DOI: 10.15439/2017F133
ISSN 2300-5963 ACSIS, Vol. 11

Simulation Driven Development – Validation of requirements in the early design stages of complex systems – the example of the German Toll System

Tommy Baumann*, Bernd Pfitzinger†, Thomas Jestädt†
*Andato GmbH & Co. KG, Ehrenbergstraße 11, 98693 Ilmenau, Germany. tommy.baumann@andato.com
†Toll Collect GmbH, Linkstraße 4, 10785 Berlin, Germany.

***Abstract*—Looking at the end-to-end processing, typical software-intensive systems are built as a system-of-systems where each sub-system specializes according to both the business and technology perspective. One challenge is the integration of all systems into a single system – crossing technological and organizational boundaries as well as functional domains. To facilitate the successful integration we propose the use of simulation models in parallel to the existing software engineering procedures. As an example we look at the German tolling system for heavy goods vehicles (HGVs) – a liability-critical system consisting of some 60 sub-systems including a fleet of more than 1 000 000 on-board units deployed in the HGVs. Since its start in 2005 the system regularly undergoes changes and updates. To mitigate the associated costs and risks we developed a microscopic discrete event simulation (DES) model of the tolling system and use it to support both the design of planned changes and the monitoring of the day-to-day operations. The model includes the dynamic aspects of the tolling system and HGVs interacting with the system. In the article we discuss the use of realistic simulation models as part of the system design process. Since simulations are heavily used by the design process it is called Simulation Driven Development (SDD).**

I. Introduction

Historically, software development focused on standalone systems [1] and even there a projects' success was far from guaranteed [2]. Taking these approaches to build interacting systems bears a high risk of inadequate integration of the various systems into a coherent end-to-end system. Of course, problems with the integration of systems tend to surface very late in the software development process with a correspondingly large impact on the schedule and the resources needed.

A. Complexity Challenge

Modern technical and socio-technical systems consist of a large number of distributed components and are characterized by architectural complexity, dynamic interactions and complex interdisciplinary functionality. The continuing technical advances – e.g. in the field of electronics, where a 50% increase annually can be assumed – are one essential driver but also the emerging systems-of-systems accelerate the growth in complexity. In addition the requirements for these systems evolve rapidly, driven by end-user demands and non-functional aspects (e.g. in safety, accessibility and comfort). However, the efficiency of the existing system design methodologies evolves more slowly, e.g. [3] mentions increases of about 25% per year. This gap between the growth of the systems under consideration and the design methodologies used in their development has become a familiar terminology since the mid-1990s – the ”system design gap” [4]. This effect has been strengthened by shortened system life-cycles and time-to-market periods necessitating novel and improved system design methodologies and tools. In fact, an organizations' capabilities to develop and maintain IT systems are both a competitive advantage and a barrier that is difficult to overcome for any competitor [5].

Regarding the challenges of the system design process most of the critical system design problems originate in the early design stages when specialists are specifying the system under a high degree of variability and uncertainty. The *European Software Process Improvement Training Initiative* (ESPITI) in 1996 showed that the probability of critical problems due to poor design decisions is over 60% in the specification phase. The main reason for this high probability is that either text-based or non-executable model based specifications are utilized. These specifications cannot be validated in an integrated manner at a system level where the overall architecture and dynamic behavior are determined. The system design uncertainty remains high and the probability of errors too. In addition, crucial design steps are not fully automated e.g. enforcing validation after a design change. Hence traditional design processes are high risk and thereby highly expensive development processes [6].

B. Facing the complexity challenge

To overcome the complexity and integration issues we propose to introduce a holistic executable specification of the overall system accompanying the complete system development process. The executable specification can at any time be validated and optimized against the requirements of the integrated system. The validated specification of the integrated system can in turn be passed on to specialist teams for subsystem development and subsequent integration.

In this manner, integration problems surface in the early design stages rather than in the final test stages. As a consequence the development time and risk are reduced, specification quality and speed increases – albeit at the added expenditure of maintaining an executable specification. However, even after the completion of the product development such executable specifications can be of use in day-to-day operation (e.g. to predict and monitor the dynamic system behavior) and continued product development (e.g. validating architectural changes in the operational context).

The remainder of this paper is split into four sections. Section II introduces our system design approach where the executable specification transports the knowledge along the design process. Technically the executable specification is implemented as a simulation model, which collects and encompasses the known system requirements as explained in section III. The interplay between the requirements, the simulation model and the requirement management process is discussed in section IV whereas section V gives an example of a system where executable specifications have been applied.

