

Process Synthesis in Assurance Based Development

Patrick J. Graydon and John C. Knight

University of Virginia

151 Engineer's Way, P.O. Box 400740

Charlottesville, VA 22904-4740

+1 434.982.2216 (voice), +1 434.982.2214 (fax)

{graydon, knight}@cs.virginia.edu

ABSTRACT

Assurance Based Development (ABD) is a novel approach to the synergistic construction of critical software systems and their assurance arguments. In ABD, the need for assurance drives a unique process synthesis mechanism that results in a detailed operational process for building both software and an argument demonstrating its fitness for use in given operating contexts. In this paper, we introduce the process synthesis mechanism of ABD. A key element of ABD process synthesis is the success argument, an argument which documents developers' rationale for believing that the development effort in progress will result in a system that demonstrably meets an acceptable balance of all stakeholder goals. Such goals include safety and security requirements for systems using the software as a component and time and budget constraints. We also present the details of a case study in which we used ABD to develop the control software for a prototype artificial heart pump.

Categories and Subject Descriptors

D.2.9 [Software Engineering]: Management — *software process models*.

General Terms

Design, Economics, Reliability, Theory, Verification.

Keywords

Software process, Software assurance, Assurance arguments.

1. INTRODUCTION

Assurance Based Development (ABD) [4, 11] is an approach to constructing software systems in which creation of the software is combined with explicit creation of assurance of the software. Using ABD, developers can be confident to the extent possible that the construction effort will succeed, and that the resulting system will be demonstrably fit for use. In this paper, we describe ABD's *process synthesis* mechanism. Driven by each software system's unique assurance needs, this mechanism produces, for a given system, a detailed *operational process* for building both the software and evidence of its fitness for use.

ABD process synthesis is centered on two rigorous arguments: a *fitness argument* and a *success argument*. Together, we refer to these arguments as ABD's *assurance arguments*. Both are derived from related work on safety arguments [6]. A fitness argument gives the developers' rationale for believing that the system being built is fit for use, including both demonstrably adequate functionality and demonstrably adequate dependability. The success argument gives the developers' rationale for believing that the development effort under way will yield an adequate system on time and within budget.

Software engineering is about choices, and we contend that development choices should be driven by the need for assurance and judged according to the assurance that they provide. In ABD, the fitness and success arguments organize and focus all of the information that is necessary to make this possible. The state of the arguments at any time during development makes developers aware of the assurance needs so as to prompt them to consider the right development choices. The arguments also provide a sound basis for judging each potential choice in the context of both the particular software development effort at hand and the development choices that have already been made.

Knowing that a critical software system is fit for use in its expected environment is essential. There are many techniques available to developers of such software, but in most cases the benefits of such techniques have been shown only in isolation and developers often cannot fully exploit the benefits that they bring. For example, developers might use formal methods for the "critical parts of the system," but they are often unable to evaluate the ensuing effect on the system as a whole. In ABD, the development process is derived from the fitness and success arguments, and the arguments are evaluated and updated if necessary continuously throughout system development. ABD developers use the process synthesis mechanism to create an operational process that they can be confident will result in software upon which stakeholders can justifiably depend. A process repair mechanism allows choices that did not support the arguments as expected to be corrected.

Assurance Based Development ensures that the technology selected to create a software system yields the correct evidence to justify the desired confidence. The ABD process synthesis mechanism fills the void between the need for a rigorous assurance argument and the mechanism by which software will be built to meet that need. Process synthesis is especially important in circumstances where certification authorities require an assurance argument, but provide little or no other guidance. Such is the case with the British Ministry of Defence's Standard 00-56 [8]. The standard

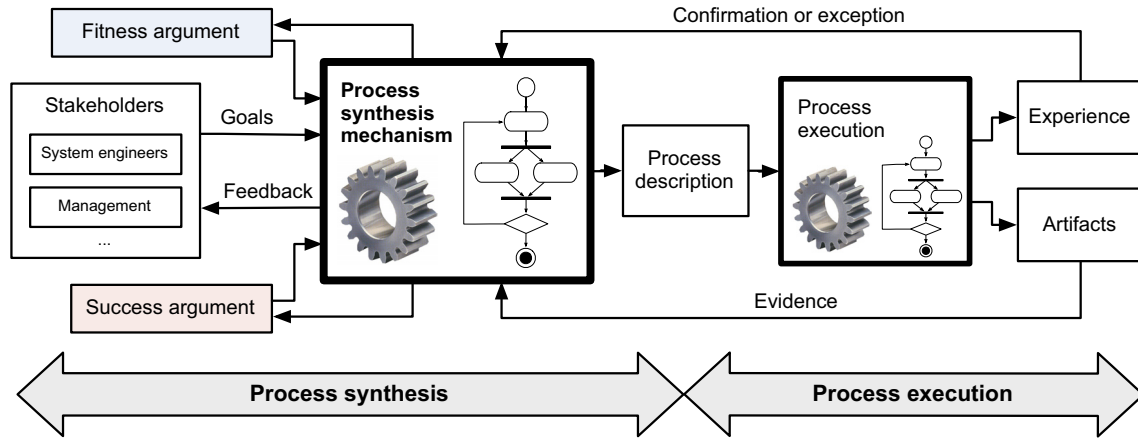


Figure 1. Synthesizing and executing an operational process in ABD

requires the creation of a safety argument, but provides no indication of acceptable software development practices.

In this paper, we present the ABD process-synthesis mechanism. In section 2, we summarize the ABD approach and detail our recent enhancements. In section 3, we describe how ABD’s fitness argument is used to demonstrate that a delivered system has all of the properties, including safety and security properties, that it must have if it is to be considered acceptable. In section 4 we introduce success arguments. We present the synthesis process itself in section 5 and a case study evaluation of enhanced ABD in section 6. Finally, we describe related work in section 7 and conclude in section 8.

2. ASSURANCE BASED DEVELOPMENT

2.1 Overview of ABD

The ABD process synthesis mechanism generates an operational process specific to the system being built. To the extent possible, this process will result, when followed, in a software system that is fit for use together with a compelling fitness argument for the system, and the process will do so on time and on budget.

