Software Development in Antagonistic and Dynamic Operational Environments: Experiences with Digital Governments

Ivan Krsul[1]
University of Florida
ivan@acis.ufl.edu

Abstract

Software systems for governmental organizations are mostly defined by procedures and policies that are, in some contexts, dynamic, seldom formally defined or even documented. Furthermore, changes in policy can be rapid and frequently antagonistic to those previously defined. Current development methods are not resilient or adaptive under change and software systems developed in dynamic and antagonistic environments exhibit significant problems that cannot be prevented using traditional software engineering methods. Software for governmental organizations needs to be secure and security, in its most general definition, can only be provided when the logic used for the requirements analysis matches closely that of the logic in place by the time the software is deployed. In this paper we show that during the implementation of the ASYCUDA++ system in the Bolivian National Customs Agency, we found that current software development practices, though technologically sound *per se*, are not tightly integrated with effective mechanisms for identification of potential dynamic and antagonistic environments and argue that, at least, software development models should provide mechanisms for adapting the software development cycle to shifts in global goals.

1 Introduction

Computer Systems are instantiations of requirements derived, generally, from a logic that defines the expected results. In off-the-shelf software, the designer defines the logic a-priori and users must adapt to the system. In large organizations, requirements are derived from a *business logic*[2] and the policies and regulations that are relevant. In governmental organizations, the business login and the system requirements are derived from regulations (i.e. laws and decrees), the resulting procedures that interpret the regulations and document the necessary details, and the policies defined by the appropriate governing body. Once defined, regulations and procedures are unlikely to change drastically, but policies can change dramatically in short periods of time.

Software systems for governmental organizations implement the government's business logic and the details of this logic are mostly defined by procedures and policies. In some contexts, policies are dynamic and seldom formally defined or even documented. Furthermore, software for governmental organizations needs to be secure and security, in its most general definition, can only be provided when the business logic used for the requirements analysis matches closely that of the business logic in place by the time the software is deployed. Given that the business logic of the organization encapsulates high-level security requirements, including access control, authentication, authorization, availability, privacy, etc., if the business logic in place by the time the software is deployed is different from the one used to define the security requirements, it is unlikely that the system will implement the correct security model.

---

[1] Ivan Krsul was the Director of the PRISMA (Proyecto de Reforma en la Implementación de Sistemas Informáticos Aduaneros) project at the Aduana Nacional de Bolivia from March 2000 to December 2001 and participated in the implementation of the ASYCUDA++ system for the Aduana Nacional de Bolivia since September 1999.

[2] The business logic of an organization is the set of high-level objectives defined for the organization and hence implicitly for the systems that must operate in that organization. The business logic is equivalent to the development goals described in [34].

A good example of this is the Bolivian National Customs Agency (Aduana Nacional de Bolivia – ANB), which for two and a half years, starting September 1999, pursued an aggressive development schedule for the adaptation of the ASYCUDA [32] system to the existing regulations and policies defined by the ANB. ASYCUDA is a computerized customs management system which covers most foreign trade procedures and has been successfully implemented in over 80 countries.

Developed in Geneva by UNCTAD (United Nations Conference on Trade and Development) [33], the ASYCUDA system is very configurable and traditionally installed without modifications to the source code, and the implementation efforts are limited to the adaptation of the local customs procedures to conform to the Kyoto convention, and the harmonization of documents and codes. The implementation of the ASYCUDA software (version 1.16, also known as ASYCUDA++) at the ANB differed considerably from the traditional implementation model used in the countries where this system was deployed[3], as extensive modifications were required to the source code of the system to conform to stringent regulatory and political requirements imposed by the Bolivian government. A reform and modernization project called PRISMA[4] was created with a team of 18 people for the effect.

This team used domain experts with more than 10 years of field experience in the design of the software requirements and standard software engineering models and practices to reduce faults, including:

- CMM for software model [1, 2] (in this particular implementation the PRISMA team reached level 3[5]).
- Techniques recommended for rapid development [14].
- Standard Risk Analysis and Requirement Management practices.
- Practices recommended for ISO 9000-3 certification. [31]
- Standard requirement management practices, including goal management and follow-up.
- Quality control practices conforming to ISO 9000-3, tightly integrated into the production environment. [31]
- Standard software testing techniques, including testing, code walk-through, operational simulations, etc.

Said models and practices successfully reduced traditional software faults in the development stages, yet the ASYCUDA++ exhibited a significant number of operational problems that had mean-repair times in excess of 12 hours[6]. Most of these operational problems resulted in violations of explicit or implicit security policies, as security requirements were embedded into the system architecture.

Posterior analysis of 178 problems reported in the development process shows that the majority of problems experienced were not due to inadequate requirement analysis, poor development practices or flawed software, but to rapidly changing regulations and policies, which in effect were creating an operational environment where it seemed that there were software faults, when in reality the abstract business logic had changed and the software systems had not followed these changes. Additionally,

---

[3] Bolivia is the first country where the source code was released and all previous implementations were off-the-shelf. The software could be configured and regulations and policies had to de adapted to fit the system.

