Success Arguments: Establishing Confidence in Software Development

Patrick J. Graydon and John C. Knight

Department of Computer Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740, USA
{graydon,knight}@cs.virginia.edu

Abstract. In this paper, we introduce success arguments. These rigorous arguments capture developers' rationales for believing that their software development efforts will succeed. We define success arguments, showing how the main success claim and the structure of a logical argument permit the developer to record and explain any form of evidence that would help to convince a skeptical audience that the effort will meet a balance of stakeholder goals that is acceptable to the stakeholders. We describe a notation for success arguments and discuss the role of success arguments in establishing confidence in the software development process. Using an example drawn from industrial experience, we illustrate how success arguments can be used to spot flaws in a given planned process. We also show how success arguments can be used to analyze existing software process models, illustrating our discussion with an analysis of the arguments underlying the Spiral model and Extreme Programming. Finally, we present some details of the engineering processes that surround success arguments, including procedures for deriving, amending, and verifying them.

1 Introduction

Unfortunately, the rate of failure of software development efforts remains stubbornly high. In this paper, we introduce the concept of a *success argument* for software development as a mechanism for addressing this situation. Constructed along with a planned development process and evolved as development proceeds, a success argument is a rigorous argument intended to convince a skeptical audience that a proposed software development effort will be successful. Using a graphic notation, the argument accumulates all the various facts, assumptions, and reasoning that underlie the software developer's process choices in a succinct form and exposes them to scrutiny. By doing so, the success argument can give stakeholders confidence that the developer's choices will result in a satisfactory system delivered on time and within budget. If such confidence does not arise, the planned process can be amended. The notion of a success argument is closely related to existing uses of argument in engineering such as safety arguments [2, 13].

A software development effort might have a wide variety of stakeholders including the individuals and organizations that fund the effort, the users who will use the resulting system, agents charged with ensuring compliance with applicable regulation, the developers themselves, the owners of systems that might be damaged or compromised if the system to be built fails, and the general public who might be affected if the system to be built fails in a way that endangers human life or the environment. These stakeholders must collectively be satisfied by a development effort if we are to call it a success. With this in mind, we define a *successful development effort* as follows:

Successful development effort: A development effort is a success if it terminates and demonstrably meets a balance of stakeholder goals that is acceptable to the stakeholders.

The stakeholders in any development effort will have a number of goals, possibly including goals regarding the time development will take, the cost of development, features of the service that the developed system will provide, and the dependability characteristics of the developed system such as safety and security. It is possible that these goals will conflict. The goal of safety, for example, might conflict (at least somewhat) with the goals of maximizing functionality and minimizing cost. Since we cannot insist that all systems meet all stakeholder goals, we must be prepared to accept systems that meet some balance of stakeholder goals. If the developers of a software-based system can justifiably claim that the system meets such a balance and the balance is acceptable to the stakeholders, then the developers can justifiably term the effort which produced the system a success.

The purpose of a success argument is to provide a mechanism for accumulating all the factors likely to affect success and to use the argument to anticipate likely sources of failure, where (obviously) *failure* is the opposite of success. Software development efforts fail for a wide variety of reasons. These reasons fall into the following four general classes:

- ***Deficient process.*** In some cases, the development process used is deficient or inappropriate for the effort. A developer strictly following the Waterfall process model, for example, might: (a) elaborate requirements; (b) draft a complete requirements document; (c) validate these through customer inspection; (d) complete a design and verify it; (e) code the system and test it; and (f) begin integration or system testing. At that point, the developer then finds that the implemented system has made the customer aware of a critical and heretofore unrecognized requirement, thereby invalidating a great deal of work. As others have pointed out, some kinds of systems, such as interactive end-user applications, are particularly vulnerable to this failure scenario [3].
- ***Deficient process instantiation.*** In other cases, the process is incorrectly or inexpertly applied. An inexperienced developer following the Spiral development model, for example, might fail to recognize a looming development risk of a type he or she has yet to encounter and go on to complete several cycles of the spiral, successfully identifying and mitigating other risks, before the unrecognized risk finally manifests itself and invalidates months of effort.
- ***Failure to affect constraints.*** In yet other cases, the software development effort might fail for reasons that have traditionally been considered outside the scope of a development process's control, but yet can be monitored, mitigated, and managed to some degree. A customer's requirements, for example, might change during the course of development. While no development process can control the customer's business practices or understanding of the problem to be solved, there are things that the developer can do to lessen the danger that such a requirements change will cause the effort to fail: user interface prototypes and executable prototypes can help the customer explore potential solutions, careful system partitioning of the system's internals can lessen the impact of certain kinds of changes, and so on.
- ***Independent constraints.*** Finally, a project can fail for reasons wholly outside of the developer's control. For example, the market might change drastically and unexpectedly thereby obviating the need for the system. Poor sales figures in one division of a company might lead to a project in another division being terminated or postponed as a cost-cutting measure, even if it was still likely that the project

would produce a system worth more than its cost. No activity that the developers could reasonably undertake would forestall such causes of failure.

The first three of these classes have been the subject of much research, and many valuable process techniques that help reduce the rate of software development failure have been developed. Similarly, many techniques exist that support the development of particular software artifacts. Combined, the number of techniques present developers with a new challenge, namely deciding which techniques to apply in a particular case. It is often difficult to recognize an erroneous selection and even more difficult to correct the situation once development is underway.

Success arguments are a novel approach to dealing with this situation. For a specific development, the associated success argument should be a compelling argument about why the developers believe that the planned development process, tools and techniques will lead to a successful development effort according to the definition above. The argument should be such that experienced independent reviewers as well as the various stakeholders find the argument compelling.

Naturally, in any development, the selection of the specific process, tools and techniques is not a purely random activity. Selections are made typically based on the experience and insights of those making the selection but with a fragmented, ad hoc and undocumented argument about why the selections are expected to work. By contrast, the success argument brings together *all* of the different aspects of the rationale for the selections and presents them in a way that enables examination and analysis by the different stakeholders. This does not guarantee success; even a compelling argument can be wrong. But we claim that the chances that this will be the case is far lower than would be the case with the present ad hoc approach.

In Section 2, we define success arguments and discuss their role in the software development process. In Section 3, we continue our discussion of success arguments by describing how the creation and maintenance of success arguments contributes to confidence in the software development activity. In Section 4, we describe a notation for recording success arguments, and in Section 5 we present an example success argument in that notation. In Section 6, we discuss how existing process models can be analyzed by recording and examining the implicit arguments which underlie them, illustrating our discussion with an analysis of the Spiral model and Extreme Programming. In Section 7 we elaborate upon the engineering processes surrounding success arguments, describing procedures for deriving, amending, and verifying the argument. We discuss related work in Section 8 and conclude in Section 9.

2 Success arguments and their role

Given the diverse ways in which a software development effort might fail, developers will need to take a variety of actions to try to forestall failure and ensure success. The success argument provides both the structure and the mechanism to record the rationale behind these actions. We define a success argument as a rigorous, logical argument supporting the following claim:

Main Success Claim: The development effort will lead to an adequate system in acceptable time and at acceptable cost.

What constitutes an *adequate system*, *acceptable time* and *acceptable cost* are system-specific. Developers will need to construct a defensible definition of each of these terms, keeping in mind the need to demonstrate acceptable trade-offs between conflicting stakeholder goals. It is of no use to argue convincingly that one will finish developing a system on time and under budget if that system does not demonstrably solve the problem that its stakeholders want solved. These definitions can and should be included explicitly in the success argument.

We note that, while the definitions of acceptable time and acceptable cost must be testable, they need not be quantifiable. One could define acceptable time, for example, not in terms of a specific delivery date, but in terms of the release to market of a competing system. This would make the strength of the argument at any time depend upon current market intelligence, but for some development efforts it may be the most straightforward statement possible of a key stakeholder need.

The remainder of a success argument supports the main success claim. In other words, it provides the rationale about why the main success claim should be considered true. The form of the success argument is a recursive structure in which claims are refined into sub-claims, argument strategies are introduced, and assumptions and planned evidence are documented. Details of the engineering of success arguments are presented in Section 7.

If the premises of a logical argument are true and its form is valid, then it is sound and its conclusions hold. Success arguments for real software development efforts are likely to contain assumptions, however, and these premises might not be true. Such assumptions, though, are made routinely and most importantly are often made *implicitly* by developers. By making these assumptions *explicit*, the success argument alerts the developer to their presence and exposes them to criticism by readers. Individual success arguments are also likely to contain errors in the form of fallacious reasoning. Again, such errors mar developer's implicit reasoning, but by making the form of the developer's rationale amenable to careful scrutiny, the success argument facilitates detection and correction of such errors.

The strength of the conclusion of an argument is limited by the strength of its premises. Some of the premises in a success argument will be assumptions, others will be observations about the system artifacts, the process being used or the problem that the system will solve, and yet others will be propositions about the effect of software engineering practices. We cannot have complete confidence in any of these. No matter how venerated the assumption, careful the observation, or scientifically supported the efficacy claim, there is always the possibility that it is wrong, even if the possibility is very slight. By showing how the premises conspire to support claims that particular stakeholder goals are satisfied, the written success argument affords the opportunity to judge whether the strength of each premise offered is sufficient given the consequences of not meeting each stakeholder goal.

The success argument includes all of the reasoning needed to support the claim that a specific software development effort will be successful. If a particular development process is to be used, the developers must cite their reasons for believing that its use will contribute to success. But developer care and competence in applying even the right process can threaten success, and so the success argument should include a sufficient reason to believe that the developers are doing as they claim and that they are sufficiently skilled to perform those tasks. Similarly, because things outside the developers' direct control can change in ways that could cause the project to fail, the argument should include the developers' reason for believing that the development activities monitor, mitigate, and manage such risks sufficiently.

Because the success argument is complete, and because it exposes all of the relevant rationale to careful scrutiny, it can give us much greater confidence that a planned software development effort will succeed than we would have without it. Software developers intend to be successful, and many plan their activities carefully so as to achieve success. Success arguments yield greater confidence that the development effort will be successful because they demand that this reasoning be complete, well-structured and written, making it subject to inspection that cannot be applied to piecemeal, ad hoc, or unstated rationales.

It is important to note that a success argument is not a recorded design rationale. The difference is not merely that design rationales are limited to design while success arguments focus on all areas of system development. A developer recording a rationale for a given system, whether it is a design rationale, testing rationale, or other kind of rationale, captures the choices that he or she has made and the reasoning supporting those choices for archival purposes. A developer creating and using a success argument does not expend effort recording information that is not directly relevant to the future success of that effort. Furthermore, because a success argument must be complete, a developer using a success argument knows that he or she has asked all the right questions. The recorded answers collectively support the main claim of future success, and any un-posed questions translate into missing support.

Because success arguments record the developers' reasons for believing that the development effort in progress will be successful, they by nature change throughout the project. As developers choose and implement specific development process steps, they gain evidence that can be added to the argument to strengthen it. Similarly, as the various activities undertaken lead to the creation of actual artifacts, the reasons for believing that the developers would be able to create those artifacts become moot and can be pruned from the argument. The argument should be updated periodically throughout development, and after each update the success argument should reflect as fact what has come to pass and show why the remaining effort will end in success. At the end of development the success argument vanishes: either the activity was successful or it was not.

3 Establishing confidence in software development

We have many examples of software systems having been built that failed to meet expectations in some significant way—late delivery, over budget, and so on—and these failures represent significant financial losses. A mechanism that could establish well-founded *confidence* in an anticipated software development effort, i.e., enable likely process deficiencies to be identified and corrected ahead of time, could reduce these failures and their associated costs.

In seeking such a mechanism, we define confidence in software development as follows:

> **Confidence in software development:** Assurance, to the degree practical, that a given software development effort will be successful.

We claim that success arguments are a mechanism for achieving this needed confidence.