Figure 1 illustrates process synthesis and subsequent process execution. Informally, synthesis begins with defining: (a) a top-level goal for the fitness argument (the details of what fitness means in this case); and (b) a top-level goal for the success argument (the details of what process success means in this case). Both arguments are then elaborated by selecting developing choices and merging the argument fragments that would result from those choices into the evolving fitness and success arguments. The operational process is derived from the choices, and that process is complete when no further refinement of the arguments are needed.

Because developers are informed during process synthesis of what remains to be demonstrated if the fitness and success arguments are to be complete and compelling, they can make fully informed choices that add to or modify the operational process. The operational process, and the artifacts that result from its execution, in turn contribute evidence to the two assurance arguments thereby refining them.

Choices carry some risk that they will not yield the results expected when the process is executed. If executing the operational process results in the expected evidence, the evidence defined in

the arguments is replaced with the actual evidence. If not, ABD accommodates the unexpected result via the process repair mechanism. Management of the process during execution is shown in Figure 1 by the two feedback loops.

In some cases, choices cannot be made at all because they depend on evidence that will accrue during development. A developer might need to build a throwaway prototype, for example, in order to obtain information needed for further process synthesis. While we expect that the bulk of process synthesis in most software development efforts will be completed early in the project, ABD allows process execution to begin before a complete process is synthesized in order to accommodate such cases.

The ABD process synthesis mechanism¹ is quite different from that surrounding traditional process models, such as the Spiral and Waterfall models. Those process models serve as templates from which developers create planned processes specific to their development efforts. This is done by instantiating the template using parameter information specific to the system of interest. Although valuable, such approaches to process synthesis do not accommodate the specific needs of a given system.

2.2 The Theory of ABD

Assurance Based Development¹ is based upon a conceptual view of software engineering which holds that developers make (and revise) a sequence of *development choices*, each of which contributes an element to an operational process. Each development choice could be characterized as an answer to the question, “How can <x> be demonstrated?”. Every time a developer decides to build a specification or a prototype, to use a specific programming language, to carry out testing of a certain form, to purchase and use specific tools, or makes any other decision that materially affects the development process or its results, he or she is, in our model, making a system development choice.

In *traditional* development, developers might not explicitly conceive of what they are doing as making a choice, or even consciously conceive of the choice or of any alternatives. If a certain programming language will be used to develop a given software system, for example, then that selection is still a choice irrespec-

1. ABD as described here is refined and extended from earlier work [4]

tive of whether the selection is because the developers: (a) explicitly considered alternatives; (b) adopted the programming language because it was dictated by a standard or by a non-functional requirement given by the customer; (c) chose the programming language because finding and hiring programmers versed in that language is easier than in some alternative; or (d) because they have always used that language and never conceive of using anything else. Even if implicitly, the *choice* to use that particular programming language has been made.

The choices that are made in a given ABD software development effort and the order in which they are made together determine the operational process for that effort. The operational process is a description of what has been or will be done in the course of developing the software, including any detail that will materially affect the development process or its results. Because development choices are made and elements of the operational process are executed throughout the software development effort, some portions of the operational process can be said to have been *planned* but not yet *executed*.

The essence of ABD is to force developers to argue explicitly that the development choices they make, taken together, will produce a system that is fit for use on time and within budget. Certainly, arguments exist in traditional development, but they are often implicit, fragmented, and approached in an ad-hoc manner. Making arguments and selection of choices explicit does not produce inefficiencies or impediments to progress. Rather, the explicit nature of selection and the need to justify choices in the two arguments makes developers aware of the choices they are making and provides the context for reasoning about those choices. We claim that making the arguments explicit, complete, and justified can only benefit the developer.

We note that ABD says *nothing* about the form of the operational process itself. All that is required is that the operational process support both arguments fully and that both arguments be both complete and compelling. Support of the arguments means that the actual evidence generated during development will be precisely that which was defined (and therefore expected) in the arguments when the process was synthesized. Because ABD does not dictate anything about the process itself, the process might well take the form of a Spiral model or any other familiar form.

3. FITNESS ARGUMENTS

The goal of ABD is to facilitate the production of a software system that is demonstrably *fit for use* in a given operating context. Every engineered system has a variety of stakeholders whose needs must be considered. In many cases, these needs conflict: the public demands a system that is as safe as practicable and also as secure as practicable, while those funding the effort demand low cost and rapid deployment. If a system is to be considered adequate, it must demonstrably meet a balance of stakeholder needs. Moreover, that balance must be acceptable to the stakeholders. Achieving an acceptable balance is crucial because demonstrating that a system meets any one goal is not sufficient. A system that is safe but fails to meet the customer's needs or provide adequate security is unacceptable. We say that a system that demonstrably meets such a balance is fit for use.

Each software system produced by ABD is accompanied by a *fitness argument*¹ giving the developers' rationale for believing that

"the system is adequately fit for use in the context(s) in which it will be operated." Other researchers have referred to similar arguments as dependability arguments [3].

Recall that ABD is based on two key concepts: (a) that argument should be used to document the rationale that the system is what it needs to be; and (b) that engineering decisions should be driven by the need to produce evidence for the argument. The notion used here of a fitness argument is more comprehensive than that of other forms of assurance argument, such as a safety argument, so as to enable the main claim of the argument has to be broad enough to encompass the needs of *all* the system's stakeholders. We include dependability considerations as well as functionality and any other considerations that might be said to bear on whether or not all stakeholders will find a given system acceptable.

3.1 Systems and Software

In many applications, some important dependability properties, such as safety and security, can only be discussed sensibly in terms of their impact on a wider system in which software is a component. For example, speaking of a software component in isolation as being "safe" makes no sense; instead, we must speak of what the software must do or refrain from doing if the wider system in which it operates is to be adequately safe or secure (or both).