[4] PRISMA: Proyecto de Reforma en la Implementación de SisteMas informáticos Aduaneros.

[5] No formal certification mechanisms were used. The implementation team self-assessed the level in three month intervals.

[6] In this paper we limit our analysis to the problems reported during the implementation of the export control phase of the project and cover the period March 2000 to October 2001.

the changes in regulations and policy observed were frequently antagonistic to those previously in place.

Many software engineering models have been designed to reduce traditional software faults, increasing the likelihood of producing secure software. However, said models are still lacking in techniques for reducing faults resulting from dynamic and antagonistic environments. Of particular interest are environments where software needs to operate securely under stringent and changing judicial and political constraints.

In this paper we refer to environments as *hostile* where the abstract business logic is dynamic and changes are antagonistic to the previous business logic, extend the work of [3, 4] and posit an analysis of software faults resulting from said operational environments and the factors that contribute substantially to their introduction in production environments.

During the analysis we identified four factors, resulting from hostile environments that significantly increase the probability of operational software faults: Environmental Opacity, Environmental Variability, Environmental Complexity, and Environmental Bias. We measured a series of features identified for each problem concluding that new mechanisms paying particular attention to dynamic and changing operational environments are needed.

2 Related Work

In [4] a fault classification scheme is presented in two broad areas: Coding Faults and Emergent Faults. The latter "result from improper installation of software, unexpected integration incompatibilities, and when a programmer failed to completely understand the limitations of the run-time modules. Emergent faults are essentially those where the software performs exactly according to specification, but still causes a fault." [4].

In [3] "the external environmental context is examined primarily from the perspective of environmental uncertainty while organizational perceptions... are taken primarily from innovation diffusion theory..." and Environmental Uncertainty is broadly classified into Environmental Complexity and Environmental Variability. "Environmental complexity is the variety of environmental factors that affects organizations... It increases the diversity of environmental information that an organization needs to handle. Environmental variability is defined as the frequency of changes in the environmental factors..." [3].

3 Software Development in Hostile Environments

Software engineering life cycles divide the development process into a number of discrete steps that range from conception, through requirement analysis, development to release and operation. The importance of a correct design has long been recognized and there are many techniques for assuring that the software indeed meets end-user expectations, from the use of critics in the design environment [6] to effective risk management [14]. However, classification of vulnerabilities and security problems has been largely limited to the analysis of Coding Faults [7, 8, 9, 10, 11, 12, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24], even though we have evidence that a large proportion of security faults are introduced early on in the development phases. For example, 22 of the 50 security flaws identified in [25] were introduced during the Requirement / Specification / Design phases.

This is particularly relevant for software that is built in environments that are subject to constant change. In [13] Finkelstein and Kramer lists *Change* as a current research challenge: "How can we cope with requirements change? How can we build systems that are more resilient or adaptive under change? How can we predict the effects of such changes?" To this list we add: What effects does change have on the security of software? How can we build systems that are secure in spite of change? The implementation of the ASYCUDA++ system at the ANB is the cornerstone of a digital government project aimed at the electronic processing of all customs operations and as such must

guarantee a high level of security that can only be achieved by reducing the operational faults of the system and by developing software engineering practices that can identify whether a change in environment would result in violations of security constraints[7].

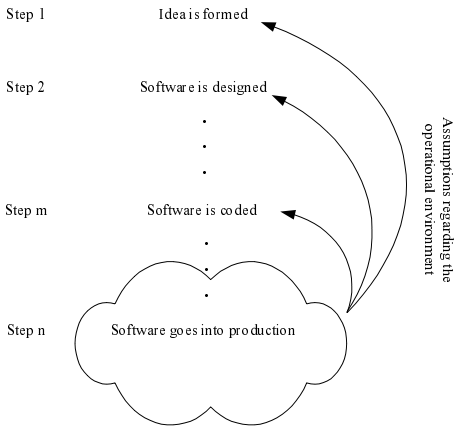

Figure 1: Environmental assumptions in the development process

Regardless of the software engineering life cycle used for a systems project, at each step in the development process engineers make a number of assumptions regarding the environment where the software will finally operate. At each step, each incorrect environmental assumption introduces a small error that has a cumulative effect on subsequent stages in the process.

The error introduced is not the product of an incorrect design, a misunderstanding of the specifications, or the product of software faults. The design is correct and, in the absence of other software faults, the implementation is correct for a system that was designed and built to operate in an environment differs from the environment in which it will actually operate.

Developers that have made faulty assumptions regarding the environment in which the system will operate, and have not introduced any traditional software faults, will build a perfectly correct system that *would* operate as designed in the environment presumed. As such, the system is devoid of software faults. There are no design errors, no mistakes in the development process, no coding flaws, yet the system will fail to operate as expected, and sometimes the failures can be catastrophic.