Existing software process models cannot do this. The literature surrounding such process models supports the conclusion that efforts similar to those for which the process model was created tend to me more successful when the process model is followed, and this is certainly an important capability. It is not the mechanism that we seek, however. Process models do not make the establishment of confidence explicit, and they do not provide a structure within which the specific needs of a particular project can be addressed. In fact, process models do not clearly delineate their scope of applicability and so can be applied incorrectly and with negative results by the unwary.

Specific software engineering techniques are likewise insufficient. The use of formal methods, for example, may establish the truth of a specific claim such as the compliance of a given source program with a formally-stated specification, but this alone is only a *partial* basis for confidence. To have confidence that is warranted, developers need to know that *all* the relevant factors of the actual development process to be followed as well as all the techniques to be used will, with high probability, produce success in a given development effort. Without a *complete* success argument, there is no reason that a rational person should conclude that a software development effort using a given process model or technique will be successful. Development failure could be brought about by any one of a myriad of factors despite the benefits of the chosen process model or development technique.

Success arguments establish confidence in software development in several ways. First, at the level of individual software development efforts, the success argument both structures and documents the argument, including its assumptions. It explains how the individual assumptions and items of evidence combine to support the conclusion that the software development effort will be successful. Second, recording the argument exposes it to examination. A reviewer, any member of the development team or an independent expert, can easily see what assumptions were made and how claims are justified. This allows him or her to: (a) question whether claims are justified in a given development; and (b) check the argument for logical fallacies that would make it invalid. Finally, the argument structure also explains the evidence: it allows a reviewer to see which items of evidence support which specific sub-claims. Knowing how the evidence is used in an argument allows a developer to gauge whether the strength of an item of evidence is sufficient given the importance of the sub-claim it supports. Without this explanation, anybody reviewing a development plan might well have access to all of the evidence that would appear in a success argument for the same project *were one constructed* and yet have no feasible way of determining whether that evidence is sufficient to provide the necessary confidence.

The benefits of success arguments do not end with their effect on individual software development efforts. Careful scrutiny of the argument associated with the use of a particular software engineering technique facilitates the development of a precise statement of the benefits that the use of that technique brings. Equally important, it also reveals the propositions about a given development effort that must be true for the technique to yield those benefits, i.e., the success argument documents the circumstances under which a particular technique can be applied and therefore also when it cannot. This statement of applicability can be recorded as a success argument fragment, with the benefits stated as conclusions and the conditions stated as premises.

The availability of premises and conclusions for individual techniques facilitates a form of software engineering quite different from that practiced by many professionals today. Rather than choosing to employ a particular technique—such as adhering to an established process model—because it tends to be generally beneficial to software development efforts that are similar to the one at hand, the use of a success argument frees the engineers building a given system to chose a combination of techniques or variants of techniques that meet the specific needs of that system. If the argument fragments associated with those choices and the evidence to be drawn from the development effort can be combined into a convincing success argument, developers can justifiably have confidence in the software engineering efforts resulting from those choices.

4 Recording success arguments

Success arguments can be written in free-form natural-language text. However, researchers in the field of safety argumentation have noted that even carefully-crafted natural language arguments can confuse readers, making it difficult for them to quickly grasp the structure of the argument [12, 13]. This difficulty has prompted the development of graphical notations for the recording of arguments. The Goal Structuring Notation (GSN), for example, is intended to facilitate the clear communication of safety arguments [12, 13]. We record success arguments in a graphical notation drawn from the elements of GSN.

Figure 1 illustrates the elements of GSN used to record a success argument. Claims are represented as *goals*, and are rendered as rectangles. Information that provides a context for the argument is included or referenced in *context* nodes, rendered as rounded rectangles. *Strategy* nodes are rendered as parallelograms, and describe how the premises offered support the goal. Premises are rendered as circular *solution*

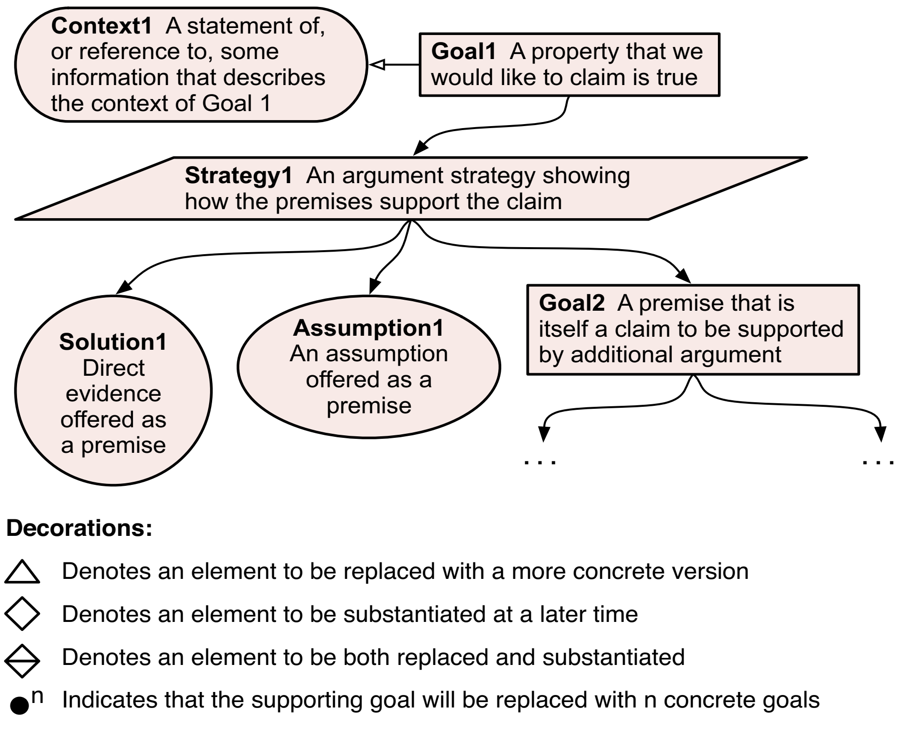

Context1 A statement of, or reference to, some information that describes the context of Goal 1
Goal1 A property that we would like to claim is true
Strategy1 An argument strategy showing how the premises support the claim
Solution1 Direct evidence offered as a premise
Assumption1 An assumption offered as a premise
Goal2 A premise that is itself a claim to be supported by additional argument
...
...
Decorations:
Denotes an element to be replaced with a more concrete version
Denotes an element to be substantiated at a later time
Denotes an element to be both replaced and substantiated
n Indicates that the supporting goal will be replaced with n concrete goals

Figure 1. Elements of a success argument

nodes in the case of direct evidence, oval *assumption* elements in the case of premises that are simply assumed to be true, or as goals when supported by further argument.

These elements of an argument are combined hierarchically. A goal at one level of the argument might be supported by one or more sub-goals that function as premises at that level and that are in turn supported by argument at the next lower level. A connecting line with a filled arrowhead may run from a goal or strategy to a sub-goal, strategy, solution, or assumption, indicating that the argument supporting the goal or described by the strategy draws support from the element to which it points. A connecting line with an empty arrowhead may run from a goal, strategy, solution, or assumption to a context element, indicating that the element from which the line originates *and any elements that support it in the argument structure* are to be taken in the context described or referenced in the pointed-to context element.

Graphic decorations indicate elements that will be changed. The triangle decoration, shown in Figure 1, indicates an element that will be replaced with a more concrete version at a later time. The diamond decoration indicates an element that requires further substantiation, such as a supporting argument in the case of a goal element or final delivery in the case of a solution element referring to evidence that has yet to be produced. These may be combined, in which case they are rendered as a diamond with a line through it. The multiplicity decoration, rendered as a filled circle annotated with a number, is attached to a connecting line to indicate that the pointed-to elements should be replaced by the indicated number of specific elements at a later time.

5 A development example and the use of argument

To illustrate the basic value and utility of success arguments, we present a historic example drawn from industrial experience, with names and certain details altered or omitted to protect the company and individuals involved. We describe a development activity that was far from successful and show how a success argument could have been used to reveal the difficulties that arose.

Jack was the software engineering manager for Alpha Software, a small software company. He and his small team had set about the task of creating the next version of Omega, Alpha Software's only product. Omega is a data processing application designed to run on a networked collection of commodity PCs. It includes features for capturing data, analyzing it using several different processor-intensive algorithms, and exporting it to a variety of commercial database systems.

The new release of Omega was intended to: (1) include some new data capture features requested by some of Alpha Software's key customers; (2) fix miscellaneous defects reported in the last release; (3) extend data export support to new versions of previously-supported database systems; and (4) add support for exporting to several new database formats. While discussing the desired changes with the sales, marketing, and support departments, Jack and his team created a work breakdown structure and a draft schedule for the work, with each developer providing input on tasks related to the features for which he or she was responsible in the previous version.

In practice he did not, but suppose that Jack had recorded his team's rationale for believing that a satisfactory new version of Omega would be ready by the planned delivery date given the available resources. To do so, he might have produced the argument shown in Figure 2. This argument is not an example of a