In ABD, a software system that will be embedded in a larger system cannot be considered fit for use in that context unless it demonstrably possesses the properties upon which the larger system's functionality and dependability rely. These properties, collectively, form a *fitness contract* between the software component and the larger system into which the software will be integrated. The contract is two-way since it includes assumptions about the larger system upon which the software can depend. Contracts might change over time as development of the larger system progresses or as developers of the software system "push back," and any written version of a contract might be incomplete.

When ABD is used to produce software to be embedded in a larger system, it does not demand the use of any particular assurance methodology for that encompassing system. If a safety argument is developed for the larger system, the argument and evidence produced for the software component has to justify the claims about that component in the system-level safety argument. If, instead, the adequate safety of the larger system is demonstrated by showing that development was carried out in compliance with an appropriate safety standard, then the software component's argument must justify concluding that the parts of the standard that apply to the software component were, in fact, adhered to. In either case, the contract between the two systems specifies what the software component may assume about the larger system and what properties the software and the software's development must have.

When building stand-alone software systems, activities such as requirements engineering, hazard analysis, fault analysis, security threat modeling, and the like might be done as part of the software system development effort rather than as part of a separate effort at the level of a wider "system." In such cases, there is no contract from an outside system specifying what the software system must do. Developers will instead have to argue from the available evi-

1. In earlier work [4], we referred to the ABD fitness argument as the assurance argument.

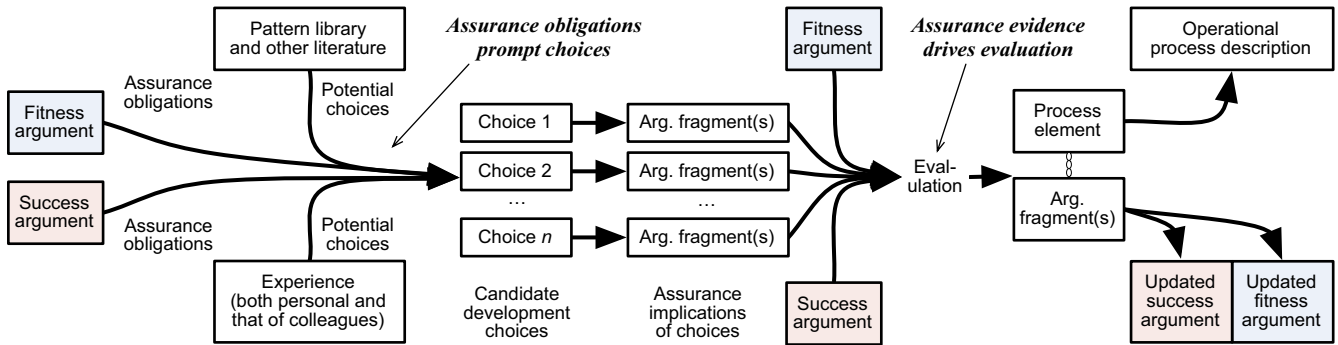


Figure 2. A process synthesis step in ABD

dence and from their assumptions about the operating context that the software system they produce will be fit for use in that context.

3.2 Documenting Fitness Arguments

To document fitness arguments, we use a variation of the graphical Goal Structuring Notation (GSN) [6]. GSN was developed to record safety arguments, and to adapt it for use in ABD, we have made minor modifications including the addition of development choice identifiers. These identifiers are discussed in section 5.2.

4. SUCCESS ARGUMENTS

Fitness arguments speak only to properties of the product. Separately, issues such as meeting development cost and schedule goals have to be considered. These goals are about development of the product, and so they are not characterized by the fitness argument.

To organize such information, and to give developers justifiable confidence that the detailed process they propose to use to build a specific system will result in success, we have introduced a different kind of engineering argument, a *success argument* [5]. A success argument documents the developers' rationale for believing that a proposed set of development activities will yield a system that is fit for use *on time* and *within budget*. Thus, for example, where a safety argument is used to guard against risks to human life or health, or damage to the environment, a success argument is used to guard against development risks.

As with a fitness argument, the text of the top-level claim of a success argument is always the same: "the effort will lead to an acceptable system in acceptable time and at acceptable cost." What is meant by the "acceptable time" and "acceptable cost" will vary from effort to effort, and the developers of any given project will define these phrases in the context of their project. Like fitness arguments, success arguments are recorded using a variation of the GSN notation.

As execution of the operational process proceeds more is learned about development risks, and elements of the success argument are either confirmed by experience or exceptions occur requiring changes in order to restore the strength of the argument (see Figure 1). At any time during development, the success argument shows what assumptions the developers have made, and the strength of the argument shows the confidence that can justifiably be placed in the claim that the effort underway will be successful.

5. PROCESS SYNTHESIS

In the ABD process synthesis mechanism (shown in Figure 1), process synthesis operates concurrently with execution of the oper-

ational process. This concurrency is shown in the figure through the feedback paths: as the execution of the operational process results in artifacts and experience, the process synthesis mechanism captures this evidence and incorporates it into the assurance arguments either confirming the argument elements or requiring that the argument(s) be changed because the operational process did not proceed as expected. Examples of the latter include test results (i.e., fitness evidence) not yielding the results expected thereby weakening the fitness argument and an activity taking far longer than expected (i.e., success evidence) thereby weakening the success argument.

An important aspect of the process synthesis mechanism is the *repair mechanism*. The process synthesis mechanism cannot guide a developer to make perfect development choices every time nor can the mechanism protect a developer completely from changes originating outside his or her scope of control. The repair mechanism accommodates the need to make changes to the operational process and the assurance arguments.

The process synthesis mechanism relies upon a structure that we refer to as *assurance links*. An assurance link couples the fragments of assurance argument and operational process description that resulted from a given choice, so as to facilitate repair.

We describe the process synthesis mechanism generally and the repair mechanism and assurance links specifically in the following sections.

5.1 Process Synthesis Overview

The ABD process synthesis mechanism, shown in Figure 2, is effected as a step performed multiple times by one or more developers. In each step, the developer performing it considers the state of the ABD assurance arguments, makes a system development choice, and modifies the operational process and the assurance arguments accordingly.