If the abstract requirements for the system change during the development and such changes are antagonistic to the business logic in place at the time requirements were specified, the effect is equivalent to that of the developer making faulty assumptions regarding the operational environment.

When a system is built in the presence of dynamic and antagonistic requirements, we may postulate that it will operate in a "Hostile Environment". Hostile not in the sense of the environment is purposely trying to damage the system, but nevertheless managing to do so. We perceive a natural resistance towards the system from the environment, that manifests itself as defects in the design or coding of the system.

Software that operates in hostile environments fails to operate, as pointed out before, not because it was badly designed, or badly coded, but simply because it was designed for an environment that is no longer present by the time the system is operational.

4 A-priori categorization of emergent problems

In this section we present a categorization of software faults for emergent or environmental faults, where these manifest themselves, mainly due to a hostile environment, during operation of the software.

[7] Note that in principle this would be an extension to the theory and practice of runtime monitoring of environmental assumptions. However, existing models require formal specification of environmental constraints and the environmental variability mentioned in this paper is at the policy-making stage, and by definition high-level and abstract.

> *"The context of software system development is changing. Systems are rarely developed from scratch; most system development involves extension of preexisting systems and integration with `legacy' infrastructure. These systems are embedded in complex, highly dynamic, decentralised organisations; they are required to support business and industrial processes which are continually reorganised to meet changing consumer demands."* [13]

This is the case of system development projects being undertaken in developing countries[8], and the implementation of the ASYCUDA++ system at the ANB is no exception, since this customs organization experienced drastic changes during implementation, creating enormous environmental uncertainty.

In particular, we found it necessary to extend the classification proposed by [3] to describe the conditions that contribute significantly to the development of software systems that are correctly designed and implemented, as required by existing software engineering practices, yet produce significant faults during operation.

As such, we traced the causes for the operational problems identified during the deployment of the ASYCUDA++ system and identified the following factors as contributing substantially to the creation of hostile environments:

Environmental Opacity refers to the degree of opacity in the operational environment, during the design, coding or deployment. The environment is opaque if developers are unable to "see" the characteristics of the operational environment [26].

Environmental Variability refers to the "*rate and volume of changes in the environmental factors. Rapid and large volume of changes could decrease confidence in predicting outcomes....*" [3].

Environmental Complexity "*refers to the diversity and interdependence of environmental factors that organizations have to contend with...*" [3].

Environmental Bias refers to the degree of bias – wishful or prejudiced thinking – in the design of the systems with respect to the actual operational environment.

5 Data Collection

During the implementation of the ASYCUDA++ system at the ANB the PRISMA project kept a detailed record of 178 different problems considered *hard to fix* by the development team – problems that were not solved within 12 hours of being reported. 60% of these problems were identified in a workshop with users of the system four months after initial deployment, and the members of the development team reported the remaining 40% six months after the initial deployment.

For each of the problems reported, we measured keywords using the following criteria:

Importance: Describes the importance of solving the identified problem. Define importance as a function of the severity of the effects of not solving the problem. If the effect is that the system can cease to function, we classify the problem as *very important*. If the effect is that the system malfunctions, but still produces meaningful results, we classify the problem as *important.* If the effect is that the system still functions correctly, we classify the problem as *unimportant.*

---

[8] Based on the MARIA system (Argentina), the Lucía system (Uruguay), the implementations of ASYCUDA in Venezuela and Bolivia, the implementation of the SIF system in Honduras, The Red Social Project (http://www.redsocial.org), the SIGADE projects, etc.

Urgency: Describes the perceived urgency of solving the problem by the development team. If the problem needs to be fixed within a month, we classify the problem as *very urgent.* If the problem needs to be fixed within three months, we classify the problem as *urgent.* If the problem can be solved after three months, we classify the problem as *not urgent.*

Impact: Measures the dispersion of the effects of not solving a problem. A problem has *global impact* if all users of the system will feel the effects of not solving the problem, *institutional impact* if all users of the institution that developed the system will feel the effects of not solving the problem, and *local impact* if only developers would feel the effects of not solving the problem.

In addition, various yes/no features were measured for each problem as follows:

Type of Problem	**Description**
Normative	These are problems that result from incorrect or incomplete norms or procedures defined *a-priori*. These norms or procedures define the desired functionality of the systems built.
Systems	These are problems that result from incorrect system design or coding.
Infrastructure	These are problems that result form deficient or non-existent infrastructure that is required for the system to operate.
Administrative	These are problems that result from deficient administration.
Support	These are problems that result form poor customer support.
Training	These are problems that result from poor training.

Table 1 – Features measured: type of problem

Entity or group that can solve the problem	**Description**
Internal	The systems group/developers can solve the problem.
Institution	Any other group in the institution (i.e. not the developers) can solve the problem.
External	People outside the institution can solve the problem.

