High-Integrity Extreme Programming
Richard F. Paige, Howard Chivers, John A. McDermid, and Zoë R. Stephenson
High-Integrity Systems Engineering Group, Department of Computer Science, University of York, UK.
{paige, chive, jam, zoe}@cs.york.ac.uk

ABSTRACT
We assess the applicability of Extreme Programming practices to engineering high-integrity systems, focusing on the characteristics of this problem domain that distinguish it from those considered more traditional for agile development. We suggest that Extreme Programming needs both extension and modification to be applicable to engineering high-integrity systems, and discuss promising extensions.

Categories and Subject Descriptors
D.2.1 [Requirements/Specifications]: Methodologies – object-oriented, structured, agile.

Keywords
Agile development, high-integrity systems, Extreme Programming.

1. Introduction
High-integrity system (HIS) engineering takes a principled, risk-based approach to building systems with substantial safety, availability, reliability and robustness requirements [11]. Such systems – e.g., in the avionics, marine, energy, and defence domains – invariably require external certification [10], often by government authorities, in order to be deployed and used. HIS include both software and hardware components, as well as mechanical components and infrastructure, e.g., airframes. When a HIS is certified for deployment, it is the entire system that is certified, not the hardware or software individually. This in itself makes engineering HISs challenging. A key challenge in the certification process is in dealing with incremental change. Certification is very expensive and requires the use of a system safety analysis process (discussed further below) and a rigorous testing process. Changes to the system sometimes occur after certification, making it necessary to repeat parts of the certification process. More often than not, the entire software-specific part of the certification process has to be repeated, at considerable cost. Ongoing research is attempting to allow modular and incremental certification to be carried out via use of architectural modelling and contracts.

A key part of the engineering process is system safety analysis [10], intended to demonstrate that there is sufficiently low risk of accident or loss. This process is based around the identification of hazards, states or conditions that, together with conditions in the system’s environment, will lead to an accident under credible conditions. The primary concern of system safety analysis is hazard management, including early identification and classification of hazards so that corrective action can be taken to eliminate, or mitigate against, hazards as part of the design process. The system safety process culminates in a safety case being produced that gathers evidence in order to justify that the system is safe. The process is split into the following steps, with the first two steps being iterated over the next two: identifying potential hazards; risk assessment; preliminary system safety assessment (including refinement of safety requirements); system safety assessment (producing evidence); and safety case delivery (producing a defensible argument that a system is safe to use in a given context).

HISs are often engineered using model-based development techniques, e.g., formal methods, Simulink and Stateflow [15] or UML models. Modelling techniques are expensive, but considered essential tools to apply in this domain, particularly as models could provide the means to control complexity and to identify the effects of change more accurately, thus giving a way to manage the costs of certification.

Another possible way to manage the costs of certification is through application of the iterative and incremental practices and principles of Extreme Programming (XP) [1]. This approach to development may have other benefits to the domain of high-integrity systems engineering, which we also can explore. It is important to assess the usefulness of XP practices in this domain, because they may offer substantial benefits, particularly towards incremental certification, while maintaining compatibility with how practitioners work: for example, iterative and incremental development – ideas at the heart of many XP practices – was the recommended form of development for the US Department of Defence in 1994.

In this paper, we consider the application of XP to engineering HISs. In doing so, we assess each practice of XP in this domain, and consider its impact on engineering HISs and the research issues that arise from the practice.

We make the case that the XP approach to development could – with some reasonable modification and extension – be made applicable to high-integrity systems engineering. The value of doing so may be to better manage resources, particularly those associated with the costs of incremental certification.

2. Issues in High-Integrity Systems Engineering
Before we proceed, it is important to clarify some of the key differences between systems and software engineering. First, we emphasise that we are focusing on systems of high-integrity: thus we assume that there is a substantial non-software
component to the system, usually consisting of both hardware and mechanical subsystems (e.g., fuel pumps). We are interested in assessing the integration of XP ideas into a HIS engineering process. Because of the interdependencies between domains in HIS, it is insufficient to consider the application of XP practices solely to the software domain in HIS engineering.

