Software Process Synthesis in Assurance Based Development

Patrick J. Graydon and John C. Knight

University of Virginia
Department of Computer Science

Technical Report
CS-2009-10

October 28, 2009

Software Process Synthesis in Assurance Based Development

Patrick J. Graydon and John C. Knight

University of Virginia, Department of Computer Science
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 process for building both software and an argument demonstrating its fitness for use in given operating contexts. In this paper, we introduce the ABD process synthesis mechanism. 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.

1 Introduction

Assurance Based Development (ABD) [12, 20] is an approach to constructing software systems in which creation of the software is combined with explicit creation of assurance of the software in the form of rigorous argument. 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.

Driven by each software system's unique assurance needs, the ABD *process synthesis mechanism* produces, for a given system, a detailed development process for building both the software and evidence of its fitness for use. In this paper, we describe the ABD process synthesis mechanism.

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 [16]. 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. To be considered acceptable, both arguments need to be compelling.

Software engineering is about choices. 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 options. The arguments also provide a sound basis for judging each option 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 a software development 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 [18]. The standard 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 define fitness for use and describe how fitness arguments can be used to demonstrate that delivered software has all of the properties, including safety and security properties, that it must have if it is to be considered acceptable. In section 3, we define success for software development efforts, explain how traditional process models attempt to facilitate success, and how success arguments document the developer's rationale for believing that the planned process will yield success. In section 4, we present the details of ABD process synthesis. In section 5 we describe a case study evaluation of ABD and its process synthesis mechanism. In section 6 we describe the software development activity that was the subject of that case study. In section 7, we present metrics and artifacts from the development activity, in section 8 we give observation that we made during the case study, and in section 9 we present the results of our case study. Finally, we present related work in section 10 and conclude in section 11.

2 Software Fit For Use

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 customers needs or provide adequate security is unacceptable. We say that a system that demonstrably meets such a balance is fit for use.

2.1 Fitness Arguments

Each software system produced by ABD is accompanied by a *fitness argument*[1] giving the developers' rationale for believing in the main fitness claim shown below. Other researchers have referred to similar arguments as dependability arguments [8].

> **Main fitness claim:** The system is adequately fit for use in the context(s) in which it will be operated.

The notion used here of a fitness argument is deliberately more comprehensive than that of other forms of assurance argument, such as a safety argument. Because achieving an acceptable balance of stakeholder concerns is crucial, the main claim of the fitness argument is broad enough to include dependability considerations as well as functionality and any other considerations that might be said to bear on whether or a given stakeholders will find a given system acceptable.

Safety and other assurance arguments have been recorded in a variety of notations, including Adelard's graphical Claims Argument Evidence (ASCAD) notation [3] and natural language text. In the case study development reported in this paper, we have used the graphical Goal Structuring Notation (GSN) [16] to document fitness arguments. We selected GSN because the notation is in active industrial use as a means of recording safety arguments, but there is no reason why ABD could not be adapted to any suitably expressive argument notation.

2.2 Systems Versus 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 exam-

[1] In earlier work [12], we referred to the ABD fitness argument as the assurance argument.

ple, 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 — i.e. not embedded 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 must instead argue from the available evidence and from their assumptions about the operating context that the software system they produce will be fit for use in that context.

3 Software Development Effort Success

3.1 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* [11]. The role of the ABD fitness argument is to address

operational risk by forcing the developer to express a rationale for believing that a software product is fit for use. Likewise, the role of the ABD success argument is to address development risk by forcing the developer to express a rationale for believing that a planned process will yield a system that is fit for use *on time* and *within budget*.

As with a fitness argument, a success argument always has a fixed main claim:

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

We define an acceptable system as a system accompanied by an acceptable fitness argument. The meaning of the terms "acceptable time" and "acceptable cost," however, will vary from effort to effort. The developers of any given project will define these phrases in the context of their project. Like fitness arguments, we have chosen to record success arguments in the GSN notation.

Fitness arguments and success arguments, in addition to having different main claims and addressing different types of risk, evolve in different ways over the course of software development. A fitness argument, though drafted early and updated throughout development, is completed when the system is completed and its conclusions should be taken as holding from that point forward. A success argument, however, is never complete: a success argument is used continuously throughout development to evaluate the likelihood of success at any given moment and becomes moot with the delivery of the system. 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.

3.2 Traditional Process Models

The ABD process synthesis mechanism is quite different from that surrounding traditional process models such as the Waterfall model and the Spiral process model [5]. The Spiral model brought flexibility to the software process. ABD is more flexible still. As we have shown in prior work [11], the Spiral process model can itself be modeled using a success argument.

We note that ABD mandates *nothing* about the form of the process the developer synthesizes. The synthesized process might well take the form of a Spiral model or any other familiar form. All that is required is that the process support both arguments fully and that the 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.

3.3 The ABD Approach to Ensuring Success

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: (a) fit for use; and (b) completed 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.

4 Process Synthesis

ABD is based on two key concepts:

1. Engineering choices should be driven by the need to produce evidence for the assurance arguments.
2. Argument should be used to document the rationale for believing that the system is fit for use.

The core of the ABD process synthesis mechanism is illustrated in Figure 1. The input to the core mechanism is the sequence of *assurance obligations* represented by unaddressed goals in the fitness and success arguments. These obligations drive *development choices* that yield process fragments which are the output of the process synthesis mechanism. Combining these process fragments yields a complete process for building both the desired software and the necessary evidence of its fitness.

4.1 Software Development As a Sequence of Choices

Assurance Based Development 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 the software development process. Each development choice could be characterized as an answer to the question, "How can <*x*> be demonstrably accomplished?". 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 development choice. Each choice simultaneously yields a portion of the development process and evidence that the process or an artifact that results from executing it has a particular property. That evidence, collectively, forms the grounds for the ABD assurance arguments.

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

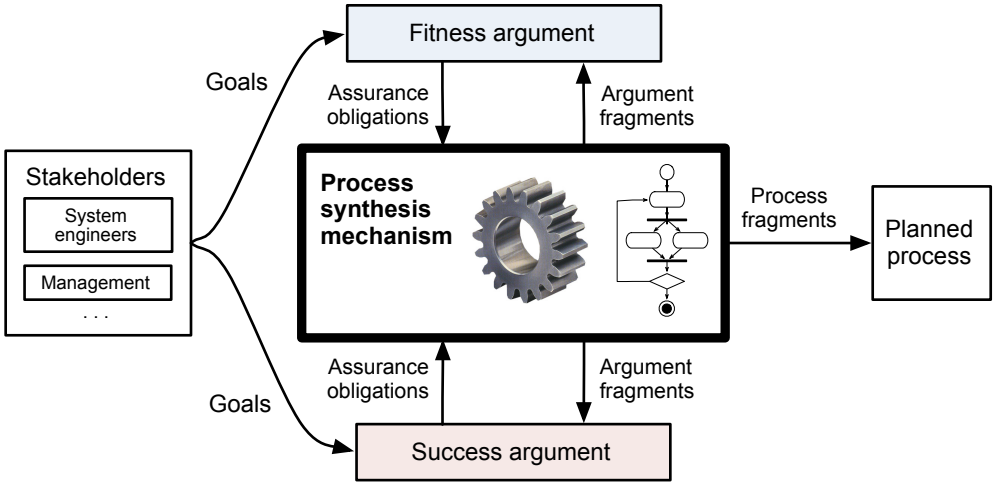

Figure 1: ABD process synthesis

will be used to develop a given software system, for example, then its selection is still a choice irrespective of whether 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) have always used that language and never conceive of using anything else. A *choice* to use that particular programming language has been made, even if implicitly.

By contrast, the choices that are made in a given ABD software development effort are explicit. These choice and the order in which they are made together determine the software development process for that effort. The synthesized 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.

4.2 The Process Synthesis Mechanism

Informally, process 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 process is derived from the choices, and synthesis is complete when no further refinement of either argument is needed.

