

Software Process Synthesis in Assurance Based Development of Dependable Systems

Patrick J. Graydon and John C. Knight
Department of Computer Science, University of Virginia
P.O. Box 400740 Charlottesville, VA 22904-4740, USA
+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.

Keywords—software dependability; software assurance; assurance arguments; safety-critical systems; software processes

I. INTRODUCTION

Assurance Based Development (ABD) [1], [2] is an approach to constructing critical 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 [3]. 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 for systems requiring high dependability 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 choices that have already been made.

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.

In this paper, we present the ABD process synthesis mechanism. In section II, 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 III, 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 IV, we present the details of ABD process synthesis. In section V we describe a case study evaluation of ABD and its process synthesis mechanism. In section VI we describe the software development activity that was the subject of that case study. In section VII, we present metrics and artifacts from the development activity, in section VIII we give observation

that we made during the case study, and in section IX we present the results of our case study. Finally, we present related work in section X and conclude in section XI.

II. 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.

A. 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 [4].

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 stakeholder will find a given system acceptable. We used the graphical Goal Structuring Notation (GSN) [3] to document fitness arguments.

III. SOFTWARE DEVELOPMENT EFFORT SUCCESS

A. 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

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

have introduced a different kind of engineering argument, a *success argument* [5]. 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*.

The meaning of the terms “acceptable time” and “acceptable cost” will vary from effort to effort. The developers of any given project will define these phrases in the context of their project. We record success arguments in GSN also.

Fitness arguments and success arguments, in addition to having differing main claims and addressing different types of risk, evolve in different ways over the course of software development. A fitness argument is completed when the system is completed and its conclusions should be taken as holding from that point forward. A success argument, however, is used continuously throughout development to evaluate the likelihood of success at any given moment and becomes moot with the delivery of the system.

B. Traditional Process Models

The ABD process synthesis mechanism is quite different from that surrounding traditional process models such as the Waterfall and Spiral models [6]. The Spiral model brought flexibility to the software process. ABD is more flexible still. As we have shown in prior work [5], 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.

IV. PROCESS SYNTHESIS

ABD is based on two key concepts: (a) engineering decisions should be driven by the need to produce evidence for the fitness and success arguments; and (b) argument should be used to document the rationale for believing that the system is what it needs to be. The core of the ABD process synthesis mechanism is shown in Fig. 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

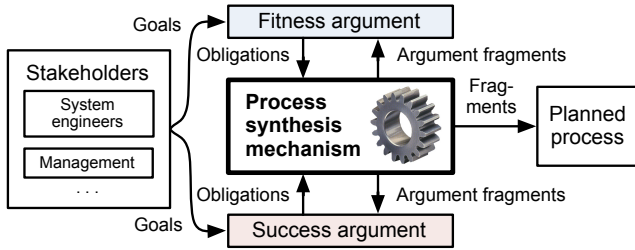


Figure 1. ABD process synthesis

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.

In *traditional* development, developers might not explicitly conceive of what they are doing as making a choice, or even consciously conceive of the choice or of any alternatives. If a certain programming language will be used to develop a given software system, for example, then that selection is still a choice irrespective of whether the selection is because the developers: (a) explicitly considered alternatives; (b) adopted the programming language because it was dictated by a standard or by a non-functional requirement given by the customer; (c) chose the programming language because finding and hiring programmers versed in that language is easier than in some alternative; or (d) because they have always used that language and never conceive of using anything else. 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 choices 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.

A. 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 ABD assurance arguments, 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.

These considerations are merely a guide, and 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 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. The cost of a poor selection is the need to repair the planned process or assurance arguments. Developers must balance this cost against the care taken in selecting a goal to address.

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. These options need not be alternatives for each other or satisfy the chosen assurance obligation in full. At the beginning of the development effort, where high-level obligations are addressed, the assembled options are frequently incomplete solutions and frequently have disparate natures. As described in prior work [1], 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. After a choice is made, the argument fragments that the choice yields are added to the fitness and success arguments and the associated process element is added to the planned process by recording it in the appropriate project documents.

