we had was, L double implies of NOT of S OR R $L \Leftrightarrow \neg(S \vee R)$ Continuing on, the NOT of S OR R implies Q by simplification $\neg(S \vee R)$ => QQ however implies S OR R. => $S \vee R$ This was one of the original definitions that we had. The fact that a book is requested implies that it is either on the stacks or on reserve. This implication is done by the definition that we did earlier.

(Refer Slide Time: 36:52)


To Prove $L \Rightarrow \neg Q$

$\neg(L \Rightarrow \neg Q)$ → Assume negation.

$\neg(\neg L \vee \neg Q)$ - rewriting

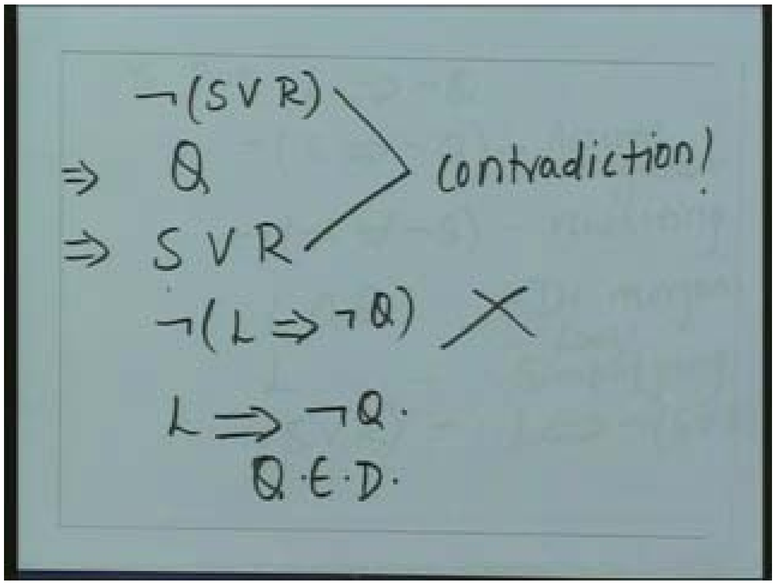
$L \wedge Q$ - De morgans Law.

L — Simplifying

$\neg(S \vee R)$ - $L \Leftrightarrow \neg(S \vee R)$

This is a contradiction. On one step we have NOT of S OR R **[** $\neg(S \vee R)$ **]**. And few steps later, we have arrived at S OR R **[** $S \vee R$ **]** which is a contradiction. Hence the assumption that we made in the beginning of this entire theorem must have been incorrect. The assumption that we made was NOT of L implies NOT Q is incorrect because of the contradiction. $\neg(L \Rightarrow \neg Q)$ → Assumption is wrong As a result of which L implies NOT Q. $L \Rightarrow \neg Q$

(Refer Slide Time: 38:26)



$\neg(S \vee R)$

$\Rightarrow Q$

$\Rightarrow S \vee R$

contradiction!

$\neg(L \Rightarrow \neg Q)$ ✗

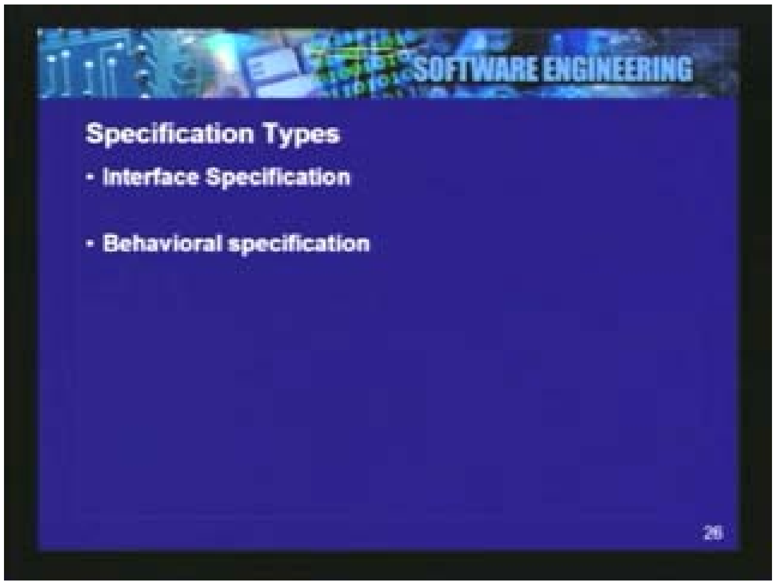
$L \Rightarrow \neg Q.$

Q.E.D.