Table 2 – Features measured: group that can solve the problem

Type of solution identified	**Description**
Policies	Solving the problem requires the creation of policies – these tend to be administrative, but can also be legal, technical or normative.
System / Technical	Solving the problem is a mater of applying existing technical resources – generally fixing bugs, system configuration, network configuration, etc.
Normative	Solving the problem requires the development of norms or procedures.
Coordination	Solving the problem requires that two of more groups –independently administered– coordinate efforts.
Communication	Solving the problem requires communication – includes emailing people, distributing documentation, publishing notices, etc.
Planning	Solving the problem requires a significant amount of further planning – generally because existing methods or techniques do not apply.

Table 3 – Features measured: type of solution identified

Cause of Emergent Fault	**Description**
Environmental Bias	The developers identified that the problem was caused in part because at some stage during the implementation of the system, somebody made a significant

	decision that was biased (i.e. it was done based on wishful thinking).
Environmental Opacity	The developers identified that the problem was caused in part because it was not possible to clearly identify the characteristics of the actual operational environment.
Environmental Variability	The developers identified that the problem was caused in part because the operational environment varied significantly between design/coding and operation.
Environmental Complexity	The developers identified that the problem was caused in part because the operational environment had a significant amount of complexity.

Table 4 – Features measured: cause of emergent fault

Security Policy Violated	**Description**
Integrity	Problem resulted in a violation of implicit/explit integrity policy
Accountability	Problem resulted in a violation of implicit/explit accountability policy
Access Control	Problem resulted in a violation of implicit/explit access control policy
Availability	Problem resulted in a violation of implicit/explit availability policy
Authentication	Problem resulted in a violation of implicit/explit authentication policy
Privacy	Problem resulted in a violation of implicit/explit privacy policy

Table 5 – Features measured: security policy violated

The following table summarizes the findings:

Result	**% of Problems**
Problems classified as "Not Urgent"	40%
Problems classified as "Urgent"	23%
Problems classified as "Very Urgent"	37%
Problems classified as "Not Important"	40%
Problems classified as "Important"	42%
Problems classified as "Very Important"	18%
Problems that were caused by one or more or Environmental Variability, Environmental Complexity, Environmental Bias or Environmental Opacity.	65%
Problems that were caused by Environmental Variability	30%
Problems that were caused by Environmental Complexity	37%
Problems that were caused by Environmental Bias	21%
Problems that were caused by Environmental Opacity	24%
System problems of technical nature that could be solved by the development team and where the solution was technical.	11%
Problems that could not be solved by the development team.	58%
Problems that did not have a systems technical solution.	86%
Normative problems	24%
System Problems	12%
Infrastructure Problems	15%
Administrative problems	53%

Support Problems	7%
Training Problems	20%
Problems whose solution required the generation of policies.	26%
Problems whose solution included technical solutions.	14%
Problems whose solution included the development of further norms, procedures, laws, etc.	22%
Problems whose solution requires significant amounts of inter-group coordination.	49%
Problems whose solution requires communication.	29%
Problems whose solutions required significant amounts of further planning.	47%
Integrity	25%
Accountability	16%
Access Conttrol	9%
Availability	28%
Authentication	11%
Privacy	4%

Table 6 – Summary of finding for 178 problems collected

6 Co-word Analysis of Operational Software Faults

In this section we present the result of applying co-word analysis to the 178 problems mentioned previously for the project implementing the ASYCUDA++ software for the ANB. As mentioned before, this project used standard software engineering models to reduce operational faults, and successfully reduced traditional software faults in the development stages, yet exhibited a significant number of problems that had mean times to repair in excess of 12 hours.

The data collected in section 5 can be easily converted into a database where each problem is described as a set of related keywords and visualizations techniques can be applied to discover the sources of the problems or strengthen our confidence in the a-priori classification developed in section 4.

Co-word analysis is a content analysis technique that is effective in mapping the strength of association between keywords in textual data. Co-word analysis reduces a space of descriptors (or keywords) to a set of network graphs that effectively illustrate the strongest associations between descriptors [29, 30].

Co-word analysis is an example of a graphical modeling technique that applies some of the ideas of association analysis [27, 28]. Graphical modeling is a variant of statistical modeling that uses graphs to display models. "In contrast to most other types of statistical graphics, the graphs do not display *data,* but rather an interpretation of the data, in the form of a *model*… Graphs have long been used informally… to visualize relations between variables." [27].

This technique illustrates associations between keywords by constructing multiple networks that highlight associations between keywords, and where associations between networks are possible.

For co-word analysis, every field in a database can be used as a keyword, or can be transformed to a series of keywords by applying the following rules:

1. If a field in the database can have the values yes, no, ?, and NA, then a keyword with the name of the field is generated if the value of the field is yes.

