DEPENDABLE
COMPUTING

Informal Reverse Engineering

A Microstandard

Version 1

Jonathan Rowanhill

November 11, 2021

1 Introduction

This document provides guidance for the assured informal reverse engineering of a software module in a software system. This microstandard assures that an informal reverse engineering process, performed manually by humans, produces a set of interface contracts for the software module that are *sufficiently complete and correct.*

Sufficiently complete and correct is defined as the interface of the target module in the target system being completely covered by explicit, defined contracts that completely capture the following:

1. Required functional and nonfunctional (for attributes of interest) behavior of the target module in response to activation and state of the target system as well as functional and nonfunctional behavior and state of the system in response to behavior of the target module.
2. Assumptions that can be made by the target module about behavior and state of the target system on target module behaviors, and assumptions that can be made about target module behavior and state on resulting system state actuation.

This standard is meant to assure that informal reverse reengineering achieves correct and complete natural language interface definition of the target module sensitive to the functional and nonfunctional needs of target systems.

1.1 Purpose

It can be extremely difficult to correctly reverse engineering the behavior of a software module. This is further complicated when the context of a system in which it operates must be considered. The result is that often a very informal set of contracts or requirements or description of functionality about a software module is maintained or used to develop testing, formal specifications, new versions of the module, and documentation. This standard exists to help assure that such a reverse engineering activity produces a definition of the target module that is useful for these activities, while being highly assured to be correct and complete, while at the same time not relying on more than human analysis of source code.

1.2 Rational Microstandard

This standard is a rational microstandard [1]. This means that the standard consists of three key components as follows:

- **Process Guidance**: This document guiding the user through conduction of process and collection of evidence in conformance with this standard.
- **Evidence Templates**: Template documents for the evidence that users of this standard must populate in partial fulfillment of this standard.
- **Argument Templates**: The rationale for why the process defined in this guidance document, along with the produced evidence, when conducted with sincerity and

competency, produces a reasonable level of assurance that the resulting reverse engineering is correct and complete.

Those that wish to comply with the standard will primarily be interested in the first two elements of the standard. Those that wish to understand the underlying reasoning behind the standard, contribute to, improve, or reuse parts of the standard, will be additionally interested in the argument templates.

Furthermore, those who wish to produce an explicit assurance case from this standard can do so by instantiating the argument templates with specific evidence produced in performing the process defined in this document.

1.3 Call for Participation

This is an open, rationale microstandard. Just as open-source software is meant to be shared with a community and collaborated and iterated upon to better serve the community, so is this standard. By having an explicit rationale in the form of assurance arguments, participants can debate the reasoning behind the standard, iterating upon it and improving it, and changing this guidance, process, and evidence as needed to improve the standard for use by the community. Improvements are always sought in

- Assurance Quality
- Readability
- Learnability
- Applicability
- Cost-Efficiency with Trade-offs and Alternatives

For more information on how to participate in the development of this on microstandard, please contact info@dependablecomputing.com.

1.4 Guidance Material

The remainder of this document is as follows. Section 2 presents the reverse engineering problem in more detail. Section 3 outlines the basic rationale (argument) of this standard. Sections 5 guides the user through the process of reverse engineering, producing the accompanying evidence needed to back the assurance arguments.

2 The Problem of Informally Reverse Engineering A Software Module

Assume that a set of software systems, **S**, share a common software source module **M** that provides important critical functionality as illustrated in Figure 1. In the figure, three systems are shown using module **M**. Each has a set of requirements that **M** is intended to satisfy.

The user's goal is to determine what the behavior of the software module M is within these systems. For this standard, it is important that behavior be captured accurately, and therefore the model of assume/guarantee contracts is applied. Contracts are developed at the interface of the target module where it is controlled by or controls the target systems of interest.

This requirement provides process, analyses, and evidence collection techniques to develop these natural language contracts for the target module M while assuring that they are acceptably correct and complete.

For the purposes of this standard, a **software source module** is any componentized unit of

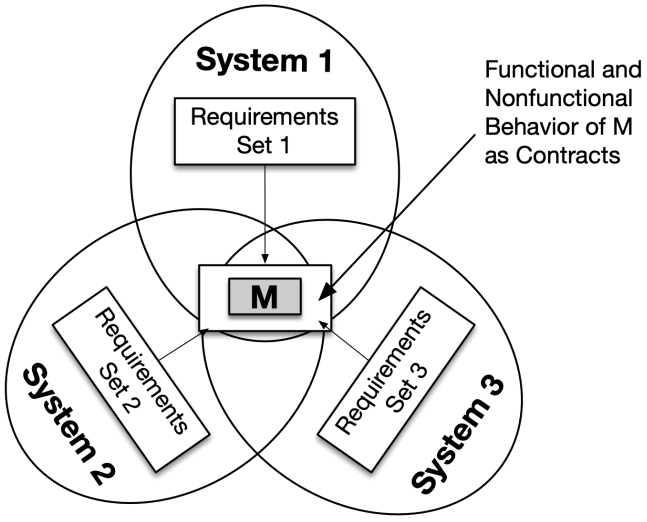

Figure 1. An illustration of the informal reverse engineering project.

traditional software source code with a clear delineating boundary. It could be a source module, file set, or class definition.

This standard is tailored to traditional software that is of static design. Data-driven-design software, *e.g.,* trained neural networks, are *not* the target of this standard. In contrast, a traditional software module defining a neuron on which such a neural net is built *would* be a good candidate. A typical appropriate module might be a message service implementation or class that stores and maintains complex data.

An applicable **software system** can be any software system described by source code and hardware. It can be a purely informatic system or a software subsystem that supports a cyber-physical system.

2.1 Reference Process Summary

The reference process for module replacement consists of four key steps.

1) Identification of the systems of interest
2) Analyzing the software module to identify its interface points with potential systems.
3) Developing interprocedural control-and-data-flow graph of the target module from points of control from potential systems to its points of control on potential target systems.
4) Further developing a copy of the above graph for each target system from points of service provision in the systems to points of control on the target module.
5) Analyzing the source code of the systems and target module to label the edges of the graphs with requirements and preconditions.
6) Extraction of contracts from the union of requirements and preconditions are edges of the graphs.

Each of these steps is accompanied by production of evidence that helps assure that the products of these steps are correct and complete.

3 Issue Tracking

Throughout this reference process, the user will be required to track and resolve issues. The user is expected to log all issues and track their resolution as is common to any other modern software engineering activity and provide such tracking as evidence of compliance with the standard.

4 Concepts for Capturing Module Behavior

For this standard, the intended behavior is captured as a set of *contracts* on the interface of module **M**.

4.1 Behavior Contracts

A contract is a set of *requirements*, both *functional* and *non-functional*, that state the guarantees of functionality and data of module **M** given guarantees of the surrounding systems **S**, along with preconditions that are assumptions that can be made by the module **M** and systems **S**.

Consider two example contracts. An example contract is presented in <<Figure Z>> to the left. The first contract, C1, states that given an input parameter of the form provided by A1, the resulting function of method F1 will be G1, with safety constraint S1. This is an example of an Assume/Guarantee contract.

F1 is an *interface point* of the module M. An interface point is any distinct design location on the interface of M where constraints between M and any systems of S must be enforced. C1 is the contract for interface point F1. Constraints G1 and S1 are *guarantees* that must be provided by an implementation of F1 for module M, given that the *assumption* A1 is satisfied by the systems S.

Another contract, C2, is presented for the data structure *interface point* on the example module. This is data that can be directly read or written to by external system elements in S, and therefore, the contracts C2 and C3 must be satisfied on read and write operations from systems of S.

4.2 Requirements

There are three classes of requirements under consideration in this standard are:

1) **Functional Requirements**: The functional behavior required of the prime software module.
2) **Nonfunctional Requirements**: Requirements about how the required functions are performed.
    a. **Dependability Requirements**: Non-functional requirements capturing required safety, security, reliability, availability, and maintainability of the prime software module.
    b. **Design Requirements:** Non-functional requirements capturing restrictions on the design and implementation of the prime software module. These can include restrictions on the source programming language and its features, as well as restrictions in software design or in required preservation of existing software features from the original software module.