The ABD process synthesis mechanism is effected as a step performed multiple times by one or more developers. In each step, the developer performing the synthesis considers the state of the assurance argu-

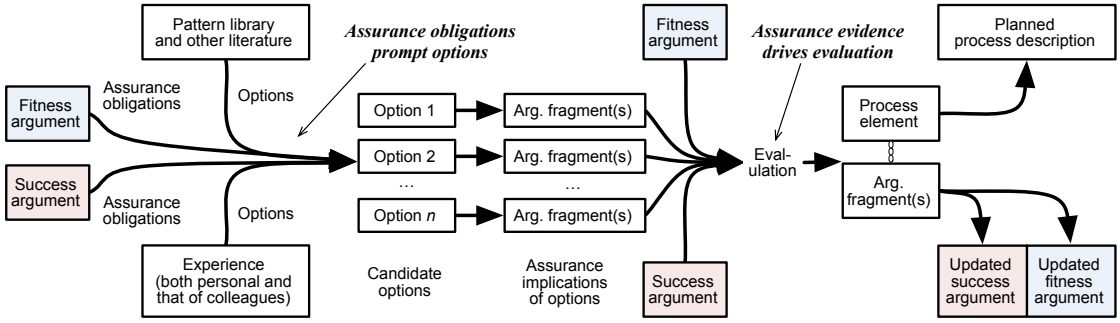

Figure 2: A process synthesis step in ABD

ments, selects an obligation to address, assembles a set of options, evaluates these based on the argument fragment(s) that each supplies, makes a development choice that will yield the necessary evidence, and modifies the planned process and the assurance arguments accordingly. Fig. 2 illustrates this procedure.

At any time during process synthesis, the unaddressed goals in the evolving assurance arguments represent assurance obligations that the developers must 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 planned process. During process synthesis, choices can be made, changed, and updated in any order that reflects their overall impact on the fitness and success 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.

Once a developer has selected an assurance obligation to be satisfied, he or she then sets about gathering a set of options 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 [12], the developer gathers options 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 propose options. Each option is then assessed using a set of criteria including whether the option supports or precludes achieving needed functionality, the likely restrictions it imposes on later choices, the costs it imposes, its feasibility, applicable standards, and any relevant non-functional requirements.

Evaluating an option 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,

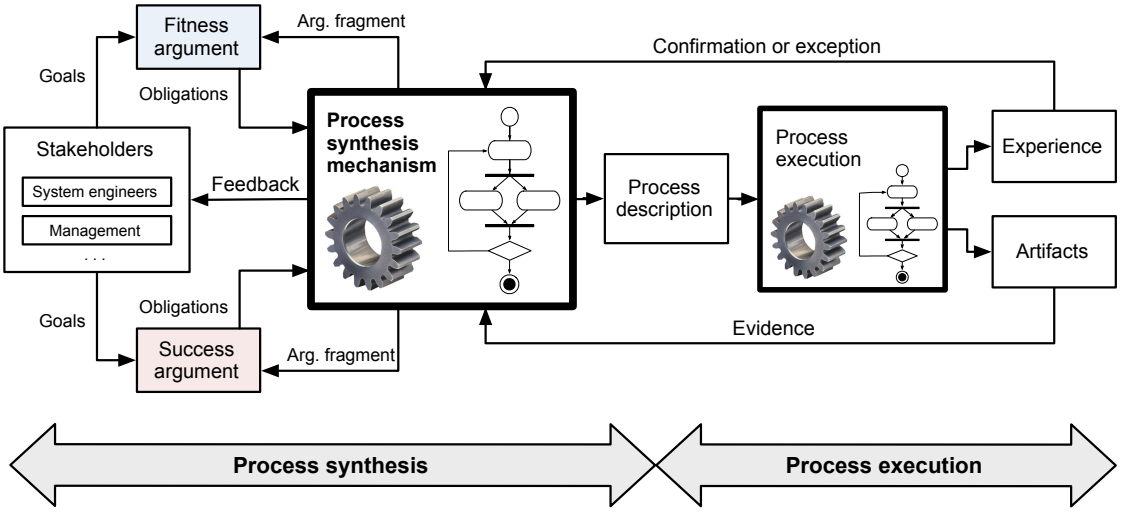

Figure 3: Synthesizing and executing a process in ABD

may preclude the subsequent use of static analysis techniques that rely on certain language features. A developer need not enumerate all plausible options or consider only options that assess each choice perfectly. Enumerating and evaluating alternative options 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 that the choice yields are added to the two ABD assurance arguments and the associated process element is added to the planned 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.

4.3 The Complete ABD Approach

In ABD, process synthesis operates concurrently with execution of the process. The complete ABD approach is illustrated in Figure 3. ABD concurrency is show in the figure through the feedback paths: as the execution of the planned process results in artifacts and experience, the process synthesis mechanism captures this evidence and incorporates it into the assurance arguments.

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 process. The 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 their process contribution will not yield the expected evidence. For example, static proof of freedom from memory leaks might prove to be infeasible because the software complexity makes comprehensive static analysis intractable. If executing the planned process results in the expected evidence, the evidence placeholder in the arguments is replaced with the actual evidence. If not, ABD accommodates the unexpected result via the *process repair mechanism* (discussed in section 4.4).

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 supports process execution before a complete process is synthesized in order to accommodate such cases.

4.4 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 all three 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 identifying the choice, enumerating alternatives, selecting one, and then modifying both the planned process description and the two assurance arguments.

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.

5 A Case Study of ABD

5.1 System Studied

In order to assess ABD process synthesis, we have conducted a case study development of a specimen safety-critical system, the University of Virginia's *LifeFlow* Left Ventricular Assist Device (LVAD) [21]. LifeFlow

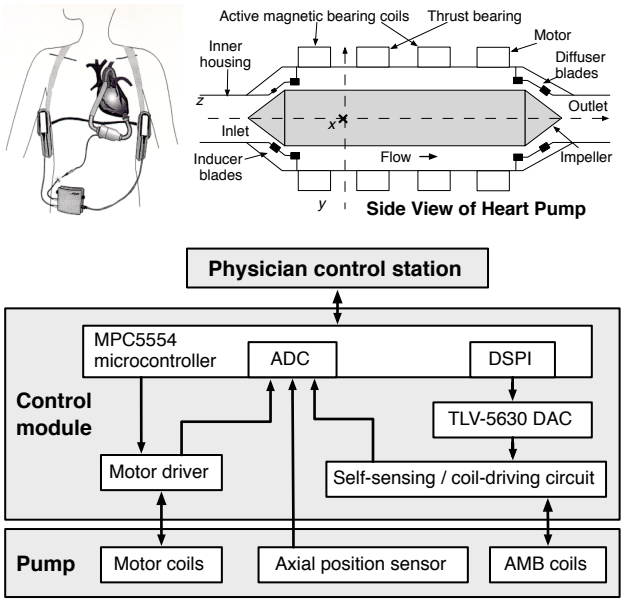

Figure 4: LifeFlow structure and use

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 dangerous blood clots.

Control of the magnetic suspension bearings is provided, in part, by a digital control algorithm running on a microcontroller. In hard real-time, the controller 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 4 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 magnetic bearing control software.

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-

Table 1: Magnetic bearing control software requirements

Functionality	1. Trigger and read ADCs to obtain impeller position vector $\vec{u}$. 2. Determine whether reconfiguration is necessary, and if so, select appropriate gain matrices **A**, **B**, **D**, and **E**. 3. Compute target coil current vector $\vec{y}$ and next controller state vector $\vec{x}$: $\vec{y_k} = \mathbf{D} \times \vec{x_k} + \mathbf{E} \times \vec{u_k}$ $\vec{x_{k+1}} = \mathbf{A} \times \vec{x_k} + \mathbf{B} \times \vec{u_k}$ 4. Update DACs to output $\vec{y}$ to coil controller.
Timing	This functionality must be provided in hard realtime with a frame rate of **5 kHz**.
Reliability	No more than 10^{-9} failures per hour of operation.

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 (MBCS) 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.

5.2 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 options 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 option was made based on this criterion?" and "Was assessment of the options 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 The LifeFlow MBCS Software Development Activity

Using ABD and following the case study protocol described in section 5.2, we created a planned process and executed it to create the UVA LifeFlow Magnetic Bearing Control Software (MBCS). The major milestones for our effort were:

1. **Evolutionary Prototype 1.** This prototype, developed while we waited for a complete requirements draft from the LifeFlow systems engineering team, included an implementation of control computation in the correct form and was subjected to thorough formal verification.

2. **Evolutionary Prototype 2.** This prototype, developed after details about the target computer system had arrived but before final control constants were available, allowed us to check the I/O instruction sequences provided with the requirements.

3. **Final Deliverable.** The final deliverable consisted of software suitable for use with as part of the LVAD First Prototype in laboratory and animal testing.

Process execution began with planning tasks, such as estimating task durations and recording tasks, durations, and dependencies in the project schedule document we decided to create. Execution finished when the last process step had been completed and the fitness argument was completed with the integration of all referenced artifacts.

