Understanding Integration Testing

Alexander Pretschner[0000−0002−5573−1201] and Lena Gregor[0000−0002−7909−5086]

Chair of Software and Systems Engineering, TUM School of Computation, Information and Technology, Technical University of Munich, Munich, Germany
{alexander.pretschner,lena.gregor}@tum.de

Abstract. What is integration testing? There are multiple definitions in the literature and relevant standards. The notion of integration testing also is understood and implemented differently across and even within organizations. We define integration testing intensionally, delineate it from unit, subsystem and system testing, and summarize analytical and constructive approaches at integration testing. We argue that *partial negative specifications* which reflect recurring integration faults can help detect and even avoid integration problems.

1 Introduction

Software is the most powerful driver of innovation today. Powered by communication over the internet, software enables the integration of things, data, concepts, and systems. Because of the enormous complexity of the integrated systems and data we build, integration is becoming ever more relevant and difficult.

In addition to conceptual integrity, the integration process requires to match hardware and software interfaces. In practice, this turns out to be a very difficult task: What does it mean for interfaces to match? How can we make sure they match? How do we check if they match? In this essay, we focus on matching software interfaces and consider hardware interfaces only in passing. While many ideas we discuss can and probably should also be used constructively, we will specifically focus on the activity of integration testing.

The literature roughly suggests two ways to understand integration tests. One understanding is that integration tests aim at testing the *combined functionality* of some subset of components, or units. The other understanding is that integration tests aim at evaluating the *interaction* in-between components, most often without saying what exactly this means or entails. In fact, one standard [1] uses both definitions within the same standard, where "integration tests" are the former and "integration testing" is the latter.

Academic distinctions may not be so relevant to practitioners. However, our own anecdotal experience is that different companies and even different teams within one company have different understandings of integration tests, and do integration tests to vastly varying extents. Some observe in hindsight that unit and integration tests seemingly tested for the same "thing;" some observe that problems found by integration tests could have been found using unit tests; and some realize that unit tests fundamentally fail to identify relevant problems.

Test pyramid. This is in line with differing perspectives on the test pyramid. The idea is to have many automated unit tests; fewer (possibly automated) integration tests; and as few end-to-end tests as possible. At the same time, different organizations today observe that the structure of their tests rather resembles ice cones, honeycombs, sand glasses, or any other shape.

The Influence of Architecture. The architecture of the system matters, of course. If we think about a system written in Java, we may have at least a clear understanding of what a unit test is. If we then integrate units and test these, this intuitively becomes an integration test. But if we test the composition of two units, what are the consequences for the *intention* of tests? If we re-use the unit tests and simply replace drivers or stubs (mocks) with the actual components, does this turn unit tests into integration tests? If we consider a microservice architecture, what is the difference between a unit and an integration test?

Granularity. Integration testing happens at several levels [2]. "Units" to be integrated include classes, components, things, (micro) services, full-fledged enterprise systems as well as electronic control units (ECUs) in cars. We will use examples from these domains throughout our essay, but our treatise aims at an understanding that is independent of the specific notion of unit. We do, therefore, not think that the conceptual ambiguities about integration tests are a result of different notions of "unit."

Development Process. In addition to the architecture and granularity, the development process clearly has an impact on the understanding and necessity of integration tests. If software is developed by distributed teams, possibly under different governance, the integration problem seems different from one single agile team that develops a specific piece of software: Intuitively, it matters if the testing process is intertwined with or even precedes development, or if it is performed after development, possibly by a different team.

Why is it difficult? While there may be no consensus on what exactly integration testing is, integration *problems* are widely acknowledged. Why do we have these problems? When system components are developed, there is both explicit knowledge and implicit assumptions about the technical context of the system: Among others, APIs and database schemas constitute explicit knowledge while the interpretation of numbers or the communication format, among many others listed in §4.1, often remain implicit. If two component implementations then make conflicting assumptions, integration problems are unavoidable.

Overview. The remainder of this essay is organized as follows. §2 introduces our formalism, the way it can be used to model development processes, and recaps the fundamentals of testing. §3 introduces different test levels. We argue that there are two kinds of integration faults that relate to (1) inadequate design decisions an6d (2) to inadequate connections at the implementation level. §4 argues that integration tests should address integration faults, defines the latter, and provides examples. A key insight will be that we cannot expect specifications for integration testing, and that lists of integration faults can be used as negative specifications for integration testing. §5 uses these insights pragmatically, and §6 concludes.

2 Preliminaries

2.1 Behavior Descriptions

Throughout this paper, we gently make use of formalism where we believe it can help clarify concepts or lead to more succinct descriptions. We see the development process as a transformation of *behavior descriptions* [3] in the spirit of Broy's work on using streams for formally founded model-based development [4]. In the simplest form, behavior descriptions relate input to output. We extend this notion to nondeterministic as well as underspecified behavior descriptions that relate input to sets of output. Where appropriate, we may assume that the formalism includes notions of logical or real time, in order to capture real-time behavior or the performance of a system. We may also assume that the formalism captures special security properties such as confidentiality expressed as hyper properties. Because the precise nature of the formalism is immaterial to our argumentation, we refrain from incorporating the latter properties and stick to the intuition that behavior descriptions are predicates whose semantics are a mapping from streams of input $\mathbb{I}$ to streams of sets of output $\mathbb{O}$,

$$(\mathbb{N} \to \mathbb{I}) \to (\mathbb{N} \to 2^{\mathbb{O}}).$$

Software and systems development is based on user requirements $\mathbb{U}$ and then possibly systems requirements $\mathbb{R}$. At least in principle, requirements are turned into *system specifications*, which are turned into *subsystem* specifications, turned into *unit specifications* and ultimately *implemented* by code. All these artifacts can be interpreted as behavior descriptions, but they usually do not exist as formal documents. The notion of subsystem is recursive where units are implemented but otherwise not further refined. Subsystems hence include units; we also use the word *component* as a synonym. We generalize each of these development steps into a transformation $\rightsquigarrow$ between behavior descriptions, where the left behavior description is turned into the right behavior description.