At any time during creation of the operational process, the unaddressed goals in the assurance arguments represent *assurance obligations* that the developers must find a way to satisfy. The developer selects from among these a goal or goals to be addressed and then seeks a way to do so. When choosing a goal to address, developers should consider: a) their area of expertise; b) the perceived risk that each goal might be infeasible to address; and c) the need to minimize interdependency and so avoid the case where developers simultaneously make mutually-incompatible decisions.

Sequencing of choices does *not* reflect the ordering of activities in the operational process. During process synthesis, choices can be

made, changed, and updated in any order that reflects their overall impact on the arguments. For example, a choice that is mandated by some applicable standard should be made first so as to ensure that its feasibility is not precluded by other choices.

Developers approach making development choices as seeking an answer to the question “How can <x> be demonstrated?”. This approach keeps the focus on developing the evidence that will be needed in the assurance arguments. Also, framing each choice in the form of a question prompts developers to think of alternative means that might be used to demonstrate the truth of the claims.

Once a developer has selected an assurance obligation to be satisfied, he or she then sets about gathering a set of candidate development choices that might be used to satisfy it. Note that multiple choices might be necessary to satisfy an assurance obligation in full. This is usually the case at the beginning of process synthesis where high-level obligations are addressed. As described in prior work [4], the developer gathers candidate development choices from a variety of sources including his or her own experience, the experience of colleagues, the relevant literature, and a library of ABD patterns. We expect that patterns will be useful in helping developers find an acceptable choice. Each candidate choice is then assessed using a set of criteria including whether the choice supports or precludes achieving needed functionality, the likely restrictions it imposes on later choices, whether it supports or precludes achieving the needed dependability, the costs it imposes, its feasibility, applicable standards, and any relevant non-functional requirements.

Evaluating a choice can be daunting since choices depend both on each other and on prior choices, and because they affect future choices. The choice to use a particular programming language, for example, may preclude the subsequent use of static analysis techniques that rely on certain language features. A developer need not enumerate all plausible candidate choices or to assess each choice perfectly. Enumerating and evaluating alternative choices requires time and effort, and so developers must balance the perceived risk of making (and thus having to redress) a poor choice against the time required to make a more considered decision.

After a choice is made, the argument fragments corresponding to it are added to the two ABD assurance arguments and the associated process element is added to the operational process by recording it in the appropriate project documents. Where evidence is not available at the time the choice is made but will result from execution of the process element, an annotation in either the fitness or success argument (as appropriate) is used to indicate that the evidence is forthcoming; this annotation is removed when actual evidence replaces the defined evidence.

5.2 Assurance Linking

The relationships between process element choices, the evidence they produce, and the associated locations in the fitness and success arguments are recorded using *assurance links*.

Assurance links need to be navigable in both directions. A developer needs to be able both to identify which development choice(s) led to a given section of one of the assurance arguments and to identify which portion(s) of one of the assurance arguments gave rise to a given choice. Both capabilities are necessary to support the repair mechanism described in section 5.3: a developer attempting to repair a portion of argument needs to be able to iden-

tify which development choice(s) might need to be revisited in order to give rise to the needed argument, and a developer contemplating revising a given choice needs to be able to identify the portion of argument that will be affected by that change.

We record assurance links by assigning each development choice a unique identifier. Each node in both ABD assurance arguments is tagged with the identifier of the choice that resulted in that node. Examples are shown in Figure 4. Likewise, each record of the choice appearing in any development artifact — a written description of the planned process, a development schedule, or any other permanent record of the choice — is tagged with this identifier in a manner suited to the artifact in question.

5.3 Repairing the Planned Process

At any point in the creation or execution of the planned process, a developer might discover a flaw in the planned process, the arguments, or both. For example, a developer might find: (a) that a previous choice has led to a goal that cannot be satisfied; (b) that a portion of one of the assurance arguments is not logically valid; (c) that a development choice did not lead to the expected evidence; or (d) that the process element(s) contributed by a choice cannot be executed because it was not feasible as stated or because critical resources have become unavailable.

In such cases, a developer must readdress one or both assurance arguments, the planned development process, or both using the ABD *repair mechanism*. First, any faults in the argument itself, such as logical fallacies, poor assumptions, unwarranted inferences, or even flaws in the notation are corrected. If the argument is still not compelling, the problem lies with a poor development choice that is infeasible or does not contribute the necessary evidence. Repair is effected by first identifying the choice, enumerating alternatives, selecting one, and then modifying both the operational process description and the two assurance arguments. The assurance links facilitate the identification of the choice corresponding with the problematic section of argument.

If no reasonable alternative can be found, the problem of the poor choice has its roots in a previous choice which must itself be readdressed. The developer must identify the prior choice(s) that influenced this one and consider alternatives to those until a suitable one can be found.

6. A CASE STUDY OF ABD

6.1 System Studied

In order to assess ABD process synthesis, we have conducted a case study development of a specimen system, the University of Virginia’s *LifeFlow* Left Ventricular Assist Device (LVAD) [12]. *LifeFlow* is a prototype artificial heart pump designed for the long-term (10–20 year) treatment of heart failure. *LifeFlow* is a continuous-flow, axial design. Magnetic bearings and a brushless DC motor will keep the pump’s impeller centered in the pump housing and turning without the need for mechanical bearings or shaft seals. Careful design of the pump cavity, impeller, and blades, aided by computational fluid-dynamics simulations, minimizes the damage done to blood cells, thus reducing the potential for the formation of fatal blood clots.

Control of the magnetic suspension bearings is provided, in part, by a digital control algorithm running on a microcontroller under the command of a higher software layer. In hard real-time, the con-

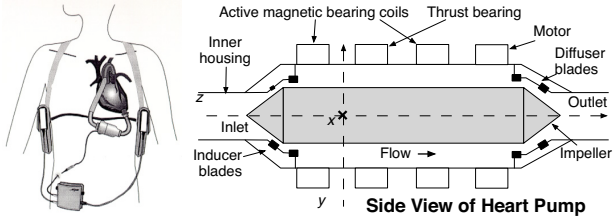


Figure 3. LifeFlow structure and use

troller must sample the position of the rotor as reported by a self-sensing circuit, compute the coil currents necessary to keep the rotor adequately centered, and direct the coil driver to achieve those currents. Individual magnetic coils can fail and such failures are anticipated to be more likely than is acceptable, and so the control software is also required to be capable of reconfiguring to a variety of backup modes in which rotor levitation is accomplished with the coils remaining after a failure. Figure 3 shows the placement of the pump, the batteries and the controller, a cross-section of the pump, and the overall structure of the controller. Table 1 briefly summarizes the requirements for the control software.