Space considerations preclude us from presenting all results including each of the development choices we made, their effects on the assurance arguments, the options at our disposal, the rationale behind the selected choice, the result of executing the synthesized process fragment, and the details of the temporal interleaving of process synthesis and execution. Instead, we summarize the development process below, presenting details of selected choices and other events in the order in which they occurred so as to illustrate ABD and its application to the LifeFlow MBCS.

6.1 Arrival: First Draft of Requirements

Our case study began when the LVAD systems engineers delivered the first draft of the MBCS requirements. These requirements define the fitness contract between the MBCS and the larger LifeFlow LVAD system. In so doing, they define fitness for use: the MBCS is fit for use in the context of the LifeFlow LVAD if it meets the documented requirements imposed upon it by the LifeFlow LVAD.

The first draft of the requirements document described the form that the control equations would eventually take, the approximate speed at which they must be computed, and the general form of the system in which the software would be used. While the draft requirements document lacked critical details such as the target hardware platform, the exact control constants, and the applicable dependability requirements, it nevertheless provided a basis upon which to begin ABD development.

6.2 Synthesis: DC-001 / Elaborating the Argument Context

The first development choice we made was to elaborate the context for the ABD assurance arguments. This choice is obvious, and we present it because we expect that many ABD projects will begin with this choice.

Obligations: The unsupported main fitness and success claims.

Options Considered:

1. Elaboration of the context.

Reasoning: This option was deemed acceptable.

Table 2: Context for magnetic bearing control software

System	The magnetic bearing control software.
Operating context	The system is a component of the LifeFlow LVAD First Prototype.
Requirements	Requirements imposed by the LifeFlow LVAD First Prototype are recorded in <*path of file*>.
Acceptable time	By the LifeFlow LVAD First Prototype delivery date.
Acceptable cost	Presently available resources and staff plus target hardware costs.

Process Fragment Contributed:

1. Elaborate the context for the ABD assurance arguments by defining the terms used in the standardized main fitness and success claims and linking to the provided requirements.

This process fragment was executed immediately after it was synthesized.

Argument Fragments Contributed: Execution of the process fragment produced by this choice resulted in the creation of GSN context elements in both the fitness argument and success argument. The content of these elements is summarized in Table 2.

6.3 Synthesis: DC-002 / Review Argument Before Execution

The second development choice we made was to require that the success argument be reviewed and found to be compelling before releasing process elements for execution. We present this choice because the the observations we made during this process synthesis activity highlight the concerns that govern process execution and the interaction between process synthesis and process execution.

Obligations: The unsupported main fitness and success claims.

Options Considered:

1. Require an acceptable success argument prior to beginning process execution.

This process synthesis step began with a suggestion: why don't we impose this rule? We reflected upon the problem this suggestion was meant to solve and determined that there was a development risk associated with starting process execution before we could justifiably be confident in the synthesized process: if

we discovered that our early decisions had created an unsatisfiable assurance obligation, we would have to revise those decisions, potentially wasting effort. While reflecting upon the question clarified the nature and impact of the proposed option, it did not cause us to consider alternative options.

Reasoning: During ABD option evaluation, we noted that it might be necessary to execute certain process elements before completing process synthesis in order to generate the information needed to synthesize an adequate process. That is, we might need to perform simple experiments or construct throwaway prototypes in order to help us choose between options. We accepted the option after modifying it to address this concern.

Process Fragment Contributed:

1. Review the success argument.
2. Do not execute any process element before the success argument has been reviewed and deemed acceptable *unless completion of the element is necessary to drive process synthesis*.

Argument Fragment Contributed: A success argument fragment citing the new policy as evidence of adequate mitigation of the risk that effort will be wasted by following an inadequate development plan.

6.4 Elided Events

For brevity, we omit details of the third development choice. This choice resulted in both the creation of a software development schedule document into which process steps could be recorded and in process tasks to estimate task durations and populate the schedule. The existence of the schedule, in turn, provided evidence used as part of an argument that the system would be completed in adequate time.

6.5 Synthesis: DC-004 / The Use of SPARK Ada

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. We present the details of this choice in order to illustrate how the ABD process synthesis mechanism functioned in practice near the beginning of our development effort.

Obligations:

1. Show that the delivered system satisfies its requirements.
2. Show that the risk that details missing from the requirements will not be made available in time for the effort to succeed has been adequately mitigated.

3. Show that the risk that the dependability goals might be impossible to meet successfully has been adequately mitigated.

4. Show that the risk that the real-time goals might not be demonstrably achievable has been adequately mitigated.

We chose, arbitrarily, to address the first obligation.

Options Considered:

1. Use a refinement approach such as the B method [1].

2. Develop a formal specification in Z [19].

3. Develop a formal specification in PVS [19].

4. Adopt the SPARK Ada programming language [4] and its associated tools to prove that the implementation complies with the low-level specification embodied in the SPARK-annotated subprogram contracts.

5. Use the Echo approach [22, 23] to formal verification. In the Echo approach, one: (a) constructs a high-level, *abstract specification* in a language such as PVS; (b) constructs a software *implementation* and a low-level specification of its behavior such as an annotated SPARK Ada program; (c) constructs an *implementation proof* showing that the source code refines the low-level specification using, for example, the SPARK tools; (d) extracts a high-level specification from the low-level specification and code using Echo tools; and (e) establishes a formal *implication proof* that the *extracted specification* refines the abstract specification using a mechanical proof checker such as PVS. The implementation proof and implication proof, taken together, form a complete verification argument showing that the code refines the high-level specification. This approach is illustrated in Figure 5.

None of these options would have addressed our goal in its entirety. The development of a formal specification, for example, might address a small part of **G_ReqSatisfied** by reducing the risk that a misunderstanding would give rise to an executable that did not satisfy the requirements, but cannot, by itself, support that goal. There were many other options we could have considered that would at least partially address the goal. We drew this list of options mainly from our experience and our interest in formal methods, but we might have considered options such as various forms of testing. We proceeded with this set of options despite its limitations because we thought it was likely to contain at least one suitable option.

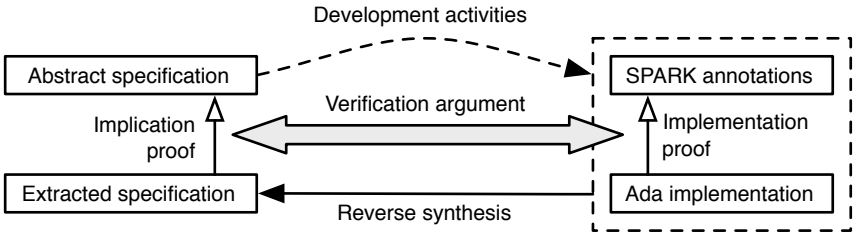

Figure 5: The Echo formal verification process

Reasoning:

- **Option 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.
- **Option 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.
- **Option 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.
- **Option 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.
- **Option 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.

We chose option 4 because 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.

Process Fragment Contributed:

1. Select and procure a compiler.
2. Build low-level specification in the form of annotated SPARK Ada subprogram declarations.
3. Implement subprogram bodies.
4. Use the SPARK Examiner to: (a) demonstrate freedom from flow errors; (b) show compliance with SPARK language restrictions; and (c) generate Verification Conditions (VCs) sufficient to show freedom from exception and compliance with low-level spec.
5. Use the SPADE Simplifier to automatically discharge VCs.
6. Build custom Simplifier rules as necessary.
7. Use the prover to prove custom Simplifier rules and VCs not discharged by the simplifier.
8. Use the POGS tool to compile a report on the disposition of each VC, and use this report to determine when all VCs have been discharged.

Argument Fragment Contributed: The fitness argument fragment resulting from this choice is shown in Figure 6.[2]

In the figure, the diamond decoration on the strategy element **ST_ArgOverRefinement** 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, including support for claims that: (a) the specification refines the requirements; (b) the low-level specification refines the specification; and (c) the executable refines the source code.

The diamond decoration on solution **S_POGSReport** indicates that the evidence represented by that solution is forthcoming. When the SPARK proofs had been completed and the report was available, we removed the decoration. If the proofs could not have been completed as expected, we would have had to repair our planned process and fitness argument.

6.6 Elided Events

For brevity, we omit the details of the next few development choices. These include the choice to perform integration testing, the choice to perform requirements-based functional testing with MC/DC structural coverage [6], and to use functional decomposition to derive the software design.