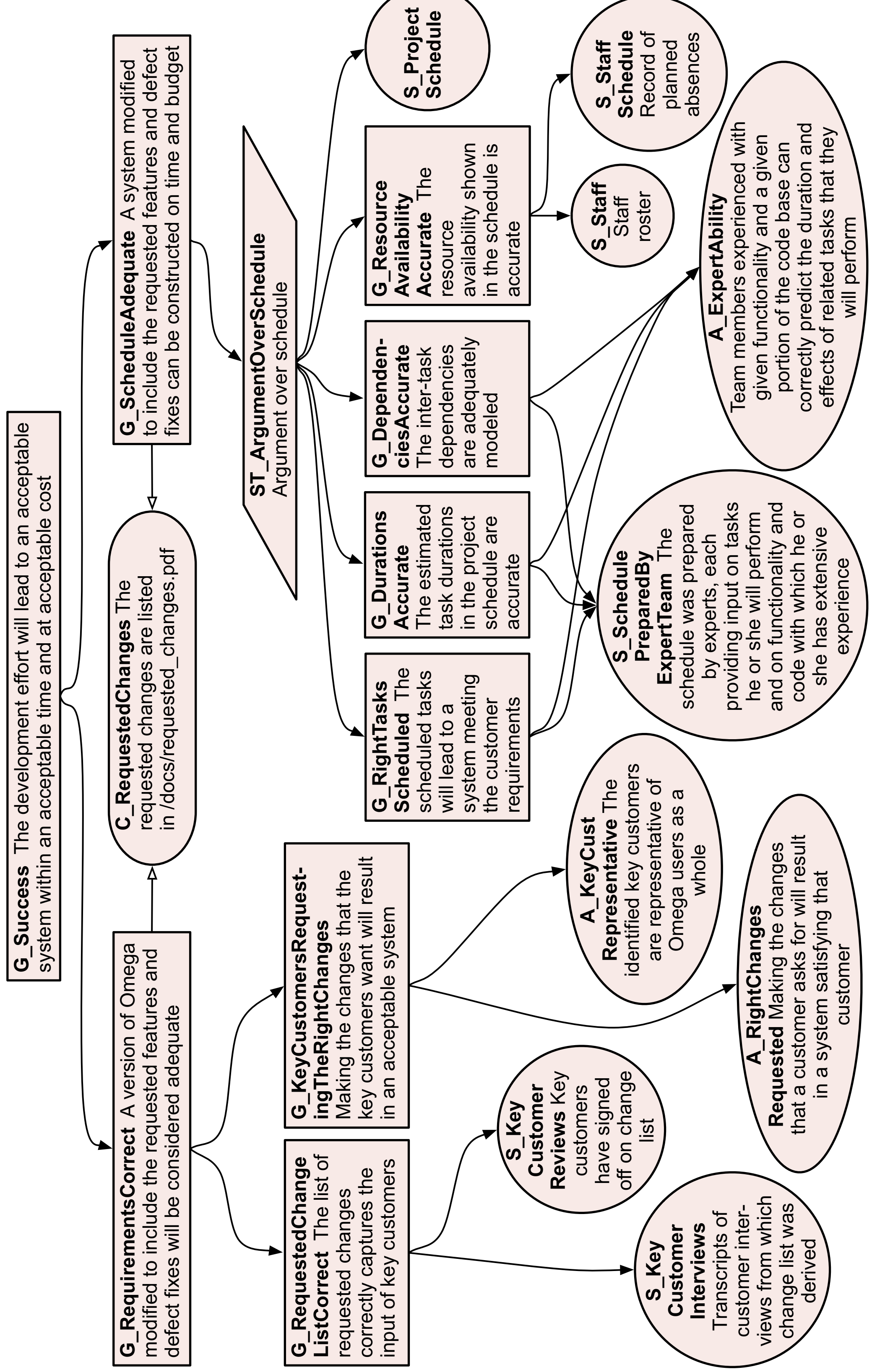

G_Success The development effort will lead to an acceptable system within an acceptable time and at acceptable cost
G_RequirementsCorrect A version of Omega modified to include the requested features and defect fixes will be considered adequate
C_RequestedChanges The requested changes are listed in /docs/requested_changes.pdf
G_ScheduleAdequate A system modified to include the requested features and defect fixes can be constructed on time and budget
G_RequestedChangeListCorrect The list of requested changes correctly captures the input of key customers
G_KeyCustomersRequestingTheRightChanges Making the changes that the key customers want will result in an acceptable system
ST_ArgumentOverSchedule Argument over schedule
G_RightTasksScheduled The scheduled tasks will lead to a system meeting the customer requirements
G_DurationsAccurate The estimated task durations in the project schedule are accurate
G_DependenciesAccurate The inter-task dependencies are adequately modeled
G_ResourceAvailabilityAccurate The resource availability shown in the schedule is accurate
S_ProjectSchedule
S_Staff Staff roster
S_StaffSchedule Record of planned absences
S_KeyCustomerInterviews Transcripts of customer interviews from which change list was derived
S_KeyCustomerReviews Key customers have signed off on change list
A_RightChangesRequested Making the changes that a customer asks for will result in a system satisfying that customer
A_KeyCustRepresentative The identified key customers are representative of Omega users as a whole
S_SchedulePreparedByExpertTeam The schedule was prepared by experts, each providing input on tasks he or she will perform and on functionality and code with which he or she has extensive experience
A_ExpertAbility Team members experienced with given functionality and a given portion of the code base can correctly predict the duration and effects of related tasks that they will perform

Figure 2. First draft of example success argument

good success argument. Rather, it is an example of the kind of rationale that, in our experience, underlies the actions of *many* software developers. Jack's rationale appealed to expert judgment, and this does give some support to the conclusion. But expert judgement is fallible. Experience with perhaps several successful product development cycles is often regarded as a sufficient basis for proclaiming the existing staff "expert", but that vastly underestimates the complexity of the underlying judgement process. A good success argument needs to be complete and compelling, and relying heavily on the judgement of what are usually self-proclaimed experts produces neither.

Although reliance on expert judgement is unreliable, a major benefit of documenting the rationale is clear in this case. All the stakeholders can be presented with the argument shown in Figure 2, and any of them might conceivably ask for more evidence. Whether or not more convincing evidence could have been obtained for this project, the argument allows the credibility of the engineers' rationale to be examined and gauged.

Alpha Software's senior management reviewed the schedule and found the 15-month schedule to be unacceptable: the sales and support departments were reluctant to make customers wait so long for the fixes and enhancements they had asked for, and the marketing department wanted the new product ready for launch by the start of an annual trade show in which the company traditionally has a marketing presence. At the same time, the developer who had been responsible for the data export features announced his intention to leave the company. Jack discussed the situation with senior management, and it was decided that the company should hire a new senior developer and two more interns. Jack adjusted the schedule, redistributing the tasks to take advantage of the new staff, and concluded that the 10-month timetable demanded by sales, support, and marketing concerns could be met.

If Jack and his team had created the argument shown in Figure 2 and attempted to update it at this point, they would have noticed that Jack's rescheduling invalidated the solution labeled **S_SchedulePreparedByExpertTeam**: tasks would no longer be performed by *the experts who estimated them*. This violation is crucial. The engineers estimating the tasks made their estimates with the assumption that they would be performing the estimated work. Delegating this work to another engineer almost certainly changed the effort required significantly, either because the engineer performing the work was less experienced and hence less productive, or because the needed changes required knowledge that the engineer performing the work had to rediscover or get from the original expert, or both. Arguably a scheduling mistake like this should not be made today, more than thirty years after the publication of *The Mythical Man-Month* [4]. In our experience, however, such mistakes do get made, and while creating and maintaining a success argument cannot force developers to use good engineering judgement, it can at least help to expose the fallacies in their rationales.

Returning to the scenario, in the revised schedule, the task of repairing and extending the data export features was delegated to one of the new interns, Jill. Jill had some previous experience using the universal database API that was used by the existing data export code. After a week spent getting settled into Alpha Software and learning about the Omega product generally, Jack assigned her the repair and extension tasks. He presented her with: (1) a list of the versions of database management systems to be supported; (2) an example record to demonstrate how a customer might use the export feature; (3) a list of the field types supported by Omega; and (4) the existing, undocumented source code. Jill considered the existing code

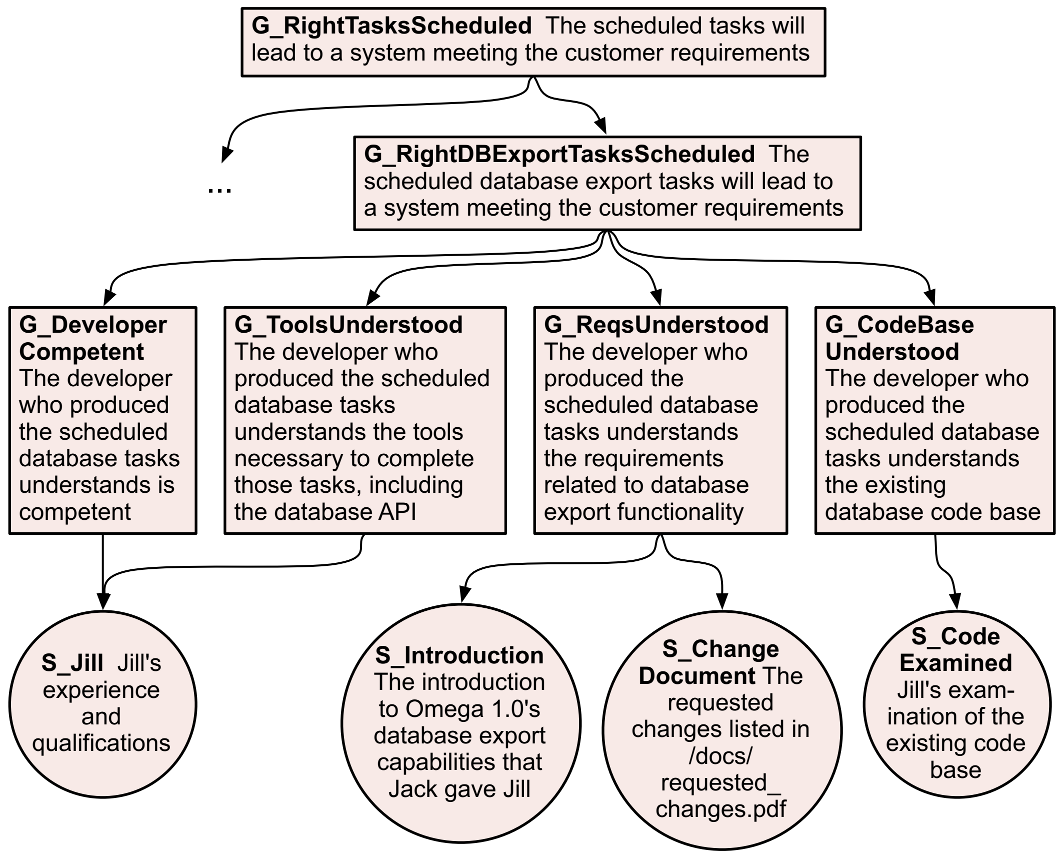

G_RightTasksScheduled The scheduled tasks will lead to a system meeting the customer requirements
...
G_RightDBExportTasksScheduled The scheduled database export tasks will lead to a system meeting the customer requirements
G_Developer Competent The developer who produced the scheduled database tasks understands is competent
G_ToolsUnderstood The developer who produced the scheduled database tasks understands the tools necessary to complete those tasks, including the database API
G_ReqsUnderstood The developer who produced the scheduled database tasks understands the requirements related to database export functionality
G_CodeBase Understood The developer who produced the scheduled database tasks understands the existing database code base
S_Jill Jill's experience and qualifications
S_Introduction The introduction to Omega 1.0's database export capabilities that Jack gave Jill
S_Change Document The requested changes listed in /docs/ requested_ changes.pdf
S_Code Examined Jill's examination of the existing code base

Figure 3. Fragment from revised example argument

and developed her own work breakdown structure for the task. Finding the existing code to be poorly structured, Jill suggested a ground-up rewrite of the data exporting code, which she predicted that she could complete in the time originally allocated for the modifications.

After a few months of development, during which Jill had produced builds of her code with progressively more functionality, the testing team completed development of a broader suite of test cases. Jill submitted her code for testing and was surprised to discover that it then failed an important test: it did not handle the arbitrarily nested tables permitted in Omega. Since Jill had assumed that each record to be exported was flat, like the sample she was given, her design was deficient, and her design and the code implementing it had to be revised at significant expense.

If Jack and Jill had updated the success argument to reflect their rationale in planning to have Jill rewrite the entire database export subsystem, they might have arrived at an argument including the fragment shown in Figure 3. The premises of the argument became even less convincing than before: why should a reasonable person believe that a simple, unstructured verbal explanation of the product's features, combined with the change list derived from customer input gave Jill a sufficiently accurate understanding of what the database export subsystem must do? Why should a reasonable person believe that an ad hoc reading of the existing code base was sufficient to allow a developer to form a reasonable opinion as to how it could be modified to meet the new requirements? Why should we not believe, instead, that the difficulty of reading and understanding another developer’s poorly structured and documented code led Jill to leap to the conclusion that it should be replaced rather than revised?

A written argument, by serving as a *complete* record of the developer’s rationale, can help us to avoid such poor practices. In addition, a reviewer reading the argument with a view toward extracting lessons learned could read the argument, observe that the requirements misunderstanding revealed during testing challenges goals **G_RightDBExportTasksScheduled** and **G_ReqsUnderstood**, and conclude that the requirements practices followed were inadequate. He or she might then suggest the preparation of a written set of requirements for the new version, design reviews, or any of a number of other software engineering techniques, with the resulting enhanced argument providing a means to gauge the improvement offered.

6 Analyzing established processes models with success arguments

In practice, all the actual processes used by software developers have an implicit albeit unrecognized success argument associated with them. These success arguments are often based on the developers’ experience: “We have built a system like this before, and we built it this way. Our effort was successful, so we will use the same process again.”

In some cases, developers take advantage of the experience and insights of others by using an established process model as the basis for the actual process they follow to build a particular system. Process models such as the Spiral model were created to address inadequacies in both purely ad hoc approaches and prior models such as the Waterfall model. By using the process model, the developer gains the ability to use the implied argument associated with the process model as support for his or her own implicit success argument.

Boehm argues that because developers following the Spiral model address risk explicitly, they accommodate one of the major causes of process failure and are thus more likely to experience a successful outcome than developers applying the Waterfall model [3]. In essence, the implicit success argument present in an actual process based on the Spiral model is more compelling than the argument associated with a process based on the Waterfall model. Recall that by *failure* here, we mean the opposite of success as defined in Section 1.