The strategy applied by the argument is to apply a generalization of Overarching Properties. There are three overarching properties. They were developed to express fundamental properties of system safety. In our case, we utilize them more generally as follows:

4.3 Completeness and Correctness

The user of the standard must show that for their target module M and systems S that defined contracts are a complete and correct representation of the *desired behavior* of module M for **attributes of interest**.

In the ontology of Overarching Properties [2], correctness and completeness mean that the contracts the user defines for module M captures all the *desired behavior* for module M in systems S.

Note the emphasis on the word “desired” in the above definition. The actual behavior of the current implementation of module M is likely to contain software bugs (design faults). It is common to discover such bugs while analyzing the software module to develop its contracts.

Therefore, it is important that the user log all potential bugs as issues and resolve them with system stakeholders. Sometimes, it will be advantageous to keep them. But many times, they

can be corrected in the contracts or reengineering of the module. In the former case, the correct contract defines the
"buggy" behavior. In the latter, it defines the desired behavior for the reengineered module.

The previous definition of completeness and correctness refers to attributes of interest. For this standard, *attributes of interest* are

1. Functionality: Functional Requirements of M
2. Dependability: Non-functional dependability requirements of M, as defined by John Knight in Fundamentals of Dependability for Engineers [3] as the overlapping quality attributes of
    a. Safety
    b. Security
    c. Reliability, and
    d. Maintainability.

The standard does not require that a particular set of these attributes be covered, only that the set of such quality attributes that is covered be carefully stated by the user in their claim about the reengineered system.

4.4 Where Behavior Contracts are Attached to Module M

We have written that the contracts defined for M are "on" the *interface* of module M. The interface of module M is any location of the module's source code, or any grouped data (primitive or data structure), where the module and the surrounding systems may interact.

4.4.1 Interface Points

For this standard, we define a point where a contract must be defined on the software module as an *interface point*. An **interface point** is any place in a software module's source code where control flow or dataflow can transition between the module and systems that utilize the module in either direction.

The control flow of a system is the flow of executed machine instructions between the system and the software module. (We ignore process interleaving except under advanced security concerns, discussed later in this section.) This flow can be in either direction, where execution of the system begins executing instructions defined by module source code, or where source code returns execution flow to flow from the surrounding systems.

A typical example of a **control-flow interface point** is at a function, procedure, or method signature in the source code of module M. Systems can call the function/procedure/method to execute code in the module. The interface point is also traversed when the body of the function/procedure/method concludes and returns to the system.

A data-flow interface point exists at any read-write of data in which the instruction (code) executes in S or M and the data exists across the border in M or S, respectively. For this work, the interface point can be attached to the data structure itself if it is in Module M, or the instruction that accesses the data if the data is not in M. An example of a data-flow interface

point would be a read or write from Module M to a global data structure in S. Another example would be a system in S writing or reading a data structure in module M.

4.4.2 Behavior Points

A **behavior point** is a subclass of interface point. It is any interface point where a system of set S instigates code execution or data state change in M. An example of a **control-flow behavior point** is a procedure definition (at the function signature) that is called by source code from one or more systems in **S** from outside of the target module **M**. An example of a **data-flow behavior point** is a static global data structure in target module M that is written to by one or more locations in systems of S from outside the target module source code. A further example of a **control-flow behavior point** would be any method of the target module **M** that handles events, signals, or traps that originate from outside of the target module.

4.4.3 Control Points

An **control point** is a subclass of interface point that defines a location in target module **M** that effects change in state of target systems **S**. A **control-flow control point** exists wherever target module **M** executes a function/method/procedure in system **S** which then results in system or environment state change. A **data-flow control point** exists wherever target module M directly writes to data objects declared in system in S. A control-flow control point can also occur if target module **M** triggers a system or trap detected by the system from hardware.

Remember that control points can result from system changes to the environment if the environment is included in a system definition (as should be the case with many cyber-physical and vehicle systems). Changing state in an operating system, file, and physical environment state through operating system calls or calls to functions/procedures/methods outside of the target module **M** all count as actuation.

4.5 Techniques to Construct Contracts

There are many techniques one can use to reverse-engineer contracts for the source code module M. If a system has extensive documentation, that can sometimes be used to determine contract terms. The arguments that accompany this guide leave room for documentation analysis and other techniques.

The current version of this standard focuses on use of reverse engineering on the source code through static analysis of software. The analysis is conducted on the module and its surrounding systems. The reason for this is that source code is the primary artifact that is likely to exist for any module formal reengineering project and contains most of the detail that the user will require.

4.6 Defining Terms

As analysts develop functional and nonfunctional requirements that lead to contracts on interfaces of the target module, they will need to be careful and precise in the use of terminology. Whenever technical concepts can be objectified, it can be useful to name them

and explicitly define them. In the reverse engineering evidence template, we provide a worksheet, called “Definitions”, as a specific location where definitions of terms can be placed.

5 Informal Reverse Engineering to Contracts

To determine, informally, complete and correct intent of software module M, one must (1) develop a clear picture of its actual behavior, (2) account for design flaws, and then (3) define its intended behavior as a reflection on its actual behavior. Using reverse engineering on source code, we can apply static analysis and dynamic analysis techniques.

5.1 Informality in Reverse Engineering

The guide refers to “informal” reverse engineering. This is because this standard does not assume contract development rises to the level of rigor of formal methods. To apply formal methods to reverse engineer the software would be above the assurance-level intended by the standard, which is meant to remain practicable.

Analysts are free to use whichever tools they see fit to perform the reverse engineering. The assurance case of this standard assumes that all information obtained from tools and analysis is correct to the extent possible. Therefore, tool dependability is the analyst’s responsibility and is assumed by the standard given the imposition of informality.

Because general static analysis is a very wide field with many variations, this standard utilizes specialized data structures and methods designed to capture all and only what is needed for the development of interface contracts for the target module M and a defined set of target systems. If the analyst uses more general tools for reverse, engineering, they should find that those data models map easily to the evidence and analysis requested in this process.

This guide applies the specific and tailored concepts of Chapter 4 to be technology agnostic. All techniques for static analysis from this chapter can be performed by hand. Use of tools is entirely optional. However, the presentation of clear evidence is not optional. The analysis spreadsheets that provide evidence to the standard are designed to be useful both for collection and interpretation of reverse engineering in addition to being the direct evidence required by the standard.

5.2 Basics of Software Static Analysis

Static analysis largely consists of inter-procedural analysis and intraprocedural analysis, for which there are many textbooks, articles, and tools. A common way of thinking about these analyses is that they produce static graphs representing the behavior of software.

- **Interprocedural Static Analysis** reveals how execution can flow between the functions, procedures, and methods of a modern software system in any execution. It produces graphs that show how control flows through a program. The nodes of the graph are its functions, procedures, and methods, and its edges are possible calls between the the methods.

- **Intraprocedural Static Analysis** produces a graph for the internals of each method/procedure/function. Each node of a produced graph is a *basic block* (a set of consecutive instructions that do not branch or loop), and its edges are the links between basic blocks resulting from branches and loops.

The basic graph contains value in that is shows a program's control-flow structure. Additional value comes from labelling the graph with **invariants**—conditions of the program that are always true during the transition represented by each edge. These invariants take on two forms: *preconditions* and *postconditions*. One can label the edges of inter and intraprocedural flow graphs with preconditions and postconditions.

- **Preconditions**: Properties of software and environment state that are invariantly true on entering a new node from an edge.
- **Postconditions**: Properties of software and environment state that are invariantly true on leaving a node along an edge towards another node.