What this example which we have kind of gone through in a fairly detailed manner is, trying to show the fact that a formal specification such as what we just wrote out for a very simple library system with two rules can be used to prove certain properties of the system itself. In this case the rules were pretty simple that book can only be in one of three different states. It can either be on stacks or it can be on reserve or it can be loaned out. And the second thing that we saw is about the book can be requested only if it is on the stack or on if it is on reserve. By that definition, we naturally came to a theorem which said that if it is loaned out it cannot be requested, which is L implies NOT Q.

What formal specifications give you is the confidence. The specification that you wrote out in the beginning of this exercise is something that is correct. Correctness is one of the properties if you remember. It is very concise. If you look at the specification, it is really consistent. Any inconsistent statements such as the one that we made can be proved to be incorrect. It satisfies all the criteria of the formal specification, at the same time it gives you the confidence of what you have written out in the specification is something that you can go forward with, in order to construct a system from it. There are different kinds of specifications. Even though there are different models based on which specification can be done, each one of the models can be applied to each one of the specification type that is being shown in the slide. The two kinds of specifications are, Interface specifications which basically tell you how subsystems are going to interact with each other and not about the details of a particular subsystem by itself.

(Refer Slide Time: 40:22)


SOFTWARE ENGINEERING

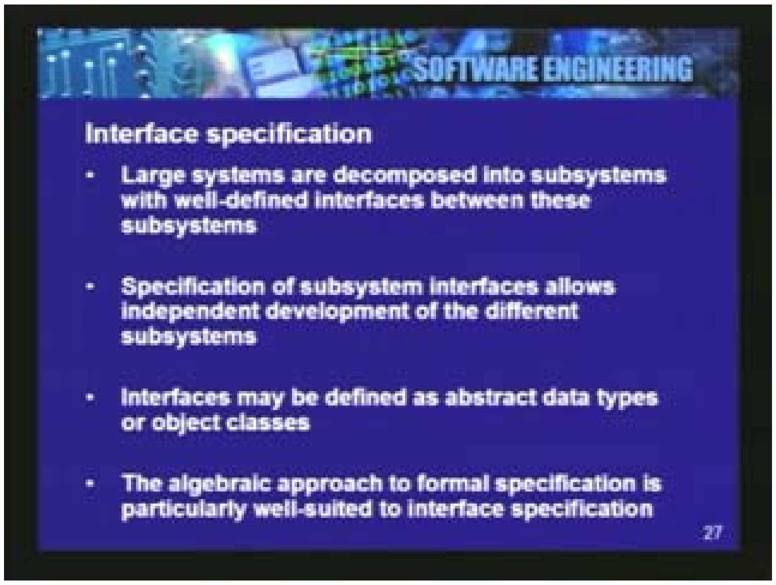
Specification Types

- Interface Specification
- Behavioral specification

26

Behavioral specifications primarily focus in on the behavior of an individual subsystem, of an individual component, of an individual module and so on. So it does not concern with inter component, inter modular and inter subsystem relationships as much as it is concerned with the internals of a single module by itself.

(Refer Slide Time: 41:09)


SOFTWARE ENGINEERING

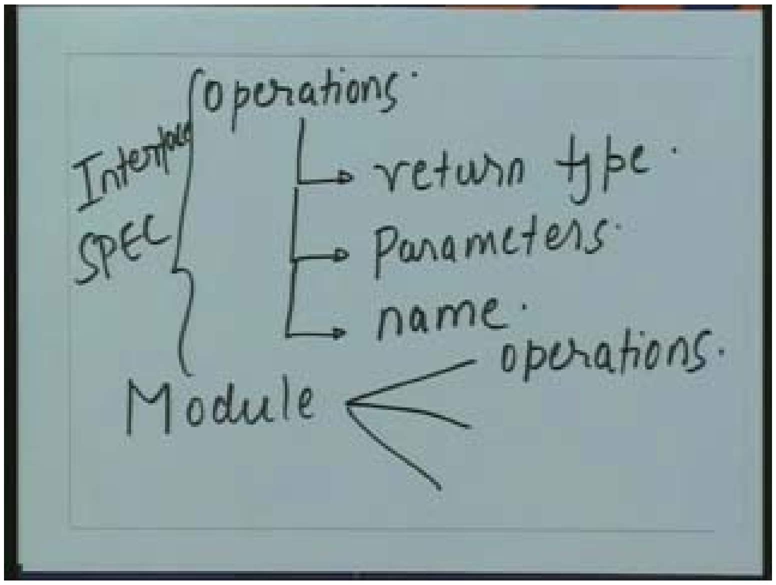
Interface specification

- Large systems are decomposed into subsystems with well-defined interfaces between these subsystems
- Specification of subsystem interfaces allows independent development of the different subsystems
- Interfaces may be defined as abstract data types or object classes
- The algebraic approach to formal specification is particularly well-suited to interface specification