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, 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

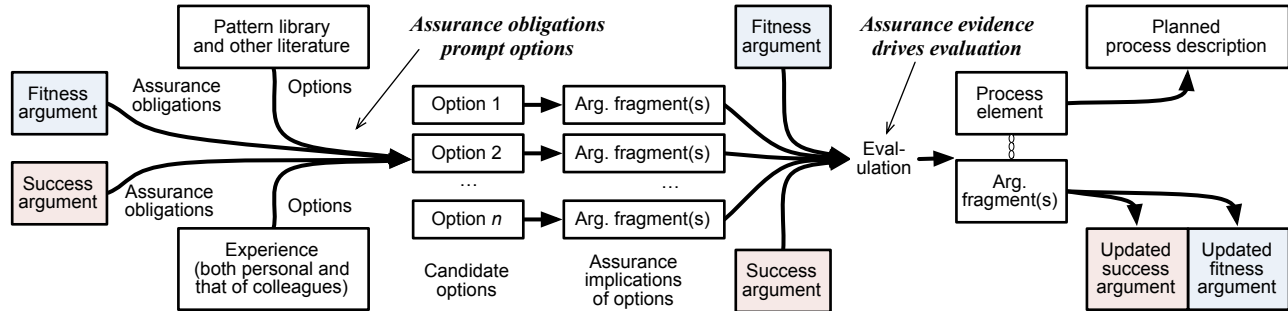


Figure 2. A process synthesis step in ABD

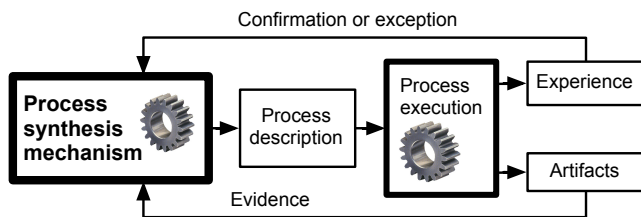


Figure 3. Synthesizing and executing a process in ABD

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.

B. The Complete ABD Approach

In ABD, process synthesis operates concurrently with execution of the process. The complete ABD approach is illustrated in Fig. 3. ABD concurrency is shown 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.

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 IV-C).

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.

C. 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 they were 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 and removing the defective choice(s) and re-synthesizing the necessary process elements.

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.

V. A CASE STUDY OF ABD

A. System Studied

In order to assess ABD process synthesis, we conducted a case study development of a specimen safety-critical system. The University of Virginia *LifeFlow* Left Ventricular Assist Device (LVAD) [7] is a prototype artificial heart pump designed for the long-term treatment of heart failure. Life-Flow has 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

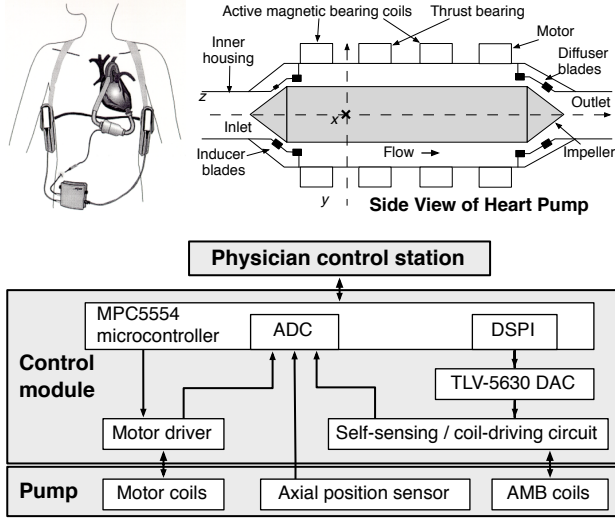


Figure 4. LifeFlow structure and use

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 under the command of a higher software layer. 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. Because individual magnetic coils can fail and such failures are anticipated to be more likely than is acceptable, 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. Fig. 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 I briefly summarizes the requirements for the 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-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. The authors obtained the software requirements from the LifeFlow developers and, in collaboration with Xiang Yin, built the necessary software as described in the following sections.

