NASA/TM—1999-208898 AIAA–99–0248

Software Development Processes Applied to Computational Icing Simulation

Laurie H. Levinson, Mark G. Potapczuk, and Pamela A. Mellor
Lewis Research Center, Cleveland, Ohio

Prepared for the
37th Aerospace Sciences Meeting & Exhibit
sponsored by the American Institute of Aeronautics and Astronautics
Reno, Nevada, January 11–14, 1999

National Aeronautics and
Space Administration

Lewis Research Center

January 1999

Available from

NASA Center for Aerospace Information
7121 Standard Drive
Hanover, MD 21076
Price Code: A03

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22100
Price Code: A03

SOFTWARE DEVELOPMENT PROCESSES APPLIED TO COMPUTATIONAL ICING SIMULATION

Laurie H. Levinson
Mark G. Potapczuk
Pamela A. Mellor
NASA Lewis Research Center
Cleveland, Ohio

The development of computational icing simulation methods is making the transition from the research realm to commonplace use in design and certification efforts. As such, standards of code management, design, validation, and documentation must be adjusted to accommodate the increased expectations of the user community with respect to accuracy, reliability, capability, and usability. This paper discusses these concepts with regard to current and future icing simulation code development efforts as implemented by the Icing Branch of the NASA Lewis Research Center in collaboration with the NASA Lewis Engineering Design and Analysis Division. With the application of the techniques outlined in this paper, the LEWICE ice accretion code has become a more stable and reliable software product.

Introduction

The LEWICE ice accretion code has been supported and maintained by the Icing Branch of the NASA Lewis Research Center since the 1980's. The original code combined already existing elements, such as flow field and trajectory calculations, with new elements modeling the physics of ice growth to create an updated system capable of predicting the evolution of ice shapes on surfaces exposed to icing conditions.

Since this original icing simulation code was created almost twenty years ago, the system has undergone many changes.[1,2,3] Features have been added, "bugs" have been fixed, and adaptations to various hardware platforms have been included. Much of this evolution has been in response to requests from the user community for enhanced features, greater reliability, increased accuracy, and improved usability.

As the development of this code has become more user-oriented, however, the use of the code by the icing community has increased, and demands for even higher levels of reliability and accuracy have also increased. Most notably, the use of LEWICE as a substitute for or augmentation to experimental testing continues to be a desired goal of both the user community and the regulatory authorities.

In order to advance the code along the path toward acceptance, the Icing Branch of the NASA Lewis Research Center has embarked upon a rigorous program of software re-engineering. This effort is designed to make the code more robust and more easily managed, and to more quantitatively identify its capabilities over the range of operating conditions it is designed to simulate. In order to accomplish this task, however, it has been necessary to move beyond the icing research domain itself and to incorporate, in addition, software engineering knowledge and expertise, as provided by the NASA Lewis Engineering Design and Analysis Division.

The first step in this collaborative effort has been to gain an understanding of standard software development processes and techniques, and to carefully consider the implications of these processes and techniques relative to the icing research development environment.

A standard software development process, as used within many software-intensive disciplines, provides a structured approach to the creation of computer codes. This typically begins with the specification of the planned system capabilities - or "requirements" - and leads all the way through distribution and support of the final software product. This type of approach is generally more traceable than that used in a research environment, and leads to better documented code. However, it also requires a greater degree of oversight and record keeping during the development effort. Traditional methods for the development of research codes have generally followed a more unstructured, less formal process in order to allow maximum flexibility to the developer. In this more informal

environment, frequently recommended software development strategies for planning, documentation, version control, and distribution of the final product have not commonly been put into practice.

In the more formal environments established outside the research realm, the development process, although normally tailored to the specific needs of an individual project, generally takes the form illustrated in Figure 1.[4] The process begins with a basic description and evaluation of user needs, then moves to a definition of the planned system capabilities, specification of the system structure, implementation into code, testing of the code on several levels and, finally, distribution of the final product and associated documentation. Throughout the process, configuration management techniques are applied to ensure traceability of all changes that may be made over the course of the development effort.

When considering the implications of this more rigorous approach relative to the development of icing simulation software, one of the outcomes is to highlight the need for the specification of requirements. This is especially true with respect to the ability of the code to accurately predict the ice shape geometry being simulated. The ability to define requirements such as this clearly relies upon the existence of some measurable standard relative to which the requirements can be specified. At the present time, however, there are no well-established acceptance criteria for ice shape modeling. While several researchers have suggested methods of comparing computed ice shapes to measured ice shapes,[5,6] there is as yet no agreement on the best approach to use in order to perform this comparison. Additionally, in none of these cases has there been a determination of how these comparisons should be assessed.

Furthermore, ice shape is just one characteristic that could be evaluated with respect to an ice accretion code. Other outputs that could be evaluated include droplet impingement limits, collection efficiencies, and heat transfer distributions. Acceptance of the software is based, in part, upon satisfaction of requirements for parameters such as these, and it is not entirely clear, at this time, precisely how to define such requirements. Similarly, it is not clear what additional requirements are necessary in order to adequately specify the essential characteristics of the system.

This paper will outline some suggestions for addressing these and other issues through the use of standard software engineering principles as applied to the development and maintenance of the LEWICE software system.

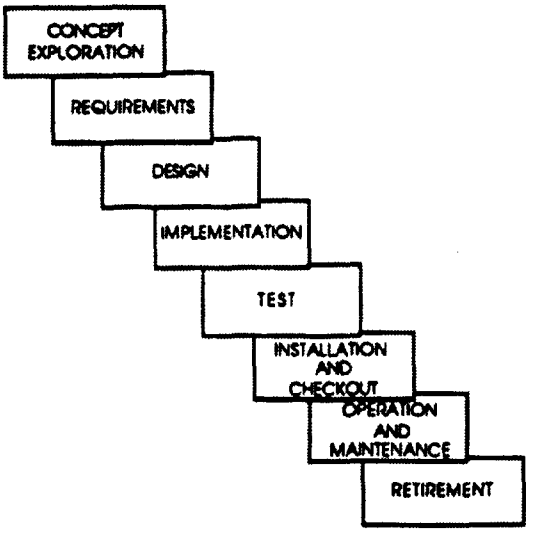

Figure 1. Typical Software Development Life Cycle

The Software Development Process

In order to better understand the issues involved in this type of software process improvement initiative, it is first necessary to understand the elements of a typical software development process. In general, when production software is created - where production software is software one might purchase and assume to be of high quality - the development process is commonly broken down into a sequence of stages through which the software progresses. Because these stages cover the entire range of activities that are to take place over the lifetime of the software system, they are, together, referred to as the software development life cycle. Within the software community, there are a variety of accepted ways to specify this life cycle. One typical formulation, the waterfall life cycle as defined by the Institute of Electrical and Electronics Engineers (IEEE), is shown in Figure 1. This basic life cycle serves to illustrate the process of creating a software system from its initial conception on. It is often referred to as an ideal life cycle since, in reality, there is rarely a clean demarcation between the various stages, or a purely sequential process, as is implied by the waterfall chart. Instead, the software system is frequently built in pieces, each of which follows the sequence of steps defined, although not all at the same time. Some portions may be better understood or may be considered more critical than others, and so may be built sooner. This incremental approach does not invalidate the waterfall model, but rather implies that there are many little waterfalls used instead.