27

Interface specifications are large systems, typically the way that they are built. They are decomposed into a set of modules it is called a set of process modularization. And each module has a well defined interface just like every other module does and the interaction between any two modules is purely on the basis of this interface nothing else can be assumed about this particular module. How it is being built or how it is being done is not the concern of the interacting module whereas it only concerns about interface specification. Interface specification is only concerned about three pieces of information. Interface specifications primarily deal with the operations.

(Refer Slide Time: 42:58)



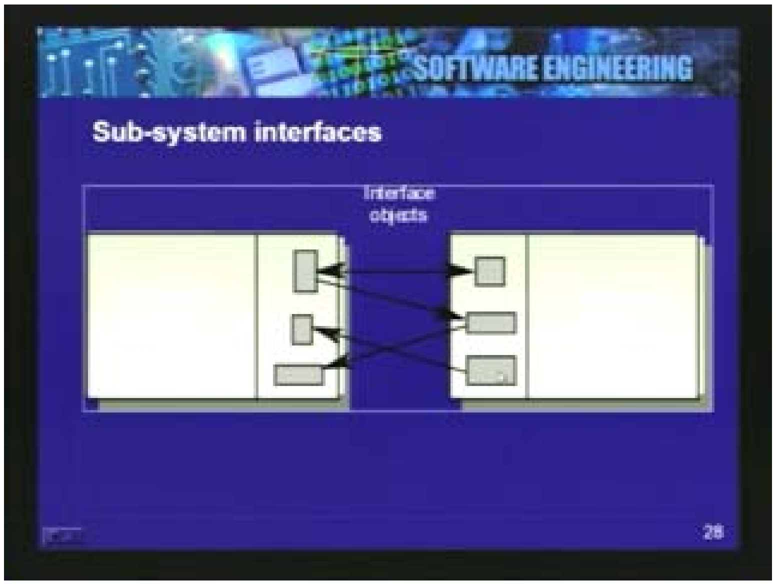
And for each operation it deals with the return type of that operation, in other words what is the type of the output that can be expected of that operation, what are the different parameters that the operations ends up taking, and of course what is the name how do i end up calling the operation by itself. Every module or a subsystem within a software specification is made up of multiple such operations and the sum of all the operational specifications that we wrote out is the interface specification.

Interface specifications can be seen in various forms if you took a language like java for example the interface specification is simply java interfaces. These kind of explain what is the name of an interface, what are the different methods that belong to this particular interface and for each method what is the return type of the method, what is the name of the method, what are the different parameters, what is the order of the parameters that it takes it in, what are the types of different parameters, are there are any other types that can be substituted for this type and so on and so forth.

The obvious advantage of these kinds of specifications is the fact that individual subsystems can be specified separately, and the development of each one of the subsystems can now go on in parallel by different developers. Where as the specification of all the individual subsystems and how they interact with each other are typically done by the system architect. But as the development can go on in parallel, it is one of the big advantages of these methods. The interfaces can be defined as abstract data types. In order for these specifications to be formal, some level of behavioral specifications also needs to be added.

When you combine interface and behavioral specification you get what are known as abstract data types. We will go into the details of abstract data types and how to construct them and what are the formalisms surrounding them in the next lecture. The algebraic formal specification that we saw earlier is obviously very well suited for interface specification. It does not concern states. Interface specifications are not concerned with state either. We saw that in the case of model based specifications it was the interaction of the state and the operations that can be defined only on that state whereas here it is very different. We have kind of gone through this diagram which basically says that different subsystems, two subsystems A and B and in the case of each subsystem is represented by a set of interfaces which is simply diagrammatically shown this way.