To summarize, in reverse engineering, the analyst is producing intraprocedural and interprocedural flow graphs. We utilize both to sufficiently understand target modules in declared systems to produce adequate interface point contracts. We use them on each as follows:

- **Analyze Systems**: Inter-and-intra-procedural static analysis are critical for analyzing the behavior of each software system in S that utilizes target module M. Static analysis allows the analyst to define contract terms at the target module's behavior points. More specifically, these analyses allow the analyst to:
  - **Synthesize Nonfunctional Requirements for Behavior Points**: To know what nonfunctional requirements must be met at the interface of a target module, we must identify services within the control-and-data-flow graph of each system of system set S that depend on the target module M at one or more *behavior points* (defined in Chapter 4). The method presented utilizes interprocedural static analysis to discover the complete interprocedural control flow between where services are well defined in each system of **S** to where they interface target module **M**. The method then is used to decompose non-functional requirements along the control graph to the boundary of an interface point, where they become contract guarantees.
  - **Synthesize Functional Requirements for Behavior Points**: Intraprocedural analysis of system code that interfaces a behavior point determines the functional requirements that are to be contract guarantees on that behavior point. Sometimes, interprocedural analysis of system code is necessary to clarify these requirements to useful level of semantics. In that case, the intraprocedural analysis used to produce nonfunctional requirements (described in the previous bullet), can be applied to construct hierarchy of functional requirements.
  - **Synthesize Assumptions Supported at Behavior Points**: Static analysis can identify constraints on use of target module data and procedures that are

intentional invariant assumptions that will hold on behavior point contracts for all systems.

- **Analyze Target Module:** Inter and intra procedural static analyses allow the analyst to understand the internal behavior of the target module.
    - **Identify All Possible Interface Points:** Intra and interprocedural analysis allow identification of all possible behavior and control points of the target module.
    - **Determine Contracts at Control Points**: Static analysis of target module M determines the functional and nonfunctional terms on control point contracts.

5.3 Mixed Control-and-Data-Flow Graph

There is one important complication in the way informal static analysis is performed in this guide: interprocedural static analysis is extended to include indirect control flow across data objects. That is, analysts must identify **mixed control-and-data-flow graphs**. One starts with a complete intraprocedural control-flow graph, and then extends it with nodes representing persistent data objects.

The mixed control-and-data-flow representation is important for modeling asynchronous processes that interact with data. This allows analyzing services across message buses, shared, memory, and other architectural models and enterprise patterns.

In a **mixed control-and-data-flow graph**, we represent writes and reads to *intra-procedurally persistent declared data* as edges in the graph. And nodes are added to the graph to represent the persistent declared data objects. **Intraprocedural declared persistent data** is any data object declared outside of the scope of a procedure/function/method. If a procedure/function/method reads or writes to a persistent declared data object, then that data object is represented by a node in the mixed graph. Any writes to the declared data from a procedure/function/method is represented by an edge from the node of the procedure/function/method to the node of the data object. Any read of the data object is represented by an edge from the node of the data object to the node of the reading procedure/function/method.

The remainder of this chapter goes into detail in how these analyses are performed.

5.4 The Reverse Engineering Function

In this step of the reengineering process, it is assumed that the engineer has access to the following input information:

1. The current source code of the target module M and the software of systems in S that interact with module M.
2. The stakeholders of system M including available developers and stakeholders in the services provided by systems in S. The latter are usually owners of the system.

The user will be expected to develop and maintain

1. Reverse engineering spreadsheets, as provided with the standard.
2. Issue tracking lists along with issue resolutions.

The user will apply a reverse engineering approach developed for this guide.

The result of the approach will be

1. Completed reverse engineering spreadsheets.
2. Issue tracks with all issues resolved.

The completed spreadsheets will contain

1. Interface points for the software module M.
2. A contract for each interface point of M.

Each contract will contain assumptions on systems of S and guarantees from M for use by S of the specific interface point on M.

The assumptions and guarantees of each contract will define, in natural language, constraints on behavior of systems in S in assumptions, and constraints on the behavior of M in guarantees.

The constraints will define, in natural language, functional requirements and nonfunctional requirements on the systems and target module over the attributes of interest.

The functional requirements will define the functional behavior of module M under stated conditions in systems S.

The nonfunctional requirements will define nonfunctional behavior of M under stated conditions in systems S and will include qualities such as performance, availability, reliability, safety, and security.

5.5 Example

Throughout this section we will refer to an example of completed analysis/evidence provided as an annex to the standard. The example targets software module CVisibilityGraph.h/cpp class as module **M** utilized by the Open UXAS software system as the sole system in system set **S**.

5.6 Process

The reverse engineering process presented here is tailored to produce both functional and nonfunctional requirements for module M contracts within the systems of S. The process is illustrated in Figure 2. The steps of the process are as follows:

1. **Definition of M and S**: First, the analyst determines what source code is the target software module M, and what software systems utilizing M will be considered. In addition, the services provided by those systems are defined. In the illustration, M is defined and two systems using M are identified. System 1 provides services SV1 and SV2, and system 2 provides service SV3.
2. **Analysis of M**: The module M is analyzed. In this process, the internal structure of M is analyzed as interprocedural control-flow graph. Static data objects declared and stored in M are also identified. The analysis identifies the interface points of M as behavior points (BP) and control points (AP). In addition, d*esign constraints*—requirements on

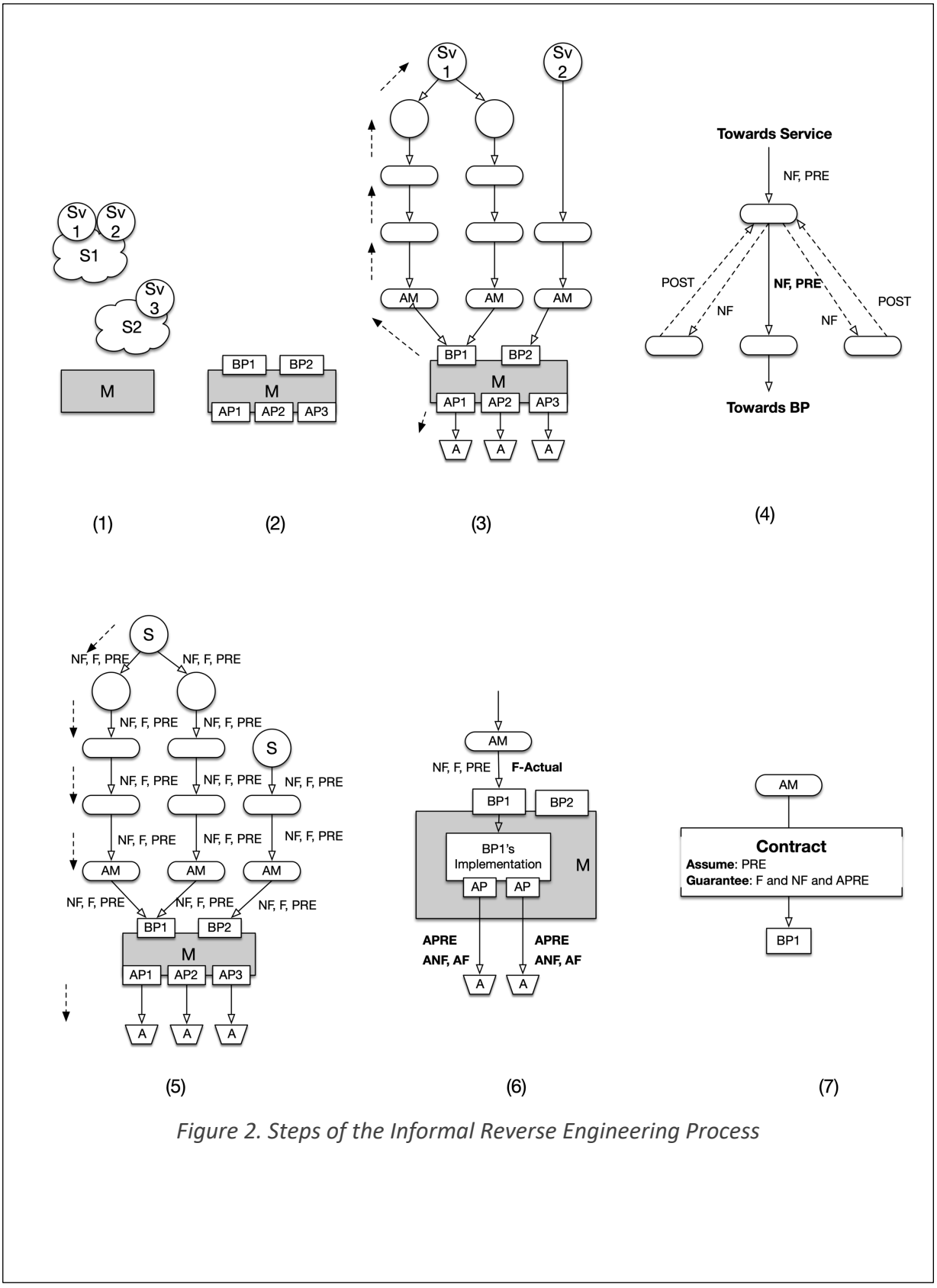

Figure 2. Steps of the Informal Reverse Engineering Process

can be identified at this time.