[2]We initially inserted an oversimplified argument, which we later expanded through the repair process. The figure depicts the repaired argument.

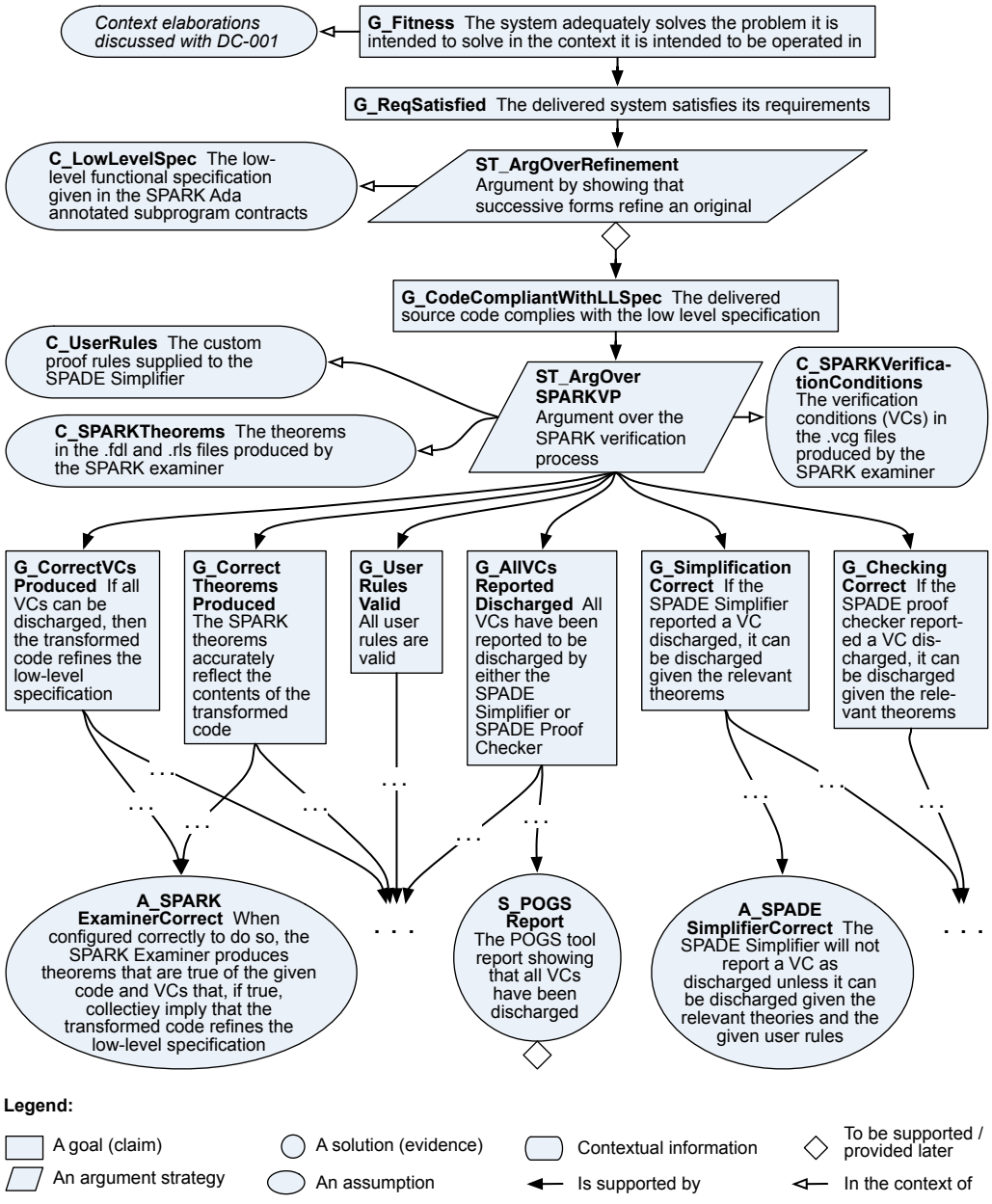

Figure 6: Assurance contribution from choice DC-004

6.7 Synthesis: DC-007 / WCET

At this point in process synthesis, we had made development choices that partially addressed **G_Req-Satisfied** by providing evidence that the delivered software would be functionally correct. We present the details of DC-007 and DC-008 both because they illustrate how we treated the belated discovery of an implicit choice and because they are examples of development choices made after the initial choices had established the broad outlines of the fitness and success arguments.

Obligations:

1. A success argument goal to show that the timing requirements could be met demonstrably.
2. A related fitness argument goal to show that the real-time requirements would be met.
3. Other obligations that we deemed less pressing.

Options Considered:

1. Analyze the worst-case execution time (WCET) 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.

Reasoning:

- **Option 1:** WCET analysis would supply strong evidence that the hard real-time deadlines would be met.
- **Option 2:** This option was deemed unacceptable because it would force us to demonstrate both that: (a) re-issuing the last frame's outputs would be done rarely, and (b) that doing so that rarely would be sufficient to keep the impeller from striking the pump's inner housing.

Accordingly, we chose option (1).

Process Fragment Contributed:

1. Select and procure a WCET tool.
2. Use this tool to perform WCET analysis of the control calculation.

The second task was a placeholder. When the first task was executed and a tool selected, we returned to the process synthesis mechanism to replace this task with detailed tasks specific to the selected tool.

Argument Fragment Contributed: Making this choice generated a fitness argument fragment that cited a report from the as-yet-unspecified tool to support a claim that the control calculations would always complete within their allotted time bounds. Like the second task in the contributed process fragment, this argument fragment was clarified after the WCET tool had been selected.

Integrating this fragment into the fitness argument required a change to the structure shown in Figure 6: 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 real-time and non-real-time requirements. The argument fragment supporting **ST_ArgOverRefinement** shown in Figure 6 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.

6.8 Synthesis: DC-008 / Real-Time Structure

The process of formulating argument fragments related to DC-007 raised two important questions: (i) of what, exactly, were we going to measure the WCET; and (ii) how would establishing that WCET contribute to knowing that the real-time deadlines would be met? These questions prompted us to realize that we had implicitly chosen to structure the software in the form of a cyclic executive. That is, we had chosen to construct a single-threaded application containing one main loop operating as a fixed-rate real-time frame, with individual tasks performed either in every iteration or in every n^{th} iteration as their scheduling needs dictate.

Obligations: Explicitly consider the real-time structure of the MBCS as our next process synthesis step.

Options Considered:

1. Use a cyclic executive design.
2. Use a real-time operating system, to be chosen later, with the control computation implemented as a task.
3. Use a concurrent design based on Ravenscar Profile tasking in SPARK Ada.

Reasoning:

- **Option 1:** We reasoned that such a structure was feasible but might complicate the implementation of a low-priority, non-real-time task such as logging, if one were to be later introduced. (We would need to divide the implementation of such a task into pieces that each demonstrably fit within the real-time frame.)

- **Option 2:** We reasoned that this option might be infeasible if no suitable operating system was available for the target hardware (which had not yet been chosen), and we noted that it might restrict our choice of compilers. We also noted that this choice would bring derived dependability requirements: we would need to show not only that the chosen OS would guarantee the necessary real-time properties but that it would not interfere with the functionality of the task implementation that we would provide.
- **Option 3:** We reasoned that this option might restrict our choice of compilers to those that provided a demonstrably suitable implementation of the necessary portions of the Ada run-time library.

Of these, we elected to remain with our original choice, option (1).

Process Fragment Contributed:

1. Design a schedule for the cyclic executive, specifying its frame rate and the conditions under which each task would be executed on a given iteration.
2. Add the cyclic executive structure to the specification.
3. Implement the cyclic executive.

Argument Fragment Contributed: Given choice DC-007, the making of choice DC-008 contributed two new sub-arguments to the success argument: (a) if the microcontroller is sufficiently fast and if it is possible to schedule new tasks brought about by requirements change, 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 it should be possible to demonstrably meet the real-time requirements. The new sub-goals in these fragments prompted a 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.

6.9 Elided Events

We omit the details of the next few events. Briefly:

1. We chose to use carefully-reviewed argument to show that our specification refines the given requirements.
2. We chose to use the Echo verification process described in section 6.5.
3. When complete requirements were not delivered on schedule, we chose to construct Evolutionary Prototype 1.

4. We chose to prepare advice for the systems engineering team regarding processor selection and did so.

5. We reviewed the arguments and repaired them accordingly. The repairs consisted of clarifications and the addition of an unsupported sub-goal to represent evidence found to be missing.