A much more thorough analysis of a process model can be conducted by developing the success argument explicitly for the model. That success argument, if it is compelling, essentially explains why the process model yields actual processes that tend to work. By examining the argument carefully, a thorough understanding of both the facets of the model that contribute to process success and those that contribute to process failure can be gained.

Examination of the success argument associated with a process model might identify shortcomings in that model. If a particular claim or assumption within the argument is found to be unsupported or unjustified, for example, we learn that, under some conditions, use of the process model might not engender success. The process model might then be modified to provide the needed evidence or avoid the unjustified assumption.

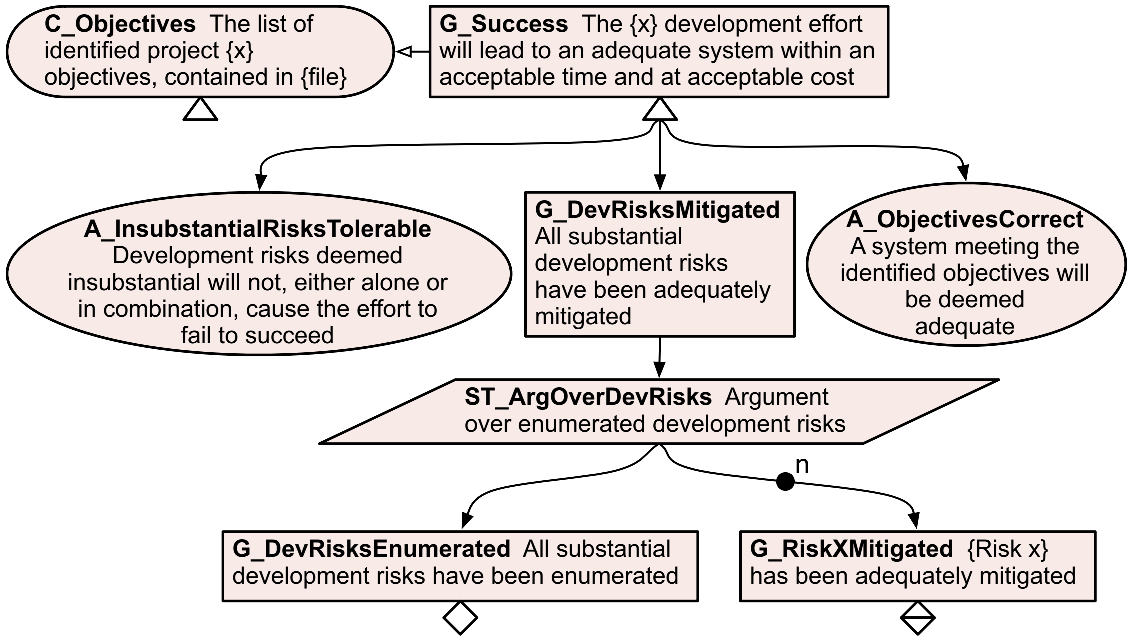
An alternative is to avoid the facets of the process model that contribute to failure in certain applications by restricting the use of the model to specific circumstances where the sources of failure cannot arise.

For example, some process models cannot be used effectively on large projects and others cannot be used in some application domains and so their use needs to be limited. These limitations are generally known but only vaguely, and developers might have difficulty determining whether a given process model is appropriate for the application at hand. By revealing the specific sub-goals that need to be supported by details from a specific application, analysis of the success argument associated with a process model can reveal precise applicability criteria.

In the remainder of this section, we illustrate this mechanism for analyzing process models by developing the key features of the success arguments for the Spiral model and for Extreme Programming. In each case, we show how the inherent features of the models imply the known model successes and identify elements of the models that are causes of development failures.

6.1 A success argument for the Spiral model

A developer using the Spiral model might reason that the development effort under way will be successful using an argument following the pattern shown in Figure 4. In the Spiral model [3], development proceeds in rounds, with developers alternately identifying areas of uncertainty that pose a significant risk to the successful completion of the project and selecting and implementing cost-effective strategies for resolving those risks. If, for example, developers believe at some point during development that one of the most significant development risks remaining is that the user interface might prove too cumbersome to support a critical use case, they may decide to perform user trials using a prototype user interface to address this risk. The developers would have to offer rationale for the claim that all substantial development risks have been enumerated, and, for each risk, rationale for the claim that the risk has been adequately mitigated.



C_Objectives The list of identified project {x} objectives, contained in {file}
G_Success The {x} development effort will lead to an adequate system within an acceptable time and at acceptable cost
A_InsubstantialRisksTolerable Development risks deemed insubstantial will not, either alone or in combination, cause the effort to fail to succeed
G_DevRisksMitigated All substantial development risks have been adequately mitigated
A_ObjectivesCorrect A system meeting the identified objectives will be deemed adequate
ST_ArgOverDevRisks Argument over enumerated development risks
n
G_DevRisksEnumerated All substantial development risks have been enumerated
G_RiskXMitigated {Risk x} has been adequately mitigated

Figure 4. Success argument pattern for the Spiral development model

The argument supporting **G_DevRisksEnumerated** would likely be a process argument. That is, the developer would argue that, given the process that was used to enumerate risks, the competence of the staff following it, etc., all substantial risks would have been enumerated. This argument would almost certainly fail to be *completely* convincing; even if developers had and used a well-developed list of common risks, we can always imagine a way for developers to overlook some risk that might be disastrous for the project. The developers must instead present an argument that is *convincing enough* given the consequences of a project failure. Note that this is the essence of what they do when questioned by managers, funding agencies, and the like; the difference is in a success argument they must do so in a well-structured, written argument.

The argument supporting each **G_RiskXMitigated** goal might take on any number of different forms depending upon how the developers elected to address that risk. If the developers elected to construct a throwaway prototype, they might use a success argument similar to the one shown in Figure 5.

This general form of this argument should be familiar to most developers, but the explicit argument representation makes an easily-overlooked obligation apparent: the developer must supply a reason to believe that the conclusions from the prototype will apply to the final system. Suppose, for example, that a developer is worried about his or her ability to build software that is demonstrably free of deadlocks. He or she might decide to use a particular technique, such as the use of an automatic model checker, to address this problem, and build a prototype to demonstrate the feasibility of the approach. If the prototype is constructed using a different programming language from that used in the final system, it is possible that the technique will not be applicable to the final system because no such tool is available for the chosen language.

6.2 A success argument for Extreme Programming

In Figure 6 and Figure 7, we present a success argument for Extreme Programming (XP) developed from Beck's *Extreme Programming Explained* [1]. That text explains the features of XP and some of the reason-

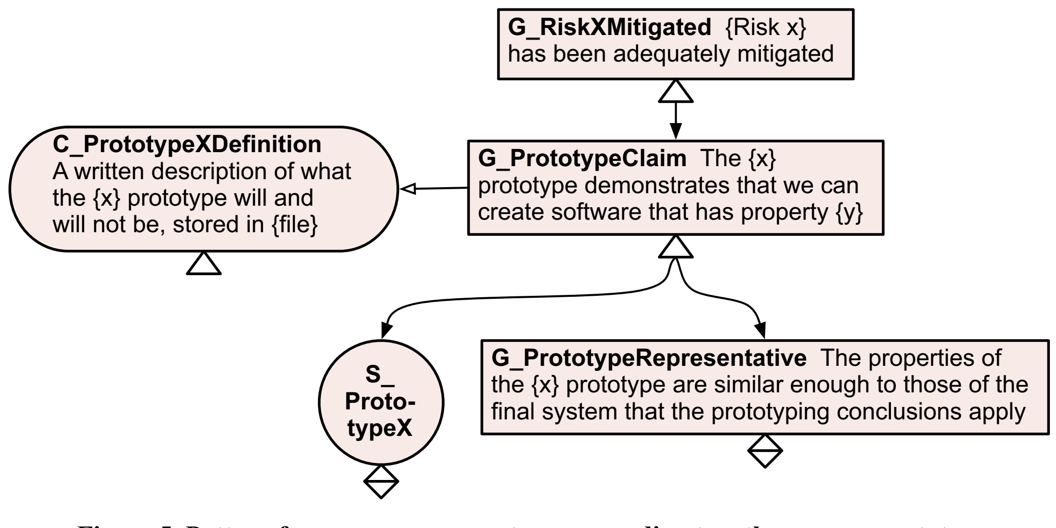

G_RiskXMitigated {Risk x} has been adequately mitigated
C_PrototypeXDefinition A written description of what the {x} prototype will and will not be, stored in {file}
G_PrototypeClaim The {x} prototype demonstrates that we can create software that has property {y}
S_ Proto-typeX
G_PrototypeRepresentative The properties of the {x} prototype are similar enough to those of the final system that the prototyping conclusions apply