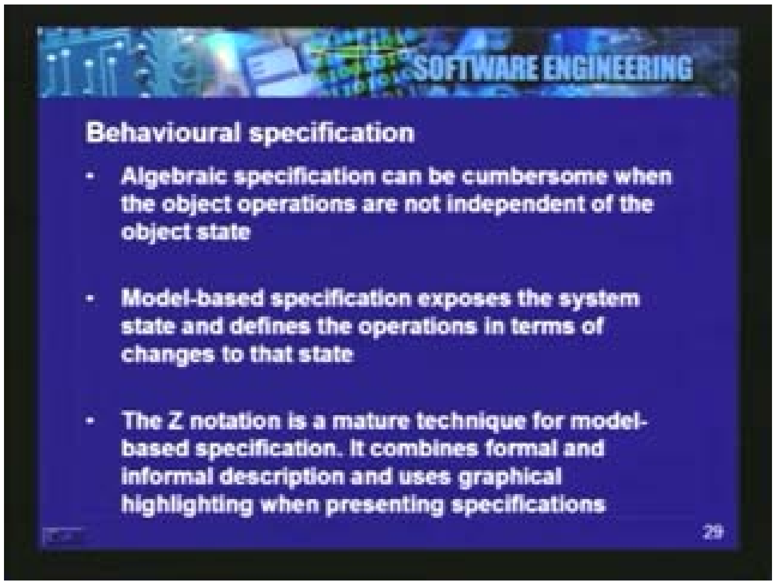
(Refer Slide Time: 44:58)


SOFTWARE ENGINEERING

Sub-system interfaces

And it shows if there are two subsystems end up interacting with each other then it can only call methods or operations on the other subsystem in a predefined way. Behavioral specifications:

(Refer Slide Time: 45:03)


SOFTWARE ENGINEERING

Behavioural specification

- Algebraic specification can be cumbersome when the object operations are not independent of the object state
- Model-based specification exposes the system state and defines the operations in terms of changes to that state
- The Z notation is a mature technique for model-based specification. It combines formal and informal description and uses graphical highlighting when presenting specifications

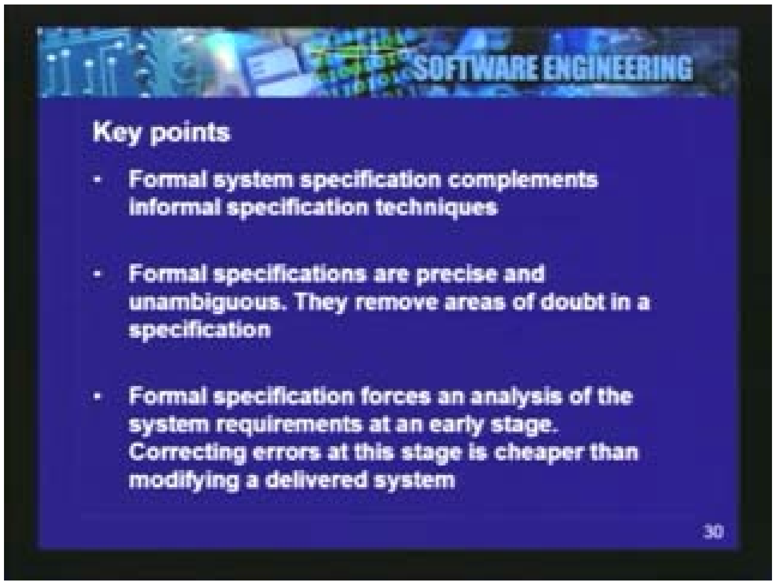
29

Algebraic specifications can end up becoming quite cumbersome when the object operations are not independent of the object state. The other thing is that simply by specifying the set of operations, it is not going to be enough. What also needs to be specified are, certain constraints on behavior that these operations can end up invoking in terms of the internal state of the object. For example any kind of arbitrary state manipulation is not going to be possible. To give a good example; Let us consider a stack which is a data structure that is used almost everyday in computer science applications.

The stack has four operations to it. You can even create a stack, you can destroy a stack, you can pop an element of the stack, and you can push an element on to the stack. These are the four very simple operations of that of a stack. One of the conditions or constraints that have to be imposed as far as the stack is concerned is that, you cannot push something on to a stack that is full. Another condition or constraint that can be imposed is that you cannot pop an empty stack, as you are not going to get anything back of that and it is an illegal operation to be done.