Nondeterminism is one way to express underspecification: if for a given input, any output is possible, this amounts to leaving open or *not* specifying the reaction to that input. The behavior description of a specification is meant to reflect the requirements—but it may take some additional decisions that were left open by the requirements. Similarly, the specification of subsystems should reflect the system behavior, but may add additional behavior. Finally, implementations of units are meant to reflect the specifications of the latter but themselves are allowed to take further decisions. If this is the case, our development steps are stepwise refinements in a formal sense. Refinement hence amounts to reducing non-determinism, commonly expressed as logical implication between behavior descriptions, or as trace inclusion at the semantic level. In order not to clutter formalism, we will not distinguish between syntax and semantics: For requirements, system specifications, subsystem specifications and implementations P, Q, $P \rightsquigarrow Q$ is a refinement step iff $Q \Rightarrow P$ or, semantically, iff

$$\forall i \in \mathbb{N} \to \mathbb{I}, t \in \mathbb{N} : (Q(i))(t) \subseteq (P(i))(t).$$

We do not think that stepwise refinement is or should be a realistic development methodology. We also do not advocate that formalized behavior descriptions should drive a development process. We are perfectly fine if behavior descriptions only exist virtually. In our context, which is independent of a specific development methodology, we use formalism solely to characterize the testing activities, not the development activities.

2.2 Development

Many design steps are functional decompositions. Because once again an exact formal definition of the composition is immaterial to our paper, we assume some composition operator $\oplus$ with a well-defined semantics that type-correctly connects inputs and outputs of the subsystems. One example for such an operator is provided by the Focus formalism [4], where the semantics simply is the logical conjunction $\wedge$. We will write $S \rightsquigarrow S_1 \oplus \ldots \oplus S_n$ to denote the decomposition of S into n subsystems and their respective connections. We will use the same symbol $S \rightsquigarrow I$ to denote an implementation step.

An idealized development process starts with a set of user requirements $\mathbb{U}$, refined into system requirements $\mathbb{R}$, in turn refined into an overall system specification S with $\mathbb{U} \rightsquigarrow \mathbb{R} \rightsquigarrow S$. The system S is functionally decomposed via $S \rightsquigarrow S_1 \oplus \ldots \oplus S_n$. Each subsystem S_i can recursively be decomposed into smaller subsystems. When the design process comes to an end, subsystems S_i are realized by implementations I_i in implementation steps $S_i \rightsquigarrow I_i$. If a subsystem S was decomposed by $S_1 \oplus \ldots \oplus S_n$ and realized by implementations $I_1, \ldots, I_n$, then $I_1 \oplus \ldots \oplus I_n \rightsquigarrow I$ is the integration step that leads to the implementation I of S. A subsystem's implementation I may be used in further integration steps to form higher-level subsystems.

While we focus on software systems in this essay, our conceptualization also works for cyber-physical systems (CPS). The behavior of CPS usually cannot be reduced to I/O relations only as it includes attributes such as weight, energy consumption, temperature, etc. Clearly lacking elegance, we can model these as part of the system output. At the level of specifications, we then need to include a subset relationship between ranges of values in order to cater to refinements. At the level of implementations, we do not consider refinements between implementations, but only between implementations and possibly hypothetical specifications.

Moreover, CPS development includes a further software-hardware integration step, concerned with the mapping of software components (tasks, communication) to hardware components (processors, memory, busses). We acknowledge the potential for confusion because of the terminology and consider this step to be a mix of *design* and *implementation* steps which include the partitioning of tasks, the choice of schedules, the exploration of and decisions about hardware design trade-offs related to cost, reliability, availability, weight, energy consumption, etc., and the deployment of software components to hardware resources.

There is a long tradition of mathematically modeling development steps as logical implications or, semantically, as trace inclusions, and thus understanding

them as refinement steps in the above sense (the only exception is the integration step, which is rather the opposite: $I \rightsquigarrow I_1 \oplus \ldots \oplus I_n$ is or should be a refinement, not $I_1 \oplus \ldots \oplus I_n \rightsquigarrow I$—but we need to implement before we can integrate, and in agile processes, we continuously integrate).

From a technical perspective, there is no need for testing in an ideal world. If during design it can be shown that $\mathbb{R} \Rightarrow \mathbb{U}$, $S \Rightarrow \mathbb{R}$, recursively that $S_1 \wedge \ldots \wedge S_n \Rightarrow S$ for all subsystems and that $I_i \Rightarrow S_i$, then the implementation I of the system satisfies $I \Rightarrow \mathbb{U}$ by construction, under the assumption that the composition of implementations reflects the semantics of the dual decomposition steps and also includes specifications of all the infrastructure components, including libraries, middleware, virtual machines, etc. Note the similarity with the communication and computation infrastructure for CPS mentioned above.

2.3 Testing

In reality, of course, things are more complex. Requirements change; specifications are inexistent, incomplete, wrong, or bad; design steps cannot be shown to be refinements; and implementations cannot be proved to be refinements either. This motivates the need for incomplete verification steps: checking the correctness of implementations $I_i \Rightarrow S_i$ is approximated using unit testing; checking the correctness $I_{j1} \wedge \ldots \wedge I_{jn_j} \Rightarrow S_j$ of subystem S_j implemented by $I_{j1}, \ldots, I_{jn_j}$ is approximated using subsystem testing; checking if the top level implementation I implements the requirements $\mathbb{R}$ is approximated with system testing; and $I \Rightarrow \mathbb{U}$ is verified during acceptance testing.

Let us assume a (sub)system S has been decomposed into $S_1 \oplus \ldots \oplus S_n$ and the S_i are implemented by I_i such that unit testing suggests $I_i \Rightarrow S_i$. Then the correctness of at least two composed unit implementations I_i with $i \in \mathbb{J}, \mathbb{J} \subseteq \{1, \ldots, n\}, |\mathbb{J}| \geq 2$ against a *hypothetical* specification $H_{\mathbb{J}}$ is checked using *integration testing* by approximating (i.e., testing) $\bigwedge_{i \in \mathbb{J}} I_i \Rightarrow H_{\mathbb{J}}$ such that $H_{\mathbb{J}} \oplus X = S$ for some equally unknown X. In case $|\mathbb{J}| = n$, we have $H_{\mathbb{J}} = S$; this is the last level of integration testing, different from system testing because the latter checks $\bigwedge I_i \Rightarrow \mathbb{R}$ rather than $\bigwedge I_i \Rightarrow S$ (cf.§§3.2 and 3.3). Jorgenson finds this distinction academic because in reality, the distinction between a system specification and a system requirements specification is not always clear-cut [5]. In practice, indeed, testing the last integration stage and system testing cannot always be clearly separated. In any case, we argue that the hypothetical specification $H_{\mathbb{J}}$ is the key to understanding what integration testing really is and should be, an insight missing in prior work [6].