To explain the various phases of the waterfall, the process begins with an examination and determination of what to do (concept exploration), followed by a further refining of those ideas into specific capabilities that are to be implemented (requirements). After these capabilities have been identified, the next step involves defining exactly how the required capabilities should be implemented (design). This design is then used to create the actual code for the system (implementation), which is subsequently checked for proper operation and corrected if and when errors are found (test). Once this activity has been successfully completed, the software is then placed in its normal operating environment and checked again for proper execution (installation and checkout). After this, the software is considered to be an operational system which may, over time, have a need for additional or revised capabilities, resulting in changes to the existing system (operation and maintenance). This operation and maintenance phase continues for the remaining time that the system is in use until, at the point where the software is deemed to be outdated or its function no longer needed, it is finally retired (retirement).

A more detailed view of this waterfall model is provided in Figure 2, below. As indicated in the figure, one of the artifacts produced during the requirements phase is referred to as a Requirements Specification. The purpose of this Specification is to document detailed information about all required software system capabilities. As such, it may well be the single most critical document produced over the entire course of a software project, as all downstream activities and decisions ultimately flow out of the information contained in this document. Among other things, the determination that must be made at the end of a project as to whether the completed system performs its intended function can only be made in reference to the system's specified requirements. While this may seem self-evident, it is nevertheless a major weakness of many software development projects.

Defining good requirements is actually quite difficult and time-consuming. According to the IEEE, requirements should be correct, unambiguous, complete, consistent, verifiable, modifiable, traceable, and ranked for importance and/or stability. They must address functionality, external interfaces, performance, desired quality attributes, and any mandated design constraints.[7] More often than not, however, vague desires of what the software might do are all that a programmer actually has to depend upon, and these may not even be documented. Frequently, the developer must make many design decisions based upon faulty information and assumptions and, as a result, may end up developing software that doesn't do what its users want at all. For these reasons, properly addressing the specification of requirements is an important step in the engineering of reliable, high quality software systems.

In the next phase of the waterfall model, the design phase, the primary output is a Design Document. The purpose of this document is to specify the intended software system structure and the associated relevant design information. The information provided can take the form of flowcharts or other pictorial representations of the system structure, and may also include items such as Program Design Language (PDL) specifications, which provide an English language description of an individual software module's logic. Whatever the form of the documentation, the activities performed and decisions made during this phase provide the foundation for the coding effort that is to take place during the implementation phase. Neglecting to perform the necessary activities and to properly address design issues generally results in code which is put together in an ad hoc fashion, and a system which is difficult to understand, test, and maintain.

As a consequence, any attempt to bypass this part of the development process will ultimately prove detrimental to both project schedule and product quality, although this fact may not be fully appreciated by those without the requisite software systems experience. The additional time required to code and test a poorly designed system, however, will quickly cancel out any short-term gains that may have been achieved by minimizing or eliminating the design phase. In addition, the generally lower quality of such a system will also have a significant impact during the operation and maintenance phase, which past experience indicates is where the majority of project time and money is typically spent. During that phase, the added cost incurred when required to implement changes to a poorly designed system will further diminish any temporary gains that may have been achieved by overlooking design in the early stages of the project.

Once the design phase has been completed, the code is then generated during the implementation phase, and subsequently checked for correctness during the test phase. At this point in the process, a common approach is to build small pieces of code, test these pieces individually, then gradually aggregate them into larger and larger pieces, which are each tested individually before being combined with other pieces and tested. Throughout the entire process, test plans, procedures, and results are documented and controlled, thereby providing not only a record of the verification process itself, but also providing the requisite traceability and repeatability in the event that errors are found and modifications to the modules become necessary.