For each system **Sys** in systems set **S**

3. **Control-and-Dataflow Analysis** of **M** in system **Sys** is analyzed. This analysis consists of constructing the intraprocedural mixed control-and-data-flow graph of each system, Sys, from all locations where services of interest (**Sv1** and **Sv2** in the figure) 'start' (called *service points*) to where they interact with the behavior points (**BP1** and **BP2** in the figure) of the target module **M**. In addition, one analyzes where each control point of module **M** (**AP1** and **AP2** in the figure) acts on state of system Sys. This service flow analysis further extends to the control points (**CPs**) of the target module. This latter portion of flow graph development can be re-used between analyses for target systems and saved for later re-use.
4. **Requirement Synthesis**: Starting with a correct definition for each service (confirmed by stakeholders), the analyst defines requirements (labelled "F" and "NF" in the figure for functional and nonfunctional requirements, respectively) and design/solution constraints ("PRE" in the figure) for each service point. The analyst then decomposes these requirements along the intraprocedural mixed control-and-data-flow graph. If it is assumed that M is largely implemented correctly at the level of natural language requirements, then one need only state and decompose nonfunctional requirements. Otherwise, it can be helpful to decompose functional requirements as well.
5. **Service Analysis**: One need not perform requirements synthesis for the entire mixed control-and-data flow of the system but must cover enough of the graph to assure correct requirements that trace to control points of module M through all utilized behavior points of module M from all services provided by a target system. The first sub-step is depicted at Step 5 of the figure. In this stage of analysis, all requirements in the produced hierarchy are checked and reviewed for quality. The result is a set of quality nonfunctional, "NF", and functional, "F" requirements at each behavior point of target module M. Also included are preconditions, "PRE", that hold at the end of each edge. At this stage, all behavior points utilized by a system Sys should trace back to services. If any edges to behavior points do not, there are missing services that must be defined with additional work to do in steps 4 and 5. (Note, however, that all edges to behavior points being covered by requirements is not a sufficient condition to conclude one has identified all services.) One must then perform a similar analysis on the actuated system elements of control points. Often there are functional and non-functional requirements on actuation that may not be obvious from the source code. Therefore, one must check with system stakeholders to identify these requirements and work backwards along the control flow to the control points, assuring that the requirements for actuation are well-stated at each AP.
6. **Behavior Analysis**: Once all systems have been analyzed, then one performs an analysis of the implementation of each behavior point (BP) in module M. For each behavior point, one 1) analyzes the source code of target module M to determine its actual functional and nonfunctional behavior (in the graph this is stated as F-actual), and 2)

analyzes the source code of M to identify the actual functional and non-functional requirements and preconditions on all control points within the behavior.

7. **Contract Consolidation**: The sets of requirements and preconditions on behavior points and control points can now be consolidated into a reduced set of contract terms for the interface of module M for a target system. For each interface point, its preconditions and requirements are reduced to a minimal set of natural language assumptions and guarantees, respectively. Across analyzed systems in set S, guarantees consolidate by conjunction, while assumptions consolidate by disjunction. This results in a consolidated contract for each interface point.

The output from the above process is a set of contracts on M that are sufficient to define its reengineering. The remainder of this chapter walks through these steps in more detail, showing the user how their results are recorded in the provided evidence spreadsheet.

5.7 Definition of M and S

Before the reverse engineering process can begin, it is important to be rigorous about what constitutes the module to be reengineered.

It is equally important to determine which systems utilizing the software module M are going to be considered. There are two reasons for this. The first and most important is because the non-functional requirements (safety, security, etc.) that module M must satisfy are determined by the services provided by systems and the ways that they use M to help provide those services [3]. In this standard, analysts will be determining how services flow through code and data in each system of interest to module M. Therefore, you need a clean and accurate definition of M and each system using M that you want to consider.

Users should first open the “Systems” tab in the “informalEngineering.xlsx” spreadsheet. Here, they should identify each system under consideration with a simple definition and enough definition of software, hardware, and physical assets to fully identify an intended system configuration that can be instantiated in the real world.

Each system under consideration should be listed with an entry in the “Systems” tab.

The team of engineers and stakeholders should review this definition of systems of interest, S, and sign on the definitions **correctness and completeness**.

Users should then open the “Target” tab in the “informalEngineering.xlsx” spreadsheet. Here, they should identify all source code artifacts and the contents thereof that define the target software module M.

The team of engineers and stakeholders should review the definition of the target module and sign off on the definition’s **correctness and completeness**.

Correctness and completeness here refer to the sufficiency of the definitions to define systems and the target module, and that the information defining them is correct and not nebulous or misleading.

Any issues detected during the definition process should be resolved at this time to the satisfaction of engineers and stakeholders before any further action is taken.

If later changes to the target module or systems takes place, change issues must be charged to the issue system and then resolved. Signoff on affected system and target definitions must be re-achieved.

5.8 Analysis of M

The analysts utilize the definition of target M and systems in S to analyze the structure of existing source code for module M.

The analyst has the following information as input:

1. The current source code of the target module M and the software of systems in S that interact with module M.
2. Clear definition of target module M and systems of interest S in the “Systems” and “Target” spreadsheets.

The engineers analyze the source code of module M and produce the following sub-analyses in the following recommended order:

1. **Access Override Analysis**: If the source code is object-oriented (*e.g.*, C++ classes), or is subject to a package-binding mechanism (*e.g.*, CORBA), then mechanisms or policies that modify traditional access capabilities between linked elements, modules, or packages might arise. These can have a profound impact on definition of target module interface points. This analysis should be considered first so that analysts are in the right ‘head space’ to consider where interface points might exist in a target module. This analysis will impact subsequent sub-analyses. Note, each system of interest could have different policy and mechanisms to override standard access assumptions.
2. **Data Analysis**: The analyst must consider what persistent data objects are declared in the target module and identify where that data can be accessed within and external to the target module. This information will be important for identifying APs when code analysis is performed in the next step. Persistent declared objects that are writeable from remote (outside the target module) system are behavior points of the target module.
3. **Code Analysis**: The analyst must consider all declared code objects (functions, procedures, and methods) of the target module. This guide and standard are restricted in scope to traditional programming languages (imperative, functional, object-oriented) with static code definitions. Through analysis od declared code objects, analysts identify the remaining behavior points of the target module. Combined with data analysis, this completes definition of traditional behavior points for the target module. It also allows the analyst to develop knowledge of intraprocedural control-and-data-flow of the target module between behavior points and control points which will be important for determining contract correctness and consistency.

4. **Unintended Access Analysis**: If security is part of the attributes of interest, then analysts should consider any security vulnerabilities or attack objectives from unknown vulnerabilities that might impact the interface surface of module M within systems of set S. Data in the "unauthorized access" tab analyses how vulnerabilities or attack objectives might increase what are interface points in the target module. This analysis might technically require consideration within systems of set S. The user of this analysis will adjust the set of behavior points and control points based on in-scope vulnerabilities based on their potential and/or presence in the target module source code and system context.

What follows is a more detailed explanation of each sub-analysis.

5.8.1 Access Override Analysis

All programming languages have the potential to override the access model directly intended by language semantics. In package models, this can sometimes be done through explicit container overrides on access policy rules. In other languages, such as Java, this is often accomplished mechanically through Proxy objects and ClassLoader modifications. In this stage of analysis, the analyst should consider the source code language of the target module, and any known mechanisms by which systems of interest in **S** might override standard access to code and data of target module M.

It is important to consider such mechanisms before diving in to analysis of target module M source code because it is important for the user to think about these concerns when examining the source code. The results of this analysis will impact the appearance of Behavior Points as the source code of module M is analyzed.