In general, when engineering a product, developers can vary four factors: the scope of the project, time, quality, and resources. In building a HIS such as an aircraft engine, the opportunities for variation are reduced. In particular:

- **Scope**: this in general cannot be varied substantially as it is constrained by the laws of physics and materials. For example, in building an aircraft engine, development is constrained by the type of engine, the airframe to be used, etc. There are no grounds for omitting features and functionality from core control because these are all essential for ensuring safety and operation (although there will be some less vital elements). If reuse is allowed, then reusing product core components is outside the scope of the main development effort, but there are still certification issues (e.g., FAA guidance on software reuse, as well as DO-178B [13]). XP seems to emphasise variation in scope.

- **Time**: it is difficult to vary the time spent on such a project as it is usually constrained by the timing of the hardware and mechanical programmes. For example, the timing of an aircraft engine software development project is usually dictated by the timing of the engine programme itself, with severe penalties if delivery dates are missed. XP seeks to avoid the need to vary time by varying other factors, particularly scope and resources.

- **Quality**: one goal of any such project is to achieve certification, and thus no variation in this factor is possible. Other quality characteristics (e.g., performance, understandability, extensibility) are also tightly constrained by the need to achieve certification. XP tends to emphasise doing the simplest thing that works (achieves successful testing, certification, etc.).

- **Resources**: variation is possible here. However, currently most changes and repairs to problems come at a high cost involving adding substantial resources to the project, and this may be reducible via use of XP practices.

Thus, any application of XP practices to this domain will be severely constrained in terms of the flexibility that is allowed; most of the flexibility will arise in the use of resources. This raises the question of whether XP practices are more effective at producing HISs than alternatives especially as XP seems to rely heavily on scope flexibility, e.g., the ability to drop functionality from consideration.

Another important issue is the approach to be taken to verification and validation when engineering high-integrity systems. Consider, for example, aircraft engine development. Any software that is written is tested using a simulation rig that mimics properties of the hardware-software interface, as well as interactions with any mechanical components. An additional testing rig that integrates hardware and software will be required. Once hardware and software have been implemented and tested, they are tested on a real engine on the ground, and later tested on a real engine in the air. Some of these tests – especially on real engines – are extremely expensive and must be considered as one-offs, e.g., testing that the engine behaves safely in the case of catastrophic mechanical failure. This is a requirement for certification. XP, by comparison, relies on rapid testing and re-testing, and in the domain of HISs, there will be a tension between rapidity and completeness.

Another point of note is that there is generally not substantial feedback from HIS customers. Feedback is usually of the form “the system works” or “the system doesn’t work”, where this is usually based on observable behaviour or a certification process. Degrees of acceptability are not available, unlike in many XP projects where customers can often prioritise elements from the functional specification.

It is also important to note that safety is an emergent property of using a system in context; it is thus inherently coupled across disparate parts of the system design.

3. **Overview of Extreme Programming**

XP is one of many agile development that targets the following development problems: the requirements are not met by the system that has been constructed; the resulting system is out-of-date by deployment; and system quality is so poor that the system is unusable. All agile development techniques – including XP – rely on two principles: the short “inspect and adapt” cycle, and the short feedback loop. Specific principles and practices recognised by XP are described below.

3.1 **XP Principles and Practices**

The guiding principles of XP provide a concrete description of XP, and it is important to judge the approach against the complete set of practices and principles, rather than those that seem to be appropriate in a given context. The principles include customer satisfaction via continuous delivery of software; embracing changes in requirements; deliver working software frequently; having developers and customers collaborate; using face-to-face conversation; and emphasising simplicity. These principles are accomplished and enabled via a number of XP practices, which are summarised below.

- **Planning game**: quickly determine the scope of the next release using business priorities and technical estimates.
- **Small releases**: put a simple system into production quickly, then release new versions on a very short cycle.
- **Metaphor**: guide all development with a simple shared story of how the whole system works.
- **Simple design**: the system is designed as simply as possible; extra complexity is removed on discovery.
- **Testing**: programmers write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.
- **Refactoring**: programmers restructure the system without changing its behaviour to reduce duplication, improve communication, simplify, or add flexibility.
- **Pair programming**: all production code is written with two programmers at one machine.
- **Collective ownership**: anyone can change any code anywhere in the system at any time.
• Continuous integration: integrate and build the system many times a day, every time a task is completed.
• 40-hour week: work no more than 40 hours a week as a rule, without working overtime a second week in a row.
• On-site customer: include real live users on the team, available full-time to answer questions.
• Coding standards: programmers write code in accordance with rules emphasising communication through the code.