2. If the field in the database can have a single value from a list then a keyword with the name of the field followed by the value of the field is generated.

3. If the field in the database can have a list of values from a well-defined set, then for every value in the field we generate a keyword with the name of the field followed by the value of the field.

We applied these rules to the features measured for the 178 problems identified during the deployment of the ASYCUDA++ system as mentioned in section 5, and used the co-word tool originally developed for [24] at Purdue University and described in [35], and obtained from our data the networks shown in Figures 2, 3, 4 and 5 networks for the parameters *max pass 1 links* = 5, *max links* = 10, *min coword* = 25.

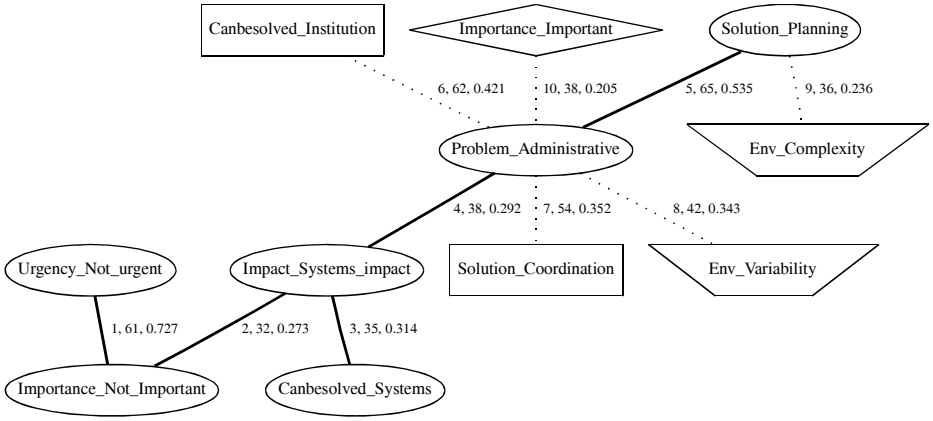

Figure 2: Graph No. 1. Density = 0.5, Centrality = 0.72
This graph shows that the problems identify were frequently administrative problems that had a systems impact, required significant amounts of further planning for the solution, were problems that were not urgent or important, and the problems had technical solutions. These problems were generally the result of policy changes that resulted in changes to the desired functionality during the development stages.

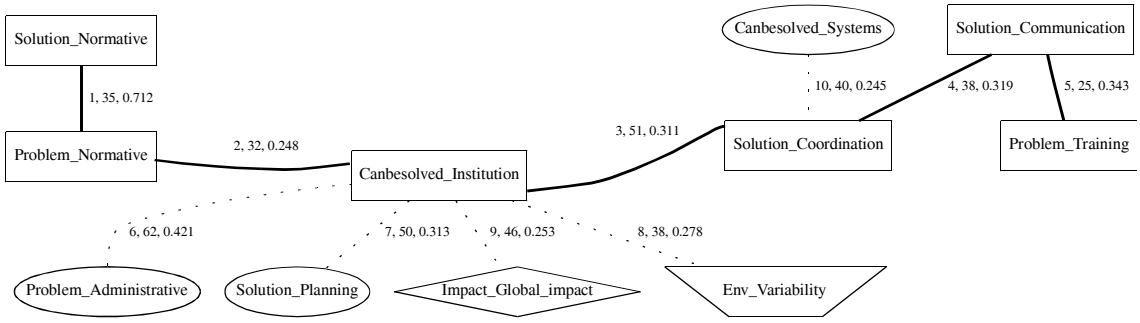

Figure 3: Graph No. 2. Density = 0.48, Centrality = 0.69
This graph shows a group of problems that were normative in nature (i.e. the regulatory framework changed or was re-interpreted during implementation), could not have been solved by the system team, required significant amounts of coordination and communication and required training or re-training personnel on the use of the system, regulations, etc. These problems are typically the result of major changes in regulation or policy, or the result of drastic changes in the interpretation of the regulations or policies during development.

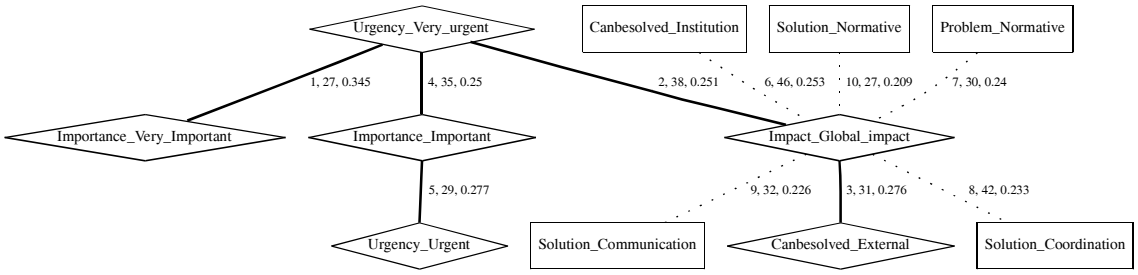

Figure 4: Graph No. 3. Density = 0.3, Centrality = 0.52
This graph shows that a group of problems identified affected the development team, the institution and the end-users (global impact), could be solved by parties external to the institution, and were very urgent and very important. These problems were typically the result of resistance by the end-users to the regulations and policies defined by the ANB (and instantiated by the system). The problems typically required political solutions that significantly affected the development team.