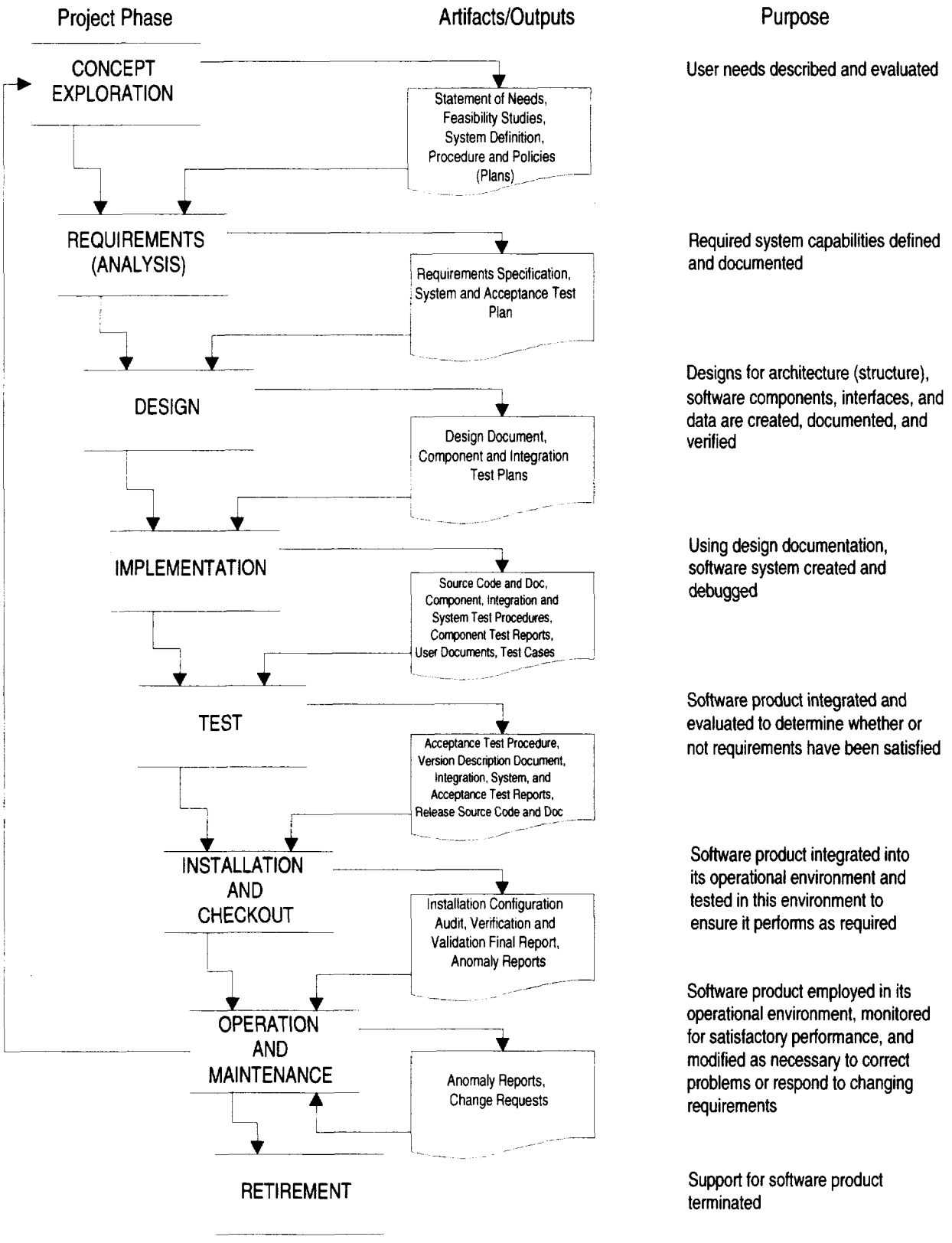

Figure 2. An "Ideal" Software Development Process

The need for such a high degree of rigor at this stage of the process can best be understood by consideration of the concept of regression testing. The IEEE defines regression testing as "selective re-testing of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements."[4] This definition helps to point out an unfortunate, and often overlooked, characteristic of software development: it is quite easy and, in fact, quite common to make an ostensibly simple change to a software module, and have that change cause unintended side effects (a.k.a., bugs) - sometimes in parts of the system far removed from the original location of the change. Therefore, in order to ensure that late changes do not adversely impact either the modified or the *unmodified* portions of a system, it is essential to have both a well thought-out approach to regression testing and the ability to repeat prior tests and reproduce prior results. The regression testing philosophy on a project, and the process established to implement this philosophy, are thus important elements of the software development process, and necessitate a certain level of planning and documentation for proper implementation.

With the testing phase complete, and the software successfully delivered and installed, the system is then considered to be in the operation and maintenance phase. During this phase, an existing system is modified either to correct problems that have been uncovered after delivery of the software or to make changes in response to new requirements. The desired changes are typically documented using anomaly reports and/or change requests, and these are then carefully reviewed prior to making any actual changes to the system's documentation or code. Although all changes made at this point are considered to be part of the operation and maintenance phase, the process used to implement these changes consists of essentially the same set of steps described above and used during the original development effort. Therefore, the operation and maintenance phase can actually be viewed as consisting of repeated implementations of the various stages of the waterfall model.

Software Process as Applied to Icing Simulation Development

Although the standard development process described above can be adapted for use in a variety of situations, there are certain characteristics of the general research environment that are not entirely consistent with this more rigorous approach. Likewise, there are also aspects of the icing simulation development domain that are not altogether compatible with the use of more formalized procedures. As with any improvement activity, therefore, when attempting to apply standard techniques to a specific environment in an effort to evolve toward a more disciplined approach, it is important to factor in both the distinguishing characteristics of the environment and their consequences for software development. If careful consideration is not given to the unique qualities and standard operating procedures of the organization, it is likely that the changes made either will not address the most critical concerns or will not prove to be a good fit for the organization. If this occurs, new approaches implemented will be unlikely to produce the desired results, as they will most likely either be unrelated to the main issues of the organization or so foreign as to eventually be discarded in favor of the original, more familiar, practices. In either case, the improvement initiative will have failed.

The Research Development Environment

In many research-oriented organizations, an emphasis is placed on individuality, creativity, and flexibility in order to provide an environment where new ideas will flourish and unique approaches can easily be pursued. In this type of environment, an individual researcher often works independently, pursuing a particular area of interest at the exclusion of other areas. Frequently, the researcher's reputation and perceived value to the organization are based exclusively on the unique knowledge and expertise that he possesses.

When the work performed in an environment such as this includes the development of analytical software for an external user community, however, the emphasis on individuality, creativity, and flexibility, which is so beneficial from a purely scientific stand-point, has some distinct drawbacks. The reasons have to do, in large part, with the basic nature of software.

Software systems are among the most complex systems ever conceived, based upon principles of human logic rather than on the natural sciences. Because of this, the results obtained when building a software system are especially dependent upon the skills of those doing the development. If, as is often the case in a research environment, only one individual is responsible for and knowledgeable about this development, then the quality and functionality of the system will be based entirely upon the capabilities and knowledge of that individual. This would represent a significant risk even in the best of circumstances; however, in the research development environment, where "research" is generally the first concern, and "development" a distant second, software systems are likely to be regarded as simply a tool of the researcher, and the methods used to develop them given little consideration. Non-software products (consider, for

example, a bridge or a car) developed without concern for the process used would commonly be expected to be of poor quality, and a similar expectation in the case of software is certainly not without justification. In fact, on the contrary, the inherent complexity and logic-based nature of software suggest the likelihood of even more serious consequences in that event.

The potential risk relative to both software system content and quality, however, is further exacerbated by the fact that, in this environment, very little about the software tends to be documented. This is most likely a consequence of both the lack of emphasis placed on standard software development methods, as described above, and the generally individualistic nature of the research environment. In this type of environment, where one individual is solely responsible for a system and has the freedom to make all decisions unilaterally, the need to document information in order to communicate it to others is minimized. With this being the case, critical information about the system is often contained only in the mind of the developer. This not only puts the organization at risk in the event that this uniquely informed individual should happen to leave the organization - taking all of his knowledge and expertise with him - but it also has other less drastic, although nevertheless significant, implications.

One such implication is that, when basic information about the system is not documented - e.g., system capabilities, specific approaches selected or rejected, the rationale for various decisions, etc. - it becomes very difficult, if not impossible, for other members of the staff or the general user community to review this information. Furthermore, without a structured process for critically reviewing implementation decisions and providing feedback to the developer, any external input that is provided will necessarily be provided in an informal, after-the-fact fashion. Often, this will be after the completed system has already been delivered to the users. Any changes made at that point will then most likely be incorporated into the system using the same unstructured process. This is especially true in a research environment, where the developer is unlikely to be particularly familiar with or concerned about software system design concepts. Although a system developed using this type of undocumented, patchwork approach may well have certain good features, it is not likely to be a cohesive, high quality system - and whatever qualities it does have, "only the developer knows for sure!"