4. Assessment of XP Practices
We now assess each XP practice from the previous section, in terms of its effect on engineering HISs. For each practice, we identify research issues associated with the application, and also discuss feasibility. The intent of this is to draw out specific high-level issues that must be addressed in experimental work.

4.1 The Planning Game
There are complications with applying the planning game to HIS. One of these challenges will be discussed under On-site Customer. A key issue is in defining the increments that take place under XP. In the planning game, user stories are created that determine functionality; increments are based on these stories. However, in HIS, the minimum size of system that needs to be built before a verifiably (and certifiably) safe design has been reached may be very large. In other words, a substantial number of increments must take place before any useful feedback can be acquired and passed back to the customer. Another issue here arises with increments once a “safe” design has been reached. Particularly, as additional increments are carried out, what size should these increments be, and how can it be guaranteed that these increments do not jeopardise the safety of the overall system.

In terms of key research challenges, the main one with the planning game is to determine how to rapidly produce a safe “base design” upon which testing and verification can be done, and from which feedback can be given to customers. One possible way forward here is to make use of product line techniques, e.g., if a family of engine controllers uses the same control laws, then this “feature” can be used to quickly produce a safe preliminary design. Another research question is the issue of incremental certification: if an initial, though incomplete safe design has been produced, and additional increments carried out, can we incrementally certify the system? Alternatively can we define increments in such a way so that re-certification need not be carried out? This may be a reasonable approach as many of the increments in building HIS are perfective and corrective, and are verified through the use of testing and simulation. This is perhaps the “make or break” for the use of XP in HIS environments. HIS developers need to be able to make incremental change cheap; if XP cannot “deliver” this in a HIS environment then one of its attractions will have been lost.

4.2 Small Releases
In general, this practice advises that the system is put into production before the entire problem is solved. This is not generally possible in HIS since the entire system involves multiple teams building very different components (some software, some hardware, some mechanical) that must be integrated. As well, certification requirements prevent an incomplete system from being released. One way forward with this practice is to allow partial releases purely for testing and simulation use (as discussed in the previous practice), but not for certification. This may imply that the size of increments (as measured in terms of functionality that is delivered to a customer) will be likely to change as the project progresses: small increments may be carried out while releases are being made for the purposes of simulation and testing, but larger increments will take place as development moves to integration of software, hardware, and mechanical components.

XP may save money via this practice: an incremental test strategy may expose problems early in the process, even though it may not result in a certified system. There are two main research questions here.

1. Can we effectively do testing and simulation using models of devices and mechanical constructs? These models must be precise enough to allow components to be executed against them, and thus must include information such as fault models for devices, etc.

2. How do we obtain flexibility via small releases, while still ensuring technical coherence sufficient for certification?

4.3 Metaphor
The Metaphor practice suggests defining a set of metaphors between customer and developer that will be used to guide the project. The Metaphor practice is controversial among XP practitioners, in part because it is thought to be difficult to explain how to use metaphors in engineering. A substantial part of the difficulty comes from combining multiple metaphors – which may be needed in HIS – that are constructed by different engineers. The challenge with this practice in HIS is that experience shows that simple metaphors do not exist: key architects for a project will have a whole-system view, and can explain this view to stakeholders and customers, but this view necessarily includes a great amount of detail. Simple metaphors, as suggested by XP, do not give enough detail for certification. Typical metaphors used in HIS include safety models, control models, and so on, which do provide separate views of the system but they are generally not simple enough to provide the communication required by XP. It is possible that simple metaphors may be of some use early in the software development process, but the issue of how much use these will be is an open question, as is when is it necessary to move from simple metaphors to detailed models during HIS development.

Thus, it is unclear how appropriate this practice is for building HIS, and observations will need to be made of typical working practice in order for us to make any conclusions.

4.4 Simple Design
This practice is directly applicable to HIS. However, we note that HIS have innate complexity due to the multitude of dependencies that exist between software, hardware, and mechanical devices, including handling failure modes. Indeed, often well over half the code is concerned with handling failure conditions. This may preclude the construction of a simple design in any absolute sense. However, the practice of producing the simplest design possible is desirable to apply and is very much common practice.
4.5 Testing
This XP practice requires unit tests to be written while development proceeds, and all tests must run in order to deliver an increment, and in order to determine when to proceed to the next unit of functionality. This practice appears to be applicable to HIS. Unit tests are certainly used in HIS development. One requirement for certification is independent verification and validation; for example, the person writing the code should not be the person doing the V&V. This suggests that in this context, one member of each XP pair might be given the responsibility for testing the outputs produced by their team-mate, and could be considered independent for the purposes of certification.