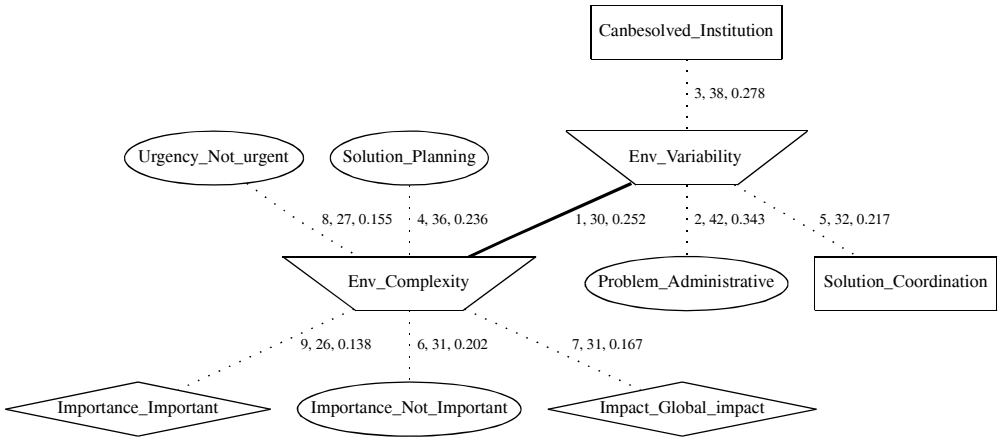

Figure 6: Graph No. 4. Density = 0.25, Centrality = 0.64
This graph shows frequent associations between Environmental Variability and Environmental Complexity. This is to be expected, as complex system are more likely to have problems when the environment changes.

In these networks, solid lines connecting nodes represent pass-1 links and dotted lines represent pass-2 nodes. Each link is labeled with a triple <*number, co-occurrence, strength*> that indicates the order in which the links were added (1 corresponds to the first link), the co-occurrence for the key pair, and the strength of the link.

The networks generated by the co-word analysis tool show the distribution of their centrality and density values. The resulting networks that have the highest centrality and density values represent the more predominant relationships among the keywords in the data. Isolated networks identify relationships with keywords that have low centrality values and hence the keywords are infrequently used in other networks. Isolated networks with high-density values indicate strong relationships among keywords in isolated groups, and point to dominant features of the database.

7 Interpretation of Results

The data collected, as summarized in Table 6, shows that aggressive use of standard software engineering methodologies is successful in reducing the incidence of traditional software faults, as 89% of the problems recorded were not systems problems that could be solved by the development team using technical tools at their disposal.

However, the total error was far larger than expected, as 86% of the problems did not have a technical solution that could be applied by the development team.

Particularly interesting is that the solution to 49% of the problems identified required significant amounts of coordination and the solution to 47% of the problems required significant amounts of further planning. This is an indication that the development team probably did not prepare adequately for the types of problems encountered. Since 65% of the problems were caused by one or more of Environmental Complexity, Environmental Bias, Environmental Opacity or Environmental Variability, it is reasonable to assume that a significant fraction of the problems encountered require either more coordination efforts or further planning.

Figures 3 and 4 show how most problems encountered required coordination and planning as a significant component of the solution, with 24% of problems being normative (relating to norms, procedures or law) and 53% of the problems being administrative (relating to general systems and institutional administration). Such problems frequently arose because of either unexpected changes or unexpected complexity in the operational environment.

Figure 6 shows the strong correlation between Environmental Complexity and Environmental Variability, and shows that the problems identified typically required significant amounts of coordination and planning and that the development team alone could not have solved said problems.

During the final stages of development, or early in the deployment of the system, the business logic (equivalent to the development goals described in [34]) had changed without due consideration to the effects on system development. Rather than the exception, such changes tend to be the norm in environments that operate under evolving political and/or judicial constraints. It is noteworthy that some of the institutional goals (as defined in [34]) were hidden from the development team and/or domain experts because of political reasons[9].

Finally, it is interesting to note that 25% and 28% of the reported problems resulted in a violation of implicit or explicit integrity or availability security policies, while only 9% and 4% of the problems violated an access control or privacy policy.