Table 1: Magnetic bearing control software requirements

Functionality	<ol style="list-style-type: none"> 1. Trigger and read ADCs to obtain impeller position vector u. 2. Determine whether reconfiguration is necessary, and if so, select appropriate gain matrices A, B, and D. 3. Compute target coil current vector y and next controller state vector x as follows: $y_k = Dx_k$ $x_{k+1} = Ax_k + Bu_k$ 4. Update DACs to output y to coil controller.
Timing	This functionality must be provided in hard real-time with a frame rate of 5 kHz .
Reliability	No more than 10^{-9} failures per hour of operation.

The LifeFlow team is presently constructing a prototype pump: (a) to determine whether the target blood damage characteristics can be achieved; (b) to demonstrate the efficiency of the impeller position-sensing mechanism; and (c) to determine whether software could be built to support the final LifeFlow system’s fitness goal.

We chose development of the magnetic bearing control software for the prototype LifeFlow LVAD for the case study because the system presents significant process challenges. We obtained the

software requirements from the LifeFlow developers and built the necessary software as described in the following sections.

6.2 Case Study Process

Ideally, we would have conducted a controlled experiment with replicates to obtain a statistically significant assessment of the effect of ABD upon software development outcomes. Such an experiment would be far too costly, and so instead we conducted a case study to determine whether any of a set of potential pitfalls would manifest in practice. In particular, our study aimed to determine whether:

1. **ABD is feasible.** ABD would be infeasible if it required the developer to perform tasks of which he or she were not capable or if the additional effort required to create and maintain the fitness and success arguments was prohibitively high.
2. **Unsupported goals in the assurance arguments are appropriate drivers for development choices.** ABD might be less effective than traditional methods if the developer was precluded from making the right choice or if the developer was distracted from the right choice by ABD’s focus on assurance.
3. **The effect of a choice on the assurance arguments is a sufficient basis on which to judge it.** ABD is based on the premise that if fitness and success can each be adequately guaranteed we will achieve all project aims. ABD would fail if any development choice brought a concept of value that could not be represented in either assurance argument.
4. **The ABD development choice criteria are the right criteria.** The effectiveness of ABD would be compromised if the set of criteria according to which developers are asked to evaluate candidate development choices is missing an important criterion or if the criteria forced developers to spend too much time considering irrelevant aspects of a choice.

To provide a basis for making these determinations, we conducted our case study development in conformance with a strict protocol that required us to answer 23 questions each time we made a choice and 5 questions each time we invoked the repair mechanism. For each development choice and each of the ABD decision criteria, we recorded answers to the questions “What assessment of each candidate choice was made based on this criterion?” and “Was assessment of the candidate choices in terms of this criterion: (a) not useful; (b) somewhat useful; or (c) critical?” and rated the difficulty of making the assessment on a scale.

For each development choice we also answered the general question “Was there a factor in making this decision that was *not* raised by analysis in terms of the criteria? If so, what was it?”. We considered whether it was clear in foresight, hindsight, or both that the assurance obligations prompted the choice, listed the traditional software engineering artifacts that a given choice might have been recorded in, and answered the question “Are there other development choices that could have been made in parallel?”.

6.3 Elaborating the Argument Goals

The first development choice we made was to elaborate the parameters in the standardized top-level goals used in both arguments. This choice is obvious. We used GSN context elements to introduce these definitions:

- **C_System:** “System” is the magnetic bearing control software.

- **C_OperatingContext:** The system is a component of the LifeFlow LVAD First Prototype.
- **C_Requirements:** Requirements imposed by the LifeFlow LVAD First Prototype are recorded in *<path of file>*.
- **C_AcceptableTime:** LifeFlow LVAD First Prototype delivery date.
- **C_AcceptableCost:** Presently available resources and staff plus target hardware costs.

6.4 Example Choices

Space considerations preclude us from presenting each of the development choices we made, their effects on the assurance arguments, the choices at our disposal and the rationale behind the selected choice. Instead, we summarize two significant development choices below, including only the most significant elements of the examples for brevity and clarity.

6.4.1 Fourth Development Choice

At this point in process synthesis, we had elaborated the development context and chosen to create a development schedule but not made any choices yielding evidence of fitness. The goals that were unaddressed in the evolving assurance arguments implied the following assurance obligations:

- **(Fitness) G_ReqSatisfied:** The delivered system satisfies its requirements.
- **(Success) G_ReqLateRiskMitigated:** The risk that details missing from the requirements will not be made available in time for the effort to succeed has been adequately mitigated.
- **(Success) G_DepGoalsUnachievableRiskMitigated:** The risk that the dependability goals might be impossible to meet successfully has been adequately mitigated.
- **(Success) G_TimingGoalsUnachievableRiskMitigated:** The risk that the real-time goals might not be demonstrably achievable has been adequately mitigated.

The question that we chose to answer was: “How can **G_ReqSatisfied** be addressed?”. The development choices that were prompted by this question were:

1. Use a refinement approach such as the B method [1].
2. Develop a formal specification in Z [10].
3. Develop a formal specification in PVS [9].
4. Adopt the SPARK Ada programming language [2] and its associated tools to prove that the implementation complies with the low-level specification embodied in the SPARK annotations.
5. Use the Echo approach [13, 14] to formal verification.