At this stage of analysis, the user fills out specific forms in the "Access Override Analysis" worksheet. The following fields (columns) are filled out to complete this analysis:

- **Declared Structure**: There are some granular units for which access policies or mechanisms might be overwritten. These is often file/modules in traditional compiled languages, with the addition of class or structure definitions in object-oriented and structured languages. For each such declared structure in target module M, the user should create a row in the worksheet and label this cell with aa sufficiently unique identifying name for the structure that can be found in the source code. In the example data provided with the standard, such a structure would be the declaration of the "CVisibilityGraph" class.
  For each declared structure, the following fields must then be filled out:
  - **Analysis Complete**: When the analyst believes the analysis is sufficiently complete (all data for the row is complete), then the analyst should place a value of "1" in this field. Otherwise, it should be left blank. If issues arise in the analysis that impact the analysis state, then this value should be left blank or returned to a blank state until all such issues are resolved.

- **Review Complete**: If the analysis for a given declared structure is complete, all issues are resolved, and reviewers have signed off on the analysis, a “1” should be assigned to this field. Otherwise, the field should be blank.
- **External Friend and Descendant Classes:** As part of this analysis, it is useful to list any “friend” or “descendent” classes, to be able to consider how their own access policy might be overwritten.
- **Override Mechanism**: In a comma separated list, list each override mechanisms applied to the declared structure. An example mechanism might be a non-standard “ClassLoader” that allows access to private members of a Java class.
- **Systems Affected**: In a comma separated list, list each system affected by one or more of the listed mechanisms. If desired, one can list the mechanisms affecting the declared structure in parenthesis following each system.

Then, the worksheet will provide the following indicator fields for the analysts’s convenience:

- **Descendant Access**: This field will automatically set a flag if there is any current friend or descent structure that can accesses the accessible contents of the given declared structure.
- **Mechanistic Access**: This field will automatically set a flag if there are any mechanisms permitting access to the internal elements of the declared structure.

The flags are not granular with respect to scope that is exposed. This must be determined in code analysis (later.)

Once the user has filled out the fields above for relevant structures in the target module, then this analysis is complete, and the analyst can move on to data analysis.

5.8.2 Data Analysis

In data analysis, the analyst determines what intraprocedural, persistent data objects are declared in software module **M.** Further analysis shows which of these data objects can be written to *directly* from outside of the target module, and therefore are **behavior points** in the module. It also supports analysts determining which are *directly* readable from outside of the target module and are therefore **control points** wherever writes to that data structure occur within the target module.

The analyst fills out a row in the table for each intraprocedural, persistent data object declared in the target module. Such an object has a name and type declaration in traditional programming languages (above assembly level.) Note, dynamic allocation of objects need not be considered. One need only be concerned with where the declarations of persistent data are made. Those locations will cover any dynamically allocated objects. Furthermore, one need not record data object declarations that only exist within the scope of a code object.

as follows:

- **Name**: The declared name of the data object in the source code. For example, this might be the name of a declared pointer of declared array, or a declared instance of a data structure/record/class.
- **Source Location**: The location within the target module M where the persistent data object is declared.
- **Analysis Complete:** This field should be filled out with the value 1 when analysis for this row of the table is complete by the authoring analyst(s). Otherwise, it should be blank. Issues are not part of determining completion.

If the analysts complete analysis for a given data object row, then the following fields present automated analysis results for identification of actuation and behavior points:

- **Access Potential:** These fields determine access potential to the data object
  - **Global Access:** This field will indicate whether all other system elements can access the data object
  - **Intended Global Access:** This field will indicate whether global access was intentional, or a design fault of the current implementation
  - **Friend/Predecessor/Desc. Write Access**: This field will indicate whether access is available to friend, predecessor super-classes, or descending subclasses of the object in an object-oriented languages.
- **Actuation Results**: These fields show whether the writing to the declared object can be considered an control point of module behavior.
  - **State Actuation:** This field shows whether module behaviors setting its values are an control point within potential systems.
  - **Intended State Actuation:** This field indicates that the state actuation potential of this declared object is likely intentional and not a design fault of the current target module implementation.

The above fields are automatically filled out when the user provides the remaining data on this spreadsheet for the given declared object row, and additional information is provided in Override Analysis (see the next sub-section) as follows:

- **Externalized/Global**: This field is marked TRUE if the data object is declared so that it will be globally visible or can be externalized to other system elements. In object-oriented languages, this typically means that the declared object is declared public. In C-like and other module languages, this means that the declared object is an exported symbol.
- **Part of Class:** This field is only filled out in an object-oriented language if the data object is part of a class definition.
- **Scope:** This field is only filled out in a language with explicit visibility scopes of the form **“**public”, “protected”, or “private”. The user should indicate the appropriate value here, leaving it blank if no visibility is explicitly declared, or if the language has no such construct.

- **Most open scope of methods exposing this actuatable for reads:** If the data has accessor methods that can return the value to outside the scope of the target module, then indicate the most open scope of such methods. The order of openness, decreasing, is typically “public”, “protected”, “private”.
- **Intended most open scope exposing this actuatable for reads**: Often, software is designed with unintended exposure of data. This field allows stakeholders to be questioned about the intended interfacing of such data via methods versus what is currently implemented. This does not cover confidentiality vulnerabilities.

When completed, the data on this spreadsheet will be sufficient for determination of which data structures are **behavior points** of the target module, and which, when written to, represent **control points** in target module behavior.

5.8.3 Code Analysis

In code analysis, the analyst determines (1) the interprocedural control-and-data-flow structure of module **M**. (2) The code is further analyzed to determine the source-code-level intended interface of module M. A sequence of analyses and labelling determines which procedures/methods/functions are behavior points for module M. The result identifies the behavior points and control points of module **M**. It allows analysts to distinguish actual exposure of code and data as interface from intended exposure of code and data. Often, the former exceeds the latter.

The standard includes a “Modular Code Analysis” tab in the “informalReverseEngineering.xsl” for code analysis. This tab helps the user analyze code and provide evidence that analysis is sufficient. Other evidence utilizing software reverse-engineering tools can be considered to replace this evidence in the assurance argument as appropriate.

For user reference, the example spreadsheet contains data from the example module analysis of “UxAS CVisibilityGraph”.

The table contains columns to be filled out by the analyst. Two basic columns are as follows:

- **Declared Code Objects**: Here, the analyst creates a row entry for any function, procedure, or method found in the target module **M**.
- **Type**: A module procedure’s use class can be placed here (e.g. constructor, method, accessor, etc.)
- **Analysis Complete**: This entry should get a value of 1 only when analysis of the procedure is complete. (All fields below have been entirely addressed. Issues that are recorded do not need to be addressed to receive analysis complete status.) The value should be withdrawn if issues exist in the analysis that must be resolved.
- **Review Complete**: The entry should get a value of 1 only when review of the analysis for the declared code object is complete. The value should be withdrawn if issues exist in the analysis to be reviewed.

What follows are fields that are automatically populated by the spreadsheet based on the data you provide in later columns. The automatically filled-out fields are:

- **Interface Points**: Sub-Fields that describe what kinds of interface points help define the declared code object.
    - **BPs:** Fields describing whether the declared code object defines a behavior point
        - **BP:** This flag is set to true if the declared code object is a behavior point
        - **Active BP**: This flag is set to true if the declared code object is a behavior point that is called at one or more places in the systems of interest.
    - **APs:** Fields describing whether the declared code object contains control points
        - **Contained APs:** This flag is set to true if the declared code object contains identified control points. This means there are one or more places in the declared code object that either call methods or write to data outside the module so that system state can change.
- **Access Potential:** This helper field helps you quickly assess what kind of calls are allowed to the declared code object under the typical rules of the source language.
    - **Global Access:** This flag is true if systems can be set up to call the declared code object from anywhere.
    - **Package Access:** This flag is true if any system element declared in the same package as the target module M can call the declared code object. This will only be true in languages with package access models.
    - **Friend/Descendant Access:** This flag is true if any friend or descendent class can call the declared code object. This will only be true in object-oriented languages.
    - **Mechanism-Based Access**: This flag will be true if a policy or mechanism of a system (such as access policy override, or mechanism (e.g., custom ClassLoader or Proxy in java) allows atypical access.