The above formalization of design and testing steps suggests a development process that reflects the V model. Yet, our perspective is agnostic to the concrete development process. The activities are the same in other development processes, e.g., agile processes. The difference is the temporal ordering of the steps (when is integration taking place, when are requirements refined) and the nature of the specifications: In an agile world, specifications come as user stories, test cases, or acceptance tests. These are user *requirements* specifications rather than *(sub)system* specifications in a traditional sense; and yet they all give rise to

behavior descriptions. Hypothetical specifications $H_{\mathbb{J}}$ for integration testing are often captured by the previously implemented user stories. They unlikely exist as *explicit (sub)system specifications* in either development process, possibly with the exception of the last stage where $H_{\mathbb{J}} = S$.

Test cases, or tests for short, consist of input streams and expected output streams, and of environment conditions. Testing is the process of applying input to a system and comparing the system's actual output to its expected output. Test cases reflect specifications and are themselves partial behavior descriptions. Sometimes, test cases *are* the the relevant specifications, which is typical for test-first approaches such as test-driven development.

Testing in most cases cannot prove if one behavior description refines another but usually provides partial evidence or falsifies a refinement relationship. That is, testing in practice usually cannot show $\bigwedge I_i \Rightarrow H_{\mathbb{J}}$ but at most $\neg(\bigwedge I_i \wedge H_{\mathbb{J}})$. With the exception of code and tests, we use behavior descriptions only for conceptualizing our perspective. They are unlikely to be amenable to formal treatment in any realistic development process anyway, so this is not a huge practical concern—and the approaches in §5 do not require formalized specifications. Yet, for gaining confidence that the implementation of a system satisfies its specification and/or addresses the users' needs, the choice of relevant, or good, test cases is crucial.

Testing aims both at gaining confidence if requirements are implemented correctly and at revealing faults. In terms of the latter, the idea that good test cases are usually defect-based is spelt out in earlier work [3], getting back to early considerations by Weyuker and Jeng [7]. The starting point of our understanding is the text book definition of test cases being good when they reveal faults. As intuitive as this definition may be from a management perspective, it immediately leads to the odd conclusion that there cannot be good test cases for a hypothetical perfect system. It is then a short step to extend the definition of good tests to reveal potential faults, and marry that to the economical or risk-based intuition that not all faults are equally problematic, tests easy to debug, or cheap to run. In this vein, we consider tests to be good if they reveal potential faults with good cost effectiveness and will, in this paper, concentrate on faults being potential.

Requirements-based tests usually do not target specific faults beyond "requirement incorrectly implemented." If requirements are prioritized w.r.t. importance, respective tests can still be "good" as they address specified (and therefore probably relevant) use cases, which makes their incorrect implementation naturally risk-based and hence helps design tests that are cost-effective.

3 Test Levels

3.1 Unit Tests: $I_i \Rightarrow S_i$?

Unit tests are meant to check if the implementation I_i of a unit specification S_i is correct, that is, if $I_i \Rightarrow S_i$ (once again note that in most cases tests do not

provide proof, but we may content ourselves with falsification). In order to test such a unit independently, in practice we need to *mock*, or *stub*, those units that are used by I_i, be that by calls to other functions, messages to other services, or signals sent to other ECUs in a car. Depending on the development context, there is a chance that there is no explicit specification S_i for I_i, or that the unit tests themselves constitute S_i. In practice, even without specifications, unit tests are written by using knowledge about requirements and context.

Unit tests reveal *unit faults* that are usually rooted and fixed in the unit itself. In the V model, unit tests immediately follow the implementation step. In agile contexts, unit tests are written before or during development in a sprint, and are executed whenever code is checked in to the continuous integration environment. Within each sprint, a subsystem consisting of several features is built.

3.2 Integration Tests: $\bigwedge I_i \Rightarrow H_{\mathbb{J}}$?

Integration takes place whenever components (that may be units) are connected to each other. "Connecting" here means that previously mocked functions are now replaced by the respective real implementations. The way functions from other units are called depends on the system: In a simple system, these may be synchronous function calls. In more complex systems, this may be asynchronous inter-process communication; the sending and receiving of messages within some middleware; or the exchange of messages via the Internet in REST-based (micro)service architectures.

Assume we integrate a subset of at least two implementations I_j of components S_j with $j \in \mathbb{J}$ and $\mathbb{J} \subseteq \{1, \ldots, n\} \wedge |\mathbb{J}| \geq 2$ for a (sub)system S with $S \rightsquigarrow S_1 \oplus \ldots \oplus S_n$. We further assume that unit tests suggest that the I_j are correct w.r.t. S_j. Testing includes the comparison with expected results, which is why we explained in §2.3 that we need to assume a hypothetical specification $H_{\mathbb{J}}$ for this subset of components to be given—which, because in most cases $|\mathbb{J}| < n$ and only in one case $|\mathbb{J}| = n$, almost certainly does not explicitly exist. However, in agile processes, $H_{\mathbb{J}}$ can be seen to capture all user stories that were built in earlier sprints, and regardless of the development process, *it comes as the integration test suite and hence reflects the testers' intentions.*

Remember that we have defined $\wedge$ as the semantics of the composition operator $\oplus$, without making a distinction between $\wedge_\alpha$ at the level of specifications and $\wedge_\gamma$ at the level of implementations. Now assume that integration tests reveal $\bigwedge_{j \in \mathbb{J}} I_j \not\Rightarrow H_{\mathbb{J}}$. The respective design, implementation, and integration steps preceding this integration test are $S \rightsquigarrow S_1 \oplus \ldots \oplus S_n$; $S_j \rightsquigarrow I_j$; and $I_1 \oplus \ldots \oplus I_\ell \rightsquigarrow I$ for some $\ell \leq n$. Because the I_j are assumed to be correct by virtue of unit testing, and because we assume the semantics $\wedge_\alpha$ of $\oplus$ to be adequate by design of the formalism, there are two possible causes for the integration test to fail: the implementation of $\wedge_\gamma$ at the implementation level or the decomposition at the specification level.