Another inevitable consequence of this individualistic, informal approach to software development is that, when only one individual has the appropriate knowledge to be able to work on a system, the timing and content of any release of that system are necessarily tied to that individual's schedule. Therefore, in contrast to a well-managed production software project, which has pre-established deliverables and milestones and uses an appropriately-sized team to meet those objectives, the typical analytical software system produced in a research environment has deliverables and milestones that are based on the individual developer's schedule. In this situation, common practice is to simply release system updates as they are completed - although without associated documentation or controls in place to clearly distinguish between different versions, it is not possible to know with absolute certainty which updates are contained in which version. This can quickly lead to confusion within the user community, and potentially even for the researcher himself, resulting in the classic "quality control problem."

The Icing Simulation Development Environment

Although the issues associated with the general research environment described above apply to the icing simulation environment as well, there are additional issues specific to the icing domain that must also be considered. These issues essentially reflect the state-of-the-art of icing simulation code development which, at the present time, is particularly lacking in two fundamental, interconnected areas - determination of system requirements and specification of system acceptance criteria. The deficiencies in these areas of the icing simulation development process are, at least in part, due to three basic difficulties associated with the modeling process for icing phenomena:

(1) Uncertainty in understanding of the physical processes

Although this is not an area of difficulty that is unique to the modeling of icing phenomena, it nevertheless has significant consequences with respect to both determining system requirements and establishing system acceptance criteria. The reason for this is straightforward. Logically, if one does not understand the basic phenomena being modeled, it will be difficult to develop an accurate model; and if a detailed, accurate model cannot be developed, implementing the model in code such that it accurately reflects reality will be even more difficult.

An example of this situation in the area of icing simulation code development is the coefficient for convective heat transfer over a rough surface, which plays a critical role in the determination of the amount of incoming supercooled water that changes to ice during a given time increment. Unfortunately, information indicating precisely what this coefficient should be is limited, and thus the determination of requirements and the creation of appropriate testing procedures to

verify this aspect of the code are especially difficult.

(2) *Inability to obtain information for testing of numerical models of the physical processes*

In addition to uncertainties in the creation of physical models, there can arise difficulties associated with testing the accuracy of those models resulting from a lack of available data. This can be due to deficiencies in measurement capabilities or to a lack of resources necessary to obtain the appropriate information.

If data is unavailable for evaluation of a software system or its subsystems, then the acceptability of the software is based on the judgement of the developer. This tends to make that element of the software only as good as the experience of the developer. Additionally, there is then no way to assess the impact of that software element on the accuracy of the overall system. The developer is left to examine influences of subsystems on the overall product by running parametric studies. If the number of such subsystems is small, then there is a chance of determining how any given intermediate result should behave in a system that is operating correctly. If however, as is more commonly the case, the number of such unverified subsystems is large, then attempting to understand what result any one subsystem should be producing becomes impractical.

(3) *Complexities in the numerical representation of the physical processes*

Finally, even if the physical process being modeled is well understood and there is sufficient information available for comprehensive testing, there can arise difficulties associated with the transformation of the defining equations of the process into the numerical algorithm required to solve those equations. This is the typical focus of much of the work in the field of computational simulation of the field equations of continuum dynamics.

Typically, an equation is developed that represents the physical process of interest and this equation, or set of equations, cannot be solved through means of mathematical analysis. As such, numerical representations of the governing equations are developed which allow approximations of the solutions to be calculated. The accuracy of the approximations are dependent on the methods used to develop the numerical representations of the original equations, the terms in the equations that are neglected, and the methods used to solve the new numerical representations of the governing equations.

Depending on the nature of the numerical equations and on the methods used to solve them, more problems of the type described above in item (2) can arise due to a lack of knowledge with respect to intermediate results obtained during the course of the overall calculation procedure.

It is the goal of the research effort to address these three problems while creating a software system that is usable, reliable, and accurate. Unfortunately, the aspects of ice accretion computational simulation described above can lead to difficulty in utilizing ideal software development techniques to produce a system with these desired qualities. Given this, the inherent deficiencies in our understanding of icing phenomena can be addressed in two ways. The first is to factor these elements into the specification of appropriate requirements, and the second is to develop a process for continual improvement of the software in a controlled manner. With this in mind, the LEWICE development team has identified several goals for process improvement that will lead to a more managed software development environment and to a LEWICE code that has the desired quality attributes.

Improvement Initiative Goals

As previously indicated, in an effort to address the issues discussed above, the NASA Lewis Icing Branch in collaboration with the Engineering Design and Analysis Division has recently begun a software process improvement initiative. The intent of this initiative is to improve code quality, as well as the overall software development environment, by incorporating appropriate new approaches into the icing simulation development process. To that end, specific "process goals" have been established to guide the improvement effort. These process goals represent the means by which the ultimate end goal of high quality software produced within a controlled, predictable environment will be achieved.

For the reasons mentioned earlier, the process goals for this activity are based both on the unique attributes and priorities of the organization and on the standard recommendations of software process improvement practitioners. In the latter case, the primary source of information and guidance is the Capability Maturity Model for Software (CMM), a detailed roadmap for software process improvement developed by the Software Engineering Institute (SEI) at Carnegie Mellon University.[8]

Process Goal 1: "Institutional Knowledge"

One of the fundamental concepts expressed in the CMM is the idea that, only with appropriate documentation, is it possible for an organization to repeat or improve upon past successes. Otherwise,

the success or failure of any particular project is based entirely upon the individual skills and dedication of those who happen to be involved in the project. This latter situation represents "individual knowledge," as contrasted with "institutional knowledge," which is knowledge that is available to any member of an organization and which outlives any given individual's association with that organization. The required documentation is twofold: first is the technical information such as system requirements and design, and second is the documentation of project plans and procedures. Explicitly defining both types of documentation not only provides the basis for process improvement, but is also a key element of any team-based activity, since these documents specify the information necessary for coordination amongst team members. It is imperative, however, not to neglect any facet of this documentation, as to do so would ultimately impair the organization's ability to repeat successful practices or implement relevant improvements.