8 Conclusions

During the implementation of the ASYCUDA++ system in the Bolivian National Customs Agency (Aduana Nacional de Bolivia – ANB), we found that current software development practices, though technologically sound *per se*, are not tightly integrated with effective mechanisms for *a-priori* identification of potential hostile environments. Through the application of traditional software engineering methods, the PRISMA projects was successful in reducing traditional software faults, but was largely unable to prevent a significant number of problems that resulted from operational environments that were either rapidly changing (variable), complex, opaque or not as expected during the design (biased).

Systems such as the ASYCUDA++ customs software typically require high levels of security and current software engineering models are unable to deal with stringent and changing judicial and political operational environments, particularly those where change cannot be predicted a-priori.

To avoid this type of problems, new mechanisms are needed that pay particular attention to dynamic and changing operational environments [13], including but not limited to:

- Qualitative observations of the decision making process used by legislative and political bodies.

[9] The mere fact that we may have hidden goals makes the requirement analysis challenging, but this is beyond the scope of this paper.

- Computer security models for dynamic operational environments.
- Tools that provide better support for the development of software in changing environments.
- Tools and techniques for predicting hostile environments.

Finally, we argue that software development mechanisms for hostile environments, as shown in figure 7, should provide mechanisms for adapting the software development cycle to shifts in global goals, particularly when the shift in goals result in conflicts with the actual implementation[10]. Alternatively, mechanisms should exist that prevent the shift in global goals when the software is unable to follow suit, by providing effective feedback to policy makers regarding the cost and effort required to adapt an evolving system to changing global goals.

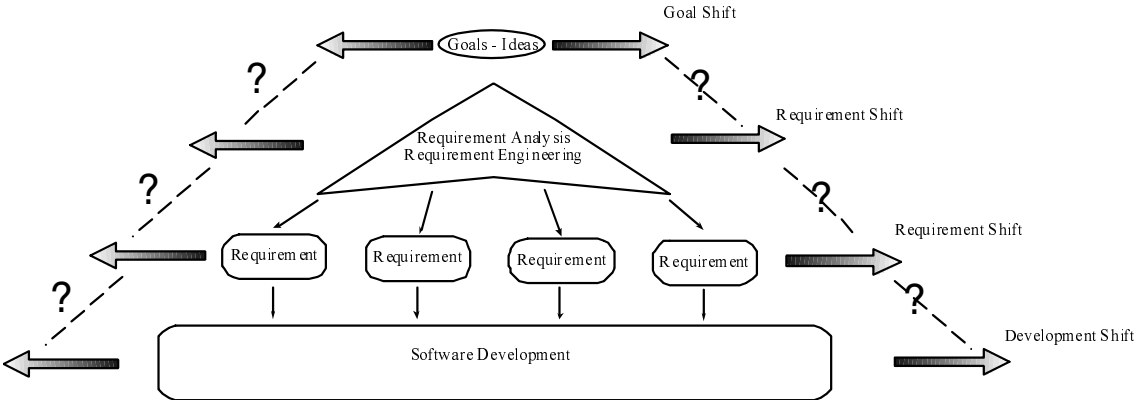

Figure 7: Goal Shifts Must Result in Implementation Shifts

9 Ackoweldgments

The author would like to acknowledge Prof. José Fortes of the ACIS Laboratory, University of Florida, for his valuable suggestions on the first draft of this paper, and to the Aduana Nacional de Bolivia, and in particular to the members of the PRISMA project, for the data collection that made this analysis possible.

10 Bibliography

[1] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis and Charles V. Weber, *Capability Maturity Model SM for Software, Version 1.1*, Technical Report CMU/SEI-93-TR-024, Software Engineering Institute, Carnegie Mellon University, May 1993.

[2] Mark C. Paulk, *Using the Software CMM in Small Organizations*, Available by request from the Software Engineering Institute, Carnegie Mellon University, 1998.

[3] Choon-Ling Sia, Hock-Hai Teo, Bernard C. Y. Tan and Kwok-Kee Wei, *Examining environmental influences on organizational perceptions and predisposition toward distributed work arrangements: a path model*, Proceedings of the international conference on Information systems, 1998.

[4] Taimur Aslam, Ivan Krsul and Eugene Spafford, *Use of A Taxonomy of Security Faults*, Technical Report TR-96-051, September 1996, Purdue University.

[10] Note that these are typical of the applications in the domain examined and we cannot generalize to application domains where there can be no goal shift (for example, in the design of a laser operating device)

[5] Simson Garfinkel and Gene Spafford, *Practical UNIX and Internet Security*, O'Reilly & Associates, Inc, 1996, Second Edition.

[6] Gerhard Fischer, Kumiyo Nakakoji, Jonathan Ostwald, Gerry Stahl and Tamara Sumner, *Embedding Critics in Design Environments*, The Knowledge Engineering Review, Vol. 8, pp. 285-307, 1993.

[7] Dodson, J., *Specication and Classication of Generic Security Flaws for the Tester's Assistant Library*, M.S. thesis, 1996, University of California at Davis.