Another issue arises with customer-written tests; we shall return to this point under On-site Customer later.

Two key research issues arise with testing and HIS. The first is with respect to incremental testing: can modified condition/decision coverage (MC/DC) testing [13], which is key in certifying HIS in the civil aerospace domain, be carried out in an incremental manner? A second issue is whether testing of HIS can be carried out quickly enough to enable the rapid feedback (i.e., supporting the rapid code-and-test process) required of agile methods. Early increments in building HISs may be large and rapid testing difficult to carry out. Moreover, some tests associated with building HIS will be impossible to carry out incrementally; witness, for example, tests carried out on a real engine while it is undergoing catastrophic failure. These tests are prohibitively expensive to carry out and thus are usually carried out once. (Usually re-certification of software after change does not require repeat of such tests, although it might if the “problem” being fixed relates to control of such failures.) Unit tests carried out using rigs or on hosts will be cheaper to carry out and can be done iteratively.

4.6 Refactoring
The refactoring practice in XP suggests transforming code (while preserving behavioural equivalence) to produce simpler designs, improve understandability, and increase overall quality. Refactoring is applied when flaws are detected, e.g., repeated code. Refactoring should rarely be needed in building HISs, since the domain and infrastructure are usually very stable. However, refactoring may be necessary when there is change due to external factors. For example, a customer may impose a change in sensor type, which will require re-engineering and refactoring of the software. Another example may be if there is a change in redundancy strategies, potentially requiring additional hardware and software redundancy to be provided. Also, there may be need for refactoring due to feedback from the safety process. In general, the software may need to be modified late in the project to control or mitigate problems in other parts of the system. Hazard analysis may also reveal the need to refactor, in order to eliminate hazards completely from the system.

In some HIS projects autogenerated code is used, e.g., produced from Simulink or Stateflow models. This code can be extremely complex and difficult to certify, and thus refactoring may be of substantial use here for simplifying architectures; it would only be possible to carry out refactoring of autogenerated code if it has been established that the code meets its requirements. However, the complexity of the autogenerated code may make it very difficult to refactor. Tool support is needed, and patterns, data mining, and pattern recognition techniques may be of use here. In general, the size and complexity of HIS makes it essential to provide tool support, not only for refactoring, but also for constructing models and autogenerating code. The key research issue here is therefore providing an expressive enough modelling language for capturing architectural and behavioural details of HIS, while enabling tool support for modelling and refactoring. At the same time, certification oversight requires clear traceability between models and autogenerated code, and the tools must provide assistance in navigating the relationships between these work packages.

4.7 Pair Programming
This practice suggests that programmers work in pairs in order to better catch errors, identify problematic code (that may need refactoring or rewriting) and to provide instant feedback on ideas. While generally a sound idea, there will need to be substantial cultural change in applying this practice to HIS, in part because of the nature of the artifacts – integrating software, hardware, and mechanical components – being produced. As well, HIS engineers have different domains of expertise, and much of their knowledge is not transferable or shareable with others. Thus, we posit that this practice could be rephrased as “pair modelling”, or “pair development”, in HIS. Certainly pure pair programming could be applied strictly at the software level, but the nature of HIS development is that the software team must communicate regularly with the hardware and mechanical teams, suggesting that there may be substantial value in making pairings along systems and software lines (e.g., one member of the pair is a programmer, the other a hardware engineer). The advantage of this approach comes with certification and independence: the system member of the pair could define requirements, which are then implemented by the software member. The different roles of the pair members enforce independence, thus enabling pairs to provide some measure of independent verification.

Given the size of typical HIS projects, it is likely that the project will involve many pairs of many different collaborating disciplines. A key research issue is to determine how this “collaborating set of pairs” process maps to the oversight process for certification. This would entail defining a strategy for implementing “pair modelling” or “pair development” within a certification process. As part of this, computer-support collaborative work tools for XP would ideally be applied.