Interaction Faults. The implementation of the "connection" (abstractly, $\oplus$ with semantics $\wedge_\gamma$) of at least two subsystems I_{j1}, I_{j2} does not work as expected by the design: The connection at the implementation level does not have the

same properties as the connection at the design level. At the level of *abstract behavior descriptions*, including both specifications and implementations, this mismatch cannot materialize: a logical $\wedge$ is an $\wedge$! However, this connection is implemented by some technical mechanism that transfers control or data. At the level of decomposing systems into subsystem specifications, we may just assume that the connection $\wedge_\alpha$ "works." The concrete implementation-level connection mechanism $\wedge_\gamma$, in contrast, is implemented by a simple synchronous method call or a remote procedure call in a full-fledged middleware (in which case we could also see the system as composed of another component, the middleware itself). It includes characteristics of the communication that are outside the control of the interacting partners, including jitter, latency, message loss, etc., and if these are instrumental for system functionality, they are a cause for integration faults. $\wedge_\gamma$ also captures the full complexities of hardware-software integration as introduced in §1.

We have defined tests to include environment conditions in §2.3. The common understanding is the configuration of the hardware and software stacks. However, one may also see characteristics of the transfer of control and data to be such environment conditions. This becomes apparent when considering electromagnetic interferences or cosmic rays as a source for a system's malfunctioning: These evidently are environment conditions.

The connection mechanism does *not* include the content, encoding, format of the messages or function calls or the way the recipient is addressed. These are the responsibility of the components that interact. At the level of the implementation, $\wedge_\gamma$ may be defective or not share all properties with its abstract counterpart $\wedge_\alpha$: It may be that $S_{j1} \wedge_\alpha S_{j2} \Rightarrow S_j$, $I_{j1} \Rightarrow S_{j1}$ and $I_{j2} \Rightarrow S_{j2}$, but $I_{j1} \wedge_\gamma I_{j2} \not\Rightarrow S_j$. This can happen, for instance, if messages are simply not forwarded by the communication infrastructure. We call these integration faults *interaction faults*. They are typically meant when integration testing is said to target the interaction between subsystems, cf. §1.

For pure software systems, the majority of integration problems does not seem to be due to the interaction itself. In contrast, interaction problems are more relevant, if not prevalent, when considering hardware components.

Design Faults. The decomposition $S \rightsquigarrow S_1 \oplus \ldots \oplus S_n$ was incorrect. That is to say, $S_1 \wedge \ldots \wedge S_n \not\Rightarrow S$ already at the specification level, but this *design fault* was not or could not be detected at the level of specifications, for instance, simply because of a lack of respective specification documents. At this level, we usually assume that the connection $\wedge_\alpha$ between the subsystems simply works correctly. However, the (possibly implicit) component specifications may have left open the communication format, message frequency, encoding, interpretation of numbers, and so on. Yet, these are the responsibility of individual components, not of the connection $\wedge_\alpha$. Implementations of the subsystems each took their own decisions. These were possibly correct w.r.t. their own possibly implicit specifications, but later turn out to be in conflict with each other. For instance, one subsystem implementation I_1 took the decision to use XML, in line with specification S_1, and another subsystem implementation I_2 to use JSON, in line with specification

S_2. These integration faults could, *in principle*, have been avoided and detected at the level of specifications during design step $S \rightsquigarrow S_1 \oplus \ldots \oplus S_n$, but have not; and therefore constitute *design faults*.

Without considering non-functional properties, Reiter calls these faults *architecture faults* [8]. We prefer the term design fault because not all design faults are architecture faults, see §4.3. Performance and security problems, for which integration testing probably is the wrong technique, are discussed in §4.5.

In practice, tests may also incorrectly suggest that the hypothetical specification $H_{\mathbb{J}}$ is satisfied, because they are incomplete. Yet, *at a later integration stage*, the system may not work as expected. This may also be because of the possibility of a fourth cause, incompleteness (or incorrectness) of the hypothetical specification $H_{\mathbb{J}}$ used during integration testing.

If $H_{\mathbb{J}}$ does not contain or contains inconsistent information about the transfer of control or data between two components, including format, timing, units, data types and data model, as spelt out in §4.1, incorrect implementation choices may be impossible to detect at the level of integration testing. In practice, the same holds for subsystem specifications S_i that are used for subsystem testing: If these do not contain information about timing, for instance, then two implementations are free to pick their own timing, and this choice may be in conflict with other choices, a recurring problem in the context of flaky tests, cf. §4.1. As mentioned in the introduction, integration faults are based on conflicting assumptions.

Depending on the development context and the need for standard conformance, there is a chance that there are specifications for full subsystems. In contrast, when integration testing is performed, the only explicit representation of $H_{\mathbb{J}}$ usually is the test cases themselves, containing input and expected output, written to reflect the testers' intentions. In practice, at this test level, engineers perform rather explorative testing to see if the interfaces of the subsystems "match." We will therefore suggest in §4.7 to approximate $H_{\mathbb{J}}$ by formulating such a specification negatively, based on commonly recurring integration faults.

3.3 System, Subsystem, and Acceptance Tests: $I \Rightarrow \mathbb{R}$? $I \Rightarrow \mathbb{U}$?

In a V modellish process, system tests are performed after system integration has taken place. Text books sometimes additionally require that the system's infrastructure be identical to the prospective deployment infrastructure. The goal of these tests is to show that a system's implementation I implements the (stipulated) requirements $\mathbb{R}$, $I \Rightarrow \mathbb{R}$. System tests are sometimes called end-to-end tests because they allow for user interactions that mimic those in the real deployment context. Good system test cases address important use cases and risky potential defects. In agile development, system tests are run continuously, where the integrated system consists of all those components that have hitherto been developed and those that need to be mocked. As there often are no explicit system requirements, checking is instead done directly against a subset of the user requirements that often come as user stories.