We were conscious of the fact that none of these choices would satisfy **G_ReqSatisfied** in full and that additional choices would be made later. Our evaluation of these choices was as follows:

- **Choice 1:** A refinement approach would bring strong evidence that the source code refines a formal specification. Choosing a refinement approach would constrain us to programming languages and tools suitable for that approach.
- **Choice 2:** Developing a formal specification is thought to help uncover defects, and formal specifications are less prone to misunderstanding than ones written in natural language. The choice of Z would necessitate translation if we later chose to use tools made for a different language.

- **Choice 3:** A formal specification in PVS notation would bring the same benefits as a formal specification in Z. Again, the choice of a specific notation restricts later choices.
- **Choice 4:** Use of the SPARK Ada language and its associated tools yields strong evidence that the code complies with the low-level specification given in the SPARK annotations. Making this choice would mandate the later selection of an Ada compiler for the target microcontroller.
- **Choice 5:** Use of Echo would bring strong evidence that the code implies the functional part of a formal specification. While the approach can in principle be used with various programming and specification languages, the tools are at present available only for SPARK Ada and PVS, thus restricting later choices. The technique and its tools are still under development; their use thus carries a risk.

The choice we selected was choice 4, for the following reasons:

Rationale: We decided that the fitness evidence produced by using the SPARK Ada programming language and its associated tools was indispensable in this effort. It was not yet clear whether a proof that the low-level specification implied a formal specification was necessary or whether a simpler approach could justifiably be used for this system. In later choices, we selected the Echo approach and the use of a formal specification in PVS.

Contributions to the Arguments: Making the choice to use SPARK Ada and its associated tools allowed us to add the argument fragment shown in Figure 4 to the fitness argument. Note the diamond decoration on the strategy element **ST_ArgOverRefinement**, which indicates that it is not yet completely supported. The choice to use SPARK Ada only partially addresses the obligation to show that the requirements have been satisfied, and so the argument fragment provides only partial support for goal **G_ReqSatisfied**. The later choices provided the remaining support.

Note also the diamond decoration on solution **S_CCPostCondProved**. Since we had chosen to establish proofs using the SPARK tools, we added this solution to indicate that the proofs would be forthcoming. When the proofs were complete, (they showed what we expect they would show), we removed the decoration. If they could not have been completed as expected, we would have had to repair our operational process and fitness argument.

In retrospect, we should have added an unaddressed goal to the success argument’s argument over development risks so as to force ourselves to consider the risk that the system team would chose a processor for which no suitable Ada compiler was available. We noted this later, and addressed the issue by advising the LifeFlow systems engineers of suitable processors.

6.4.2 Seventh and Eighth Development Choices

At this point in process synthesis, we had made development choices that partially addressed **G_ReqSatisfied** by providing evidence that the delivered software would be functionally correct, but we had not addressed **G_TimingGoalsUnachievableRiskMitigated** or provided evidence that the real-time requirements would be met. Accordingly, the question that we chose to answer during the seventh process synthesis step was: “How can we ensure that the real-time goals will demonstrably be met?”. The development choices that were prompted by this question were:

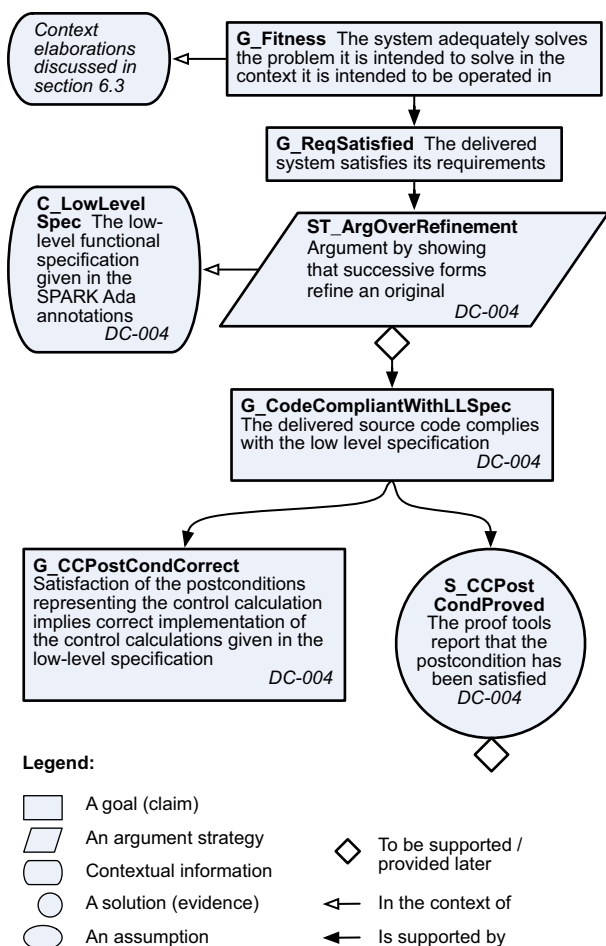


Figure 4. Assurance Contribution From Choice 4

1. Analyze the worst-case execution time (WCET) analysis of the control calculation using a technique to be chosen later.
2. Use a watchdog timer to stop the calculation and re-issue the control outputs from the last frame if the deadline would be otherwise be missed.
3. Select a real-time operating system and write the control computation as a periodic task.
4. Design the control computation as a periodic Ada task conforming to the Ravenscar Profile.
5. Use a cyclic executive design.

The choices we selected were (1) and (5). We reasoned that choice (2) was unacceptable because it would force us to demonstrate that re-issuing the last frame’s outputs would be done rarely and that doing so that rarely would be sufficient to keep the impeller from striking the pump’s inner housing. We rejected choice (3) because we were unsure that we would be able to argue convincingly that a system using an operating system would be adequately dependable. We rejected choice (4) since it would force us to use, and thus argue the dependability of, an Ada runtime implementing tasking. Choices (1) and (5), we reasoned, would supply strong evidence that the real-time deadlines will be met, although choice (5) might complicate implementation if the software is extended to handle other tasks because it might require decomposing those tasks into pieces.