What is the result when you pop a stack that has just been pushed with an element for example? That is kind of behavioral specification that is not done very well with respect to the interface specifications that we just talked about. It is very incomplete, as a result of which you would also need certain constraints on behavior which will be put forward in abstract data types like we will see in the future. The other thing is that model based specifications ends up exposing state and all the operations are defined only in terms of changes to the state. And the ‘Z’ notation that we saw earlier is one of the languages for expressing model based specifications; Z and VDM.

(Refer Slide Time: 47:51)


SOFTWARE ENGINEERING

Key points

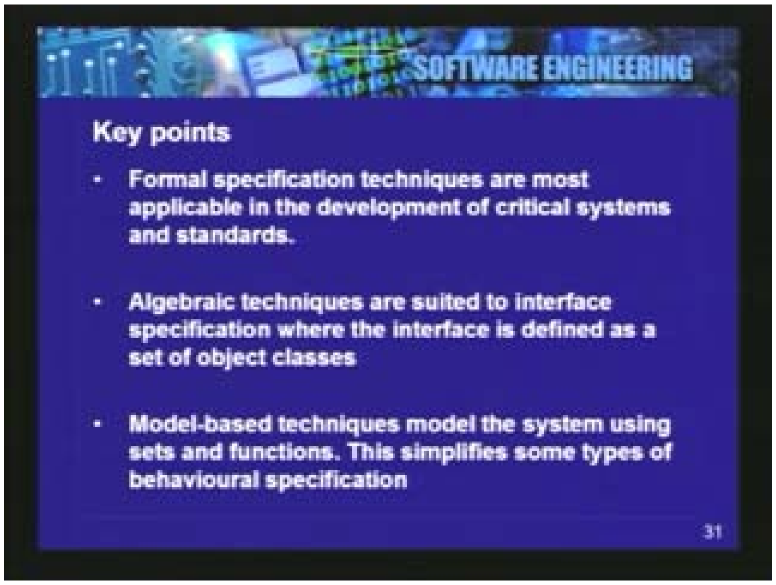
- Formal system specification complements informal specification techniques
- Formal specifications are precise and unambiguous. They remove areas of doubt in a specification
- Formal specification forces an analysis of the system requirements at an early stage. Correcting errors at this stage is cheaper than modifying a delivered system

30

Z is a fairly matured technique for model based specifications and users highlighting etc for presenting the specifications themselves. Some of the key take away as far as this introductory lecture to formal specification goes are formal system specifications complements informal specification techniques. Although the wish is that you can go from a requirements definition document which is written by the user to a design, really going through formal specifications. It is almost always appropriate for you to try and write down the natural language specification corresponding to this requirements document.

The natural language specification can then be given to an expert who is well versed with the mathematical notations who can then come up with a specific or more formal specification. So they end up complementing informal specification techniques. Also the other thing to consider is that, almost invariably the entire system cannot be specified formally. You will have to take the critical parts of the system and then make sure that these are the parts that need to be absolutely fool proof or these are the parts that cannot undergo any error what so ever. For example the part that does the test firing on the rocket on the space shuttle. Things like that would have to be specified formally, whereas the rest of the system is typically just specified using natural language or informal specification techniques. And then the two come together and complement each other very well. Formal specifications by their nature are unambiguous, they are very concise and precise and they remove areas of doubt in a specification.

(Refer Slide Time: 50:05)


SOFTWARE ENGINEERING

Key points

- Formal specification techniques are most applicable in the development of critical systems and standards.
- Algebraic techniques are suited to interface specification where the interface is defined as a set of object classes
- Model-based techniques model the system using sets and functions. This simplifies some types of behavioural specification

31

If you take a look at natural language specification and if a particular piece of specification open to a multiple interpretation, then this is another place where formal specifications can be used. And they force the analysis of the system requirements at the very early stage. So what ever errors, or inconsistencies or lack of completeness there exists within the specification is caught at a very early stage in the process. Like we have seen before several studies have proved catching errors at an early stage is extremely beneficial. They cost a lot less to fix when they are caught upfront as supposed when they are caught in the tail under the process where the design might change, the implementation would have to change and so on.

They are most obviously applicable in the area of critical systems development and standards in this area. Algebraic techniques and model based techniques exist. They are the two kinds of formal specifications. Algebraic techniques are primarily focused on the interface whereas model based techniques are primarily focused on the state of the module which is being specified. They revolve around the state whereas algebraic specifications evolve primarily around the interface specification. We shall see an example and some of the detailed techniques that can be used to develop abstract data type which are really formal algebraic specifications in the next lecture.

Thank you.