In addition to the perspective that system testing aims at verifying if $I \Rightarrow \mathbb{R}$, it is not uncommon to see system testing as the last stage of integration testing,

or at least not to insist on a clear-cut distinction [5]. As explained above, in this case the hypothetical specification $H_{\mathbb{J}}$ is not hypothetical anymore but instead is given by S, i.e., $H_{\mathbb{J}} = S$. We then test if the development $S \rightsquigarrow I_1 \oplus \ldots \oplus I_n$ actually was a refinement.

The idea of system testing recursively applies to *sub*systems S_i of a system S with $S_i \rightsquigarrow S_{i1} \oplus \ldots \oplus S_{in}$ and an implementation $I_i = I_{i1} \oplus \ldots \oplus I_{in}$. Subsystem testing checks if $I_i \Rightarrow \mathbb{R}_i$ holds for subsystem requirements $\mathbb{R}_i$, as long as these are available. We may also take again the perspective that a subsystem test for S_i actually is the last stage of the integration tests for the implementation I_i of S_i. In this case, the specification of S_i is actual rather than hypothetical.

Subsystem and system testing are done w.r.t. requirements. Integration testing is done w.r.t. possibly hypothetical system specifications.

Finally, acceptance tests verify if a system implementation I satisfies the user requirements, $I \Rightarrow \mathbb{U}$. Requirements specifications are likely incomplete, and respective faults may incorrectly be considered to be integration faults. We will consider three examples in §4.4.

4 Integration Faults

If good test cases target potential faults, cf. §2.3, then it is natural to require integration tests to target specific integration faults. As it turns out, there is a huge body of literature and practice on recurring faults in different system types. Interestingly, very few specifically target integration faults.

4.1 Examples

There is a wide range of known integration faults, including different expectations regarding units, encoding, data models, formats, data types, value ranges, frequency, quantization, message synchronicity, message or call ordering, schedule, information freshness and timeouts. Popular examples include the Mars climate orbiter that crashed due to unit inconsistencies where one component expected data in metric units of Newton-seconds and another component sent data in Imperial units of pound-seconds. Ariane-V exploded because two connected components operated with different representations, and hence value ranges, of integers. Buffer overflows can be seen as a further example of integration problems related to value ranges: input strings in one component are longer than expected by the memory allocated to buffer variables in other components. In microservice architectures, mismatches between communication formats JSON and XML are frequently observed. Moreover, as microservices are meant to maintain their own data and thus data model, changes to the model in one service are not always consistently implemented in other services. Respective inconsistencies are not uncommon in contexts where the data model frequently changes, as seen in the insurance sector with frequently changing regulations. Finally, flaky tests can be understood as integration problems. One source of test flakiness

are timeouts, where testers specify timeouts that have not been specified at all during design and implementation of the system to be tested.

We do not claim this list to be complete. Indeed, there is a huge body of literature regarding recurring faults for different system types. Some of these works include integration-relevant faults considering various aspects of integration, among them integration faults in monolithic (e.g. [9]), object-oriented (e.g. [2]), or cyber-physical systems (e.g. [10,11]), interface faults (e.g. [12,13,14]), and faults in the composition of services (e.g. [14,15]).

4.2 Integration Faults as Design Faults

In this essay, the lack of space makes us focus on design rather than interaction faults. We are fully aware of the importance of interaction faults specifically in CPS. That said, let us turn our attention to why *design faults* occur, and why they recur. We have already observed that in practice, the boundaries between the last stage of integration testing and system testing are blurred. Textbooks indicate that system testing is requirements-based and tests if $I_S \Rightarrow \mathbb{R}_S$. We have seen that system testing sometimes also is understood as the last step of integration testing where it is checked if $I_S \Rightarrow S$. Integration testing takes some subset of components and checks their composition against a hypothetical specification. In agile settings, that specification corresponds to previously implemented user requirements. We may speculate that in agile settings, integration testing corresponds to the flavor of integration testing that targets correct functionality rather than correct interactions between components.

We have argued that specifically software integration faults often are a special form of design faults. For simplicity's sake, let us consider the interaction of only two correct implementations I_1 and I_2 with specifications S_1, S_2 in $S \rightsquigarrow S_1 \oplus \ldots \oplus S_n$. Integration tests are executed against a hypothetical specification $H_{\{1,2\}}$ and fail: $I_1 \wedge_\gamma I_2 \not\Rightarrow H_{\{1,2\}}$. If the implementation of the connection $\wedge_\gamma$ between components is correct, the problem must be the *design* with $S_1 \wedge_\alpha \ldots \wedge_\alpha S_n \not\Rightarrow S$.

That the problem is a specification problem does not contradict the intuition that it can be *repaired* at the implementation level: It will often be possible to modify implementations I_i such that $\bigwedge I_i \Rightarrow H_{\mathbb{J}}$, independent of the specifications, and as long as $H_{\mathbb{J}}$ is consistent. Note that we use the term modify rather than refine here: implementations usually are fully refined. After this fix, it may be that $I_i \not\Rightarrow S_i$ which means that we would need to update the specification.

It is important to remember that $S_1 \wedge_\alpha S_2 \not\Rightarrow S$ can be due to the following situation: Both S_1 and S_2 leave open the same implementation choice, e.g., the communication format JSON or XML. This means that implementations I_1, I_2 that refine these specifications can decide differently, thus introducing an integration fault. And yet, this is a *design fault*: When designing $S \rightsquigarrow S_1 \oplus \ldots \oplus S_n$, the communication format should have been chosen consistently! However, as we lack ways to understand $S_1 \wedge S_2 \not\Rightarrow S$ at *design time*, we are likely to detect this problem only *after implementation*.

In sum, the problem can often be fixed in the implementations, but the root cause is the design. In order to understand integration faults, we therefore study them in terms of what can go wrong at the design level.