6. We chose to assume, for the purposes of the prototype, that single-precision floating-point arithmetic is sufficient.

7. We constructed a formal specification of the control calculations in PVS.

8. We implemented the control calculations in SPARK Ada.

9. We performed basic testing on the prototype executable.

10. We completed Echo formal verification of the control calculations.

6.10 Delivery: Evolutionary Prototype 1

At this point in the development effort, we had successfully completed development of a prototype that included formally-verified control calculations in the correct form. This prototype, in turn, became evidence for our success argument. Successful completion of the process elements involved in the prototype's construction suggested that we would be able to complete similar process elements when building the final software.

6.11 Elided Events

After we completed Evolutionary Prototype 1, the systems engineering team delivered an updated requirements document. Additions to the document included the specification of Freescale's MPC5554 microprocessor [10] but not the final control constants. As a result, we made development choice DC-014, opting to proceed with construction of a second evolutionary prototype that would demonstrate our ability to target the selected processor. Following choice DC-014, briefly:

1. We chose to use AdaCore's GNAT Pro High Integrity Edition Ada compiler [2].

2. We chose to search for an appropriate WCET tool.

3. We chose to conduct a small experiment to gauge the likely size of the final executable and did so.

4. We repaired the success argument's treatment of development risks related to insufficiently fast or capacious memory.

5. We chose the layout of executable segments in the processor's various memories.

6. We repaired the success argument to add a goal of adequate mitigation of the risk of error in the hardware I/O details provided in the new requirements draft.

7. We repaired the fitness argument's treatment of assurance to include a new goal: assurance that the executable will write to hardware registers in the prescribed order.

6.12 Choice: DC-019 / Inspection of Register Write Order

Repair of the fitness argument resulted in the creation of an unsupported subgoal, **G_ControlRegsWrittenInCodeOrder**. We present the details of development choice DC-019, which addressed this goal, as an example of choice made toward the end of the development effort. Unlike process synthesis near the beginning of the project, in which the assurance obligations we addressed were broad and general, the assurance obligation we faced during later choices were quite specific.

Obligations: We mapped variables in our SPARK Ada program to the registers controlling the microcontroller's ADC and other peripheral units, and marked these with the `Atomic` pragma. Unfortunately, while the language semantics guarantee that the writes to each variable will be atomic and will not be reordered, they do not guarantee that writes to different registers controlling the same peripheral will not be re-ordered with respect to each other. Checking revealed that our compiler was not re-ordering these writes. However, we needed a guarantee of this property.

Options Considered: We considered only one option: inspection of the disassembled binary. Had inspections proven unsatisfactory, we might have considered alternatives such as writing all interactions with memory-mapped registers in hand-coded assembly language.

Reasoning: We deemed this option acceptable.

Process Fragment Contributed: As a result of this choice, we modified our planned process to: (1) call for the use of the `objdump` tool to disassemble the compiled binary; and (2) include a formal inspection of the disassembly to confirm the write order.

Argument Fragment Contributed: This choice contributed the fitness argument fragment shown in Fig. 7. Because the disassembly is correct and inspection shows that the write order in the disassembly is correct, we believe that the write order in the compiled binary is correct. The strength of this belief rests upon confidence in the adequacy of the inspection protocol (**G_InspProtocolAdqt3**) and upon confidence in the disassembly tool and our use of it (**G_DisassemblyCorrect**).

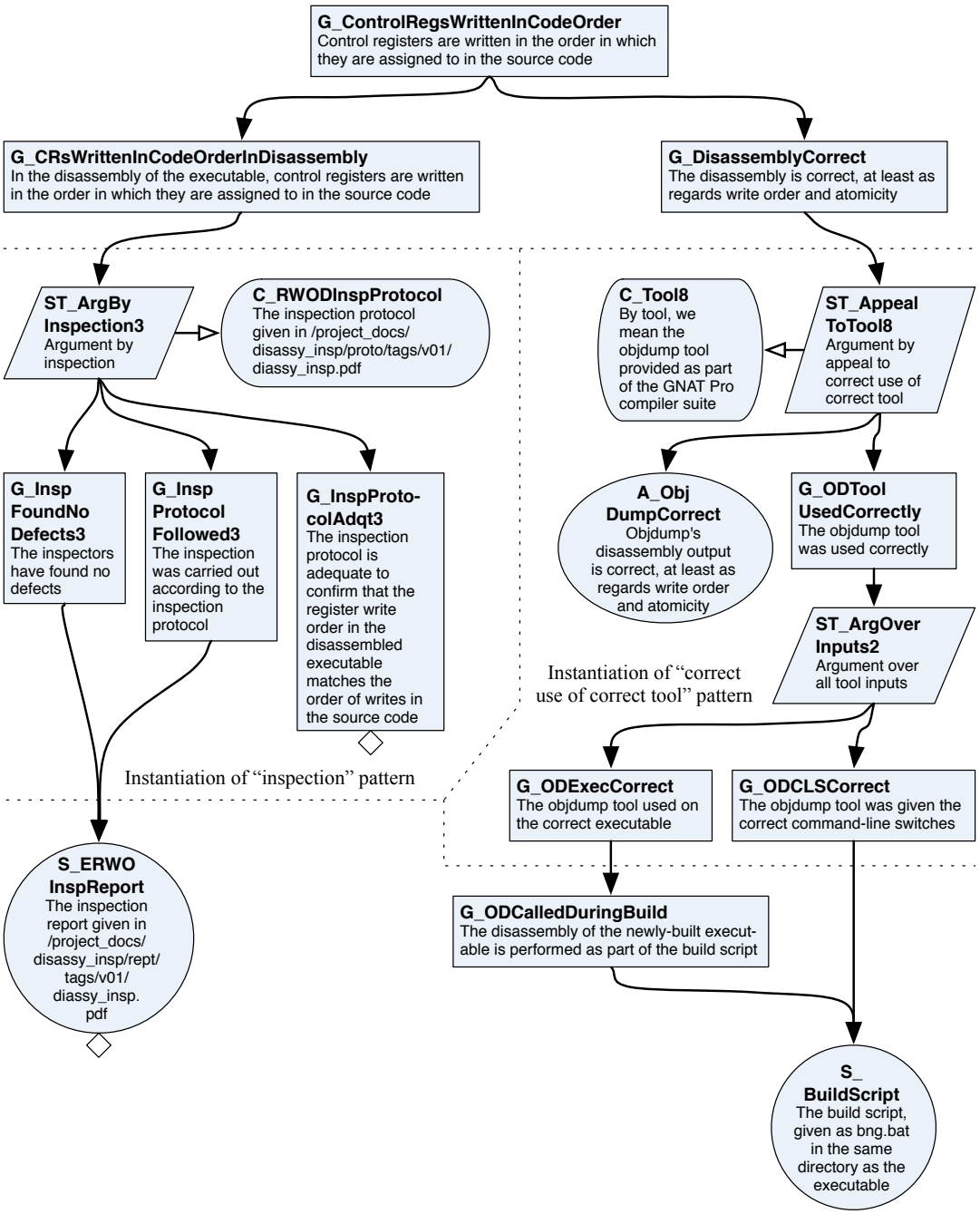

Figure 7: Sub-argument demonstrating memory-mapped register write order

6.13 Elided Events

We again omit the details of the next few events. Briefly:

1. We discovered an issue that precluded debugging any executable larger than 1.5 KiB and asked our tool vendors for help in isolating the cause of that issue.
2. We revised the goals of Evolutionary Prototype 2 to enable further progress despite our debugging difficulty.
3. We completed a miniature prototype capable of serial communication with the DAC.
4. We discovered errors in the hardware I/O details given in the requirements document and obtained a corrected version from the systems engineers.
5. With the help of our tool vendors, we discovered the cause of our debugging issue and implemented a solution.
6. We revised again the goals for Evolutionary Prototype 2 in light of the lifted restriction.

6.14 Delivery: Evolutionary Prototype 2

At this point in the project, we had completed construction and evaluation of Evolutionary Prototype 2. The completion of this prototype offered evidence for use in the success argument. Because the prototype included the hardware I/O functionality that the final system would have, its completion provided evidence that development risks related to the correctness of hardware I/O details in the requirements and to implementation of these in SPARK Ada had been adequately mitigated. In addition, because the program texts of Evolutionary Prototype 2 and the final system were not expect to differ in a way that would substantially contribute to code size or run time, successful completion of the prototype also provided evidence that development risks related to speed and memory size were unlikely to manifest.