The remaining fields of a declared code entry are data from analysis entered by the analyst. They are as follows:

- **Access Attributes**: Attributes of access of the declared code object
    - **Externalized/Global**: Set to TRUE if the declared code object has a externalized symbol that can be linked and called from any other code, as in languages such as C. Otherwise set to false.
    - **Part of Class**: Record the name of the class that owns the declared code object if the code object is a method or procedure owned by a class. This field will apply to typical class methods as well as static class procedures, *etc.*, as found in various object-oriented languages.
    - **Scope**: For languages that declare explicit scope of code objects, indicate the scope of this object. (Typical values are “public”, “protected”, and “private”. The spreadsheet also recognizes “package” scope (such as with java when an explicit scope is not declared.)
- **Behavior Potential**: Field that help determine the behavioral design of the defined code object.

- **Activation Analysis:** These fields help the analyst determine whether the declared code object is a behavior point, and if and where it is currently utilized by systems of interest.
  - **Remote Calls**: Explicit calls to the code object from locations outside the target module within systems of interest. A comma separated list shows the location of each call within each system of interest.
  - **Local Calls**: Explicit calls to the code object from within the target module. A comma separated list shows the location of each call in the module.
  - **Direct Data Object Received/Handler Of Potentially Remote Triggered Objects/ Object Changes**: The target module might contain handlers that are called by systems when a remote object is received (e.g. notification handler), or when a remote object changes state (e.g. event handler). Such PBPs that are handlers are important to recognize in the target module M because they may interact with larger systems indirectly, through the appearance or modification of data. Here, one recognizes if a method (that might otherwise be private, for example) is called when the PBP is a handler of events on objects.
  - **Handled Asynchronous Signals/Traps:** Here, one determines whether the Module Behavior is called in response to system or hardware level signals or traps that could be generated outside of the target module. This is another indirect way that a system might interact to cause behavior in the target module, and these must be noted carefully. For example, if a procedure is set up through a system call to be the handler of a hardware trap, this must be noted, as the procedure is now an interface point of any system that can throw the hardware signal.
  - **Module-Local Exceptions/Signals/Traps Generation Sites**: This cell should list all places in the target model M that can throw the exception/signal/traps recognized by the handler in the field above. This can be tricky and depends on hardware, operating system, and other conventions, and therefore must be carefully analyzed.
  - **External Exceptions/Signals/Traps Generation Sites**: This should list all places in the systems of set S that can throw the exception/signal/traps recognized to be handled by the PBP. This can be tricky and depends on hardware, operating system, and other conventions, and therefore must be carefully analyzed.
- **Return Value Behavior:** Sub-fields describing data returned by the declared code object, including direct return function values, indirect value returns through pointer and reference parameters, and potential thrown exceptions.
  - **Returns a Value**: The procedure returns a value type. This is true and should be filled out even if the returned value is always the same or null. Void return types in C-like languages do not return a value.

        - **Thrown or Propagated Explicit Exceptions Outside Its Scope**: The procedure can throw exceptions that will return from the procedure and pass a signal to the caller. This is a type of returned value. Some languages make this explicit in a procedure's signature. Others do not. All exceptions thrown by source code, including by any code called by the current procedure, must be considered.
        - **Propagates Unchecked Exceptions Outside Its Scope**: Some languages have exceptions that can be thrown without any explicit indication in source code. These must be considered in any language that applies such a mechanism. (For example, Java "RuntimeException".)
        - **Reference or Pointer Parameters That Can be Written To:** If a procedure has non-constant reference parameters or pointer parameters (referencing parameters that can be written to), then their contents can return a changed value. These must be indicated in this field as they can affect system state.
- **Actuating Potential:** These sub-fields record analysis results for how the declared code object actuates systems, that is, changes persistent state of a system where the state is not visible outside the target module.
    - **Externalized Calls that Change System State:** The analysts must determine what calls from target module M outside target module M impact system state.
        - **Calls to Outside Module That Can Change System State:** If the declared code object calls a procedure outside of the module, then any change to system state (visible outside of target module **M**) that results from that call is an actuation. Therefore, any such call must have its further call-chain inspected to determine if there are any potential state change. This is any state change, including from unchecked exceptions that are not caught, data writes, system calls, etc. The locations of the initial calls are recorded in this field. This can be a difficult and detailed analysis of source code outside M**.**
        - **System Calls/Traps/Signals**: System calls/traps/signals made by the declared code object can change system state. They are to be recorded here.
    - **Changes to Externally Visible Data:** Here is listed any writes to intraprocedural, persistent declared data inside or outside target module **M** that is visible (can be read) outside of the target module. Such data writes represent changes to system state.
        - **Externally Visible Data Objects Written To:** The analyst records where in the procedure modifies the persistent data visible to outside systems. This includes persistent data in target module **M** and outside target module **M**.

Completion of the above analysis products for all declared code objects in the target module M completes initial analysis of module M. With the above information, the analysts has sufficient

and sufficiently clear information from which to identify all interface points requiring a contract within the module. The spreadsheet's automatically triggered fields aid this identification.

5.8.4 Security Analysis

The worksheet "unintended access consideration" is designed to enable listing of vulnerabilities in systems of interest that allow unintended access to declared code and data objects. A full security analysis approach is beyond the scope of the current standard. Users should fill out the worksheet with the data from adequately complete security analysis as follows:

- **Class / Object/ Structure**: A declared code or data object or class or structure of module M that has unintended access.
- **Vulnerability:** The vulnerability that allows the access
- **Provides Read Access**: TRUE if the vulnerability allows read of the declared object, otherwise FALSE.
- **Provides Write Access**: TRUE if the vulnerability allows write to the declared object, otherwise FALSE.
- **Provides Execution Access**: TRUE if the vulnerability allows execution of the declared object, otherwise FALSE.
- **Vulnerability Mitigation Requirement**: An acceptable security requirement that will prevent the vulnerability from being exploited. The analyst can utilize multiple rows to list requirements that are sufficient for different sets of systems of interest.
- **Systems to Which it is Applicable:** The systems of interest for which a particular vulnerability and mitigation requirement are applicable and to be applied, respectively.

5.8.5 Module Analysis Signoff

Before proceeding with later analysis stages, stakeholders should agree that the analysis of the system is sufficiently complete or correct. Signoff data is collected on the "Analysis Signoff" worksheet. Here all parties are indicating that the module analyses above are practicably complete and correct, including that there are no outstanding identified issues, and that all analyses and review were performed with acceptable quality.

5.9 Service and Module Flow Analysis

In this part of the reverse engineering process, analysts will determine the mixed control-and-data-flow graphs for the systems of interest relevant to the target module. The resulting graphs will have two key properties that must be assured:

- The system's graphs should at least completely capture the flow of control and data from services of interest to behavior points of the target module. This is step 3 in Figure 2.
- The system's graphs should at least completely capture the flow of control and data from utilized behavior points by the system on the target module to all utilized control points of the target module. This is the graph part of step 6 in Figure 2.

Both steps can be achieved through a manual, backwards static analysis.

5.9.1 Manual, Backwards Static Analysis

In this approach to constructing intraprocedural control-and-data-flow graphs, one starts with the interface model of the target model M, the already constructed code of module M, the already performed module analysis for M, and the source code of the target system Sys as input. One then manually inspects the source code of target system Sys to construct sufficient control-and-data-flow graphs over the system S to the behavior points M utilized by the system. One then constructs the graph from all control points of the target module M to all behavior points of the target module. This analysis is performed in reverse-direction from control-and-data flow. The procedure is as follows:

1. Start with the set of nodes **NS** consisting of each behavior Point (**BP**) of the target module **M**. Make each node **N** of **NS** a node in the control-and-data-flow graph.
2. For each node **N** of **NS**
    a. Remove **N** from **NS**
    b. If **N** contains sufficient direct semantics so that its functional and non-functional requirements in providing services in the target system can be completely and correctly states in natural language, then
        i. Determine the set of services provided by **N**
        ii. Add a service point node, **SP**, to the graph for each identified service provided by **N** that is not already present in the graph.
        iii. Add an edge to the graph from each service point node, SP, to N, where the service of **SP** is partially or completely implemented by **N**.
    c. Otherwise
        i. If **N** is a code object (*e.g.*, procedure, method, function declaration, or "goto" location label), then
            1. Find each code object **C** within System **S** that calls or jumps to **N**. For each code object **C**, add **C** as a control node in the control-and-data-flow graph, add an edge from **C** to **N** in the graph, and add **C** to **NS**.
            2. If **N** is a dynamically registered handler or triggered call on events, signals, object changes, *etc.* in system **S**, then the temporary object representing the event, trap, signal, object, *etc.* is represented as a data object **D**. For each such data object **D** of which **N** is a reactor/handler, add **D** as a data node in the control-and-data-flow graph, add an edge from **D** to **N** in the graph, and add **D** to **NS**.
            3. Find each persistent data object **D** within System **S** that is read in code object **N**. For each data object **D**, add each **D** to the control-and-data-flow graph as a data node, add an edge from **D** to **N**, and add **D** to **NS**.
        ii. If **N** is a data object (*e.g.*, persistent declared data structure instance, class instance, *etc.*), then find each code object **C** within System **S** that

writes to **N**. For each writing code object C, add **C** as a control node in the control-and-data-flow graph, make a data-flow edge from **C** to **N**, and add **C** to **NS**.