4.8 Collective Ownership
This XP practice allows anyone involved in a project to change the code at any time. This practice is not applicable to HIS. The software that is to be written in a HIS requires substantial sub-domain knowledge, e.g., of fuel pumps, pressure sensors, starting systems. Not all people involved in such a project will have the domain knowledge required to modify code outside their own specific sub-domains; moreover, specialisation of knowledge is commonplace for HIS developers (i.e., engineers tend to stay in a given role when they move between family members, for example, always working in fuel systems). This provides the wrong level of granularity of specialization and knowledge to allow collective ownership.
4.9 Continuous Integration
This practice requires new code to regularly be integrated with existing code, and all tests re-run on integration. This practice is applicable to HIS, subject to there being a suitable test harness in which to run tests, e.g., a hardware and/or airframe simulator. This practice is tightly coupled with the Testing practice, and the research issues discussed in the Testing section apply to this practice as well, i.e., those related to size issues of increments.

4.10 Forty Hour Week
This practice is an important part of the XP philosophy. We do not assume that it is inapplicable to HIS but need to carry out observations of working practice to validate this hypothesis. The usual caveats made in the XP community about introducing this practice into organisations apply equally to HIS development.

4.11 On-Site Customer
XP states that a customer should be available full-time on the project, to answer questions from the developers. In general, there may be a customer team available to answer questions, so as to avoid customer burn-out and allow special domain knowledge to come to the forefront. Leaving aside the usual challenges with dealing with customers, there are specific complications to this practice in HIS. Foremost among these is that it is invariably the case that projects deal with a large number of customers, given that HIS involve a multitude of different components of different types. A typical project will involve many subdomains (often all within one company, but external contracts may also be involved), different levels of product integration (e.g., software integrated with electronics integrated with hydromechanical devices), and multiple concerned stakeholders. In an aircraft systems project stakeholders will include subdomains within the managing company, airlines, certification authorities, and many others. Thus, expecting a single customer, with knowledge of the system and knowledge of requirements, or even a team of customers with such capabilities to be available is unrealistic. There will be multiple customers that will need to be consulted at different times, and these customers may not always be available, because the large number of subdomains involved makes it extremely difficult to identify a priori which people should be involved. Moreover, it is extremely difficult to ask such a substantial number of people to be available full-time on the XP team, as this will invariably reduce the contributed business value of the acting customers. In general, the complexity of HIS customer requirements appear to reduce the agility of the process and the rate of feedback.

The job of the XP coach is also made more difficult as they may have the added responsibility of ensuring that developers talk to the right customers, and that customers make themselves available at the right time. Thus, coaches need business skills as well as engineering skills.

It is thus unclear how this practice applies to HIS; clearly it is desirable to do so, but there are logistic issues that need to be resolved, and these will depend on the project. The best way forward may be to use classical management structures, and to have a HIS project manager or chief architect act as full-time customer, providing them with the ability to dynamically involve other domain-specific experts as necessary. Again, the practicality of this approach needs to be assessed and, if necessary, argued against the increased provision of business value to the HIS organisation. An approach to this problem might be through the use of collaboration environments, see the conclusions.

4.12 Coding Standards
This XP practice is directly applicable to HIS, but we would extend it to be applicable to other design descriptions (e.g., models produced for certification), and would also require it to be applicable to safe subsets of programming language [14].

4.13 Summary
Table 1 summarises our discussion with respective to each XP practice and its applicability (full, partial, inapplicable) to engineering high-integrity systems.

