BluePrint: A systems development process supported by machine assisted reasoning

A. Ward

Coherent Technology Limited, 25 The Heights, Wallington, Fareham PO16 8TL, UK

Abstract

BluePrint is a software engineering environment. It provides support for the practical use of formal specification and machine assisted reasoning. The environment maintains system-wide completeness and consistency checking, that supports the control of modern risk-driven development processes. Proofs of required system properties can be developed at the specification level, and semantic checking is available to prove that an implementation meets the specification and therefore will also have these properties. Highly automated testing and Failure Modes Effects Analysis are possible.

Appropriate process models for the use of BluePrint are described. These models emphasise tackling major threats early, and have short planning cycles to keep flexibility. High risk in particular subsystems is controlled by the use of early partial specifications. Support for the management of change in the specification encourages the incorporation of learning about the problem and solution spaces. Proof of important system properties is conducted at the abstract specification level where irrelevant implementation detail does not obscure the arguments.

The engines of the toolset originate in three illustrious research organisations in the USA and are well proven. They consist of formal specification and design languages, supported by syntactic and semantic checkers and an advanced theorem prover. Interfaces to major modern implementation languages are available. These engines are fully functional but difficult to use without a support environment.

The BluePrint environment adds representation of the relationship between documents in the environment. This allows browsing of the documents, and automated completeness and consistency checking. A general mechanism of establishing higher level obligations that must be discharged
by lower level refinements of the system allows the flexible incorporation of design insights at the time they occur, rather than when an appropriate level of refinement is met.

The environment aims to support current best engineering practice through the provision of machine assisted reasoning based on unambiguous specifications. Detailed, tedious tasks are automated to free engineers for creative design work. The environment is not restrictive in prescribing the order of engineering tasks: it merely reports on the detailed status of system construction.

1 Introduction

This description of the BluePrint environment is addressed to practicing software engineers with some knowledge of software engineering process design and some understanding of the place of formal specification techniques.

The context for the use of the environment is set by describing the system development process that might be followed. This allows the engineering goals to be brought out. The toolset and its environment is then described in two parts: firstly the core technology and where it comes from, and then the support given to the use of that technology by the BluePrint environment.

The BluePrint environment gives highly practical support to the crucial, abstract phases of system development. It is designed to leverage the best skills and the best practice of good software engineers.

2 System development process

Formal systems development has been characterised by a rigidly top down refinement model of development. In sharp contrast the BluePrint toolset aims to add support for formal specification and reasoning about system properties to the best current process models. Rigorous proof of crucial system properties and semantic checking take the place of testing as the primary validation tool. This complements rather than contradicts modern risk reduction strategies. Engineers work by oscillating between high level abstractions and low level implementation detail as they test solution strategies against aspects of the problem. The process described here supports the way that the best engineers work by adding tool support for formal specifications to remove ambiguity and for machine assistance for reasoning that is otherwise slow and error prone.

2.1 General process model

The general process model is based upon the Evolutionary Learning Model. The important features of this model are:

- Overall risk is controlled by tackling the problem areas posing the
greatest threat first. Areas where there are major uncertainties and lack of knowledge are generally high on the list.

- Flexibility is maintained by a short planning cycle where the current situation is evaluated, plans are made for the next work item, the work is performed and then checked against the criteria set in the plan. The goal is to capture the maximum amount of knowledge from the work performed.

- The process is shaped by the set of deliverables and their associated standards that are decided on initially. The order of deliverable creation is governed by the risk driven planning process.

- For the process as a whole the understanding that the engineers have of the problem and solution spaces is given higher priority than production of the system: lack of understanding causes error and delay.

This model is considerably enhanced by having reasoning tools capable of detecting inconsistency and incompleteness at the system specification level.

Feedback from this modeling led to changes in the design ... this sort of design-time verification is all too rare in software development.2

It is also a great benefit for the toolset environment to track the semantic relationship between all the documents in the system so that complicated dependencies can be properly dealt with. The difficulty with following advanced process models that circumvent the main pitfalls of conventional development has largely been the management of the dynamics. Having an environment that can report on inconsistencies between documents of all sorts from system specification through to code, and not just syntactic inconsistencies but differences in meaning, is a major enabling technology.