- If in order to repair the design fault, both S_1 and S_2 *must* and can consistently be refined into S_1' and S_2' such that $S_1' \wedge S_2' \wedge S_3 \wedge \ldots \wedge S_n \Rightarrow S$, then both S_1 and S_2 were underspecified. The implementations took conflicting decisions, probably inadvertently. This is the case that we studied above: *both* specifications leave open the communication format. Note that there is no unique component to blame for this integration fault.
- If it is sufficient to refine *only one* component specification S_1 into S_1' (or symmetrically, S_2 into S_2') and thus ensure that $I_1' \oplus I_2 \oplus \ldots \oplus I_n \Rightarrow S$, then S_1 initially offered implementation choices that *could* conflict with the design of the rest of the system, and I_1 happened to precisely take a decision that did conflict. This happens, for instance, if an implementation I_2 of S_2 is re-used (and thus equated with S_2) and performs computations with 32-bit numbers while S_1 *didn't specify* if numbers were 32 or 64 bits long, but some implementation I_1 decided to do 64 bits. It also happens when the integration of components leads to a buffer overflow.
  In this case, there is a conflict between two implementations that could have been avoided by fixing one component specification and making sure that it is correctly implemented.
- It may also be *impossible* to simultaneously refine both S_1 and S_2 into a consistent subsystem specification, i.e. for all refinements $S_i' \Rightarrow S_i$ and correct implementations $I_i \Rightarrow S_i$, we have $S_1' \oplus \ldots \oplus S_n' \not\Rightarrow S$. This is the case if component specifications took conflicting design decisions, for instance, if S_1 requires its implementations to communicate with I_2 at 100Hz and S_2 requires its implementations to communicate with I_1 at 33Hz. In this case, either one or both specifications (and their respective implementations) need to be modified, *not refined*, in order to conform with the behavior of the respective other component.
- Finally, one may be tempted to think that in addition to the above possibilities, it is possible that *any* of the two specifications be refined in order to fix the integration problem, but not necessarily both: In this case, $S_1 \rightsquigarrow S_1'$ with $S_1' \Rightarrow S_1 \wedge S_2' = S_2$ *or* $S_2 \rightsquigarrow S_2'$ with $S_1' = S_1 \wedge S_2' \Rightarrow S_2$ fix one specific observed integration problem.
  That is, both $S_1 \oplus S_2 \rightsquigarrow S_1' \oplus S_2$ and $S_1 \oplus S_2 \rightsquigarrow S_1 \oplus S_2'$ are possible refinements that fix the integration problem. However, this means that there is underspecification in both specifications for some inputs. If we remove this underspecification in one component, then implementation choices in the other component may, once again, lead to the same integration problem. In practice, this is unlikely to happen because only one implementation is actually changed. We have seen before that it is possible to resolve the integration problem at the level of either *implementation*. However, if the unchanged implementation is re-used with its unchanged specification in a different context, then this may lead to future integration problems.

For simplicity's sake, we have argued with two rather than more components. It is of course possible that an integration fails when three but not two components are integrated. Then, for instance, it may be possible to refine S_1 and S_2 as well as S_2 and S_3, but there still is an inconsistency in $S_1' \oplus S_2' \oplus S_3'$. The characterization of different integration faults, however, remains the same.

4.3 Other Design Faults

Are there design faults that are *not* integration faults? Intuitively, this is the case, for instance, if the intended functionality of one single component S_i happens to be incorrectly specified. This, however, begs the question of the reference w.r.t. which S_i is incorrect. There seems to be one special case where we do not need this external reference: a component S_1 is obviously problematic regardless of the rest of the system if it is *impossible* to refine (or implement) it in a way such that for *any* refinement of the other units $S_2' \oplus \ldots \oplus S_n'$, we have $S_1 \wedge S_2' \ldots S_n' \not\Rightarrow S$. In this case, the design of S_1 is "broken," must be modified, and cannot be repaired by the implementation of the remaining system.

Unfortunately, this case is of rather theoretical interest, as in practice the misbehavior of one unit often can be compensated by other units (exception handling, error correction, redundancy). However, we may accept the practical intuition that there is a design fault in the definition of S_1 if "the other units *could but should not* fix the problem in S_1" and refrain from attempting to formalize this idea.

Design faults which are not integration faults can also lead to the violation of non-functional requirements, including performance and security, cf. §4.3.

4.4 Requirements Rather than Integration Faults

Some faults may seem like integration faults but really are due to incorrect or incomplete requirements for a system at a higher level.

To start with, consider a car2infrastructure component that is meant to help optimize traffic. In order to do so, it provides information to vehicles about the situation behind a turn at a crossing: pedestrians, children, other cars, buses, etc. The input to such a component are signals derived from camera or infrared or other sensors; the output is a representation of the real world behind the turn. Acceptance testing such a system may then lead to challenging situations for the recognition capabilities of the controller—but what really matters is the safety of traffic, the throughput of the overall system, and so on. In other words, the relevant properties are system-level properties that transcend the properties of the controller subsystem. The controller is a subsystem, not the top-level system.

Higher-level properties often do not make sense at a lower level. This is the case for drone swarms, for instance, where one part of the system's desired behavior is to maintain a minimum distance to other drones. This is *controlled* by each individual drone but can only be observed if there is at least a second drone, i.e., after the top-level system "drone swarm" has been put in place.

Now consider an automated cruise control component, or ACC. Phantom jams occur when changes in the (usually decelerating) movement of one car are reflected and amplified by the following car, leading to a further amplification of the movement by the next following car, and so on. This phenomenon, called string instability, is provoked by many ACCs on the market [16]. Possibly overlooked by the designers of the respective control algorithms, it is however relatively easy to fix in the control algorithms themselves [17]. The point here is that the phenomenon cannot be observed by (acceptance) testing one individual ACC alone: String stability is a property of the higher-level system within which the ACC is embedded: traffic management. If string stability is ignored during the design of the ACC, it is unlikely that the ACC induces string-stable behavior. However, upon integrating it into cars and then into traffic, the property becomes relevant. From the perspective of the overall traffic, we may—incorrectly—see a lack of string stability as an interaction problem.