3. Repeat the above procedure by starting with a set of nodes **NS** containing each control point of the target module **M**. The terminating condition is altered to occur not when service points are reached, but instead when behavior points are reached.

The above algorithm can be augmented with existing static analysis tools. Many tools will produce intra-procedural control-flow graphs for a system. One can utilize such tools to automatically construct a control-flow graph. Then, utilizing the above algorithm, one can "follow" the control-flow graph backwards from behavior points, augmenting the graph with data flow and service points.

The main challenges of building a control-and-data-flow graph are knowing

1) How to follow dynamic call links through data objects, such as in object and notification handlers.
2) How to observe reads on persistent data and observe where such data is written from.
3) How to identify when a data object or function has clear functional and nonfunctional requirements at the level of real-world service.

Each of these is beyond the scope of this guide. The first two can be learned from static analysis literature and guides. The later requires understanding requirements in problem definition of real-world services, as defined in "Fundamentals of Dependable Computing" by John C. Knight.

Ultimately, identifying where service-level functional and nonfunctional requirements can be stated for a given control (*e.g.*, function, method) or data (*e.g.,* structure or array) object and then identifying the defined services will take practice.

The evidence model provided in "informalReverseEngineering" only shows the resulting analyses. Quality of this work must be determined by peers and signed-off on as a statement of quality.

5.9.2 Recording the Flow Graph as Direct Evidence

The "InformalReverseEngineering.xlsx" workbook includes a worksheet "Control-and-Dataflow Graph" for recording the captured control-and-dataflow graph. The columns of the spreadsheet are as follows:

- **Controlled**: The unique name of the controlled node (end node) of an edge in the graph(s).
- **Type**: The type of controlled or controller node in the graph. This can be one of "BP", "control", "data", "subservice", "service", "CP".
- **Controller**: The unique name of the controlling (start node) of an edge in the graph(s).
- **Type**: The type of controlled or controller node in the graph. This can be one of "BP", "control", "data", "subservice", "service", "CP", "start node".

The user can place the data for their control-and-data-flow graph here in table form, and this is considered sufficient evidence to show that an analysis was performed.

5.9.3 Review Evidence

Assuring correctness and completeness of the resulting graph benefits from a review step. In this step, the analysts review the resulting produced graph(s) for the systems of interest to be sure that:

1. All the services of interest have been identified for each target system.
2. For each service of each system, a covering set of service points have been identified in the control-and-dataflow graphs.
3. All of the {target module}'s behavior points utilized by each system are correctly and completely identified
4. For each system, the control and dataflow of the graphs between the service points and utilized behavior points appears correct and complete when compared against system's source code.
5. For each a given system, the control and dataflow of the graphs to utilized control points appear correct and complete when compared to the target module's source code.

This evidence is recorded in the spreadsheet "informalReverseEngineering" as signoffs from analysts. The last is recorded in the worksheet "Control-and-Dataflow Signoff". Evidence for the behavior points is referenced from the earlier module work.

5.10 Requirement Synthesis

Once one has identified adequate service points for the services of a system, then the flow of control and data from service points to the behavior points of the target module defines the paths of functionality over which the identified services are implemented in the system utilizing, in part, the target module. This is sufficient information to use as a guide for reconstructing functional and non-functional requirements on the target module behavior points (BPs) and control points (CPs). The process of doing so consists of:

1. Defining the functional and nonfunctional requirements of each service represented by service points. Also list any valuable preconditions that will be true when the service is initialized.
2. Decomposing the functional and non-functional requirements and preconditions into a set of entailing requirements and resulting preconditions on each edge of the control-and-data flow graph leading towards at least one behavior point. This is done for each node **N** for which one has already labelled requirements on all its incoming edges.
    - For requirements, the analyst considers all edges leading out of node **N**, examining the code and data of each node on the other end of each edge. The analyst determines what functional and nonfunctional requirements are 1) satisfied by the code and data of nodes at the end of each edge coming from **N**, 2) and that entail the requirements of edges directly leading to **N**. If one cannot find such a set for a node N, then it is possible that the software studied contains

significant design flaws. Otherwise, one should be able to deduce requirements to hold on each outgoing edge from N.
    - For preconditions, the analyst considers what the design and implementation of the given node N cause to be true on each edge leading out of N given what is true (requirements and preconditions) on the edges leading into N.
3. The process repeats recursively until all control and data flow is considered to all edges leading to all behavior points of the target module such that requirements and preconditions reach closure / stabilize.
4. The contract of requirements on each behavior point (BP) of the target module is the conjunction of the set of requirements on the incoming edges to the BP.
5. Repeating the above procedure to extend from behavior point nodes to extend functional and nonfunctional requirements decomposition from behavior points to control points within the target module source code.

5.10.1 The Sub-Steps of the Process

The steps of the requirement synthesis process are as follows:

1. Given a target system Sys with modeled control-and-data-flow graph produced previously and a blank "Requirement-Labeled Flowgraph" worksheet in "informalReverseEngineering.xml"
2. Copy all the edges from the control-and-data-flow graph worksheet "Control-And-Dataflow Graph" to the worksheet "Requirement-Labeled Flowgraph".
3. For each edge that starts (has a controller) that is a service node, **SN**, create a new row in the table where the controller is left blank and the controlled element SN.
4. For each service node **SN** of the new graph created in step 3:
    a. Create an entry in the Requirement Flowgraph worksheet where the Controller columns are labelled with the service node SN, and the Controller columns are left blank.
    b. Define the functional and nonfunctional requirements of the service represented by the node **SN** to this edge.
    c. Add the preconditions that hold on service initiation to this edge.
5. Iteratively perform basic static analysis on the source code represented by each node of the flow graph.
    a. For a node **N** of the graph with incoming edge set **IE** and outgoing edge set **OE**,
        i. Examine the source code to deduce what each edge of **OE** is required to guarantee to entail the union of the requirements on the incoming edges **IE**
        ii. Given the preconditions that are entering N on edges IE, consider what preconditions are implied on outgoing edges in OE. The more one analyzes the internal structure of the procedure (intraprocedural control-flow), the more detailed this static analysis can be. Dynamic analysis (e.g. symbolic execution) can supplement.

   b. Repeat step 5 until all requirements on all edges of the graphs reach stable closure.
6. For each edge ending at a behavior point of the target module, review the software with stakeholders to determine which aspects of the design of the implementation of the behavior point, and subsequent control-and-data-flow to control points, should be preserved. This might include re-use of data structures, organization of code, use of design paradigms, etc. Each of these is captured as a "design constraint" non-functional requirement on edges representing control and data flow from the system to behavior points, and on edges representing the internal solution of the target module.

The result of this algorithm are requirements for each interface contract of each behavior point of the target system. These contracts are relative to the system of interest. If the target module is utilized by many systems, a sufficient reference set of systems of interest must be similarly analyzed.

5.10.2 Step 4b: Determining Functional and Nonfunctional Requirements at Service Points

Functional requirements for services are those aspects of service to the environment performed by a system of interest. Given a target system Sys, each of its services will perform some operation on the surrounding environment as a real-world problem solution. The key to developing a list of service-level functional requirements is to avoid, to the extent possible, any implication of a solution design. That is, the design of the system providing the service should not be captured in these requirements.

The example workbook includes sample service requirements for Route Planning and an Operating Region Declaration sub-service.