Making these choices allowed us to add support for **G_TimingGoalsUnachievableRisk** to the success argument in the form of a new sub-argument: (a) if the microcontroller is sufficiently fast and if scheduling new tasks brought about by requirements change is possible, then we should be able to create a suitable schedule for our cyclic executive design; and (b) if WCET analysis of the code for each task and the executive structure is possible, then we should be able to demonstrably meet our real-time requirements. These new sub-goals were what prompted the later choice to provide the LifeFlow systems engineers with criteria for the selection of the microcontroller so that they could select one compatible with our development choices.

Making these choices resulted in an argument fragment that used evidence from the real-time schedule and WCET analysis to demonstrate that the real-time requirements had been met. Integrating this into the fitness argument required a change to the structure shown in Figure 4: we added a layer of argument between **G_ReqSatisfied** and **ST_ArgOverRefinement** that decomposed the obligation to show that the requirements had been met into an argument over different kinds of requirements. The argument fragment shown in Figure 4 was kept to show that the functional requirements had been met, and the new argument fragment was added to show that the real-time requirements had been met.

6.5 Artifacts Completed

At the time of this writing, final control constant values were not available, and so we could not complete development of the final software artifact and integrate it with the LifeFlow bearing hardware. Rather, we completed the implementation with temporary control constants and developed all the other necessary artifacts. Specifically, we have completed:

- A PVS specification for the magnetic bearing controller.
- An implementation of the control algorithm written in SPARK Ada. This implementation consists of 242 non-blank, non-comment lines of SPARK code and 133 lines of annotations.
- A refinement proof, checked by the PVS proof verifier, showing that the low-level specification embodied in the SPARK annotations complied with the PVS specification.
- Automatic proofs, generated by the SPARK proof tools, that the control algorithm implementation satisfied the low-level specification embodied in the SPARK annotations.
- A fitness argument that explains how this evidence, together with the remaining evidence that we will generate for the final system, supports the conclusion that the system is fit for use.
- A success argument giving our rationale for believing that, when the final control constants are available, we will be able to complete construction of the final system in acceptable time and at acceptable cost.

We tested this implementation against a reference simulation of a simple physical system and a controller for it, and found that the SPARK implementation’s output was acceptably close to that of the reference implementation, including when we forced the system to reconfigure. Testing of this sort would not be sufficient to demonstrate the functional correctness of the final software, but such a demonstration is not needed since the argument about dependability in the fitness-for-use argument depends primarily on proof.

6.5.1 The Fitness Argument

Execution of the operational process yielded the software that we sought along with the associated fitness argument. At the time of writing, the fitness argument contained 42 GSN elements, including 5 solutions, 11 context elements, 3 strategies, 3 assumptions, and 20 goals. Three goals remain unresolved pending decisions that we cannot make until the necessary details from the LifeFlow LVAD system are made available to us.

6.5.2 The Success Argument

At the time of writing, the success argument contained 41 GSN elements, including 4 solutions, 9 context elements, 10 assumptions, and 18 goals. The argument is divided into two main lines of reasoning: (a) a constructive portion that uses our development schedule as a basis for reasoning about the amount of time construction will require (and thus, in large part, the cost of development); and (b) an argument over enumerated development risks that gives our reasons for believing that each had been adequately mitigated.

6.6 Observations

The success argument enabled us to deal with the two major development risks that we faced: incomplete information from the systems engineers and possible inadequacy of the complete package of chosen technology. Incompleteness of, and uncertainty and change in, the constraints imposed on software projects from the outside are an unfortunate reality in many development efforts. In this effort, the incompleteness was severe: the requirements that we were initially given did not specify the microcontroller to be used, the dependability requirements, or the precise control constants to be used. Some of these details arrived later, while others, such as the control constants, are still unavailable as of the time of this writing.

Because the constraints of the overall system development schedule required that we not delay development, we proceeded with development with incomplete requirements. This was not an exception in our execution of ABD, but rather a deliberate development choice that met our specific needs: we proceeded only after arguing convincingly that, when details such as the correct control constants were available, we would be able to integrate them into our software and regenerate the evidence needed to support the fitness argument. The flexibility gained by using the success argument, rather than a rigid model of a software lifecycle, as the foundation for planning allowed us to accommodate the impact of missing information in an explicit and orderly way.

The second major development risk, possibly inadequate technology, is a serious issue because of its potential impact on the LifeFlow LVAD system dependability. The LifeFlow system requires that the active magnetic bearing control software be ultra-dependable, and so we sought to demonstrate that the combination of tools and techniques that we selected would achieve that dependability. This assurance obligation shaped our evolutionary prototype: while laying out the argument associated with the prototype, we convinced ourselves that our choices as to what should be included from the prototype, what should be assumed, and what could be excluded were appropriate given what its development was meant to demonstrate.

In ABD, the success argument is the focus for development risk. In this development effort, an argument over enumerated develop-

ment risks formed the bulk of our success argument. Whenever a choice created a development risk, as occurred with our choice to use SPARK Ada without first knowing the target architecture, we added this risk to the success argument as an unsupported goal so that the risk would be addressed by subsequent choices. Had cost and schedule been of greater concern to us, we might have demanded more support for a constructive line of argument that established the accuracy of our schedule estimates and predicted acceptable time and cost. We might have, for example, decided to conduct a detailed review of the schedule and the time estimates contained within it, or decided to periodically review and update the schedule so as to ensure that it remained accurate and consistent as development progressed and experience accumulated.

Maintaining both assurance arguments forced us to think critically about the conclusions that could be supported by the use of a particular tool or technique. When thinking about how to integrate evidence from functional testing into our fitness argument, for example, we noted that the test cases would be derived from the requirements, and not the specification. Thus, the test results would be evidence that the code complies with the requirements but could say nothing about the correctness of the specification.

When we first began to synthesize the process, we were faced with broad assurance obligations that no single development choice could fully address. As a result, we found ourselves considering “alternative” choices that spanned a broad spectrum of categories, a situation that became much less common after we had made several choices. Our first decision, for example, could easily have been the choice of a programming language, the choice of a specification technique, or the choice of a verification technique. Upon making any one of these choices, we would have decomposed the assurance obligation into a part that was supported by our choice and the remainder, which would serve to prompt later choices.