Table I
MAGNETIC BEARING CONTROL SOFTWARE REQUIREMENTS

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

B. 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. 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?”.

VI. THE MBCS DEVELOPMENT ACTIVITY

Using ABD and following the case study protocol described in section V-B, we created a planned process and executed it to create the UVA LifeFlow Magnetic Bearing Control Software (MBCS). Space considerations preclude us from describing each choice, its rationale, its process contribution, and its effects on the assurance arguments. Instead, we present the details of three typical process synthesis steps in order to illustrate the process synthesis mechanism. Additional detail can be found elsewhere [8].

Earlier process-synthesis choices had yielded the following process contributions: (a) the specification language selection was PVS; (b) the programming language selection was SPARK Ada; (c) the compiler selection was the Ada-Core high-integrity Ada compiler; and (d) the verification technique selection was the Echo verification approach [9] which was used to create a *refinement argument* structure in the fitness argument. Separately, the systems engineers chose the MPC5554 microprocessor as the target computer.

The example choices we present here are those associated with the hard real-time requirements and one of those associated with the use of the MPC5554 microprocessor with no operating system.

A. Choice DC-007: WCET

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.

Options Considered:

- 1) Analyze the worst-case execution time (WCET) of the control calculation using a method 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 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 rarely would be sufficient to keep the impeller from striking the pump’s inner housing.

Accordingly, we chose option (1).

Process Fragments 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 of the fitness argument: we added a layer of argument that decomposed the obligation to show that the requirements had been met into an argument over real-time and non-real-time requirements. A new argument fragment was added to show that the real-time requirements had been met.

B. Choice 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: 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 their portion of the 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.

We elected to remain with our original choice, option (1).

Process Fragments 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 allowed us to add two new sub-arguments to the success argument: (a) if the microcontroller is sufficiently fast and if scheduling new tasks brought about by requirements change is possible, then we should be able to create a suitable schedule for our cyclic executive design; and (b) if WCET analysis of the code for each task and the executive structure is possible, then we should be able to demonstrably meet our real-time requirements. 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.

C. Choice DC-019: Inspection of Register Write Order

Obligations: We mapped variables in our SPARK Ada program to the registers controlling the MPC5554’s ADC and other peripheral units, and marked these with the `Atomic` pragma. Checking revealed that our compiler did not reorder writes to these registers with respect to each other. However, we misread the Ada language semantics and failed to see its guarantee of this write order. Consequently, we sought our own evidence that write order was preserved.

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 Fragments Contributed:

- 1) Use `objdump` to disassemble the compiled binary.
- 2) Inspect the disassembly to confirm the write order.

Argument Fragment Contributed: This choice contributed the fitness argument fragment shown in Fig. 5. Because we believe that the disassembly is correct and that 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 our confidence in the adequacy of the inspection protocol (**G_InspProtocol-Adqt3**) and upon our confidence in the disassembly tool and our use of it (**G_DisassemblyCorrect**).

Subsequent Repair: Review of the argument fragments associated with this choice given in a draft of this paper revealed our misunderstanding of Ada language semantics. The process was repaired to remove the unnecessary inspections.

VII. METRICS AND ARTIFACTS

A. Development Choices And Repairs

During the course of development, we made a total of 27 development choices. Most of the first development choices had to be made in sequence: only 4 of the first 12 choices could have been made in parallel with at least one other choice. In contrast, all of the last 8 choices could conceivably have been made in parallel. 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.

We invoked the repair process a total of 44 times. The bulk of repair came at the end of the project and consisted of refinements to and clarifications of our arguments.