Developing service-level nonfunctional requirements is often a matter of subject expertise and beyond the general scope of this guide. Each attribute has associated expert analyses approaches. The arguments of this standard assume separate microstandards to develop these service requirements for each nonfunctional attribute type.

5.10.3 A Simple Safety Analysis for Step 4b

For illustrative purposes, this guide includes a very basic form of FHA safety analysis for service-level requirements. The assurance argument for this approach is found in "safety-synthesis.gsn". The accompanying process consists of the following steps:

1. The analyst makes sure that the functional service requirements are complete for each service of the system before proceeding.
2. The analyst works with stakeholders to enumerate the accidents and losses that can occur because of use of the target system's services. For each service, one works with stakeholders and experts to enumerate each of the accidents or losses that would be unacceptable.
3. The analyst then enumerates hazardous states of the world that would presumably result from application of the service. A hazardous state is a service state that is "one step away" from leading to an accident or loss. At the service level, these hazards are

reflected in environment state that can emerge from use of the service. In the example "informalReverseEngineering.xml" evidence for UxAS, hazards relate to potential accidents involving vehicles using routes produced by the Route Planner service.

4. Finally, for each service, the analyst enumerates service failures that might result in each enumerated hazard. Example failures for the analyzed service are presented in the example UxAS data.

The evidence required for the backing assurance argument is store by the analyst in the "Service FHA" tab of the "informalReverseENgineering.xlsx" spreadsheet. It consists of the following fields for the above activities:

- **System**: The target system of concern
- **System Signoff**: Signoff from stakeholders and experts that the system service-level FHA is sufficiently complete and correct.
- **Accidents**: A list of unacceptable accidents or losses that can arise in the system.
- **Accidents Signoff**: Signoff from stakeholders and experts that the system service-level accidents list is sufficiently complete and correct for the given system.
- **Service-Related Hazards**: Hazardous states in the environment of the system that can arise from provision of the system's services.
- **Hazards Signoff**: Signoff from stakeholders and experts that the system service-level hazards list is sufficiently complete and correct for the given accident.
- **Service**: The service(s) that might influence entering the enumerated hazardous states.
- **Service Failures:** The failures of each service that might result in each service-related hazard.
- **Service Failures Signoff**: Signoff from stakeholders and experts that the system service-level failures analysis is acceptably correct and complete for the given functional service definition.

5.10.4 Step 5: Basic Forward Requirements and Preconditions Analysis with Iterative Closure

The goal of the analyst is to develop a set of functional and nonfunctional requirements for each edge of the control-and-dataflow-graph. This is done through inference on implication. Assume a node **N** of a requirement-labelled control-and-data-flow graph. The analyst's goal is to deduce the requirements on each edge leaving N based on the union of all requirements on edge entering **N**.

Because of cycles in control-and-dataflow graphs, it will not always be possible to complete analysis of each node of N in a single iteration. Sometimes, a node's outgoing edges must have partial requirements defined until looping edges are analyzed on return to previous nodes. One must then update all edges leaving N again with the newly input requirements and continue until the requirements on all edges stabilize. This is referred to as closure under static analysis.

In such cases where only some information is available, that is deduced across the flow graph to further nodes until resolution is reached. Many iterations on sections of the control-and-

dataflow graph may be required. The reader is encouraged to examine the literature in static code analysis and Floyd-Hoare logic to formally learn the requisite skillset.

Finally, it should be noted that entailment and closure only occur when considering potential requirements on control and dataflow that is not explicitly represented in our graphs. At any given node N, there is a set of outgoing edges ES that are paths found in backwards static analysis from control and/or behavior points. Additional outgoing control and data flow might occur at node N that does not reach behavior and control points. It is assumed in this guide that these semantics are taken into account when reviewing source code or data at a node to determine an appropriate decomposition on those edges that are explicitly represented.

In contrast to requirements, in which the user attempts is attempting to reverse-engineer a set of satisfactory postconditions for each edge, precondition analysis merely determines what is implied by the operation of a given node leading to an edge. For more information, the user is referred to classic static analysis with Floyd/Hoare logic and inference over control flow graphs and workflow charts. Like requirements synthesis, looping can cause a need for iterative analysis of the graph structure.

Unlike requirements synthesis, the more detail of *intraprocedural* static analysis one performs, the more precise one can make determined preconditions. The graphs produced in this standard are inter-procedural. Intraprocedural analysis (analyzing over control flow within a procedure (*e.g.*, branching statements and for loops in a procedure) is beyond the scope of this standard and is discussed in more detail in static analysis literature.

5.10.5 Step 6: Design Preservation

Design preservation is the task of identifying any elements of the current target software module that should be preserved in its re-engineering. Everything else that will exist in contracts for the target module will either be functional or nonfunctional requirements describing what the target module's behavior must be like. In contract, design preservation places constraints on what the solution to the behavior must be like. These are recorded as "design constraint" nonfunctional requirements.

It is critical to work with stakeholders of a system, including code maintainers, to understand the principals of design and implementation that should be preserved in the target module.

For each edge of the control-flow graph that terminates at a behavior point, all aspects of design and implementation of that behavior point in the current target module that are to be preserved should be recorded as design constraint nonfunctional requirements.

5.10.6 Nonfunctional Analyses

This standard includes consideration of functionality and dependability attributes of interest.

5.10.7 Review Evidence Generation

Additional fields are present for each row of the requirement-labeled control flow. They are designed to help track which analysis work has been completed and reviewed. The following

fields of the "Requirement-Labeled Workflow" tab in the evidence spreadsheet should be filled out as analysis work commences:

- **Requirement Analysis Completion Checks**: For each edge of the graph, indicate which types of requirements have been analyzed and considered for the edge. Leave blank if not yet complete or place a value of 1 if the analysis is complete for the given requirement type. One need only complete analysis for the attributes of interest.
- **Signoff**: When peers have reviewed the analysis results for a given edge and agree that it is sufficiently complete and correct, then they place their name in this field.

5.11 Contract Consolidation

Once the control-and-dataflow graphs are complete and correct with respect to form and labeling with requirements, one can extract an interface contract for each behavior and control point.

- The contract for each behavior point is the conjunction of all sets of requirements and disjunction of all sets of conjoined preconditions on incoming edges to the behavior point. It also includes conjunction of all design constraints of edges implementing the behavior point in the target module.
    - A contract takes the form of an assume/guarantee contract.
        - The assumptions are the preconditions on the incoming edges to the behavior point.
        - The guarantees are
            - The requirements on the incoming edges to the behavior point.
            - The design constrains on any edges between the behavior point and any utilized control points.
- The contract for each control point is the conjunction of all sets of requirements and disjunction on all sets of conjoined preconditions on incoming edges to the control point.
    - A contract takes the form of an assume/guarantee contract
        - The assumptions are the preconditions on the incoming edges to the control point.
        - The guarantees are the requirements on the incoming edges to the control point.

Assurers should perform the above conjunction and disjunction operations, and then present the contracts for peer review.

Contracts for a set of systems are created by merging the contracts for each interface point for each system, where guarantees (requirements) are conjoined and preconditions are disjoined.

5.12 Outcome

The outcome of the process should be a set of contracts. Based on the evidence provided, following reference process, stakeholder should be assured, to the extent that evidence is of quality and accuracy from the perspective of evidence reviewers, that that informal reverse engineering has produced accurate and complete natural language contracts on the interface between the target module M and a system of interest S.

6 Requirement Quality

A second element of the standard is determining that the contracts produced for the target module in the target system consist of assumptions and guarantees that contain statements of sufficient quality.

We can think of the assumptions and guarantees on contracts as requirements on the target system and target module interface points. Therefore, we apply a simple Dependable Computing requirement quality microstandard: “microstandard-requirementquality-v1”.

7 References

1. Knight, John C., and Jonathan Rowanhill. "The indispensable role of rationale in safety standards." International Conference on Computer Safety, Reliability, and Security. Springer, Cham, 2016.
2. Holloway, C. Michael. *Understanding the Overarching Properties*. National Aeronautics and Space Administration, Langley Research Center, 2019.
3. Knight, John. "Fundamentals of Dependable Computing." CRC Innovations in Software Engineering and Software Development: Boca Raton, FL, USA (2012).