6.15 Elided Events

Much of the activity that followed the release of Evolutionary Prototype 2 consisted of repair activity, and much of that was in the nature of clarification. Having developed a broad approach that had produced working prototypes and demonstrated the feasibility of our chosen formal verification strategy, we focused on perfecting the fitness argument. Where the fitness argument was vague, we clarified. Where detail was missing or assumed, we added detail. Where the argument could make use of an argument fragment used elsewhere, we repeated the familiar fragment so as to make the argument structure easier to parse. These

changes resulted in many new argument elements added near the leaves of the hierarchical structure. The fitness argument grew dramatically but retained its general high-level form.

The choices made during this time added detail to the general process I had already synthesized. They were:

1. The choice of *which* WCET tool to use, replacing a "TBD" in the planned process.
2. The choice to formally document the configuration management procedure that we had been using as a matter of course, taking particular care with features that we now knew to provided critical fitness evidence.
3. The choice to use inspection to confirm the correctness of the frame synchronization logic given in our specification.
4. The choice to use inspection to verify that specific inputs to the WCET tool are correct.
5. The choice to use inspection to confirm that the right kind of floating-point arithmetic is used. (Formal verification shows that the computation would be correct if real-valued arithmetic was used in lieu of floating point; the inspections show that we are using floating-point arithmetic in a reasonable way as a substitute for perfect arithmetic.)
6. The choice to rely solely upon testing to confirm the correctness of logging features that were added to facilitate field debugging. (Formal verification confirms that the logging implementation will not interfere with other functionality, but does not show that logging itself functions correctly.)
7. The choice to use a hardware probe to capture test execution traces non-intrusively in order to provide evidence of the structural coverage of the functional test plan.

6.16 Delivery: Final Deliverable

Completion of the planned development activities (with some exceptions discussed in Section 7.6) has yielded both a fully-functional implementation of the MBCS and a full formal verification of that implementation. This implementation will be turned over to the systems engineering team for integration testing and use in the LifeFlow LVAD First Prototype.

7 Metrics and Artifacts

7.1 Development Choices And Repairs

During the course of development, we made a total of 27 development choices, and invoked the repair process a total of 44 times. The spacing of these activities over time is shown in Figure 8.

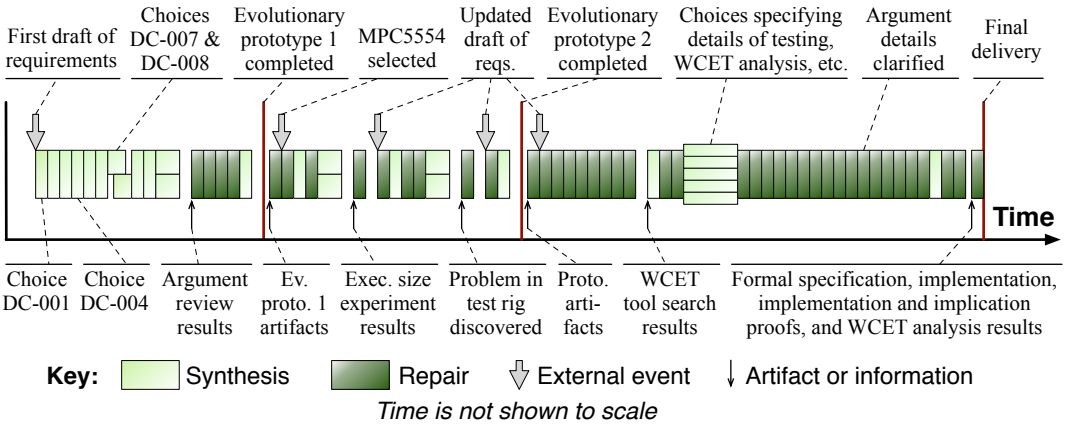

Figure 8: Development choices and repairs

The figure depicts several choices as though they were made in parallel by developers working independently. These choices were actually made in sequence by one developer. Our case study protocol, however, directed us to consider whether each choice *could* have been made in parallel with the previous choices. Choices are shown in parallel in the figure where our response to the study question indicated that parallel decision-making was possible. Since the study questions do not include a question about repair activities being possible in parallel, all repair activities are shown sequentially even though it is likely, in our opinion, that some of them could have been made in parallel.

Choices were more frequent than repairs at the beginning of development. This relationship reversed near the end of the project as we refined and clarified our arguments. Choices that could only be made sequentially were common near the beginning of the project but became rare later on. We hypothesize that this is a result of the changing nature of choices: the first choices suggest broad, partial solutions to abstractly-stated problems, whereas later choices provided narrow, tailored solutions to very specific, isolated problems.

7.2 The Formal Specification

We created a specification to serve as the basis for implementation and formal verification. This specification contained:

- A formal portion written in PVS. The content of the formal portion is summarized in Table 3.
- An informal portion written in natural language. The informal portion describes the mapping between formal variables and signals visible at the computer interface, specifies how signals are to be

Table 3: Content of the PVS portion of the MBCS specification

	Lines
Blank and comment lines	87
Control and other constants	57
Type declarations	35
Needed arithmetic theorems	17
Input and output conversion	32
Control calculation	23
Real-time frame synchronization	24
TOTAL	275

sampled or emitted, defines the state that the hardware should be initialized to, and provides an informal explanation of the formal specification.

7.3 Implementation Source Code

The MBCS implementation consisted of:

- **SPARK Ada source code.** The implementation contains 2,510 lines of SPARK Ada, distributed as shown in Table 4. Of the 579 total lines implementing the control calculations, 127 contain control constant declarations. Of the 1,185 lines implementing the hardware interface, 739 contain the register bitfield type declarations, memory mapped variable declarations, and associated pragmas. Some of the remaining bulk of the hardware interface is a result of writing constant declarations so that each bit field in the constant is named and given a value on a separate line.

- **Library routines in non-SPARK Ada.** The GNAT Pro compiler generates calls to `memcpy` and `memset`. Since we are not deploying a runtime library containing implementations of these routines, we have provided our own implementation. Because the implementation uses Ada access types, it is not legal SPARK and cannot be analyzed by the SPARK tools.

- **A startup routine in PowerPC assembly language.** The startup routine, which configures the MPC5554's external bus, consists of 106 instructions over 133 lines and contains no loops or conditional expressions.

Table 4: Measures of the SPARK Ada implementation

	Control calculation	Main program / cyclic executive	Hardware interface	Logging	Support	TOTALS
Code lines	309	47	1,185	92	34	1,667
Comment & annotation lines	198	46	268	16	28	556
Blank lines	72	21	168	23	3	287
Total lines	579	114	1,621	131	65	2,510
End of line comments	0	0	211	0	0	211
Public types	6	0	1	0	32	39
All type definitions	22	0	21	6	32	81
Public sub-programs	1	1	11	1	0	14
All sub-program bodies	5	3	12	1	0	21
All statements	83	22	101	19	0	225
All declarations	47	7	323	34	0	411
Logical SLOC	146	29	424	53	33	685

Table 5: Implementation proof VCs and how these were discharged

	Proved by or using				
	Examiner	Simplifier	Proof checker	Review	TOTAL
Assert or post	12	13	13	16	54
Runtime check	0	92	0	1	93
Refinement VCs	19	0	0	1	20
TOTALS	31 (19%)	105 (63%)	13 (8%)	18 (11%)	167

7.4 The Implementation and Implication Proofs

The SPARK Ada implementation proofs consisted of 167 verification conditions produced by the SPARK Examiner. We discharged these using the methods shown in Table 5. The 18 VCs discharged by review consisted of: (a) 16 checks that the value in a control register conforms to the type of the variable bound to that register; (b) a check that required knowledge of the hardware floating-point–to–integer conversion semantics; and (c) a VC checking that a hardware input routine was equivalent to the artificial proof function created to represent it in the Echo process.

The Echo implication proof, checked by PVS, shows that the low-level specification embodied in the SPARK annotations complies with the PVS specification. The proof is divided into 54 formulas: 19 type correctness conditions (TCCs) and 35 implication lemmas. These formulas were proved successfully in 210 seconds on a dual 1 GHz machine with the following exceptions:

- A TCC and two implication lemmas related to the time type. We use a `mod 2 ** 64` type in Ada to represent time values derived from the MPC5554's 64-bit time base. This type, however, is given as an unbounded integer in the specification. The difference is not problematic as the time base will not "wrap around" in centuries of operation.
- A TCC related to the type of the cells in the controller input, state, and output vectors. These are represented as single-precision floating-point variables in the implementation but as real numbers in the specification. The LifeFlow LVAD control engineers assure us that they can prove that these values will not exceed the limits of single-precision floating-point storage.