B. 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. 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.

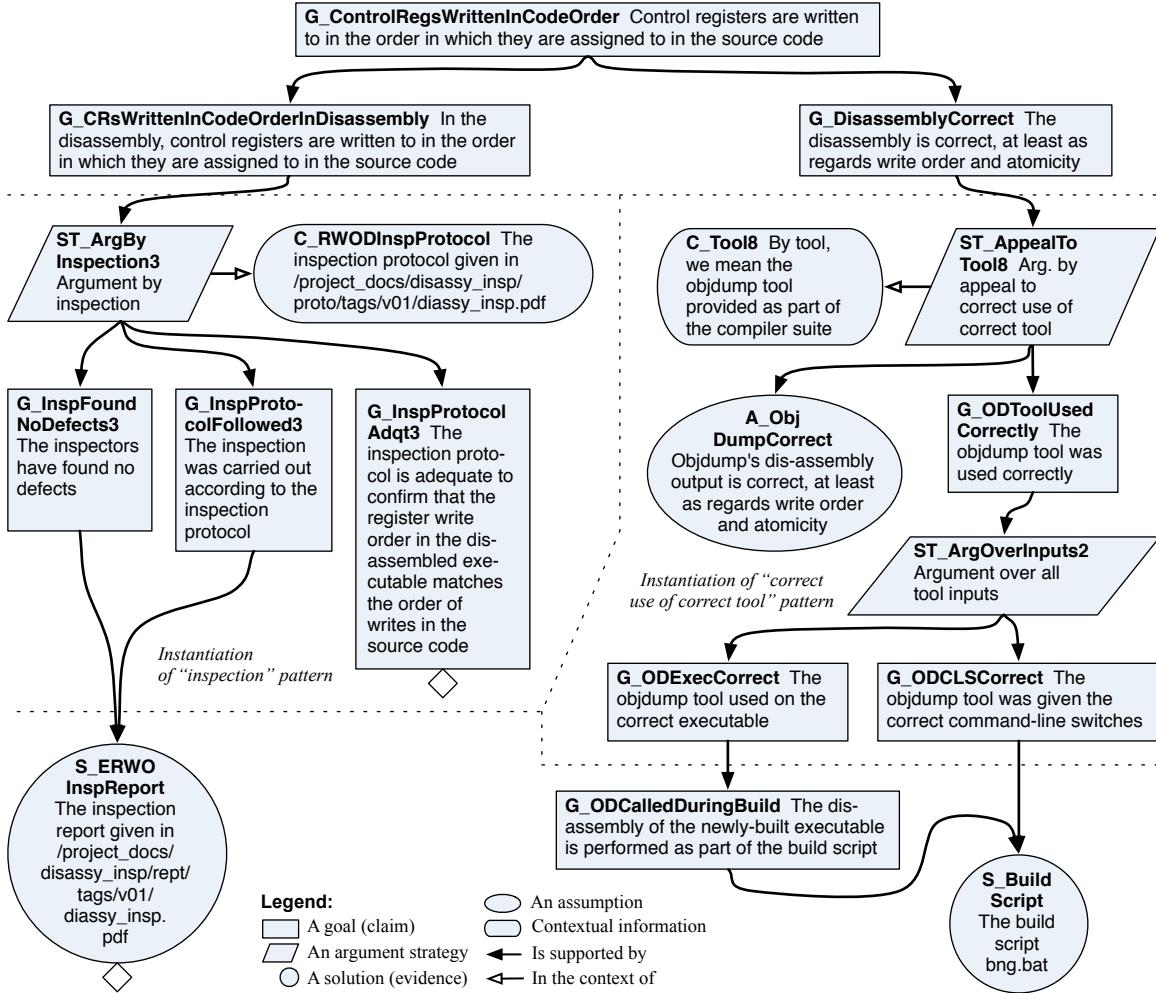


Figure 5. Sub-argument demonstrating memory-mapped register write order

VIII. OBSERVATIONS

A. ABD Enabled Progress Despite Missing Information

The success argument enabled us to progress despite incomplete requirements from the systems engineers. Incompleteness of requirements and uncertainty and change in requirements are unfortunate realities 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 we could not delay development, we proceeded with development despite incomplete requirements. This was not an exception in our execution of ABD, but rather a deliberate development choice that met our specific needs: we proceeded only after arguing convincingly that, when the missing details were made available, we would be able to integrate them into our software and regenerate the evidence

needed to support the fitness argument. The flexibility gained by using the success argument in lieu of a rigid software lifecycle model allowed us to accommodate the impact of missing information in an explicit and orderly way.