Figure 5. Pattern for success argument corresponding to a throwaway prototype

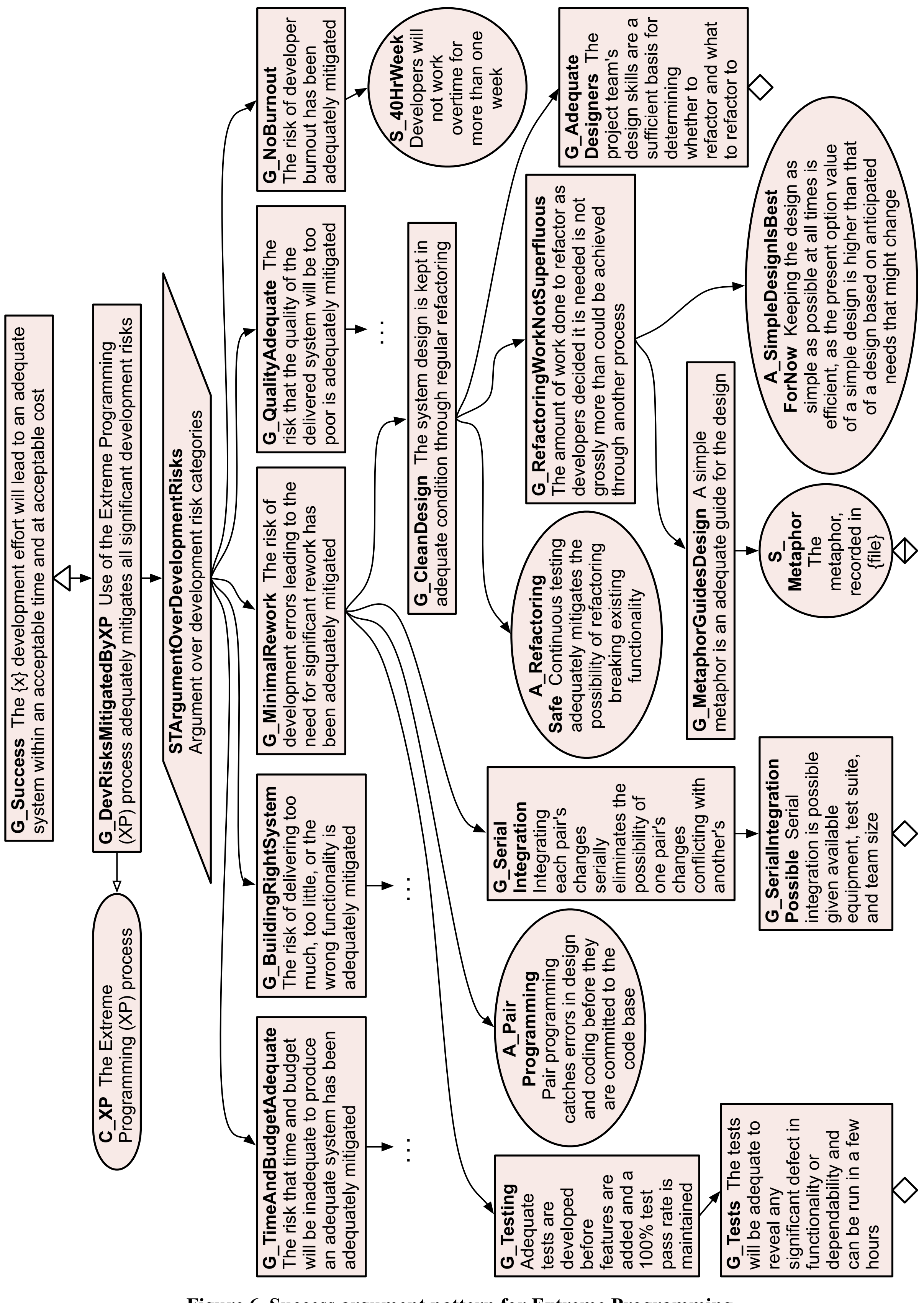

G_Success The {x} development effort will lead to an adequate system within an acceptable time and at acceptable cost
G_DevRisksMitigatedByXP Use of the Extreme Programming (XP) process adequately mitigates all significant development risks
C_XP The Extreme Programming (XP) process
STArgumentOverDevelopmentRisks Argument over development risk categories
G_TimeAndBudgetAdequate The risk that time and budget will be inadequate to produce an adequate system has been adequately mitigated
G_BuildingRightSystem The risk of delivering too much, too little, or the wrong functionality is adequately mitigated
G_MinimalRework The risk of development errors leading to the need for significant rework has been adequately mitigated
G_QualityAdequate The risk that the quality of the delivered system will be too poor is adequately mitigated
G_NoBurnout The risk of developer burnout has been adequately mitigated
S_40HrWeek Developers will not work overtime for more than one week
G_CleanDesign The system design is kept in adequate condition through regular refactoring
G_Testing Adequate tests are developed before features are added and a 100% test pass rate is maintained
A_Pair Programming Pair programming catches errors in design and coding before they are committed to the code base
G_Serial Integration Integrating each pair's changes serially eliminates the possibility of one pair's changes conflicting with another's
A_Refactoring Safe Continuous testing adequately mitigates the possibility of refactoring breaking existing functionality
G_RefactoringWorkNotSuperfluous The amount of work done to refactor as developers decided it is needed is not grossly more than could be achieved through another process
G_Adequate Designers The project team's design skills are a sufficient basis for determining whether to refactor and what to refactor to
G_Tests The tests will be adequate to reveal any significant defect in functionality or dependability and can be run in a few hours
G_SerialIntegration Possible Serial integration is possible given available equipment, test suite, and team size
G_MetaphorGuidesDesign A simple metaphor is an adequate guide for the design
S_ Metaphor The metaphor, recorded in {file}
A_SimpleDesignIsBest ForNow Keeping the design as simple as possible at all times is efficient, as the present option value of a simple design is higher than that of a design based on anticipated needs that might change
. . .
. . .
. . .

Figure 6. Success argument pattern for Extreme Programming

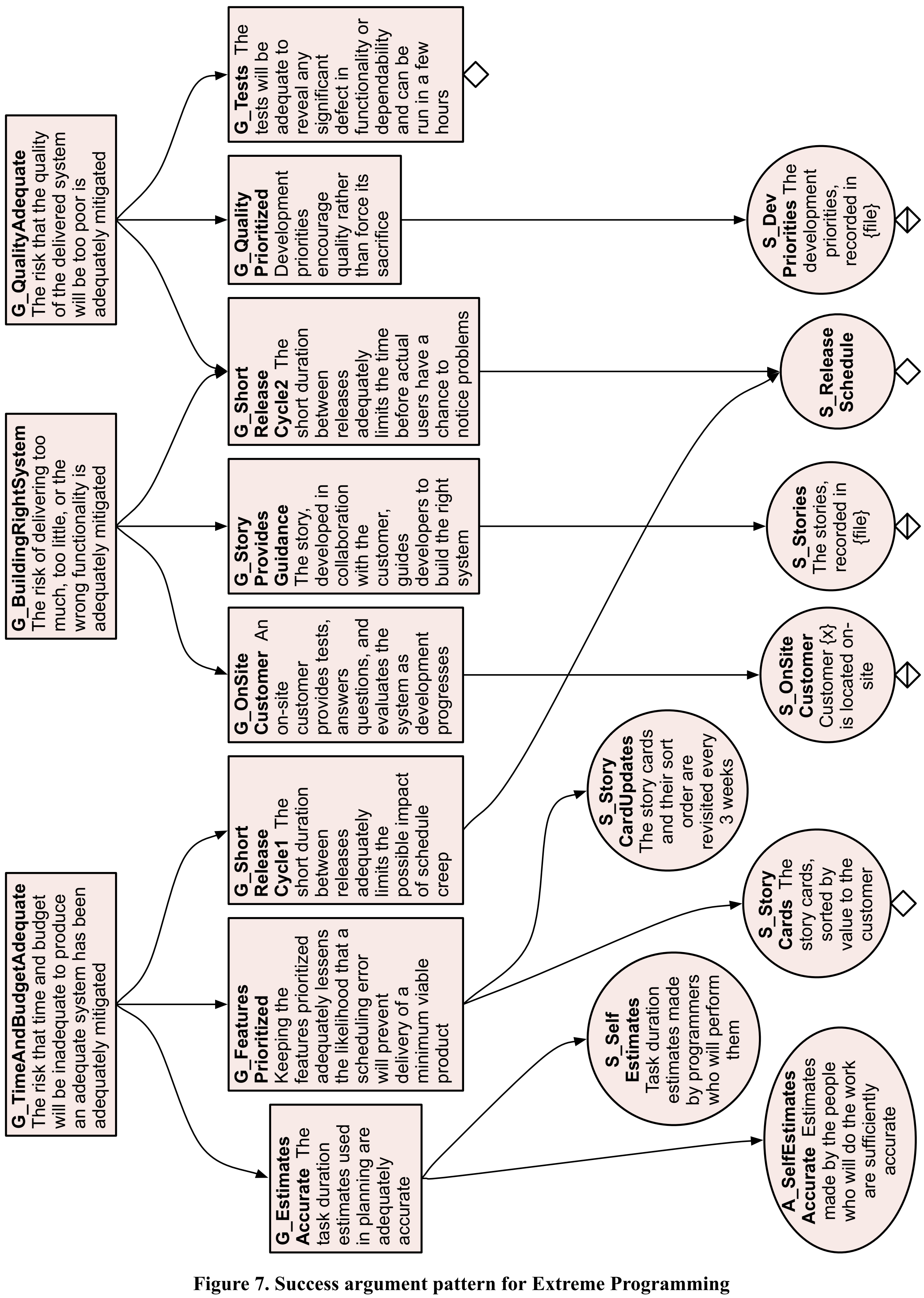

G_TimeAndBudgetAdequate The risk that time and budget will be inadequate to produce an adequate system has been adequately mitigated
G_BuildingRightSystem The risk of delivering too much, too little, or the wrong functionality is adequately mitigated
G_QualityAdequate The risk that the quality of the delivered system will be too poor is adequately mitigated
G_Estimates Accurate The task duration estimates used in planning are adequately accurate
G_Features Prioritized Keeping the features prioritized adequately lessens the likelihood that a scheduling error will prevent delivery of a minimum viable product
G_Short Release Cycle1 The short duration between releases adequately limits the possible impact of schedule creep
G_OnSite Customer An on-site customer provides tests, answers questions, and evaluates the system as development progresses
G_Story Provides Guidance The story, developed in collaboration with the customer, guides developers to build the right system
G_Short Release Cycle2 The short duration between releases adequately limits the time before actual users have a chance to notice problems
G_Quality Prioritized Development priorities encourage quality rather than force its sacrifice
G_Tests The tests will be adequate to reveal any significant defect in functionality or dependability and can be run in a few hours
A_SelfEstimates Accurate Estimates made by the people who will do the work are sufficiently accurate
S_Self Estimates Task duration estimates made by programmers who will perform them
S_Story Cards The story cards, sorted by value to the customer
S_Story CardUpdates The story cards and their sort order are revisited every 3 weeks
S_OnSite Customer Customer {x} is located on-site
S_Stories The stories, recorded in {file}
S_Release Schedule
S_Dev Priorities The development priorities, recorded in {file}

Figure 7. Success argument pattern for Extreme Programming

ing behind them rather than argue to a skeptical audience that the use of XP on certain kinds of systems by certain kinds of teams will always result in success. While adapting the text to the latter purpose and converting it into our graphical notation, we chose to simply collect into an argument the justifications for individual XP practices that we found in the text. We might have added nodes or made other modifications that are not part of Beck's original justification in order to make the argument clearer or more compelling, but we wanted to avoid editorializing to the degree practical. However, even our crudely translated argument can be analyzed to reveal two important characteristics of XP: the limited confidence that it supplies, and the situations in which its use is contraindicated.

The limited confidence that XP engenders can be seen in the success argument as follows. Unlike the success argument associated with the Spiral development model, the XP argument relies for support in large part on assumptions about the benefits that each element of XP provides. For example, in XP developers keep the design as simple as possible at any given moment rather than designing for anticipated needs. The reason that this practice is thought *not* to lead to excessive rework is a general argument based on an options cost model. Designing for the moment should be cheaper because: (a) it is often difficult or impossible to predict future needs correctly; and (b) the cost of building and maintaining support for the wrong needs is high.

Beck offers this rationale to support the practice generally, and so it is represented in our argument by the assumption that it holds for the development effort in progress. However, we cannot have as much confidence in the truth of the success claim in this argument as we might in an argument that relied more heavily upon evidence. This is because such assumptions cannot be examined the way that specific observations about a development effort and its artifacts can be. This limitation should be obvious to a reviewer attempting to determine whether the argument is sufficiently convincing. How much confidence would he or she place in **A_SimpleDesignBestForNow**, for example?

The situations in which the use of XP is contraindicated are revealed in the success argument as follows. If in the success argument for a given application an assumption is not acceptable or an unsupported sub-goal cannot be supported, the truth of the conclusion is threatened and so we should not use XP. Here are three examples where the use of XP is contraindicated:

- In situations where future needs are better known, such as situations in which the customer has supplied a detailed specification and there is some reason to believe that this specification will not change significantly over the course of the project, a reviewer might express little confidence in **A_SimpleDesignBestForNow**.
- In XP, pairs of programmers alternate between adding features as part of a task and integrating those changes into the shared code base, with integration serialized so that the changes made by one pair of programmers do not inadvertently break the functionality added by other pairs. This serialization forms a bottleneck, and so large teams will find it difficult to support **G_SerialIntegrationPossible**.
- XP also relies upon a comprehensive suite of tests, with developers determining whether their changes have properly completed a task by determining whether all tests in the test suite pass. If the system is safety-critical, for example, or if tests are expensive or require the use of hardware with limited availability, it might not be possible to create a test suite that would support **G_Tests**.

We do not claim that these limitations are unknown. However, they are revealed by analysis of the success argument, and other limitations might well be revealed when applying success arguments to individual development activities.

7 Success argument engineering

Success arguments are complex entities, and they are quite different from all the other artifacts with which software engineers are familiar. Since the notion of rigorous, documented argument is unfamiliar to most engineers, we present in this section the details of deriving, amending and verifying success arguments.

7.1 Deriving success arguments

Success arguments are similar in many ways to safety and other types of assurance arguments. As such, many of the necessary techniques for working with success arguments can be adapted from the existing literature. The ideas that we present here are, in part, refinements of those presented by Kelly for safety arguments [12].

The word "process" is tied closely to the notion of success arguments, and it is important that we be clear about what we mean when we use the word. To provide that clarity, we define a *planned process* as follows:

> **Planned process**: A planned process is a process that has been defined in response to the need to undertake a specific software development effort. A planned process might be defined to create a new product or to complete a partially completed product.

In the remainder of this paper, unless stated otherwise, the word process is taken to mean a planned process.