Finally, let us consider the problem of feature interactions [18], well studied in telecommunication systems, and also common in other distributed systems such as cars. Feature interactions occur when the combination of one or more features lead to undesired or at least unspecified system behavior. For instance, if n_1 enables call forwarding to n_2, which in turn enables call forwarding to n_3, should calls to n_1 be redirected to n_3, even though n_1 didn't explicitly say so? Similarly, if n_1 has blocked outgoing calls to n_2, but n_3 has enabled call forwarding to n_2, should a call from n_1 to n_3 be redirected to n_2?

These situations share the commonality that systems are embedded within other systems, and acceptance tests at a lower component level—the traffic controller, drone swarms, the ACC, the individual feature—naturally cannot reveal the problems at a higher level. Seeing them as *interaction problems* from the higher level is incorrect: They actually are *design* problems in the higher-level systems. We are aware that avoiding this kind of situations without knowing the potential problems upfront is a daunting task.

4.5 Non-Functional Properties

Performance, a system-level property, is influenced by architecture, be that the result of too many indirections in a multi-tiered architecture, be that the result of an inadequate network topology. Performance also is influenced by unit-level choices of adequate data structures—in which case bad performance could be traced back to unit faults at the level of design or implementation.

Similarly, security is a system-level property. Its violation can result from both inadequate implementations and inadequate architectures. We have discussed buffer overflows as integration problems above; we would argue along similar lines for most injection attacks. We also know that some definitions of confidentiality such as non-interference are non-compositional: individual systems may be secure, but their composition is not. Moreover, system-level security of course is influenced by the mechanisms that transfer control, possibly attacked by system call interposition or in-memory process patching, or data: communication channels compromised by a variety of man-in-the-middle attacks.

Non-functional properties like security and performance cross-cut all possible executions of a system and hence naturally matter when (sub)systems are integrated. Security and performance problems are rooted in protocol design, architecture, interaction, and implementation and can hence be seen as all three design, interaction, and implementation faults. Yet, because they manifest at the level of the overall (sub)system, integration testing is not the right level nor the right technique for checking these properties. Specifically, reviews or static analysis tools may be more adequate for the identification of security problems.

At this stage, it is of course natural to consider the notion of *emergent properties*, a topic, however, that we see outside the scope of this paper.

4.6 Locating Integration Faults

The above integration problems between implementations result from inconsistent design choices made during development of the integrated components. The reason for the possibility of inconsistency often is underspecification. Later design choices that, taken together, led to inconsistent behavior, have been taken inadvertently or because incorrect assumptions on the implementation of the respective other components were made. We have already seen that the integration problem can be repaired in the *implementation of any component*: Both modifications $I_1 \oplus I_2 \rightsquigarrow I_1' \oplus I_2$ and $I_1 \oplus I_2 \rightsquigarrow I_1 \oplus I_2'$ solve the problem, which can cater to both design and interaction faults.

This raises the question of where to locate integration faults. At the level of the *implementations*, there is no unique component to blame. As an example, consider two connected components where one receives an input that the other component processes and then fails with a buffer overflow. Truncation of the string could be done in either unit; and usually performance considerations lead to one choice or the other. It is a typical problem that the second unit later is re-used but assumes that the string has been truncated to the length of the buffer. The underlying problem is that the specifications of the two units did not agree on a maximum length for the input.

Even if we can fix an integration problem in implementations, our considerations show that the root cause often is the *design*, not the implementation. As we have seen, integration faults at the design level can be repaired by refining only one or by refining or modifying both specifications. If the implementation is not modified accordingly, this process may entail that one of the implementations is inconsistent with its specification after this refinement or modification.

Unit tests, in contrast, immediately indicate the location of the fault. And of course, if a specification had been refined in order to avoid later integration problems, this refined specification could have been used for unit testing, and unit tests would possibly have revealed an implementation fault. This explains the common intuition in practice that some integration problems could and should have been detected with unit tests, as introduced in §1.

It is tempting to include the notion of "fault for which there is no canonical location to fix it" into a definition of integration faults. Indeed, we believe that this is a useful perspective in terms of understanding integration testing. In

general, unfortunately, we cannot see, *beforehand*, if a test targets a fault that can be fixed by touching several alternative components. This seems, however, possible if we restrict ourselves to the above class of specific integration faults.

4.7 Integration Faults for Negative Specifications

The above argumentation essentially entails the following definition of integration testing: the verification if a composition of components—that often do not form a "full" subsystem which may come with an explicit specification—matches the implicit specification of this composition. We agreed that these specifications usually do not actually exist but are hypothetical instead.

In practice, testers write integration tests that themselves constitute this very partial specification. We now suggest to simply see the above list of integration faults as negative specifications, of course tailored to a specific context. $H_{\mathbb{J}}$ then may come as "absence of unit mismatch" or "absence of format mismatch," and the tester needs to find a way to instantiate these statements in a given context. This leads to a mostly methodological approach at integration testing: checklists of both design and interaction faults, some of which have been published in the literature [2,9,12,13,14,15], thus yielding an extensional but not an intensional definition of integration tests, which is the point of this essay. Obviously, it is even preferable to use these checklists during design and implementation of the system, in order to avoid rather than to identify them at a later stage.

5 Constructive and Analytical Approaches

Checklists. The use of checklists with known integration faults, tailored to a specific context, has been discussed in §4.7. While this of course cannot guarantee the absence of integration faults, awareness of respective problems is a powerful first step, specifically when systems are developed in distributed teams.

Contract-Driven Design. One recurring attempt at embedding precise partial specifications into the design process is the use of contracts [19,4,20]. We would argue that while the idea has been around for sixty years now, there are few places where it is implemented in practice. Let us venture to speculate that the reason is of a methodological rather than a technical nature: engineers need guidance w.r.t. what constitutes an adequate level of abstraction for these contracts. It seems like the "absence of recurring faults," adequately contextualized and formalized, can be a realistic level of abstraction for writing contracts.

Contract-Based Testing. One way to target rather simple integration faults such as data type, format, or range of values mismatch, is contract-based testing for service-oriented testing, which is more "lightweight" than other integration tests because it allows to test the interaction of two services in isolation: For consumer-driven contract testing, the developers of the consumer manually write tests to specify simple conditions to the interface of their provider. The provider team can then use these tests to verify if their service actually fulfills these conditions. Provider-driven contract testing works in the opposite order.