2.2 Developing the archetype

The archetype is a proven system specification. The style of specification used in BluePrint is to describe only the external behaviour of system and modules. This is important in developing an archetype as it must avoid both over-constraining the set of implementations that can fulfill the specification, and under-constraining the set so that undesirable implementations are included.

The archetype is the foundation of reasoning about system properties. It completely and accurately defines the system to be built, and how to test it, and how to do Failure Mode Effects Analysis etc. To check that an implementation meets the archetype specification, there is a design layer that
describes how the mapping is to be performed. In practice engineers may start with a solution idea for part of a system, and reverse engineer an existing implementation back to its implementation independent abstractions, before attempting to integrate the ideas at the archetype level.

Domain analysis of a limited application area produces general descriptions of the features of all applications in the area. The analysis gives re-use at the specification level where there is a requirement for more than one similar system, or where the requirements are expected to undergo rapid change. Focusing on domain analysis rather than the immediate system requirements is the way to make a re-use strategy productive. Formal specification of such information and analysis is the ideal way to capture, store and use it.

The archetype is thus the lynchpin of the entire system and should be given maximum priority. To summarise its place in the process:

- **Implementation independence** Maintain complete independence of implementation concerns at this level to maximise the power of reasoning about system properties.

- **Captures domain knowledge** Do domain analysis wherever it can be justified to maximise the generality and hence usefulness of the information captured.

- **Early, rigorous requirements analysis** Capture the system requirements into formal specifications so that reasoning about completeness and consistency can be done at the earliest possible time.

- **Integrate reverse engineered abstractions** Where potential solutions are well understood, reverse engineer the solution through the design phase to integrate it properly with the archetype. This is the ultimate test for successful integration of the subsystem solution.

### 2.3 Proving important system properties

For the great majority of engineers, building proofs of system properties is a new activity. There is nothing new about the activity of reasoning about specifications to ascertain whether they support the desired system properties, although without tool support this has had to be less rigorous than the ideal. The ability to detect design flaws at the system specification level has immense economic rewards.

Some examples of system properties will help to focus the argument. In a rail signalling system, it may be an important property of the system as a whole that two trains can never occupy the same track segment simultaneously. This is an aggregate, or integration property of the system and will depend on the correct functioning of several subsystems. It is entirely
reasonable, however, to expect the process of system development to supply a proof that this property is preserved in all system states.

A more computer science oriented example would be that in a database management system which had been tuned to give the maximum possible parallel execution, all transactions were still serialisable. Again this is a function of the integration of many subsystems.

Establishing proofs of system properties is essentially a programming task. BluePrint allows a highly modular approach to constructing proofs, and the prover itself maximises feedback about the state of a proof and what is needed to complete it. Thus the engineer sets modest proof goals and develops a proof script that attempts to achieve the goals. This script is interactively debugged until the goals are achieved. These achieved proofs are then available as results on which to base higher level goals. Work proceeds essentially by verification of properties at a module level, building towards complete proofs of system properties.

This exercise of establishing proofs is likely to cause re-evaluation of the specification in various places, and of course this is the aim of the exercise. The importance of modularity and separation of abstractions at the proof level is to minimise the amount of rework necessary when changes are made to the specification.

Building proofs is the heart of the process as specifications that cannot be shown to support crucial system properties are of little or no value. The act of establishing that a given archetype does indeed satisfy the requirements provides confidence that the overall problem has been understood and that a satisfactory implementation is achievable. The normal run of integration problems shows that it is likely that a conventional process proceeds through the implementation phase without this understanding having been achieved.

### 2.4 Designing to optimise the implementation

As indicated above, engineers do not attempt a clean separation between specification and design. Often there is a well understood solution in mind while specification is taking place. Since working with mature solutions is likely to reduce the overall risk it is desirable to support this way of working. While working on specifications, implementation snags and difficulties come to mind. The integration of these insights is handled by the construction of a pattern of obligations that must be fulfilled for a design to be complete.