Process Goal 2: Use of a Team-Based Approach

The use of a team-based approach for software development provides several advantages over the typical individualistic approach often used in a research environment. Some of these advantages are simply a consequence of the sharing of knowledge and workload that occurs with the use of a team. For example, from an organizational perspective, a team-based approach spares the organization the risk of relying on a single individual's ability and availability in order to complete a project. On the other hand, from the individual developer's perspective, this approach helps to prevent the stress and schedule pressure that result from having sole responsibility for the release of a software system.

An additional advantage of this approach is that, by having multiple individuals with diverse backgrounds and capabilities working on the same project, the overall system quality can be greatly enhanced, as each team member can bring unique knowledge and skills to the project.

Process Goal 3: Use of Software Formal Inspections

Software Formal Inspections are essentially well defined, structured meetings where project work products are reviewed in an effort to remove defects. They have, without exception, been found to enhance software system quality and reduce costs - especially when used often and early in a project. If handled appropriately, inspections can also help to ensure that desired capabilities are incorporated into a system, and that the best approaches for implementing and verifying those capabilities are used.

Process Goal 4: Preparation of a Flexible, Well Thought-Out, Well-Structured Design

Although a good design is a basic requirement of any high quality software system, the uncertainties that currently exist in the icing simulation modeling process, as well as the overall complexity of the process, make a well-planned, flexible design all the more critical. Given the continually changing state of knowledge within the icing research field, it is not sufficient to simply build a system that incorporates well-understood features currently desired by the user community. Instead, it is important to build a system that can also be easily updated to incorporate new capabilities and techniques without compromising the existing system. This is accomplished by taking the time to prepare a carefully thought-out, well-structured, flexible design, which allows for modifications to the system with a minimum of impact.

Process Goal 5: Application of Rigorous Software Management Practices

In addition to the critical importance of documentation, another fundamental principle expressed in the CMM is the concept that, without management discipline on a project, any engineering improvements will likely be sacrificed to schedule and/or cost pressures at the first sign of trouble. For this reason, it is important to tackle management issues early in a process improvement initiative if other changes are to endure.

The types of activities that must occur in order to bring management discipline to a project are described in the CMM as follows:

"...software managers for a project track software costs, schedules, and functionality; problems in meeting commitments are identified when they arise. Software requirements and the work products developed to satisfy them are baselined, and their integrity is controlled. Software project standards are defined, and the organization ensures they are faithfully followed. The software project works with its subcontractors, if any, to establish a strong customer-supplier relationship."[8]

Process Goal 6: Implementation of Pilot Activities and Application of Lessons Learned to Future Icing Branch Projects

During the first phase of this initiative, establishing appropriate processes for use during the development of the LEWICE software, and improving the LEWICE code quality, are without question important objectives with intrinsic value of their own. However, from the standpoint of the process improvement initiative, these activities are also being

used to gain experience in implementing this type of effort in an icing simulation development environment. The intent is to then make use of this experience and, in the future, apply the lessons learned to similar activities on the remaining Icing Branch projects.

The LEWICE Pilot Project

Based on the goals and principles outlined above, an in-house effort aimed at defining processes for software management and development was begun. Using the existing version of LEWICE as the starting point for a pilot project, a team was assembled to identify the most critical areas to tackle and the specific processes that were needed. The team consisted of icing code developers and software management experts, in order to incorporate knowledge of the software, goals of the developers for software improvement, and an understanding of the processes required to achieve those goals.

Since the effort began with an existing version of LEWICE, it is important to note that this activity actually began in the maintenance phase of the software life cycle. However, the original code was developed without the use of rigorous software principles as described above, and hence most of the artifacts that would normally have been created along the way did not exist. With this being the case, it would have been an overwhelming task to attempt to re-create all of the missing documentation at once during the early stages of the activity. In addition, as the software was not developed using the standard approach described above, there were also other changes that could be made to improve the quality of the code itself. Therefore, priorities had to be established which balanced all possible efforts against the existing time constraints for the project. To do this, it was necessary to consider, once again, the overall life cycle and the various artifacts typically associated with each phase as compared with the current state of the LEWICE software.

As was described previously (see Figures 1 and 2, above), the standard software development life cycle begins with concept exploration, which is then refined during the next phase into system requirements. In the case of LEWICE, although requirements for the software had obviously been determined to some level, as code had been created, these requirements were never documented. For the current activity, although the team decided that it would be necessary to document requirements prior to formal testing, it was determined that this explicit definition of requirements should be delayed initially in favor of focusing on the design and implementation aspects of the existing code. This approach could provide two major advantages. First, it would result in more immediate benefits to the user community; and second, it would aid team members in obtaining a more thorough understanding of the code prior to the definition of requirements. It was also apparent that there was an urgent need for configuration management processes and tools, in order to help manage the release process and to control the proliferation of versions that had caused serious problems in the past. Therefore, this also was selected as an area to be addressed early in the project.

Based on the above decisions, work then began immediately on these high priority tasks. With respect to code design and implementation, an effort was made to restructure the code to make it more accessible to anyone who might be called upon to work with it. Specifically, elements such as common blocks, argument lists, read/write statements, namelists and declaration statements were modified to have a consistency in structure throughout all of the subroutines in the code. Prologs, or module headers, were created to provide standard information about each subroutine within the code listing itself. (The prolog template used to insert this information can be found in the appendix of this paper.) Additionally, mundane elements of the code such as subroutine names and variable names were altered to reflect a naming convention agreed to by the team. Finally, subroutines were made more modular and were restructured to be more logically connected in terms of their function within the operational flow of the code. As a result, the code has become better organized and in a condition more amenable to further development.

In conjunction with the above code modifications, a LEWICE Software Development Standards document was also created. This document currently consists of the LEWICE Coding Standard, the LEWICE Development Practices, and the LEWICE Automated Revision Control Procedure, although additional sections may be added, as required, over the course of the project. The first section of the document, the Coding Standard, contains specifications for filling in module prologs, naming conventions for the various elements of the code, and other code-specific requirements. The Development Practices section contains general development guidelines established to provide a consistent programming philosophy and to ensure this consistency across all phases of the project and amongst all team members. The Automated Revision Control Procedure contains detailed instructions for use of the RCS automated revision control system, including information on the specific commands and directories to be used and the low-level processes required in order to access the system. The totality of this information is intended for use by all code developers in their daily efforts.

In addition to the Software Development Standards, a LEWICE Software Configuration Management Plan is also currently under development. This Plan contains the specifics on configuration management activities such as configuration identification, control, audit and review, and also details project configuration management responsibilities. The purpose of a configuration management plan is to define the programmatic aspects of how the software and its associated documentation are to be controlled, as well as how new versions are to be released. The Plan is intended for use by all project team members in the quest for overall control of the software development effort.