A success argument does not exist in isolation; it can only be *derived* from the planned process to which it applies. Thus, there is no notion of building a success argument, only deriving one as the associated process is developed. The development of the process itself is a highly creative and informal activity, and it involves a lot of trial and error with various ideas based on experience, capabilities, available resources, schedule, and so on. However, deriving the success argument during process development provides developers with a unique new capability, namely a process *assurance check*.

Even as it is being derived, the success argument must have various properties. For example, an argument fragment intended to support a subgoal must be locally compelling. If it is not, then it casts doubt on the associated process fragment. By examining the evolving, partially complete success argument, developers can assess the quality of the partially complete process and repair it (and the associated success argument) if necessary.

Using the success argument in this way during process creation does not obviate the need to examine the argument systematically and comprehensively when it is finally complete. The benefit of doing so lies in the fact that, if the completed success argument is sound, complete and compelling then the underlying complete process from which it was derived is very likely to be successful.

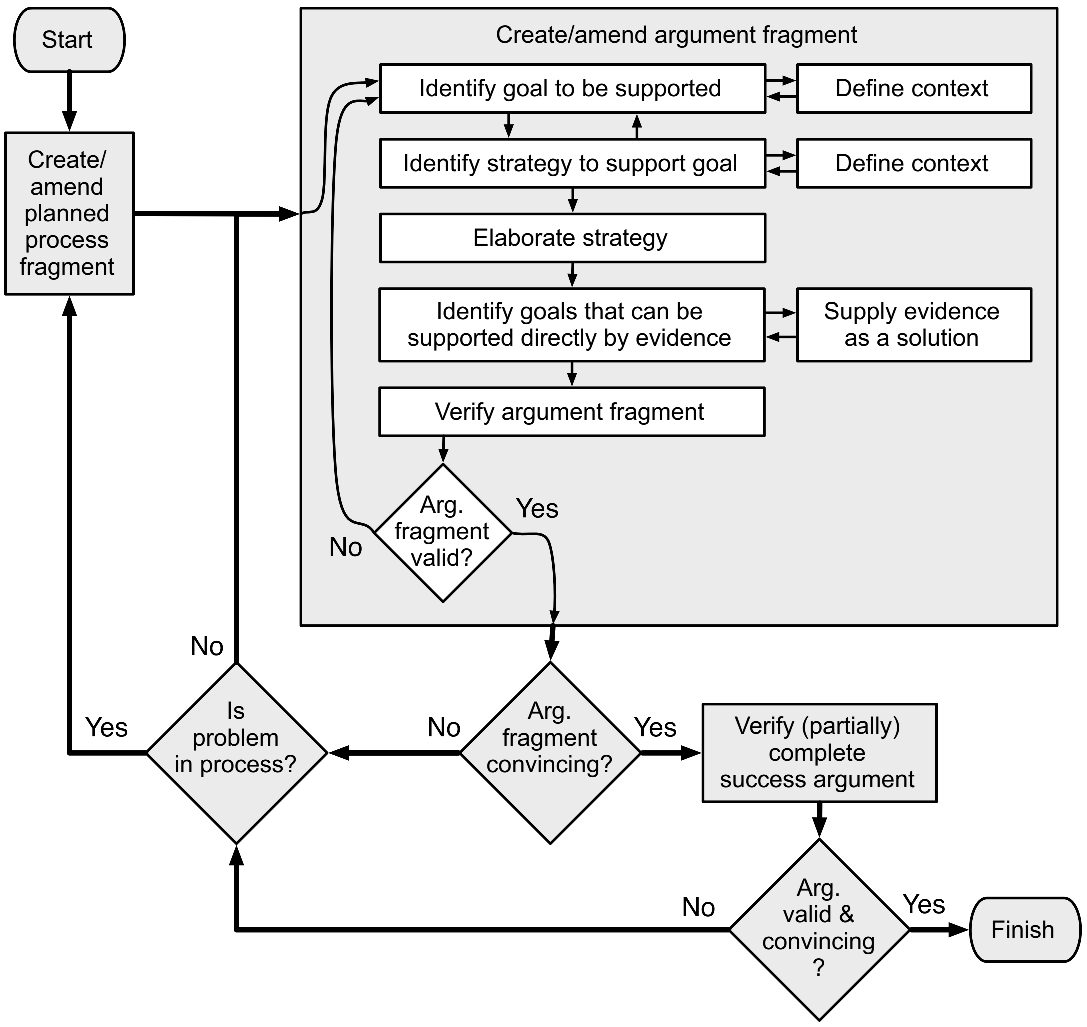

Start
Create/amend argument fragment
Identify goal to be supported
Define context
Identify strategy to support goal
Define context
Elaborate strategy
Identify goals that can be supported directly by evidence
Supply evidence as a solution
Verify argument fragment
Arg. fragment valid?
Yes
No
Create/ amend planned process fragment
No
Yes
Is problem in process?
No
Arg. fragment convincing?
Yes
Verify (partially) complete success argument
No
Arg. valid & convincing ?
Yes
Finish

Figure 8. Derivation of success argument

A detailed presentation of the derivation of a success argument is shown in Figure 8. The derivation begins with the development of the top-level goal of the success argument. This requires instantiation in the context for the top-level goal of the three critical terms, *adequate system*, *acceptable time* and *acceptable cost* as discussed in Section 2.

With these terms defined, a fragment of the process is created and the corresponding fragment of the success argument is derived. The size of the fragment will depend on the preferences and judgements of the developers involved. The creation of the process fragment can proceed in any manner deemed suitable by the developers. Derivation of the argument fragment, however, must be systematic because arguments have a precise structure. Fundamentally, argument derivation should follow the following six steps [12]:

- ***Identify the goal to be addressed.*** Success arguments are derived one fragment at a time, with each fragment supporting one goal.
- ***Define the context for the goal to be addressed.*** The meaning of the statement inside a goal bubble is often context-dependent; clarify the context as needed.

- ***Identify the strategy that will be used to support the goals.*** How will the goal be supported? Will the argument be over risks? Will the argument use multiple legs for additional support?
- ***Define the context for the strategy.*** If the strategy is of the form *argument over* <*x*>, a list of all <*x*>'s is necessary context and must be identified.
- ***Elaborate the strategy.*** What sub-goals are necessary to support the chosen goal given the chosen strategy?
- ***Identify goals that can be supported directly by evidence.*** Identify any of the new subgoals that can be supported directly by evidence, and add that evidence to the argument in the form of a solution.

With the process partially created, the argument fragment is examined and checked for two properties: (1) whether it is well formed; and (2) whether it is compelling as it stands. If the argument fragment is not well formed, then obviously the derivation needs to be revisited and the argument adjusted. If the argument fragment is well-formed but not compelling, then it means either: (a) that some part of the process fragment is inadequate and the process has to be revised; or (b) that the argument fragment is inadequate and it has to be revised.

The process is completed by repeating the above activity, incrementally increasing the size of the both the process and the argument fragments until both are considered complete. The final step is argument verification (see Section 7.4).

Most realistic processes cannot be created in their entirety at the outset of a project because later process steps depend upon earlier process results. Constructing a prototype, for example, is a common process step whose results will likely affect decisions about later process steps. Thus development has to proceed with an incomplete process.

Deriving a success argument has to accommodate this variability. This is straightforward in all circumstances except for the need to proceed when the process and therefore the success argument are not defined completely. Within the structure of the argument, this situation can be documented as an unelaborated goal. Proceeding, however, means that developers must be confident that a satisfactory elaboration will be possible later.

Proceeding with one or more unelaborated argument goals is a significant but inevitable risk in many developments. However, such risks can be mitigated significantly by using a success argument because they are pinpointed in the argument. Represented as unelaborated claims, they are clearly defined and very visible thereby providing the best chance that the risk will be addressed at the appropriate time.

7.2 Amending success arguments

A success argument reflects a moment in project time: at any point in a project, it documents why the developers believe *at that moment* that the software development effort *ahead* will be successful. As the project progresses, new decisions and activities will result in additional support for this claim, and claims about events in the past will become moot. Indeed, the entire success argument will eventually become moot: after a project terminates we know whether it was successful or not. Whatever reason we might have had for thinking that it would be successful is of historical interest only, so all that remains is for us to learn from whatever mistakes we might have made.

As development progresses, moot claims should be removed from the argument in order to minimize its size and complexity, to keep it sound, and to make it as compelling as possible. If the argument predicted that the developers would be able to achieve something and they did, then referring to that achievement as a historical fact offers more succinct and compelling support for other elements. If the prediction was incorrect, then the support for those other elements has collapsed, and additional argument and evidence must be brought in to offer the needed support.

It is helpful to draw lessons from whether arguments correctly predict what actually transpires. To facilitate doing so, it is worthwhile keeping a historical record of what the argument was at several points during development. But an up-to-date argument should contain no elements that refer to past activities in the future tense. The past activities have resulted in the present state of the software development effort and its artifacts, and that state is a fact that can be used as evidence.

Removing argument fragments that are associated with development successes is trivial. Dealing with elements of the argument that prove to be erroneous is quite the opposite. Any error in the argument, by definition, threatens the main success claim and any sub-claims between them; if the invalid portion of the argument is not superfluous and cannot be replaced with sound support for the same sub-claim, the main success claim is left insufficiently supported and the project might fail. Suppose, for example, that a success argument contained a sub-goal in which successful completion of a critical subsystem depended on the delivery of a new compiler from a vendor. If the compiler were not available, the sub-goal would not be met, the subsystem could not be competed as expected, and more than likely the entire development would fail to meet the main success claim because of late delivery.

Of course, setbacks in development are not uncommon, and must be dealt with no matter what. With a success argument available, however, the developers have the twin advantages of: (a) guidance about what the changes to the actual development process must accomplish in order to rectify the situation; and (b) assurance after the decision is made that the changes will solve the problem.

A flaw in a success argument can take one of three forms:

- ***Defective Evidence.*** Evidence that was expected to be delivered was deficient in some way. The evidence might be delivered late or it might not have the characteristics that were anticipated in the argument. Test results, for example, might take longer to acquire than expected or the product might fail tests at an unexpectedly high rate.
- ***Invalid Assumption.*** An assumption might prove to be invalid. Availability of equipment might have been assumed, for example, and in practice it turned out that the equipment was not available to the extent that was expected.
- ***Fallacious Argument Fragment.*** A fragment of the argument might be fallacious. An argument might have been made, for example, that requirements would be complete and fully documented by some particular time. That in and of itself is not a fallacy. However, the evidence for this argument is that the technique to be used has never failed previously in the application domain and with the customer involved. This argument relies upon the assumption that there are no significant differences between the prior development efforts and this one, and this assumption *must* be included explicitly in the argument. Not to do so is a fallacy known as unsupported generalization.

Requirements elicitation, capture and validation are known to be difficult problems even when carried out with great care, and so relying on prior experience can be problematic. A specific technique that has always produced success for a particular team in the past might fail to do so because of a seemingly insignificant difference in the circumstances. Stating this makes the limited strength of the argument more obvious and suggests to developers that they need to monitor the assumption carefully.

Fortunately, amending a success argument provides precisely the framework needed for a development process to recover from a problem, even a major one. The existing success argument has been shown to be deficient in some way, and an amended success argument has to be created. In doing so, the essential process steps will be implied by the changes arising in the success argument. The amendment algorithm is shown in Figure 9.