The existence of a system specification implies the existence of a set of obligations that a design must fulfill. The design, of course, ultimately raises implementation obligations. The toolset environment can check the completeness with which these obligations are fulfilled. This allows the ready use of design fragments. Implementation solutions can be reverse engineered to specifications. Design fragments can be used to note preferred
implementation at specification time. There is no constraint on the order of the system development stages beyond risk and productivity arguments. The toolset environment will check that the eventual obligation structure is consistent and completely fulfilled, thus freeing the engineer to work in an effective fashion.

The necessity to build proofs based on the system specifications makes it much more likely that the archetype will not over-constrain the solution, as these proofs become progressively harder when over-specification is present. The design stage therefore has maximum latitude to optimise the implementation with respect to resource constraints (time, memory etc.). It should also deal with error handling.

*BluePrint* has a high level design language for general use that has constructs to allow the handling of concurrency. It also has a series of interface languages that describe how the specification is mapped onto implementation in the corresponding target languages. It is this refinement of the design down to the code module level that allows the semantic checking of the implementation.

Compared to design using a conventional CASE environment, the *BluePrint* environment is capturing the semantics of an application, so that checking can be done that the meaning of system components is consistent across interfaces. CASE environments really only capture syntactic information and communicate the structure of a design via the various supported diagrams. It is unlikely that this is sufficient to detect high level design problems before the implementation stage.

### 2.5 Implement, check and test

On many software projects, system testing consumes more than half the total budget and proves to be a tail wagging the dog. The role of testing is to establish an appropriate level of confidence that the software meets the requirements. For very high integrity systems this is technically infeasible, but even for more conventional systems the real possibility of failing system tests and the necessity for re-engineering plays havoc with project planning.

The premise of the *BluePrint* toolset is that this is not a necessary aspect of software engineering, and that the combination of high integrity specifications and the ability to machine check their implementation gives an altogether more manageable process. Implementation can take its place as a tightly controlled, brief phase with early and effective quality control.

Module interfaces are completely tied down in the design phase, leaving only algorithm selection as engineering activity at implementation time. In general only a subset of the features of the chosen implementation language will be supported by the semantic checking facilities, with activities like the casting of pointer types effectively outlawed. This is only to say that integrity and verifiability are given higher priority than 'efficient' implementation.
The toolset enables the part of a project budget normally spent on extensive testing to be used to improve the integrity of the system definition. The role of execution testing is much reduced. Testing is still needed to establish confidence in system operation, but its cost is much reduced by two factors. Firstly the full definition of the semantics of the system allows full automation of test case and test results generation. Secondly failure at system test level should be effectively eliminated so that confidence generation is a much more rapid activity.

3 The toolset engines

The engines of the BluePrint toolset are a mature set of tools that have been used on substantial projects. The increased use of these tools rests only on general software engineering education and their embedding in a supportive environment such as BluePrint. The engines are fully proven and fully functional but until now have relied on a rare skill set for their effective use. All the developers of the engines are keen to see the BluePrint environment produce greater utilisation.

The implementation independent specification language (LSL) is very stable, reflecting its remoteness from the main areas of engineering development. New implementation languages need their own interface language (LIL) so there is a steady stream of new development here. The field of formal design languages especially for concurrent systems is showing continuous development. The prover (LP) is already well respected and is being enhanced constantly in various ways.

3.1 The background and provenance of the engines

The tools here termed the engines of the BluePrint environment were developed by a small team at Digital Equipment Corporation System Research Center, Massachusetts Institute of Technology and Carnegie Mellon University. They centre around an algebraic specification language called Larch that has been in use for fourteen years. There is a substantial body of research literature describing widely different uses of Larch, and there is an excellent book on the software engineering background (Liskov, 1986).

The major use of Larch to date has been in the specification of DEC's new Alpha chip set. This was a breakthrough both in the size of the complete specification and in the level of assurance of the hardware design. The prover has been used in a wide range of projects.

The industrial flavour to the provenance of the tools has lent an emphasis on engineering rather than computer science. The Larch languages emphasise the usefulness of partial specifications and the importance of modularity in building large systems. The prover emphasises detailed control by the
engineer, and lots of feedback, rather than the inclusion of heuristics and complicated proof strategies. The aim is to support the engineer in his task rather than demonstrate the power of the tools.