Fault Injection. Approaches for test adequacy assessment through integration fault injection into software systems or their interfaces are rather sparse, e.g. interface mutation for programs in C [21,22,23]. The above taxonomies and catalogues of integration faults can be used to inject realistic integration faults into component interfaces, which is the subject of our own current work.

6 Conclusions

Integration tests target integration faults. Integration testing is the process of verifying if some composition of components satisfies its specification. In contrast to subsystem tests, it is almost certain that such specifications do not exist. Instead, they come as the very integration tests that testers write. In addition to explorative testing if component interfaces match at this level, we have suggested to use the absence of integration faults as negative specifications and use this information to design integration tests.

We have argued that integration faults usually are *design faults*, at least for pure software systems without complex transfers of control or data. If the characteristics of the communication between two components, including latency, packet loss, etc., can impact the correct functioning of the system, integration testing also needs to target *interaction faults*. This is the case for hardware components where the nature of the communication channels very much matters and often constitutes the source of integration problems, e.g., network, wiring, voltage, or electromagnetic interference problems.

The methodological challenge behind integration faults is that during development, design decisions are taken that are based on assumptions on the technical context of a set of components. Integration faults materialize during implementation when these assumptions turn out to be incorrect.

Our conceptualization intensionally defines integration tests, is agnostic to the development process, and may help explain some of the confusion about integration testing: When practitioners feel in hindsight that unit tests could have detected integration problems, this is likely because shared assumptions were not understood, refined, and made explicit. When integration tests seemingly yield the same results as unit tests, then there likely is awareness of shared assumptions, but these are not explicit part of specifications and thus unit tests. If integration tests fail to reveal integration faults, they were probably not designed with potential integration faults in mind.

Acknowledgments. Tiziano Munaro provided valuable insights into the relationship between integration tests and software/hardware integration. David Marson observed the relationship between interaction faults and environment conditions in the definition of tests. Lars Alvincz, Daniel Elsner, Joachim Fröhlich, Anja Hentschel, Silke Reimer, Matthias Saft, and Horst Sauer helped us understand why there are so different perceptions of integration testing. Manfred Broy was instrumental in relating the abstract notion of refinement to practical development processes and in his skepticism about emergent properties. The anonymous reviewers provided invaluable feedback.

References

1. "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary," *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, Aug. 2017.
2. M. Winter, M. Ekssir-Monfared, H. M. Sneed, R. Seidl, and L. Borner, *Der Integrationstest: Von Entwurf und Architektur zur Komponenten-und Systemint.* 2012.
3. A. Pretschner, "Defect-based testing," in *Dependable Software Systems Engineering*, vol. 50 of *NATO Science for Peace and Security*, pp. 141–163, IOS, 2017.
4. M. Broy and K. Stølen, *Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement.* Springer, 2001.
5. P. Jorgenson, *Software Testing—A Craftman's Approach.* CRC Press, 2002.
6. M. Broy and A. Pretschner, "A model-based view onto testing," in *Model-Based Testing for Embedded Systems*, CRC Press, 2011.
7. E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Software Eng.*, vol. 17, no. 7, pp. 703–711, 1991.
8. H. Reiter, *Reduktion von Integrationsproblemen für SW im Auto durch frühzeitige Erkennung und Vermeidung von Architekturfehlern.* PhD thesis, TUM, 2010.
9. H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proc. Conf. on Software Maintenance*, pp. 290–301, 1990.
10. A. M. Madni and M. Sievers, "Systems integration: Key perspectives, experiences, and challenges," *Systems Engineering*, vol. 17, no. 1, pp. 37–51, 2014.
11. N. G. Leveson, "Role of software in spacecraft accidents," *Journal of spacecraft and Rockets*, vol. 41, no. 4, pp. 564–575, 2004.
12. J. A. Duraes and H. S. Madeira, "Emulation of SW Faults: A Field Data Study and a Practical Approach," *IEEE Trans. on SW Engineering 32(11):849–867*, 2006.
13. T. Nakajo and H. Kume, "A case history analysis of software error cause-effect relationships," *IEEE Trans. on SW Engineering*, vol. 17, pp. 830–838, Aug. 1991.
14. S. Bruning, S. Weissleder, and M. Malek, "A Fault Taxonomy for Service-Oriented Architecture," in *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pp. 367–368, Nov. 2007. ISSN: 1530-2059.
15. K. S. M. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea, "A Fault Taxonomy for Web Service Composition," in *Service-Oriented Computing*, pp. 363–375, 2009.
16. G. Gunter, C. Janssen, W. Barbour, R. E. Stern, and D. B. Work, "Model-based string stability of adaptive cruise control systems using field data," *IEEE Trans. Intell. Veh.*, vol. 5, no. 1, pp. 90–99, 2020.
17. R. E. Stern, S. Cui, M. L. Delle Monache, R. Bhadani, M. Bunting, M. Churchill, N. Hamilton, R. Haulcy, H. Pohlmann, F. Wu, B. Piccoli, B. Seibold, J. Sprinkle, and D. B. Work, "Dissipation of stop-and-go waves via control of autonomous vehicles: Field experiments," *Transportation Research Part C 89:205–221*, 2018.
18. P. Zave, "Secrets of call forwarding: A specification case study," in *Proc. Formal Description Techniques VIII*, vol. 43, pp. 169–184, Chapman & Hall, 1995.
19. B. Meyer, "Applying 'design by contract'," *Computer 25(10):40-51*, 1992.
20. P. Derler, E. A. Lee, S. Tripakis, and M. Törngren, "Cyber-physical system design contracts," in *4th Intl. Conf. on Cyber-Physical Systems*, pp. 109–118, 2013.
21. M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Integration testing using interface mutation," in *Proc. 7th ISSRE*, pp. 112–121, IEEE, 1996.
22. M. Delamaro, J. Maidonado, and A. Mathur, "Interface Mutation: an approach for integration testing," *IEEE Trans. on SW Engineering 27:228–247*, Mar. 2001.
23. M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur, "Interface mutation test adequacy criterion: An empirical evaluation," *Empirical Software Engineering*, vol. 6, pp. 111–142, 2001.