[8] Matt Bishop, *Analyzing the Security of an Existing Computer System*, Proceedings of the Fall Joint Computer Conference, Pages 1115-1119, November 1986.

[9] Peter Gavin, *Designing Secure Software*, SunWorld, Published electronically at http://www.sun.com/sunworldonline/, April 1998.

[10] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, Diego Zamboni, *Analysis of a Denial of Service Attack on TCP*, Proceedings IEEE Symposium on Security and Privacy, May 1997.

[11] Weinliang Du and Aditya P. Mathur, *Vulnerability Testing of Software Using Fault Injection*, Purdue University, 1998.

[12] Ivan Krsul, Eugene Spafford and Tugkan Tuglular, *A New Approach to the Specification of General Computer Security Policies*, COAST Laboratory, Department of Computer Sciences, Purdue University, COAST Technical Report 97-13, January 1998.

[13] Anthony Finkelstein and Jeff Kramer, *Software engineering: a roadmap*, Proceedings of the conference on The future of Software engineering, Limerick, Ireland, Pages 3 - 22, 2000.

[14] Steve McConnell, *Rapid Development: Taming Wild Software Schedules,* Microsoft Press, 1996.

[15] John D. Howard, *An Analysis of Security Incidents On The Internet: 1989 - 1995*, Ph.D. Thesis, Carnegie Mellon University, April 1997

[16] Richard A. DeMillo and Aditya P. Mathur, *A Grammar Based Fault Classification Scheme and its Application to the Classification of the Errors of TeX*, Software Engineering Research Center, Purdue University, Technical Report SERC-TR-165-P, 1995.

[17] Eugene H. Spafford, *Extending Mutation Testing to Find Environmental Bugs*, Software Practice and Principle, Volume 20, Number 2, Pages 181-189, February 1990.

[18] Michal Young and Richard N. Taylor, *Rethinking the Taxonomy of Fault Detection Techniques*, Software Engineering Research Center, Purdue University, September 1991.

[19] Matt Bishop and Dave Bailey, *A Critical Analysis of Vulnerability Taxonomies*, Department of Computer Science at the University of California at Davis, Technical Reporte CSE-96-11, September 1996.

[20] Matt Bishop, *A Taxonomy of UNIX System and Network Vulnerabilities*, Department of Computer Science at the University of California at Davis, Technical Report CSE-95-10, 1995.

[21] Matt Bishop and Michael Dilger, *Checking for Race Conditions in File Accesses*, USENIX Association, Computing Systems, Volume 9, Number 2, Pages 131-152, 1996.

[22] Richard R. Linde, *Operating system penetration*, National Computer Conference, 1975.

[23] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo and D. A. Webb, *Security Analysis and Enhancement of Computer Operating Systems*, Institute for Computer Science and Technology, National Bureau of Standards, Technical Report NBSIR 76-1041, 1976.

[24] Ivan V. Krsul, Software Vulnerability Analysis, Ph.D. Thesis, Purdue University, May 1998.

[25] Carl Landwher, Alan Bull, John McDermott and William Choi, *A Taxonomy of Computer Program Security Flaws*, Naval Research Laboratory, Technical Report NRL/FR/5542-93-9591, November 1993.

[26] Dietrich Dorner, *The logic of failure: why things go wrong and what we can do to make them right*, Metropolitan Books, 1996.

[27] D. Edwards, Recent Advances in Descriptive Multivariate Analysis, Royal Statistical Society Lecture Note Series, Chapter 7, *Graphical Modelling*, pages 135-156, Clarendon Press, 1995.

[28] Leonard Kaufman and Peter J. Rousseeuw, *Finding Groups in Data*, John Wiley & Sons, Inc., Wiley Series in Probability and Mathematical Statistics, 1990.

[29] Neal Coulter, Ira Monarch and Suresh Konda, *Software Engineering as Seen Through Its Research Literature: A Study in Co-Word Analysis*, Journal of the American Society for Information Science (JASIS), Volume 49, Number 13, Pages 1206-1223, November 1998.

[30] J. Whittaker, *Creativity and Conformity in Science: Titles, Keywords, and Co-Word Analysis*, Social Science in Science, Volume 19, Pages 473-496, 1989.

[31] James L. Lamprecht, *ISO 9000: Implementation for Small Business*, ASQC Quality Press, 1996.

[32] ASYCUDA - Automated SYstem for CUstoms DAta. www.asycuda.org.

[33] United Nations Conference on Trade and Development. `www.unctad.org`.

[34] Annie I. Antón, *Goal-Based Requirements Analysis*, Second IEEE International Conference on Requirements Engineering (ICRE `96) , Colorado Springs, Colorado, pp. 136-144, 15-18 April 1996.

[35] CoWord analysis tool. COAST Laboratory (Purdue University), ACIS Laboratory (University of Florida), http://www.acis.ufl.edu/~ivan/coword