<table>
<thead>
<tr>
<th>XP Practice</th>
<th>Applicability</th>
<th>Key Research Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning Game</td>
<td>Partial</td>
<td>How do we do safe increments?</td>
</tr>
<tr>
<td>Small Releases</td>
<td>Partial</td>
<td>What is the optimal size of increments for HIS?</td>
</tr>
<tr>
<td>Metaphor</td>
<td>Partial</td>
<td>Do simple metaphors exist? And if so, are they useful in this domain?</td>
</tr>
<tr>
<td>Simple Design</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Full</td>
<td>Can we carry out testing in an agile way in this domain? What is suitable testing infrastructure for this domain?</td>
</tr>
<tr>
<td>Refactoring</td>
<td>Partial</td>
<td>What are usable modeling languages for dealing with HISs?</td>
</tr>
<tr>
<td>Pair Programming</td>
<td>Partial/Full</td>
<td>What is the most effective use of pairs in HIS engineering?</td>
</tr>
<tr>
<td>Collective Ownership</td>
<td>Inapplicable</td>
<td></td>
</tr>
<tr>
<td>Continuous Integration</td>
<td>Full</td>
<td>What tools and infrastructure support agile testing for HIS?</td>
</tr>
<tr>
<td>Forty Hour Week</td>
<td>Partial</td>
<td>What is the effect on culture?</td>
</tr>
<tr>
<td>On-Site Customer</td>
<td>Partial</td>
<td>How do coaches manage different combinations of customers?</td>
</tr>
<tr>
<td>Coding Standards</td>
<td>Full</td>
<td>What coding standards are applicable?</td>
</tr>
</tbody>
</table>

Table 1 – Summary of XP Practices in HIS Engineering

5. Extensions to XP Practices
Our assessment of the XP practices leads us to suggest four additional practices and techniques that are necessary in building HIS. We discuss each of these briefly.
5.1 Safety Process
The XP practices and principles do not explicitly include a safety process. It is insufficient to simply include “safety” as a requirement and expect that the XP process will iterate to a safe system. Safety is a whole-system process that must include consideration of hardware- and mechanical-software interfaces. Given the incremental nature of XP it is necessary to have composable representations of safety behaviour that can be designed (or perhaps automatically generated from models) during XP iterations; this is an open research issue. There also may be benefits to using a product family-based approach in conjunction with XP practices here. It is an open question how to link a safety process with XP practices.

5.2 Static Analysis
Static analysis techniques are frequently used in building HISs (for example, schedulability analysis, fault and error propagation analysis, etc). These techniques will need to be integrated with the XP iterative process, and thus it is critical that the techniques and supporting tools be incremental and highly efficient.

5.3 Process Risk Management
This is difficult in HIS projects, and XP does not appear to provide mechanisms to make it easier. It is vital to add explicit risk mitigation and management techniques to the XP process.

5.4 Design Representations
HIS projects usually involve models of some kind for representing system designs (e.g., written in Simulink). These models tend to be complex, and heavy with detail. But this level of detail is necessary to master the inherent complexity of the architecture of HIS products. These models are considered useful by HIS developers, and they must be integrated with XP practices. Autocode generation will be of use here, as well as testing and simulation frameworks that work on models.

6. Summary and Conclusions
We have addressed the compatibility of XP and high-integrity systems engineering. There is substantial compatibility, suggesting that it is worthwhile to attempt to integrate the approaches in industrial case studies. It is also evident that XP and HIS are complementary: XP aims at rapidly producing working code that satisfies a customer; HIS engineering produces systems that satisfy governmental and external regulation and are safe to use in civil and defense industries.

There appear to be several key questions to address in evaluating the usefulness of combining XP and HIS engineering. We call this approach High-Integrity Extreme Programming (HIXP) in the following.

1. What is a useful definition of increment in HIXP? This definition must satisfy the requirement of providing useful, rapid feedback to the multitude of customers in HIS engineering, as well as leading to a system that is eventually certifiable.

2. What is a useful testing infrastructure that permits the different kinds of testing and simulation that occur in HIS, while still enabling rapid feedback? At which increments during HIXP can and should this infrastructure be used?

3. What guidance and training must be provided to HIXP coaches in order to facilitate customer feedback and deal with the range of customers inherent in HIS engineering?

4. Can the pair programming/modelling practice be used to enable independent assessment within pairs, for the purposes of leading to certification? This will require negotiation and discussion with the certification authorities, e.g., the Civil Aviation Authority.

5. How will the extensions/additional practices be integrated with the typical XP approach?

It is clear that, in order to address such questions, considerable experimentation, involving industrial software engineers, will be needed. We are currently in the process of defining such experiments but it is difficult to see how to carry them out effectively, without posing project risk. One idea we are exploring is the use of collaborative software engineering environments which would enable a research team in a University to collaborate with engineers in a company, without the need for co-location. If the project used is realistic, not real, then useful understanding can be gained without imposing undue risk. Also, such collaboration environments may provide an effective means of overcoming the difficulties noted about having an “On-Site Customer”.

References
15. Matlab/Simulink, see http://www.mathworks.com/