SLIDE 1

With the growing significance of computer systems within industry and wider society, techniques that assist in the production of reliable software are becoming increasingly important. The complexity of many computer systems requires the application of a battery of such techniques. Two of the most promising approaches are formal methods and software testing.

Formal specification languages are mathematically-based languages whose purpose is to aid the construction of systems and software. Often backed by tool support, they can be used to both describe a system and also then to analyze its behaviour, possibly verifying key properties of interest.

The use of a formal specification or model eliminates ambiguity and thus reduces the chance of errors being introduced during software development. Naturally, there still remains the issue of obtaining a formal specification that matches the actual customer requirements and this is complicated by the tendency for stated requirements to change during development.

SLIDE 2

The primary idea behind a formal method is that there is benefit in writing a precise specification of a system, and formal methods use a formal or mathematical syntax to do so. This syntax is usually textual but can be graphical. A semantics is also provided, that is, a precise meaning is given for each description in the language.

A specification of a system might cover one or more of a number of aspects, including its functional behaviour, its structure or architecture, or even cover aspects of non-functional behaviour such as timing or performance criteria.

A precise specification of a system can be used in a number of ways. First, it can be used as the process by which a proper understanding of the system can be articulated, thereby revealing errors or aspects of incompleteness. The specification can also be analyzed or it can be verified correct against properties of interest.

A specification can also be used as a vehicle for driving the development process, either through refining the specification towards code or by direct code generation. Of course, a key aspect of the development process is testing, and a specification can also be used to support the testing process. Indeed, the purpose of this paper is to explore this issue in some depth.

There are a number of different ways to write a precise specification. One approach is to build a model of the intended behaviour, and languages such as Z [Spivey 1988; 1992], Vienna Development Method(VDM) [Jones 1991] and B [Abrial 1996] do so by describing the states the system could be in together with operations that change the state.

Model-based languages such as Z, VDM and B can describe arbitrarily general systems, and have potentially infinite state. This generality has a drawback in that it makes reasoning less amenable to automation. This drawback is not, however, present in the case of finite state-based specification languages.

As their name suggests, finite state-based languages define their state from a finite set of values, which are often presented graphically with state transitions representing changes of state akin to operations in a notation such as Z. Examples of such languages include finite state machines (FSMs) [Lee and Yannakakis 1996], Specification and Description Language(SDL) [ITU-T 1999], Statecharts [Harel and Gery 1997] and X-machines [Holcombe and Ipate 1998].

Concurrency can be given a very elegant algebraic treatment, and process algebras describe a system as a number of communicating concurrent processes. Examples include Communicating sequential processes(CSP) [Hoare 1985], Calculus of communicating systems(CCS) [Milner 1989] and Language Of Temporal Ordering Specification(LOTOS) [ISO 1989a].

Many systems are built with a combination of analog and digital components. In order to specify and verify such systems it is necessary to use a specification language that encompasses both discrete and continuous mathematics. There has been recent interest in these hybrid languages, such as CHARON [Alur et al. 2000; Hur et al. 2003].

SLIDE 3

The B method [2] is a strategy for software development in which an abstract model of a system is transformed into an implementation via a series of steps that progressively concretise the abstract model. These steps or stages are referred to as refinement steps, and a model M i+1 of a system at stage i + 1 is said to refine the model M i at stage i .

Each refinement step adds more details to the system.

Refinement steps generate proof obligations to guarantee that
the system works correctly.

The behaviour of each refinement has to be provably consistent with the behaviour of the model in the previous step, keeping a palpable behavioural relation with its abstraction.

B models are called machines, and are composed of (1) a static part: variables, constants, parameters and invariants; and, (2) a dynamic part: operations, that describe how the system evolves. B machines use predicate calculus (essentially predicate logic and set theory) to model properties.

SLIDE 4

Event-B [4] is another formal method for modelling complete developments of discrete transition systems. Event-B was introduced by J-R. Abrial, and is derived from the B method. Unlike in B models, the static part of Event-B models is separated from the dynamic part, and is referred to as “contexts”. Thus, Event-B models are composed of machines (the dynamic part. e.g. variables, invariants, events), and contexts (the static part. e.g. carrier sets, constants). Three basic relationships between machines and contexts are used to structure a model:
• A machine sees a context.
• A machine can refine another machine.
• A context can extend another context.

SLIDE 5

The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. The platform is open source, contributes to the Eclipse framework and is further extendable with plugins.

Eiffel is not only a programming language but also an object-oriented method to build high-quality software. As a method, it brings some design principles:

As mentioned above, Eiffel distinguishes between “commands” and “queries”. Even if not enforced by any compiler, the Eiffel method strongly encourages following the Command/Query Separation principle: A feature should not both change the object’s state and return a result about this object. In other words, a function should be side-effect-free. As Meyer likes to present it: “Asking a question should not change the answer.”

Another important principle, which is Information Hiding : The supplier of a module (typically a class) must select the subset of the module’s properties that will be available officially to its client (the “public part”); the remaining properties build the “secret part”. The Eiffel language provides the ability to enforce this principle by allowing to define fine-grained levels of availability of a class to its clients.

Another principle enforced by the Eiffel method and language is the principle of Uniform Access, which says that all features offered by a class should be available through a uniform notation, which does not betray whether features are implemented through storage (attributes) or through computation (routines). Indeed, in Eiffel, one cannot know when writing x.f whether f is a routine or an attribute; the syntax is the same.

Structure of an Eiffel program

The basic unit of an Eiffel program is the class . There is no notion of module or assembly like in .NET, no notion of package like in Java (no import -like keyword).
Classes are grouped into clusters , which are often associated with a file directory. Indeed, an Eiffel class is stored in a file (with the extension .e); therefore it is natural to associate a cluster with a dire ctory. But this is not compulsory. It is a logical separation, not necessary a physical one. Clusters may contain subclusters, like a file directory may contain subdirectories.
An Eiffel system is a set of classes (typically a set of clusters that contain classes) that can be assembled to produce an executable. It is close to what is usually called a “program”.
Eiffel also introduces a notion of universe . It is a superset of the system. It corresponds to all the classes present in the clusters defined in an Eiffel system, even if these classes are not needed for the program execution.
Eiffel uses a notion of root class , which is the class that is instantiated first using its creation procedure (the constructor) known as the root creation procedure . An Eiffel system corresponds to the classes needed by the root class directly or indirectly (the classes that are reachable from the root creation procedure). The universe contains all classes in all the clusters specified in the system.
The definition of what an Eiffel system contains is done in an Ace file , which is a configuration file written in a language called LACE (Language for Assembly Classes in Eiffel).

EventB2Java generates JML-specified Java implementations of Event-B models. Contributions by Néstor Cataño, Tim Wahls, Camilo Rueda and Víctor Rivera.
EventB2JML translates Event-B machines to JML-specified Java abstract classes. Contributions by Néstor Cataño, Tim Wahls, Camilo Rueda and Víctor Rivera.
EventB2Dafny translates Event-B proof-obligations into the input language of Dafny. Developed by Néstor Cataño.
EventB2SQL translates Event-B machines to Java implementations that make the state of a machine persistent by storing it in a database.
EB2ALL (Beta Version) supports automatic code generation from Event-B to C, C++, Java and C#.
Tasking Event-B supports generation of multi-tasking Java, Ada, and OpenMP C code from Event-B.
B2C translates Event-B models to C source code, which may then be compiled using external C development tools.
EHDL The plug-in enables VHDL code generation from formal Event-B models automatically.