Of course if a specification is accurate and complete, then it follows that it can be automatically translated into another formal language. At this level, other formal approaches are equivalent. Indeed the development of the BluePrint user interface is designed to shield the user from the Larch syntax if they do not wish to interact with it. As indicated above however, at the level of engineering large specifications there are important differences between formal approaches. Larch has been chosen as the core technology of BluePrint to facilitate the construction of arbitrarily large systems and to maximise the benefit of partially formal approaches.

3.2 The functionality of the range of engines

Using the ‘C’ language as an example implementation language the major engines in BluePrint are:

- The language for the capture of implementation independent specifications is the Larch Shared Language (LSL) which has a syntax checker and a translator into a form used by the prover.

- The Larch Prover (LP) is the reasoning engine.

- High level design and design for concurrent implementations is done in a language called General Concurrent Interface Language (GCIL). It has a syntax checker and a translator into LSL. It can be used as an interface language for C++.

- The ‘C’ interface language is the Larch C Language (LCL). It has syntax checking tools and tools to generate the ‘C’ language header files. LCL describes the mapping of of LSL specifications onto implementations.

- Semantic checking of ‘C’ code is done by a tool called LCLINT working on the LCL interface specification.

In addition to these main components there are prettyprinters to the main formatting technologies.

The range of languages currently supported with Larch interface languages (LILs) is ‘C’, ADA, VHDL, Pascal, CLU, Modula-2 and Modula-3. GCIL can be used for C++.

There is a Handbook of LSL traits specifying a lot of basic functionality such as the properties of integers and sets. The building blocks for the definition of more complex abstract data types are thus already in place.
3.3 Major skills needed to use the engines directly

To use the Blueprint engines outside the supporting environment relies on a rare skill set. Obviously it has been done successfully before, and is likely to continue, but to deliver the advantages of the approach reliably in an industrial setting needs the sort of process support that the Blueprint environment gives.

The difficulties in using the Larch tools directly are chiefly:

- Judgment as to the sufficient-completeness of a specification under development. It is quite tricky to decide the number of axioms constraining the meaning of an interface to make the meaning complete without introducing contradiction by over-specification.

- Some fluency with predicate calculus is required to develop appropriate axioms in specifications and to understand enough of the operation of the prover.

- The strategy to use in building a proof, and how to modularise the proof satisfactorily requires considerable insight.

Of course in developing the Blueprint environment these difficulties have been studied with a view to removing obstacles wherever possible, and providing strategy suggestions and ways of learning the appropriate skills elsewhere. Despite the commercial potential of software engineering using this type of approach, effective engineering relies on supporting the engineer with his current skill set, not on insisting on a superior approach.

4 The Blueprint environment

The environment is designed to support the development process described in the first section. To achieve this, the environment must track dependencies between documents and allow the user to browse the structures thus created. The status of documents can be modeled so that only fully checked documents are used to derive results elsewhere in the system. The interface to the engines of the toolset can be improved to use context sensitive generic activities such as syntax checking rather than forcing the user to select the appropriate tool.

Having detailed semantic connections between various document types making up a software system allows the development of much more comprehensive support for the housekeeping side of software development than is possible in conventional systems. Configuration management for example can be much more easily and comprehensively supported given these connections. These are the types of application for tool support that make a major efficiency contribution to the engineering task and free skilled staff for creative work that cannot be automated. It is envisaged that there will
be steady development of support for these types of housekeeping task. The initial environment is described below.

4.1 The user interface model

The user interface consists of representations of the documents comprising the system, and of tools to operate on them with. One or more documents are selected in the interface and then a tool is selected. The command interface for the tool is then made available to control the operation of the tool on the documents.

One of the types of tool available is a viewer that gives a graphical representation of the dependencies between documents in some respect. This may be an inclusion or an inheritance relationship, it may be a refinement relationship or it may be an obligation / obligation fulfillment relationship. One capability of viewers is to allow graphically controlled browsing of the documents and their relationships.