II. The system design approach Simulation Driven Development (SDD)

Simulation Driven Development (SDD) is a system design approach for complex distributed systems and processes. It is characterized by applying modeling and simulation technologies during the whole system life-cycle (resp. product life-cycle). At its core is an executable system specification that exists during the whole system life-cycle encapsulating the current knowledge of the system, starting with the systems' conceptual design, followed by the design, implementation and test stages up to the day-to-day operations of the system and further development activities. At any time, the executable system specification represents the virtual prototype of the system to be built, the system under design or the system operated. The executable specification is kept up-to-date even after the real system or a real prototype is available. In that way it is at any time possible to test the system – either under construction or in operations – against its specification. In the SDD approach testing is preventive [7] before the system in constructed or the change is implemented. In a sense, SDD extends the test-driven-development paradigm [8] to the level of the systems' requirements.

In particular SDD emphasizes the integrated system as a whole and the dynamic coupling effects between the subsystems. One consequence of the increased knowledge of the integrated system as well as the system awareness, is a rapid improvement of the specification quality particularly in the very early design stages. This is in turn equivalent to a higher accuracy of the specification, i.e. less errors are expected in later test stages. Overall we expect SDD to increase the design and implementation speed and to reduce the overall development and operational risk considerably.

A. *Executable system specification*

In general an executable system specification defines the functional and non-functional properties of a system in a formal, consistent, and self-contained manner to enable processing [9]. Functional properties define the tasks of the system including information processing in relation to data, operation ("what the system should do" [10]) and the systems' behavior ("a behavior that a system will exhibit under specific conditions" [11]). Non-functional properties are more difficult

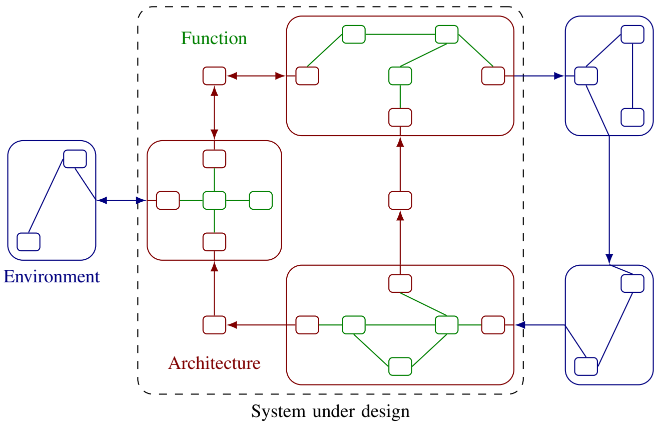

Figure 1. An executable system specification encompasses functional, architectural and environmental components.

to pin down – there is not even a simple consensus on the term and its use [12]. They are used to describe the circumstances necessary to render the required functionality, e.g. the performance requirements, quality properties and constraints (e.g. environmental and implementation constraints, platform dependencies [13] or the typical properties summarized as dependability [14]: availability, reliability, safety, confidentiality, integrity and maintainability).

In contrast to a system specification as a natural language text or non-executable models, executable system specifications are expressed by means of executable models [6]. These models include three component types (see figure 1):

- Functional components: Realization of functional system requirements (e.g. sending toll data at a certain time)
- Architectural components: Realization of non-functional system requirements (e.g. communication protocols and network topology of interacting subsystems, platform limitations)
- Environmental components: Description of operational scenarios with respect to mission objectives, and use cases of the system (i.e. dependability as listed above).

B. *The SDD design process*

The SDD design process consists of the typical design phases in system development: analysis/conception, design, implementation and test (see figure 2). However, in the SDD case all phases are accompanied by virtual and real prototypes which in turn are connected to a central requirement repository. This repository of all known system requirements enforces a revision control environment to store and manage prototype versions.

Each phase of the product development has different interactions with the requirements repository: During the analysis phase specifications are derived on a conceptual level concerning the systems' operational scenarios or use cases. Both functional and non-functional requirements are derived from the specification and entered into the repository. This set of requirements is the starting point to implement an