As an example of amending a success argument, consider the failure mentioned above that resulted from the late delivery of a compiler. Assume that the threat to the completion goal, in the form of the

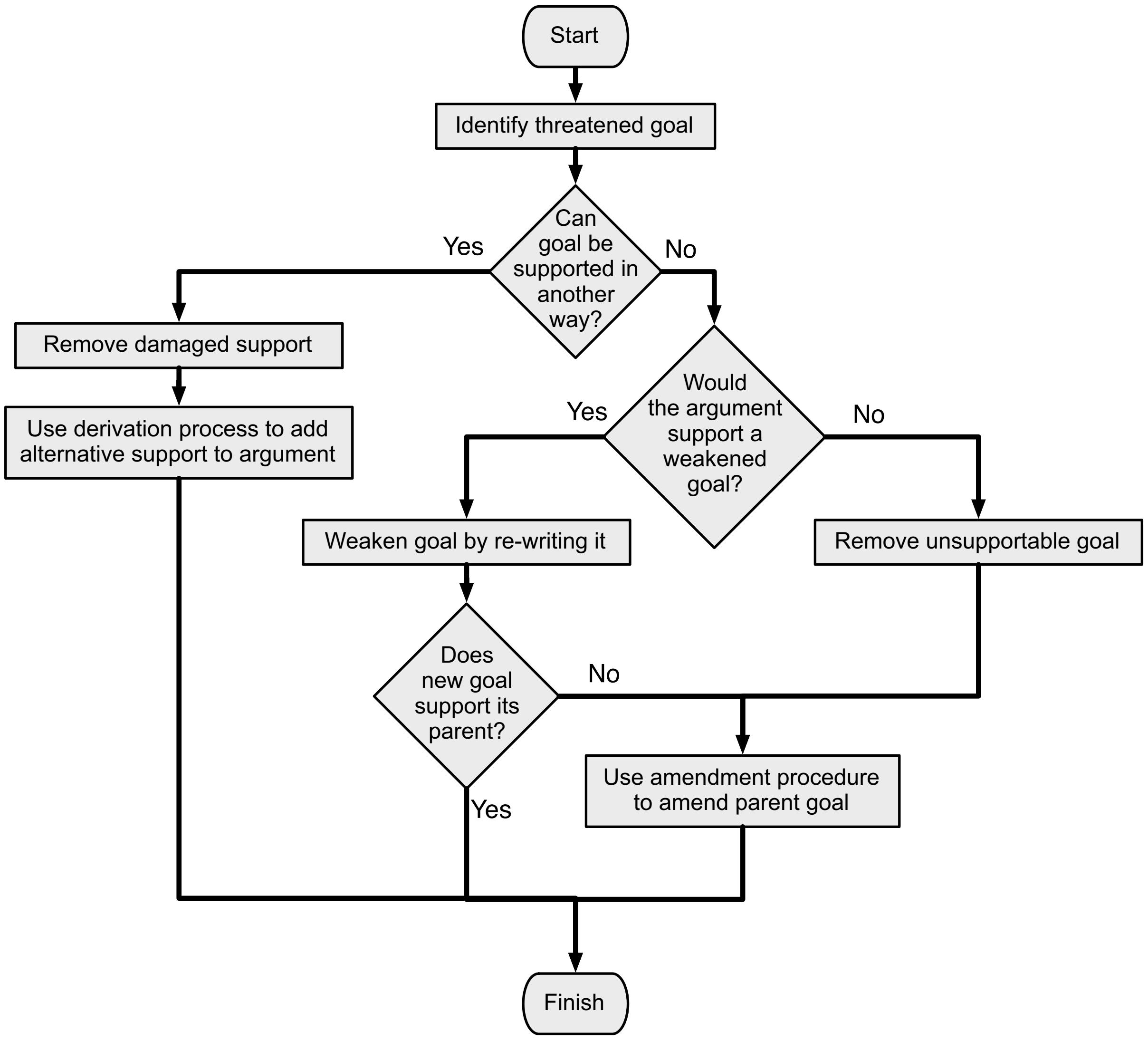

Start
Identify threatened goal
Can goal be supported in another way?
Yes
No
Remove damaged support
Use derivation process to add alternative support to argument
Would the argument support a weakened goal?
Yes
No
Weaken goal by re-writing it
Remove unsupportable goal
Does new goal support its parent?
No
Yes
Use amendment procedure to amend parent goal
Finish

Figure 9. Amending a success argument

invalid assumption that the new compiler would be delivered on time, was revealed by the vendor at the time that delivery had been expected. The options open to the developers, though obvious, are worth examining. They are:

- ***Support the goal in another way.*** The developers could elect to remove the invalid assumption, change the planned process to make use of a different compiler (perhaps incurring extra effort to modify the build process and perhaps the source program) and derive new support for the sub-goal from this new planned process step.
- ***Weaken the claim.*** The developers could weaken the sub-goal so that it claims completion by a later date. This might require weakening parent goals so as to permit a later delivery of the product.
- ***Eliminate the goal.*** The developers may decide that there is no way to support the claim and that it is unacceptable to weaken it. In this case, they would eliminate the claim and look for an alternative means of support for its parent goal. This might take the form of refactoring the system to replace the sub-system with one that does not rely upon the properties of the unavailable compiler.

In each case, by developing a repaired argument and reviewing it, the developers are able to see precisely what the impact of that choice would be. By building a repaired success argument, the ramifications of the option on the entire development activity can be seen. It is possible, for example, that the use of a different compiler is infeasible because of the negative impact that the use of the new compiler would have on the product's performance. In traditional development, this type of difficulty is sometimes hard to see, and those responsible for making a selection between choices might not realize the possible impact.

Repairing the success argument provides the opportunity to examine the complete effect of a process change and to present the results in a compact and complete way to the stakeholders and other experts. In essence, the success argument has to be rebuilt using what is now known to be different evidence, assumptions, or structure, and this leads to what must be a complete and compelling argument about the success of the remaining part of the development activity.

7.3 Success argument patterns

Developers constructing success arguments, like developers constructing safety and other forms of engineering argument [12], will likely benefit from a library of carefully constructed argument *patterns*. Just as design patterns capture and name software structures that provide certain desired functionality, a success argument pattern captures and names carefully checked argument fragments. The pattern shown in Figure 10, for example, is recorded in the same graphical notation that we use for success arguments generally and shows how one might argue that the risk of miscommunication of a specification has been adequately mitigated through the use of a formal specification language.

The pattern in Figure 10 reminds developers of some of the claims that the use of a formal specification language supports, and of the evidence and additional argument that they must supply. In this example, unsubstantiated goal **G_HHNLSCRiskMitigated** exists to remind developers of one of the limitations of formal specification languages and the consequences of that limit: because formal languages are semantically void, a formal specification must include some natural language text, if only to identify the real-world

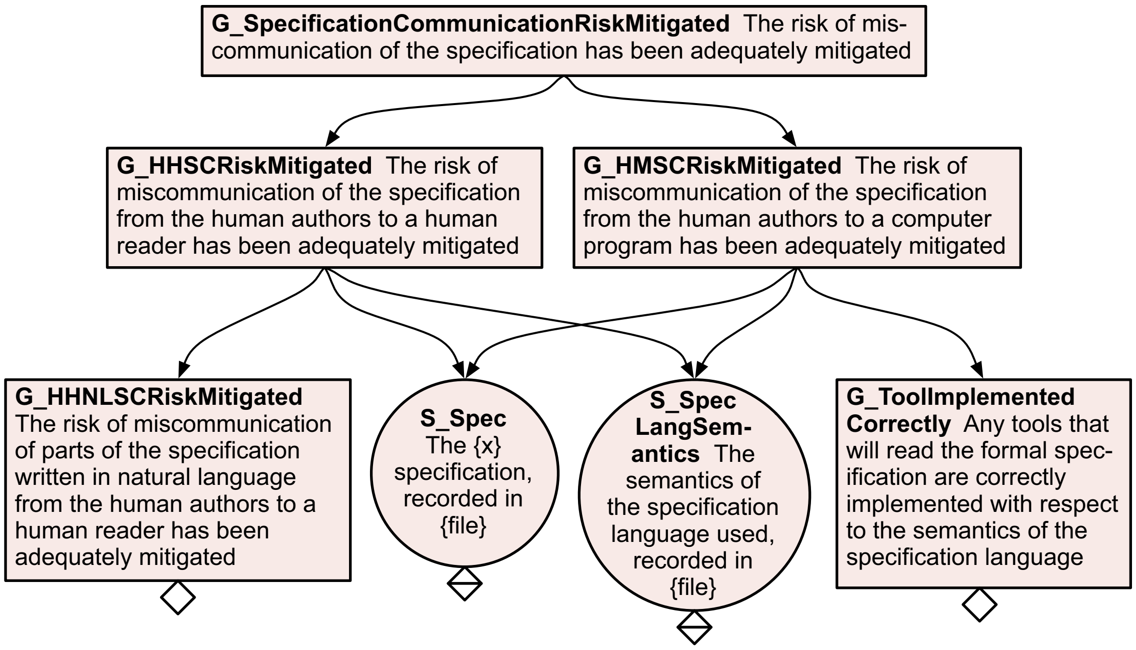

G_SpecificationCommunicationRiskMitigated The risk of miscommunication of the specification has been adequately mitigated
G_HHSCRiskMitigated The risk of miscommunication of the specification from the human authors to a human reader has been adequately mitigated
G_HMSCRiskMitigated The risk of miscommunication of the specification from the human authors to a computer program has been adequately mitigated
G_HHNLSCRiskMitigated The risk of miscommunication of parts of the specification written in natural language from the human authors to a human reader has been adequately mitigated
S_Spec The {x} specification, recorded in {file}
S_Spec LangSemantics The semantics of the specification language used, recorded in {file}
G_ToolImplemented Correctly Any tools that will read the formal specification are correctly implemented with respect to the semantics of the specification language

Figure 10. Pattern for arguing miscommunication risk mitigation via formal specification

entities corresponding to entities in the formalism. The pattern prompts developers to compensate for this limitation by adequately mitigating the risk of miscommunication in the non-formal portions of the specification, perhaps by the use of a technique such as CLEAR [20].

In addition to documenting the arguments made possible by the use of a particular tool, technique, or software development process, patterns can be used to document complete canonical arguments. In many cases, successful developments can be repeated, and so whatever argument justified the already completed activity can be reused. Such an approach does not mean that the argument would merely be copied and ignored. A canonical argument would not contain the specific details of the upcoming activity, and so it would have to be properly instantiated. This would mean defining the top-level claim, checking all of the development step parameters, and reviewing the argument to make sure that it is valid.

The use of a canonical argument does provide a significant benefit, however, because the basic structure of the argument would be complete and the overall argument structure would be valid. Canonical argument patterns might be particularly beneficial to developers in organizations that routinely conduct software development activities using a specific set of tools and techniques to solve problems in a specific application domain, since their development and use would allow for corporate knowledge about what works for a particular organization and what does not to be rigorously documented and made available for reuse.

Just as patterns allow pattern developers to capture information about argument structures that work, *anti-patterns* allow developers to capture information about argument patterns that do not work. Anti-patterns representing logical fallacies, for example, help developers to avoid writing invalid arguments.

7.4 Verification

It is important that the success argument be sound. An unsound argument could give developers false confidence in a flawed development plan, preventing them from taking corrective action until either it is too late or such action has become much more expensive. The aim of verifying a success argument is to ensure, to the degree practicable, that the argument is sound, i.e., we must ensure that the argument is both valid—its structure does not violate the rules of argument—and based on true premises.

Careful human review, of a form similar to the reviews that have been proposed for other forms of engineering argument [14], should form the basis of a success argument verification technique. Such reviews should be conducted with a care and frequency appropriate to the likely cost of the consequence of overlooking a defect in the argument. If project deadlines are tight or the consequence of a project failure is extreme, reviews should be scheduled more frequently and conducted more thoroughly than if time is ample or the cost of a project failure is low.

A taxonomy of argument fallacies [10] and a library of anti-patterns [12] have been developed for safety arguments, and these can be used initially as guides to help reviewers to identify flaws in success argument structures. In the future, they might be tailored for use with success arguments.

Although we have not done so, automated tools could be developed to conduct static analyses of success arguments. For example, an argument could be scanned to check that the referenced evidence exists, but human reviewers would need to determine whether the evidence means what it is claimed to mean in the argument. Human assessment is needed to determine, for example, whether the strength of the evidence is sufficient given the importance of the claims it supports and the degree to which the argument structure relies upon that evidence for the support of those claims.