Editors both syntax directed and forms based are the next main class of tools to allow the creation and change of documents. Editors automatically alter the status of documents as described below.

The documents themselves have locking mechanisms so that not only multiple views but multiple users can access the documents simultaneously.

4.2 The document status model

All components of the software system are regarded as documents that have status, and behaviour when interacting with tools, that depends upon their status. The bare text within a document is not accessible to the user apart from via the interface provided by the environment.

Documents have a status derived from the history of tool actions on that object. Thus an object that has been edited but not syntax checked has a different status from one that has been checked, and that is different again to one that failed its checks. Since documents in general have dependencies on other documents, checks may need to be propagated to other documents.

Rather than do the maximum amount of automatic checking that can be done after every action, which will certainly be wasteful of machine resources and users’ time, the document status is used to flag the enablement of certain actions. These actions can then be run as required by direct user requests: if a document needs to be checked because it is included in another that the user is using in building a proof, then it will be checked at that time.

In general a specification of some basic properties may be used in many derived specifications. Changing this basic specification will require a large amount of rechecking and re-proving. Since the change made may be exploratory in nature the user must be able to control the order in which derived changes are propagated.
4.3 Obligations and obligation discharge

The finished structure of a system consists of a series of layers with the most abstract specifications at the top and the finished implementation at the base. The connection between these layers is an obligation relationship, where somehow the lower layer has to fulfill or discharge the obligations laid upon it by the upper layer. The structure does not imply a temporal order of constructing the layers, but it is the way that verification is performed and must be complete so that the system properties proved in the archetype can be assured in the implementation.

The generality of the obligation structure means that it can be used as a general means of tracking issues that are important in a design. When design decisions are made, for instance that it would be most efficient if a dataset was stored in sorted order, then an obligation that the dataset be sorted can be raised at the same time as designing the module that requires the data in that order.

One result of this structure is that the boundary between implementation independent specification and the introduction of implementation constraints and a move to using a design language can be left flexible to suit the application. An application with severe constraints that dominate the architecture may see obligations phrased in design language in the system specification, whereas a completely top down approach to system building may see continual refinement of the implementation independent specification almost to the code module level.

Verification consists of the completeness check that all obligations have documents that purport to fulfill them, and the semantic check that the specifications, designs or code that is in the documents does discharge the obligations. These tasks are highly automated in the BluePrint environment.

4.4 Testing and FMEA

Testing is not a technically important activity in the BluePrint environment. Validation and verification activities have deliberately been moved away from testing to a property proving and semantic checking framework. The two reasons for this change are that testing comes too late in the lifecycle to affect the outcome of a project if there are problems, and that testing can never result in the levels of assurance required in safety critical systems.

The construction of test data and expected results is a straightforward task given an accurate and complete specification. Since testing is still required for confidence generation and to check the overall process, then the provision of test automation facilities in the BluePrint environment is still economically important as it affects the cost of software development. The reduction in the time and cost of testing both demonstrates the nature
of the change in the development process and is the economic justification for the change.

Failure Modes Effects Analysis (FMEA) is also possible given a formal specification and its mapping to the implementation. The effects on the external interface of a system can be tracked from the failure of a component. This shows the degree to which the semantics of the entire system have been captured. The basic idea is that the failure is simulated in the implementation, and the ‘error’ generated in the checking procedures is propagated until the effects at the system interface become visible.

Detailed FMEA is the route to the building of much more robust systems that can correctly handle all the expected failure modes in a reasonable fashion.

5 Summary

The BluePrint engineering environment manages to build on and support the best current engineering practice without being unduly prescriptive about the way it should be used. It delivers the ability to analyse and learn about specifications and high level design without resorting to implementations, prototype or not. It keeps the analysis of system properties away from implementation issues which can only confuse and obscure.

The basis of the toolset is a collection of mature, well proven engines that power the environment. To these are added process support to help manage large developments and to improve verification accuracy by automatically tracking and checking the discharge of obligations.

Tool support allows the system specification to track requirements changes, and remain the definition of the meaning of a system. This has a dramatic effect an all development stages from requirements capture to maintenance.

References