7.5 The Fitness and Success Arguments

The fitness argument grew over the course of development from the single, unsubstantiated, top-level goal mandated by the ABD process to a final total of 350 GSN elements as shown in Table 6 and Table 7. The widest argument step in the final argument derived support for one goal from 5 child elements. The longest support path from a solution or assumption to the final argument's top-level goal was 26 elements.

Over the course of this effort, the success argument grew from its ABD-mandated top-level goal to a peak size of 50 GSN elements before becoming moot with final delivery. Its widest argument step — 10 elements — consisted of an argument over all enumerated development risks.

7.6 Artifacts Not Completed

Due to time and budget constraints, we did complete all development activities. The purpose of this case study was to evaluate ABD process synthesis, not to obtain a software artifact fit for use with a human patient. Accordingly, we synthesized a process fit for a software development team with more resources and forwent executing some of the costly or time-consuming process steps that we synthesized. In particular:

Table 6: Success Argument Metrics

Revision	All elements	Gs	STs	Cs	Ss	As	Js	Uninstan-tiated	Unsup--ported	Max. depth	Max. branching
	1	1	0	0	0	0	0	0	1	1	
1	6	1	0	5	0	0	0	0	1	1	
2	10	4	0	5	1	0	0	0	2	4	2
3	14	7	0	6	1	0	0	0	3	4	2
4	16	9	0	6	1	0	0	0	5	4	4
5	20	13	0	6	1	0	0	0	9	4	8
6	20	13	0	6	1	0	0	0	9	4	8
7	20	13	0	6	1	0	0	0	9	4	8
8	21	14	0	6	1	0	0	0	10	4	8
9	22	14	0	6	1	1	0	0	10	4	8
10	22	14	0	6	1	1	0	0	10	4	8
11	25	15	0	7	1	2	0	0	10	4	8
12	34	17	1	8	2	6	0	0	14	6	8
13	39	20	1	8	4	6	0	2	15	6	8
14	39	20	1	8	4	6	0	2	13	6	8
15	43	20	1	8	4	10	0	2	12	6	8
16	42	18	2	9	4	9	0	0	5	7	8
17	40	18	2	9	2	9	0	0	7	8	9
18	40	18	2	9	2	9	0	0	7	8	9
19	44	19	2	10	3	10	0	1	8	8	9
20	44	19	2	10	3	10	0	0	6	8	9
21	45	19	2	10	4	10	0	0	5	8	9
22	46	20	2	10	4	10	0	0	6	8	10
23	49	21	2	10	5	11	0	0	6	8	10
24	49	21	2	10	5	11	0	0	6	8	10
25	52	23	2	11	5	11	0	0	7	8	10

Table 6: Success Argument Metrics

Revision	All elements	Gs	STs	Cs	Ss	As	Js	Uninstan-tiated	Unsup--ported	Max. depth	Max. branching
26	50	25	2	11	5	7	0	0	4	10	10
27	50	25	2	11	5	7	0	0	4	10	10
28	50	24	2	11	6	7	0	0	4	9	10
29	50	24	2	11	6	7	0	0	4	9	10
30	50	24	2	11	6	7	0	0	4	9	10
31	50	24	2	11	6	7	0	0	4	9	10
32	50	24	2	11	6	7	0	0	4	9	10
33	50	24	2	11	6	7	0	0	4	9	10
34	49	24	2	11	6	6	0	0	4	9	10
35	49	24	2	11	6	6	0	0	4	9	10

Table 7: Fitness Argument Metrics

Revision	All elements	Gs	STs	Cs	Ss	As	Js	Uninstan-tiated	Unsup--ported	Max. depth	Max. branching
	1	1	0	0	0	0	0	0	1	1	
1	4	1	0	3	0	0	0	0	1	1	
2	4	1	0	3	0	0	0	0	1	1	
3	4	1	0	3	0	0	0	0	1	1	
4	4	1	0	3	0	0	0	0	1	1	
5	13	5	3	4	1	0	0	0	5	8	2
6	21	9	4	5	3	0	0	0	8	8	2
7	25	11	5	6	3	0	0	0	10	9	2
8	28	12	5	6	4	1	0	0	12	9	2
9	32	14	5	7	5	1	0	0	16	9	3
10	34	18	3	7	4	2	0	0	16	8	3

Table 7: Fitness Argument Metrics

Revision	All elements	Gs	STs	Cs	Ss	As	Js	Uninstan-tiated	Unsup--ported	Max. depth	Max. branching
11	33	17	3	7	4	2	0	0	16	8	3
12	39	19	4	8	5	3	0	0	17	9	3
13	34	17	3	8	4	2	0	7	14	8	3
14	41	20	4	9	5	3	0	9	16	8	3
15	41	20	4	9	5	3	0	9	16	9	3
16	34	16	2	8	5	3	0	9	14	8	3
17	38	20	2	8	5	3	0	9	17	8	4
18	42	22	2	8	7	3	0	10	17	8	4
19	42	22	2	8	7	3	0	10	17	8	4
20	42	22	2	8	7	3	0	10	17	8	4
21	42	22	2	8	7	3	0	10	17	8	4
22	43	23	2	8	7	3	0	10	18	8	4
23	43	23	2	8	7	3	0	10	18	8	4
24	45	23	2	8	9	3	0	10	18	8	4
25	45	23	2	8	9	3	0	10	18	8	4
26	45	23	2	8	9	3	0	10	18	8	4
27	68	41	3	10	10	4	0	6	23	11	4
28	94	56	5	13	13	6	1	10	32	13	5
29	103	62	5	12	18	5	1	13	33	15	5
30	103	60	5	14	17	6	1	15	31	15	5
31	108	61	5	15	19	7	1	17	32	20	5
32	111	61	5	16	18	9	2	17	31	20	5
33	233	102	36	42	25	20	8	31	47	24	5
34	270	120	43	48	26	25	8	33	49	25	7
35	308	139	46	52	36	27	8	30	48	28	7

1. We did not complete functional testing with MC/DC structural coverage. During process synthesis, we decided *how* such testing *would* be performed. We decided, for example, that structural coverage would be ascertained by tool-supported analysis of test execution traces. We also decided to use a high-speed probe to capture the execution traces directly from the embedded microcontroller executing them so as to avoid the need to modify the test binary by instrumenting it to log trace data.

2. We did not perform the planned inspections. We did, however, choose what to inspect and which properties to inspect for.

We also left some fitness goals, such as **G_InspProtocolAdqt3** in Figure 7, unsupported. In modern practice, many such goals are implicitly assumed to follow from the experience and professionalism of the responsible engineers. We do not intend to address the question of how well such goals can be supported explicitly in this work.

Because we did not complete these activities, we cannot claim that the software we produced meets its dependability requirements. The fitness argument, however, shows that *if* we had, and *if* these activities had resulted in the expected evidence, we *would* have obtained software in which confidence was justified.

8 Observations

8.1 Success Arguments Enable Progress Despite Missing Information

The success argument enabled development efforts to progress despite incomplete requirements from the systems engineers. Incompleteness of, and uncertainty and change in, requirements are an unfortunate reality in many development efforts. In this effort, the incompleteness was severe: the initial requirements did not specify the microcontroller to be used, the dependability requirements, or the precise control constants to be used. The control constants in particular were not made available until near the end of development.

Because development could not be delayed, we proceeded with development despite incomplete requirements. This was not an exception in the execution of ABD, but rather a deliberate development choice that met the development effort's specific needs: we proceeded only after arguing convincingly that, when the missing details were made available, they could be integrated into the software and the evidence needed to support the fitness argument generated. The flexibility gained by using the success argument in lieu of a rigid software lifecycle model allowed accommodation of the impact of missing information in an explicit and orderly way.

8.2 Success Arguments Systematically Addressed Technology Risks

In this development, the risk of choosing inadequate technology was 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 selected would achieve that dependability. This assurance obligation shaped Evolutionary Prototype 1: while laying out the argument associated with the prototype, we made choices as to what should be included from the prototype, what should be assumed, and what could be excluded that were appropriate given what the prototype's development was meant to demonstrate.