7.5 Root cause analysis

Investigating process failures so as to avoid them in the future is usually referred to as *root cause analysis*. The analysis proceeds by determining what happened and why, and then process changes are developed to avoid repetition of the problem. Root cause analysis is a fundamental component of process improvement. In the Capability Maturity Model (CMM), for example, a key process area (KPA) at level 5 is Defect Prevention (DP).

As an example, consider a development activity in which the product is delivered several months late. An investigation reveals that the cause of the delay was insufficient computer availability for the core development team. Developers were unable to complete system builds promptly because the computers being used had slow processors, insufficient memory, and outdated, slow hard disk drives. In this simple case, the obvious process change is to include an explicit check on resource adequacy at the outset. In more complex processes, conducting root-cause analysis, including the determination of suitable changes, is much more problematic.

Root cause analysis is closely tied to success arguments. Having to amend a success argument during development frequently means that the rationale for the success of the planned process was wrong. If the success argument predicted something that did not come to pass, then something went wrong that those who developed the planned process and derived the success argument did not anticipate. Finding the defects in the success argument, therefore, provides a means of guiding root-cause analysis: each defect corresponds to a part of the associated planned process that needs to be changed so that the ensuing revised success argument is, in fact, correct.

This concept is similar to the use of safety arguments in the investigation of accidents and incidents in what is referred to as the Pandora approach [9]. In Pandora, a systematic review of the safety argument is conducted using a taxonomy of known fallacies [10]. Based on defects found in the safety argument, recommendations are generated that allow better practices to be developed.

This same approach can be followed using success arguments as the basis of root-cause analysis. The flaws in the success argument pinpoint the factors that led to the development failure. Details of transforming the entire Pandora approach to success cases is left to future work.

Returning to the example used earlier, the success of the planned development effort was based on the assumption that there would be sufficient computing and storage resources. This assumption was unwarranted. Although easily seen in this simple example, in complex multi-phased development efforts such erroneous assumptions will not be obvious. Avoiding similar process failures in the future requires making various process changes (which ones does not matter), and subsequent success arguments need to include explicit provision for avoiding the erroneous assumption.

8 Related work

Because success arguments are a framework for explaining why what a developer is doing will lead to a successful outcome, they are by nature related to much of the existing software engineering literature, and so we summarize only the most relevant work here.

Safety cases. Safety cases, a specific form of assurance cases, are used quite widely in Europe. In some domains, their use is mandated by regulation. Success arguments use rigorous argument in support of a different goal: the success of a software development effort rather than the safety of the resulting system.

Graphical notations, such as the Goal Structuring Notation [12, 13], have been proposed for recording safety arguments in a manner that aims to make the argument structure more apparent to a reader than natural language text. The notation for recording success arguments that we introduce in Section 4 is GSN, with minor modifications to suit the differing purposes of success and safety arguments.

Safety-case patterns [15] have been developed to capture common strategies for arguing many types of assurance claims. Patterns allow insights gained in the construction of a safety case to be passed on to other safety cases, and they help to ensure that all the required evidence for a specific type of argument has been provided. Likewise, safety case anti-patterns communicate common forms of fallacious assurance arguments. The success argument patterns and anti-patterns we propose are similar in form and purpose to safety-case patterns and anti-patterns. However, where safety-case patterns and anti-patterns describe ways to argue and to avoid arguing the safety of a system, success argument patterns and anti-patterns describe ways of arguing that a given software development effort will be successful, or ways to avoid doing so.

Review techniques for safety cases [14] have been proposed and a taxonomy of common safety argument fallacies [10] developed. Careful human review can be used to verify the soundness of a safety argument, and a list of known fallacies help engineers avoid making and reviewers spot common structural flaws in safety arguments. The argument verification techniques we propose are minor modifications to the techniques already developed for use with safety arguments.

Other forms of argument in engineering. The existing uses of argument in engineering are not restricted to safety alone. Arguments have also been proposed as a mechanism to demonstrate the security of systems [8], compliance with standards [5], and even to assess a planned system transition [19]. Again, success arguments represent the application of logical argument to yet a different kind of engineering claim.

Bayesian belief networks. Bayesian Belief Networks (BBNs) have been proposed as a way to analyze confidence in engineering arguments. Littlewood and Wright [17] have shown that analyzing multi-legged arguments using BBNs can reveal subtleties in interactions of arguments that might not be readily obvious. This capability might prove useful in the verification of success arguments, by helping developers to understand how much an argument relies upon the truth of particular premise given its structure and the needed confidence in the conclusion.

Problem-Oriented Software Engineering. In Problem-Oriented Software Engineering (POSE) [11], a problem statement is repeatedly transformed to remove constraints on environmental variables and other characteristics that prohibit direct implementation to yield a specification and, finally, an implementation. In POSE, the justifications for these transformations can be recorded, forming an argument that the implementation adequately solves the problem it is intended to solve. This argument has a different conclusion from success arguments: its main claim is that the system solves the problem it was intended to solve, not that the development effort will yield an adequate system in an acceptable time and at acceptable cost. The wider scope of a success argument's main claim permits reasoning about schedule, cost, and the effect of techniques and processes in a way that cannot be done in POSE.

Evidence-based paradigms. Evidence-based paradigms for research and practice are standards by which to evaluate these activities that focus on the use of objective evidence in support of a claim to their goodness. Evidence-Based Medicine (EBM) argues that we should assess the quality of evidence in analyzing the risks and benefits of potential treatments [6]. Similarly, Evidence-Based Software Engineering holds that we should assess proposed software engineering processes and products using direct evidence that they live up to their claims as opposed to relying exclusively on, for example, the CMM level of the responsible development organization or compliance with a standard. A success argument provides a structure for showing how the evidence coupled with a given software engineering process will contribute to the success of a given software development effort using it.

Design rationales. Numerous notations and techniques have been created to support the recording of a designer's rationale for making a particular design decision. The Qualitative Decision Management System, SIBYL [16], is an example of such a system and its associated notation. In keeping with the purpose of design rationales, which is to make the rationale for past choices available to developers making subsequent choices, systems for recording design rationales allow users to specify many questions, each answered with an argument. The main difference between design rationale arguments and success arguments is that success arguments support a well-defined goal that is broad enough that the supporting argument can touch all aspects of software development, whereas no design rationale recording technique of which we are aware attempts to combine the design rationale arguments into a comprehensive argument that the development effort will be successful.

9 Conclusion

We have introduced success arguments, a mechanism that permits developers to have justifiable confidence that a planned development activity will be successful. A success argument is a rigorous and compelling argument intended to convince a skeptical audience that a particular software development effort will terminate and demonstrably meets a balance of stakeholder goals that is acceptable to the stakeholders. The argument accumulates all the various facts, assumptions, and reasoning that underlie the developer's software development choices in a succinct form and exposes them to scrutiny. Thus, the success argument is uniquely capable of providing comprehensive assurance that a planned process for a particular development activity will succeed.

By virtue of its content and form, a success argument provides feedback to the developers of a planned process that can be used to refine that process. If the planned process does not yield a compelling success argument, then developers need to determine why that is the case and make suitable changes.

A simple way to think of a success argument is that it is a magnifying glass with which to examine processes. That examination can lead quickly to numerous opportunities for process improvement both for a specific development and for sets of similar developments within organizations. These opportunities come at multiple levels and in multiple time frames.

In the short term, even an imperfect success argument created by inexpert developers can yield important insights into the planned process from which it is derived. By contrast with rather imposing approaches to process improvement that are replete with extensive documentation and tools (such as the CMMI), a success argument is a small sharp knife. The knife costs very little, can be understood quickly and easily, and it can be applied immediately by individuals, groups, or complete organizations.

In the long term, success arguments provide a framework for determining whether the use of specific process elements will yield success. Success patterns can precisely describe the effects and requirements of process models and smaller process elements. Developers incorporating these process elements into their planned processes will know what must be true of their development efforts for the process elements to be effective, what effect they can expect, and how those effects will combine with the effects of the other process elements they have selected to produce success. Moreover, the precise characterization of the particulars of each development effort recorded in that effort's success argument can help to guide root cause analysis and software engineering. The knowledge developed in these ways can then be codified in success argument patterns and anti-patterns, bringing ever greater confidence to software development.

Acknowledgements. This work was sponsored in part by National Science Foundation grant CNS-0720794 and in part by National Aeronautics and Space Administration grant NAS1-02117.

References

1. Beck, K. *Extreme Programming Explained*. Addison-Wesley, 2000.
2. Bishop, P. and R. Bloomfield. "A Methodology for Safety Case Development." <http://www.adelard.co.uk/resources/papers/index.htm>
3. Boehm, B. "A Spiral Model of Software Development and Enhancement." IEEE Computer, May 1988.
4. Brooks, F. *The Mythical Man-Month*. Anniversary ed. Addison Wesley Longman, 1995.
5. Cyra, L., and J. Gorski. "Supporting Compliance with Safety Standards by Trust Case Templates." Proc. of ESREL 2007, Volume 2, Norway, 1367—1374, 2007.
6. Eddy, D. M., *Evidence-Based Medicine: A Unified Approach*, Health affairs (Project Hope) 24 (1): 9-17. 2005.
7. Graydon, P., J. Knight, and E. Strunk. "Assurance Based Development of Critical Systems." Proc. of 37th Annual International Conference on Dependable Systems and Networks. Edinburgh, U.K., 2007.
8. Goodenough, J., H. Lipson, and C. Weinstock. Arguing Security — Creating Security Assurance Cases. https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/assurance.html. Accessed April 2008.
9. Greenwell, W. *Pandora: An Approach to Analyzing Safety-Related Digital-System Failures*, Ph.D. Dissertation, University of Virginia, 2006.
10. Greenwell, W., J. Knight, C.M. Holloway, and J. Pease. "A Taxonomy of Fallacies in System Safety Arguments." Proc. of 24th International System Safety Conference. Albuequerque, N.M., 2006.
11. Hall, J., L. Rapanotti, and M. Jackson, *Problem-Oriented Software Engineering*. Open University, U.K., Tech. Rep. 2006/10, October 2006.
12. Kelly, T. *Arguing Safety — A Systematic Approach to Managing Safety Cases*, Ph.D. Dissertation, University of York, U.K., 1998.
13. Kelly, T., and R. Weaver. "The Goal Structuring Notation - A Safety Argument Notation." Proc. of the Dependable Systems and Networks 2004 Workshop on Assurance Cases. July 2004.
14. Kelly, T. "Reviewing Assurance Arguments - A Step-by-Step Approach." Proc. of Workshop on Assurance Cases for Security - The Metrics Challenge, Dependable Systems and Networks. July 2007.
15. Kelly, Tim, and J. McDermid. "Safety Case Patterns – Reusing Successful Arguments." Proc. of IEE Colloquium on Understanding Patterns and Their Application to System Engineering, London, Apr. 1998.
16. Lee, J. "SIBYL: A Qualitative Decision Management System." Artificial Intelligence at MIT: Expanding Frontiers, P. Winston and S. Shellard (eds.). MIT Press: Cambridge, MA, 1990.
17. Littlewood, B., and D. Wright, The Use of Multi-legged Arguments to Increase Confidence in Safety-Claims for Software-Based-Systems: a Study Based on a BBN Analysis of an Idealised Example, CSR Technical Report (2005)
18. MoD, "00-56 Safety Management Requirements for Defence Systems," U.K. Ministry of Defence, Defence Standard, Issue 3, December 2004.
19. Nguyen, E., W. Greenwell, and M. Hecht. "Using an Assurance Case to Support Independent Assessment of the Transition to a New GPS Ground Control System." Proc. of the International Conference on Dependable Systems and Networks. Anchorage, AK, June 2008.
20. Wasson, K. *CLEAR Requirements: Improving Validity Using Cognitive Linguistic Elicitation and Representation*. Ph.D. Dissertation, University of Virginia, 2006.