This observation raised a concern: what if the first choice is wrong, thus making a later choice impossible? We chose to address this issue by adopting a policy of not constructing artifacts until we had performed enough process synthesis to have an acceptable success argument. Given this policy, if our first choice turned out to be a mistake, we knew that we could “unmake” the choice at a cost no larger than the time needed to repair the assurance arguments and operational process description.

6.7 Results Of The Case Study

From the case study, we are able to draw preliminary conclusions about our study goals. Our first study goal was to determine whether ABD is feasible, and our experience suggests that it is. Although our study subject was small, its size belies its complexity. Meeting the requirements meant meeting significant real-time deadlines, operating on an embedded target with no operating system, interfacing with analog signals, and reconfiguring following coil failures. Despite these challenges, we observed no difficulties that we foresee challenging developers building other systems.

The second goal was to determine whether unsupported goals in the assurance arguments are appropriate drivers for development choices. In total, we made 13 choices. In 8 of these, we followed and recorded a direct line of reasoning from the assurance obligations to the choice that we made. In 3 other cases, we did not recall being prompted by the obligations. In the remaining 2 cases, we realized that we were considering an assurance obligation arising

from a development risk not yet added to the success argument. In all choices, there was a discernible relationship to the obligations in the arguments. We conclude that assurance obligations were an appropriate driver for development choices in this case.

The third goal was to determine whether the assurance arguments are a sufficient basis for judging a choice. Our case study protocol forced us to consider all of the value that a given choice might bring. We did not observe a value that could not be added as evidence to one of the arguments. We conclude that effect upon the assurance arguments was a sufficient basis upon which to judge choices in this case.

The fourth goal was to determine whether the ABD decision criteria are the right criteria for evaluation of choices. In 8 of the choices we made, we determined that the decision criteria covered all aspects of the choice. An unrepresented aspect that we observed frequently was effect upon schedule. We conclude that effect upon schedule should be added as a criterion.

7. RELATED WORK

Multiple standards designed to promote software assurance have been developed, but they demand the same prescription of all the systems they cover, whether this will achieve the necessary assurance or not. ABD instead compels the developer to assess the dependability needs of each part of a system and make development choices accordingly; the developer can thus economize in some parts of the system while remaining assured that the system as a whole will be adequately dependable.

Other safety-critical software development work is assessed via a safety case. Some standards [8] and researchers [6] call for safety cases to be constructed early and updated often during system development and subsequent change. Building on this idea, ABD interleaves development and assurance in such a way that the assurance case can be used to drive system development.

Problem-Oriented Engineering [7] aims to create a system and an argument that it is fit for use. In POE, the problem to be solved is documented, possibly using a Problem Frames notation, and progressively transformed, via transformations justified by the particulars of the effort, into an implementable specification. While POE is intended to produce systems which demonstrably solve a given problem, ABD is concerned with a wider problem: producing operational processes that produce software that demonstrably solves a given process and does so on time and within budget.

8. CONCLUSION

Software development processes are grounded in choices. Developers make choices between technologies, between event orderings, about the adequacy of performance, and other issues. In traditional development, the mechanism for making choices is implicit and ad hoc.

Assurance Based Development brings the notion of rigorous argument to the problem of selecting choices. ABD generalizes the notion of argument to include an argument for fitness for use of the product and an argument for success of the process. By doing so, ABD provides a comprehensive basis for development choices.

We have presented the details of how the ABD arguments operate to drive the process synthesis activity. We have illustrated ABD

with a case study of the development of software for a sophisticated, safety-critical application.

9. ACKNOWLEDGEMENTS

We thank Brett Porter of AdaCore and the AdaCore corporation for their support of this project, and Paul Allaire and Houston Wood for details of the LifeFlow LVAD. Work funded in part by NASA grants NAG-1-02103 & NAG-1-2290, and NSF grant CCR-0205447.

10. REFERENCES

- [1] Abrial, J.R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
- [2] Barnes, J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.
- [3] Despotou, G., and T. Kelly. "Extending the Safety Case Concept to Address Dependability." *Proc. of the 22nd International System Safety Conference*, August 2004.
- [4] Graydon, P., J. Knight, and E. Strunk. "Assurance Based Development of Critical Systems." *Proc. of Dependable Systems and Networks*, Edinburgh, UK, 2007.
- [5] Graydon, P. and J. Knight. "Success Arguments." *Technical Report CS-2008-10*, University of Virginia. July 2008.
- [6] Kelly, T.P. "A Systematic Approach to Safety Case Management." *Proc. of SAE World Congress*, Detroit, MI, March 2004.
- [7] Hall, J.G., L. Rapanotti. "Assurance-driven design" *Proc. of the Third International Conference on Software Engineering Advances*, Sliema, Malta, 2008.
- [8] MoD, "00-56 Safety Management Requirements for Defence Systems," U.K. Min. of Defence, Def Std, Issue 3, Dec. 2004.
- [9] PVS Specification and Verification System.
<http://pvs.csl.sri.com/>
- [10] Spivey, J.M., *The Z notation: a reference manual*, Prentice Hall 2001.
- [11] Strunk, E. and J. Knight. "The Essential Synthesis of Problem Frames and Assurance Cases." *Proc. of 2nd International Workshop on Applications and Advances in Problem Frames*, co-located with 29th International Conference on Software Engineering, Shanghai, May 2006.
- [12] Untaroiu, A., H.G. Wood, and P.E. Allaire. "Implantable Axial-Flow Blood Pump for Left Ventricular Support." *Proc. of the 45th Int. ISA Biomedical Sciences Instrumentation Symposium*, Copper Mountain, CO, April 2008.
- [13] Yin, X., J.C. Knight, and W. Weimer. "Exploiting Refactoring in Formal Verification." *Proc. of Dependable Systems and Networks*, Lisbon, Portugal, 2009.
- [14] Yin, X., J.C. Knight, E.A. Nguyen and W. Weimer. "Formal Verification By Reverse Synthesis." *Proc. of the 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2008)*, Newcastle, UK, Sept. 2008.