With work on the above tasks sufficiently underway, system testing issues were then addressed. Among the different types of testing approaches possible, one method of validating LEWICE results was to check them against experimental results obtained from actual tests run in the Lewis Icing Research Tunnel. This work has recently been conducted for a broad range of icing conditions and has been reported by Wright.[9] While this was a significant step in the process of validating simulation results with real test cases, it is not sufficient to confirm that the simulation code properly performs all of its intended functions.

In order to verify that the software is acceptable in this regard, it is necessary to generate a set of test cases that cover the basic software functions and which can be used in regression testing. This test suite must be developed in relation to actual requirements in order to verify proper operation of the code relative to the required functions. Before such a test suite can be developed for LEWICE, therefore, a comprehensive set of requirements must first be defined. Work on these requirements has now begun and will be one of the most significant accomplishments of this activity.

Process Improvement Project Status

For purposes of obtaining a better understanding of both the overall progress to date and the tasks that remain to be accomplished in the future, a comparison of the individual improvement initiative goals versus the specific activities either implemented, in progress, or planned is provided below.

The first goal, *institutional knowledge*, is concerned with the availability of relevant documentation, which includes system requirements and design as well as project plans and procedures. This goal is being addressed at the present time by documenting and implementing the LEWICE Software Development Standards and the LEWICE Software Configuration Management Plan, and by preparing the LEWICE Software Requirements Specification. In addition, once the Requirements Specification is complete, the LEWICE Software Test Plan and Test Procedures will be developed. This testing documentation will be used both for formal system testing and to ensure repeatability of results when future modifications to the LEWICE code are made. The entire set of documents will then form the foundation upon which future improvements to LEWICE can be built, and will also serve as examples for other software development projects within the Icing Branch. Furthermore, once the documented processes have been put into practice on an everyday basis, they will then be evaluated as to their good and bad points, and updated accordingly. In this way, both the product and the process used to prepare the product can sustain continuous improvement.

Progress toward the second goal, *use of a team-based approach*, is indirectly being made via the development of the Requirements Specification. Once an initial version of this document has been prepared, other knowledgeable individuals, whether at Lewis Research Center or in the broader icing research community, can then be involved in the refinement of requirements and design approaches. This goal thus leads directly to the third goal, *use of Software Formal Inspections*. Although formal inspections are just one means of obtaining input, by using a highly structured technique such as this, one can facilitate discussions on features and implementation approaches. However, as with any team-based activity, such formalized discussions are only possible after preparing the necessary documentation. For the LEWICE project, current plans indicate that requirements documentation should be available in the very near future. Therefore, to further the above goals, a training session on Software Formal Inspections has been planned for early 1999. Following the training, formal inspections can then be held to review portions of this requirements documentation, as appropriate.

In addition to the need for documented technical information, however, meaningful progress toward team-based activities also requires a certain level of control and standardization that can only be achieved with the implementation of rigorous management and development processes. Without the coordination of information and activities amongst team members that a well-structured environment provides, attempts to implement a team-based approach or to utilize Software Formal Inspections will not achieve the benefits that would otherwise be expected. Therefore, implementation of the LEWICE Software Development Standards and the LEWICE Software Configuration Management Plan are important steps

in the effort to move toward a team-based approach. In the future, integration of rigorous requirements management practices into the process will further enhance the outcome of this approach.

The fourth goal, *preparation of a flexible, well thought-out, well-structured design,* has been addressed in a preliminary fashion by providing icing researchers with introductory experience in two commonly utilized design techniques: data flow modeling and PDL-based module specification.

The first technique, data flow modeling, is a graphical method used to depict the flow of data through a software system. This technique is particularly helpful in identifying how best to subdivide a system into cohesive components, and can be used to aid in the requirements specification and/or system design process. In the case of the existing LEWICE system, top level data flow diagrams have been developed (see Figures 3 and 4) as a means of defining the appropriate structure to be used in specifying current LEWICE system requirements. This activity has not only provided valuable experience with data flow modeling, but has also aided in the development of a well-organized Requirements Specification. In so doing, it has also helped to lay the foundation for future requirements definition and design specification activities on the project.

The second technique, PDL-based module specification, has also been utilized on a trial basis on the LEWICE project. This was done in order to gain experience in using the structured language approach to module specification and to evaluate the usefulness of this approach relative to future development efforts. For this initial effort, however, it was readily apparent that, to attempt to apply this technique to all of the existing subroutines of LEWICE would not only have been an immense job, but would also have been extremely difficult given the inherently unstructured nature of the modules. This being the case, as PDL is primarily a design technique, use of PDLs solely for the purpose of documenting existing modules was not felt to be a reasonable course of action.

All the same, structured design techniques will be applied during future Icing Branch development efforts. Initially, these will be efforts associated with modules being redone for future versions of LEWICE; later, they may be for modules associated with entirely new software development efforts. As of this point in time, a recreation of several LEWICE modules using the above techniques is already planned, and will provide project team members with additional opportunities to become familiar with these methods.

Progress on goal five, *application of rigorous software management practices,* has already begun with the enumeration of software standards, plans, and procedures in our current documentation. In the future, management practices will be supplemented by adding metrics and tracking activities to improve understanding of the development process and to be better able to create reasonable development schedules. In addition, an increased emphasis on the requirements process is planned. These activities will be ongoing throughout the development effort.

The sixth goal, *implementation of pilot activities and application of lessons learned to future Icing Branch projects,* has been addressed in part by virtue of the work accomplished to date on the LEWICE project. In the future, as controlled processes and planned activities are piloted on LEWICE, these new approaches will then be used as the basis for adopting the same or similar techniques on other Icing Branch projects. This process improvement activity is thus the beginning of a new way of doing business in a research environment, and can be seen as forging a new link between two groups who have traditionally worked in very different ways.

Concluding Remarks

This project ultimately has as its goal the development of a high quality, user-friendly, robust software system. The application of software management practices to an existing research analysis tool has led to interesting adaptations of these practices in order to work toward the project goal and utilize the resources and talents available within the desired time frame. One of the more important lessons learned from this effort so far has been that this is an ongoing process.

This paper has been written to indicate progress made to date and to suggest areas where further work is needed, both with respect to the current version of the LEWICE code and for future code development efforts. The first and most important outcome of this project has been to comprehend the implications of applying software management processes to the creation and development of research software systems and to identify how these processes can best be applied to the LEWICE ice accretion code. That is, the software development life cycle described above dictates specific activities that must be undertaken to develop, verify, and maintain any large software product. Our development team has employed this waterfall life cycle model to plan and undertake the efforts needed to improve the usability and reliability of the LEWICE code.