B. 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.

C. Patterns Could Capture The Bulk Of The Arguments

Patterns have proven to be a useful tool for capturing experience in many disciplines. We will develop a library of ABD pattern as part of future work. During the process of creating, revising, and reviewing the MBCS fitness and success arguments, we identified twelve repeated argument constructs that together account for 23 of the 49 success argument elements (47%) and 146 of the 308 fitness argument elements (47%). These constructs might be turned into ABD patterns to benefit future ABD developers. Fig. 5 shows two example constructs that might be turned into ABD patterns; others include *argument over development risks*, *argument over refinement chain*, *argument by machine-checked proof*, and *argument by appeal to functional testing*.

IX. 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.

X. RELATED WORK

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.

An important quality standard in the software area is the Common Criteria for Information Technology Security Evaluation [10]. 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.

Other safety-critical software development work is assessed via a safety case. Some standards [11] and researchers [3] 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 [4] and provided methods for proposing and evaluating dependability trade-offs [12].

Problem-Oriented Engineering (POE) [13] 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) [14] 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.

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.

Other research has been conducted on the relationship between process and assurance and the mechanics of including process details in the argument [15], [16].

XI. 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 making 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 R. Dewar and B. Porter of AdaCore and the AdaCore corporation for their support of this project and criticism of this paper, P. Allaire and H. Wood for details of the LifeFlow LVAD, and X. Yin for assistance with verifying the MBCS. Work funded in part by NASA grants NAG-1-02103 & NAG-1-2290, and NSF grant CCR-0205447.

REFERENCES

- [1] P. Graydon, J. Knight, and E. Strunk, "Assurance based development of critical systems," in *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2007, pp. 347–357.
- [2] 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 ICSE, Shanghai, China, May 2006.
- [3] T. Kelly, "A systematic approach to safety case management," in *Proc. of the Society for Automotive Engineers 2004 World Congress*. Detroit, Michigan, USA: IEEE, 2004.
- [4] G. Despotou and T. Kelly, "Extending the safety case concept to address dependability," in *Proc. of the 22nd International System Safety Conference*, August 2004.
- [5] P. Graydon and J. Knight, "Success arguments," University of Virginia, Technical Report CS-2008-10, July 2008.
- [6] B. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, May 1988.
- [7] A. Untaroiu, H. Wood, and P. 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.
- [8] P. Graydon and J. Knight, "Software process synthesis in assurance based development," University of Virginia, Technical Report CS-2009-10, October 2009.
- [9] X. Yin, J. Knight, E. 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.
- [10] *Common Criteria for Information Technology Security Evaluation*, July 2009, Version 3.1, Revision 3, Final.
- [11] Ministry of Defence, "Safety management requirements for defence systems," U.K. Ministry of Defence, Defence Standard 00-56, December 2004, issue 3.
- [12] G. Despotou, D. Kolovos, R. Paige, and T. 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.
- [13] J. Hall and L. Rapanotti, "Assurance-driven design," in *Proc. of the 3rd International Conference on Software Engineering Advances*, Sliema, Malta, October 2008.
- [14] B. A. Kitchenham, T. Dybå, and M. Jørgensen, "Evidence-based software engineering," in *Proc. of the 26th International Conference on Software Engineering (ICSE)*, 2004.
- [15] I. Habli and T. Kelly, "Achieving integrated process and product safety arguments," in *Proc. of the 15th Safety Critical Systems Symposium (SSS)*. Springer, February 2007.
- [16] —, "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.