In ABD, the success argument is the focus for development risk. In this development effort, an argument over enumerated development 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 repeated requirements delays not reduced our schedule concerns to "complete as soon as practicable," 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.

8.3 Maintaining Arguments Forced Us To Understand The Process In Detail

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 are evidence that the code complies with the requirements but say nothing about the correctness of the specification. This distinction is both subtle and crucial. Many legs of any fitness argument, whether implicit or explicit, will be supported by evidence that, although imperfect, is the best available.

Developers must understand both the limitations inherent in each process element and project artifact and the implications of these limitations upon assurance of fitness if they are to construct a system that stakeholders can justifiably trust. Writing a sufficiently detailed argument both forces the developer to consider the specific meaning of the specific actions he or she plans to take and exposes that understanding to direct criticism.

8.4 The Scope of Choices Narrowed Over Time

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 considered options that spanned a broad spectrum of categories. 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. Later choices, in contrast, tended to address tightly focused assurance obligations.

8.5 Patterns Could Capture The Bulk Of The Arguments

Patterns have proven to be a useful tool for capturing experience in many disciplines. Since no pattern library yet exists for ABD, we did not create our fitness and success arguments by instantiating documented patterns. During the process of creating, revising, and reviewing these arguments, however, we have noted twelve patterns that might benefit future ABD developers. These twelve patterns might have been instantiated to produce 23 of 49 (47%) of success argument elements and 146 of 308 (47%) of fitness argument elements. Examples of instantiations of two of these patterns are shown in Fig. 7.

9 Results Of The Case Study

From the case study, we are able to draw 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, and doing so with high levels of assurance. 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 28 choices. In 19 of these, we followed and recorded a direct line of reasoning from the assurance obligations to the choice that we made. In 6 other cases, we did not recall being prompted by the obligations, but made a choice that addressed an obligation in whole or in part. In 2 cases, we realized that we were considering an assurance obligation arising from a development risk not yet added to the success argument. In the remaining case, we formalized a choice that we had made implicitly before the goals that it addresses were added to the arguments as detail added during repair. Had this detail been added when the choices that gave rise to it were made, as might have happened had we the guidance of a pattern, this choice too might have followed clearly from an obligation. In any case, in all choices except the last, there was a discernible relationship to the obligations in the

arguments. We conclude that assurance obligations were an appropriate driver for development choices in the case of this software development activity.

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 20 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.

10 Related Work

Prescriptive Standards. There are many standards designed to promote software assurance that prescribe a set of process elements that all efforts to develop compliant software must include whether this will demonstrably achieve the necessary assurance or not. (Some of these assign software to a small range of *software integrity levels*, but offer a fixed prescription for all software in each level.) ABD instead compels the developer to assess the unique dependability needs of each part of a system and make appropriate choices; the developer can thus economize in some parts of the system while remaining assured that the system as a whole will be fit for use.

The Common Criteria. An important quality standard in the software area is the Common Criteria for Information Technology Security Evaluation [7]. This standard defines evaluation criteria for software systems in security applications, and the overall approach is to define basic development requirements and then to assess their application and efficacy for a given system. Seven levels of assurance are defined with development rigor increasing as the level of assurance increases. The Common Criteria are a prescriptive standard and, as such, have value in establishing assurance but rely on prescriptions such as the use of formal methods (EAL 7). There is no attempt to derive development technology choices from assurance, merely from the overall security goal.

Safety Cases. Other safety-critical software development work is assessed via a safety case. Some standards [18] and researchers [16] call for safety cases to be constructed early and updated often during system development and subsequent change. ABD takes this advice to its logical limit. Other research has extended the safety argument concept to address all aspects of dependability [8] and provided methods for proposing and evaluating dependability trade-offs [9].

Problem-Oriented Engineering (POE). POE [15] 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 processes that produce software that demonstrably solves a given process and does so on time and within budget.

Evidence-Based Software Engineering (EBSE). EBSE [17] calls upon developers to pose precise, answerable questions to researchers and to use research results to guide development technology choices. Because ABD process synthesis forces developers to pose such questions, the use of ABD is one way in which a developer could participate in EBSE.

Design Rationales. Extensive work has been done on recording design rationales. ABD's arguments differ from recorded rationales in two important ways: (1) they explicitly record a *complete* rationale for the *entire development process* rather than an isolated rationale for each choice; and (2) rather than attempt to record everything, an ABD developer records only what is needed to justify the assurance argument conclusions.

Process Evidence In Assurance Arguments. Other research has been conducted on the relationship between process and assurance and the mechanics of including process details in the argument [13, 14].

11 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.

The form of our case study does not permit us to make strong claims about how ABD would perform in larger development efforts, on efforts to build software with different requirements, or on efforts conducted by other teams. Our study process did, however, force us systematically to examine this development effort

for specific kinds of evidence that, if present, might rebut our hypotheses. Our failure to find evidence that ABD did not work in the context of this team and this project despite systematically looking for that evidence gives us confidence that, for this team and problem, and in those ways, ABD did work. We anticipate extending our results with further studies on other specimen systems and encourage others to replicate our efforts with other teams on other projects.

Acknowledgments

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

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] AdaCore. GNAT Pro High-Integrity Family. Web page. http://www.adacore.com/home/products-/gnatpro/development_solutions/safety-critical/.

[3] Adelard LLP. The Adelard Safety Case Development (ASCAD) Manual. Web page. http://www.-adelard.com/web/hnav/resources/ascad/index.html.

[4] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[5] B.W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[6] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

[7] *Common Criteria for Information Technology Security Evaluation*. July 2009. Version 3.1, Revision 3, Final.

[8] G. Despotou and T.P. Kelly. Extending the safety case concept to address dependability. In *Proc. of the 22nd International System Safety Conference*, August 2004.

[9] G. Despotou, D. Kolovos, R. Paige, and T.P. Kelly. Defining a framework for the development and management of dependability cases. In *Proc. of the 26th International System Safety Conference (ISSC)*, Vancouver, Canada, August 2008.

[10] Freescale Semiconductor. MPC5554 Product Summary Page. Web page. http://www.freescale.-com/webapp/sps/site/prod_summary.jsp?code=MPC5554.

[11] P. Graydon and J. Knight. Success arguments. Technical Report CS-2008-10, University of Virginia, July 2008.

[12] P.J. Graydon, J.C. Knight, and E.A. Strunk. Assurance based development of critical systems. In *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 347–357, June 2007.

[13] I. Habli and T.P. Kelly. Achieving integrated process and product safety arguments. In *Proc. of the 15th Safety Critical Systems Symposium (SSS)*. Springer, February 2007.

[14] I. Habli and T.P. Kelly. A model-driven approach to assuring process reliability. In *Proc. of the 19th International Symposium on Software Reliability Engineering (ISSRE)*, Los Alamitos, CA, USA, November 2008.

[15] Jon G. Hall and Lucia Rapanotti. Assurance-driven design. In *Proc. of the 3rd International Conference on Software Engineering Advances*, pages 379–388, Sliema, Malta, October 2008. IEEE Computer Society.

[16] Tim Kelly. A systematic approach to safety case management. In *Proceedings of the Society for Automotive Engineers 2004 World Congress*, Detroit, Michigan, USA, 2004. IEEE.

[17] Barbara A. Kitchenham, Tore Dybå, and Magne Jørgensen. Evidence-based software engineering. In *Proc. of the 26th International Conference on Software Engineering (ICSE)*, 2004.

[18] Ministry of Defence. Safety management requirements for defence systems. Defence Standard 00-56, U.K. Ministry of Defence, December 2004. Issue 3.

[19] J.M. Spivey. *The Z notation: a reference manual*. Prentice Hall, 2001.

[20] E. Strunk and J. Knight. The essential synthesis of problem frames and assurance cases. In *Proc. of 2nd International Workshop on Applications and Advances in Problem Frames*, co-located with 29th International Conference on Software Engineering, Shanghai, China, May 2006.

[21] A. Untaroiu, H.G. Wood, and P.E. Allaire. Implantable axial-flow blood pump for left ventricular support. In *Proc. of the 45th Int. ISA Biomedical Sciences Instrumentation Symposium*, Copper Mountain, CO, April 2008.

[22] X. Yin, J.C. Knight, E.A. Nguyen, and W. Weimer. Formal verification by reverse synthesis. In *Proc. of the 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, Newcastle, UK, September 2008.

[23] Xiang Yin, John C. Knight, and Westley Weimer. Exploiting refactoring in formal verification. In *Proc. of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, Lisbon, Portugal, June 2009.