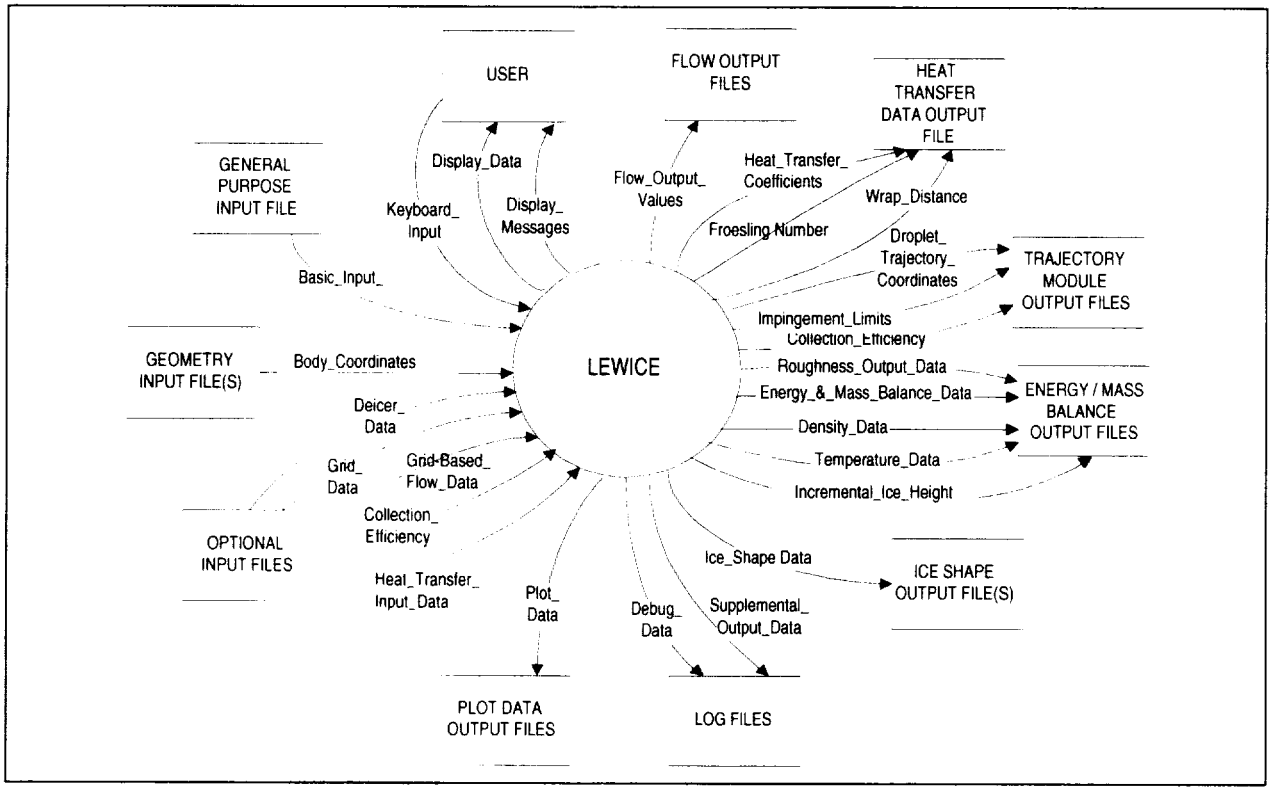

Figure 3. LEWICE Context Diagram

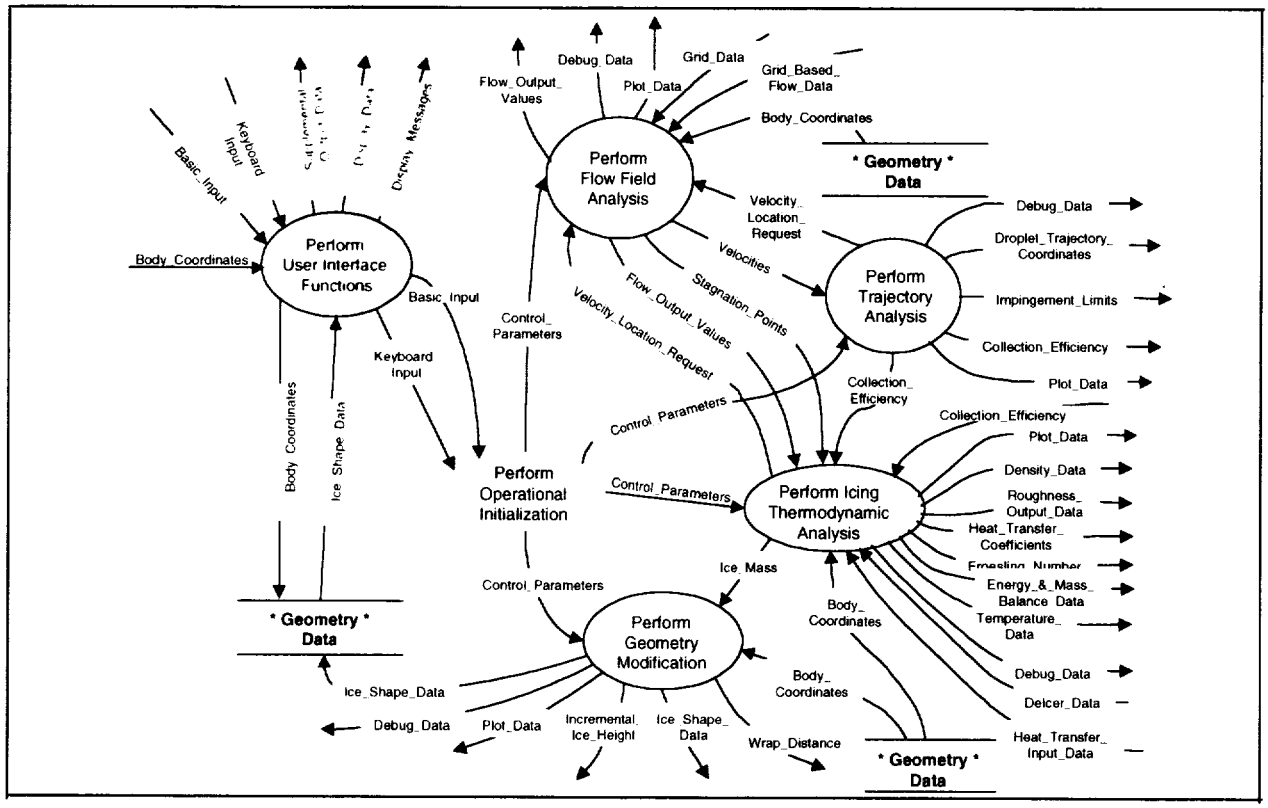

Figure 4. LEWICE Top Level Data Flow Diagram

To date the team has accomplished the following:

- Formulation of a LEWICE Software Development Standards document which consists of a Coding Standard, Development Practices, and Automated Revision Control Procedures. This document describes the current procedures to be used for construction and maintenance of the LEWICE software
- Development of a baseline version of the code which conforms to the majority of the Coding Standard provisions. This effort consisted of restructuring of subroutines, regrouping of common blocks, consistent implementation of naming and numbering schemes, enhanced modularization, and creation of prologs for each subroutine
- Testing of the baseline version for range capabilities and current ice shape modeling capability by comparison to a large database of experimental ice shape tracings
- Implementation of team-based planning and development efforts
- Implementation of an automated revision control system to archive and track versions of project work products, and to assist in the implementation of the project's overall configuration management process

Further work is needed and is indeed planned to apply all of the elements of the waterfall model to the LEWICE code. As such, the team is currently or will in the near future undertake the following activities.

- Development of a Software Configuration Management Plan which will contain specific information on configuration identification, control, audit, and review. The plan will also detail project configuration management responsibilities
- Preparation of a LEWICE Software Requirements Specification document to define the expected capabilities of the LEWICE software and to form the basis for formal system testing and future code modifications
- Implementation of Software Formal Inspections to aid in the review of project work products and, in particular, to ensure compliance of the software with specified requirements
- Preparation of a LEWICE Software Test Plan and Test Procedures to be used to verify that the LEWICE software meets all specified requirements
- Performance of formal system acceptance testing according to the documented procedures
- Implementation of changes to the LEWICE code as required to ensure that the released system satisfies all specified requirements and that the code conforms to all provisions of the Coding Standard

While there is still much to be done, the LEWICE software is currently more reliable and under better control than when this project was initiated. It is the expectation of the authors that the methods outlined in this paper will continue to be refined and applied and that the lessons learned from this pilot project will guide us in future software development projects. It is now clear that, with activities of this sort, the participants are embarking on a continual journey.

References

1. Ruff, G.A. and Berkowitz, B.M., "User's Manual for the NASA Lewis Ice Accretion Prediction Code (LEWICE)," NASA CR 185128, 1990.

2. Wright, W.B., "Update to the NASA Lewis Ice Accretion Code LEWICE," NASA CR 195387, Oct. 1994.

3. Wright, W.B., "User's Manual for the Improved NASA Lewis Ice Accretion Code LEWICE 1.6," NASA CR 198355, June 1995.

4. "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std 610.12-1990, IEEE Software Engineering Standards Collection, Sept. 1994.

5. Ruff, G.A., and Anderson, D.N., "Quantification of Ice Accretions for Icing Scaling Evaluations," AIAA Paper 98-0195, Jan. 1998.

6. Wright, W.B., and Potapczuk, M.G., "Comparison of LEWICE 1.6 and LEWICE/NS with IRT Experimental Data from Modern Airfoil Tests," NASA Contractor Report, Jan. 1997.

7. "IEEE Recommended Practice for Software Requirements Specifications," IEEE Std 830-1993, IEEE Software Engineering Standards Collection, Sept. 1994.

8. Paulk, Mark C. et al, "Capability Maturity Model for Software, Version 1.1," CMU/SEI-93-TR-24, Feb. 1994.

9. Wright, W.B. and Rutkowski, A., "Validation Results for LEWICE 2.0," to be published as a NASA CR, 1999.

Appendix

A prolog template, reproduced below, is being used to insert documentation directly into the LEWICE 2.0 source code listing. This standard header provides key information about each subroutine immediately prior to the code for that routine, and is included here to illustrate the approach being used consistently throughout the code to make the software more understandable and more manageable.

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C  module_name [- one-line description of module function]
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C  Purpose:
C
C
C  Input Data:
C
C  Output Data:
C
C  File:
C
C  Module Header:
C
C     executable subroutine/function statement
C
C  Argument List:
C
C    Argument Name      Type      Use             Description
C    -------------  ----------  ---  ---------------------------------
C
C
C
C  Global Variables:
C
C    Common Name   Variable Name      Type      Use       Description
C    -----------  ---------------  ----------  ---  ---------------------
C
C
C
C  Primary Local Variables:
C
C    Variable Name      Type                    Description
C    -------------  ----------  ----------------------------------------
C
C
C
C  External References:
C
C                       Module Name                          File
C    ------------------------------------------------  --------------
C
C
C
C  Invoking Routines:
C
C                       Module Name                          File
C    ------------------------------------------------  --------------
C
C
C  Development History:
C
C          Author            Date           Description of Change
C    --------------------  ----------  ---------------------------------
C                                     Original version
C
C Notes:
C
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1999	Technical Memorandum

4. TITLE AND SUBTITLE

Software Development Processes Applied to Computational Icing Simulation

5. FUNDING NUMBERS

WU-548-20-23-00

6. AUTHOR(S)

Laurie H. Levinson, Mark G. Potapczuk, and Pamela A. Mellor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E-11497

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM—1999-208898
AIAA-99-0248

11. SUPPLEMENTARY NOTES

Prepared for the 37th Aerospace Sciences Meeting & Exhibit sponsored by the American Institute of Aeronautics and Astronautics, Reno, Nevada, January 11-14, 1999. Responsible person, Laurie H. Levinson, organization code 7760, (216) 433-2663.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Categories: 02 and 61 Distribution: Nonstandard

This publication is available from the NASA Center for AeroSpace Information, (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

The development of computational icing simulation methods is making the transition from the research realm to commonplace use in design and certification efforts. As such, standards of code management, design, validation, and documentation must be adjusted to accommodate the increased expectations of the user community with respect to accuracy, reliability, capability, and usability. This paper discusses these concepts with regard to current and future icing simulation code development efforts as implemented by the Icing Branch of the NASA Lewis Research Center in collaboration with the NASA Lewis Engineering Design and Analysis Division. With the application of the techniques outlined in this paper, the LEWICE ice accretion code has become a more stable and reliable software product.

14. SUBJECT TERMS

Ice formation; Aircraft icing; Software engineering; Reverse engineering; Simulation

15. NUMBER OF PAGES

21

16. PRICE CODE

A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	