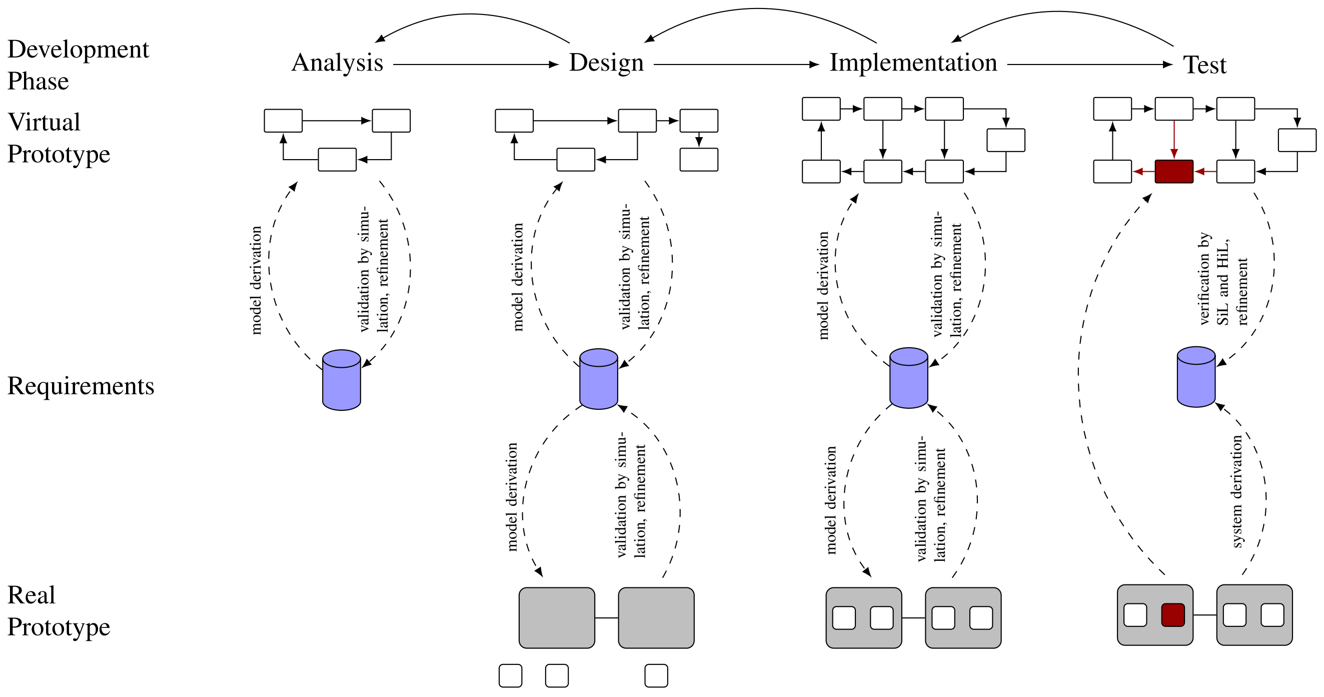

Figure 2. Design process of Simulation Driven Development: The requirements repository (center) provides the baselines for the virtual and real prototypes as well as the real system.

executable specification – a virtual prototype of the system to be developed. The simulation (i.e. running the executable specification with a given set of parameters) aims at validating the specification already at the level of the whole integrated system during this initial phase. In addition, the transition from natural language specifications to executable ones will automatically generate more detailed and rigorous specifications.

The design phase enhances the knowledge of the system under consideration in two directions: The solution space is explored through different virtual prototypes, e.g. to scope varying architectures or behavioral aspects. At the same time, each virtual prototype becomes in itself more specific by adding the necessary behavior and parameters to allow measuring its performance. Depending on the complexity and runtime the optimization can be delegated to an automatic optimization algorithm [15] – in that case properties of the simulation environment become themselves requirements, e.g. the execution performance. At the same time real prototypes are introduced to validate the design resulting in a feedback-loop: the requirements from both sides, the virtual and real prototype, are related and affect each other. The design phase ends when the variability of the potential solutions is reduced to a single solution specification – the starting point for developing the real system.

In the subsequent implementation phase the emphasis shifts from the virtual prototype to the real system under construction. However, the executable specification is kept up-to-date and mirrors the known requirements. In that way simulations are part of the decision making process: Implementation variants can be explored and compared through simulations. Each design decision is transported to the real prototype via the requirements repository and the executable specification – the latter being a representative of the whole integrated system which itself is still under construction [9]. Similarly design changes from the real prototype are transferred to the virtual prototype via the requirements repository to keep both prototypes consistent.

Finally the test phase is characterized by applying the refined virtual prototype to component tests, i.e. the validation and verification of the components through their defined interfaces [10]. To that end the soft- and hardware components implemented in the real prototype are integrated into a whole working system via the virtual prototype rather than the real world components (or artificial models thereof): So called Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) tests. In addition the virtual prototype is kept to support for the future system development, e.g. when new operational scenarios or new system architectures emerge. In any step, changes to the requirements can automatically invalidate parts of the virtual prototype and are the trigger to adapt and repeat the simulation runs.

III. Handling requirements in the SDD approach

Requirements are descriptions of how a system should behave, or of a system property or attribute [10], [16]. However, gathering the right set of requirements is a non-trivial task: In the early design phases the requirements will still be rather abstract and the impact on the real-world system is difficult to gauge. This is a particular problem for non-functional requirements, e.g. the behavioral aspects of the system [11] or its underlying architecture. Yet it is well-known from software cost models, that the initial investment in choosing the right architecture is important: A NASA recommendation [17] suggests a sweet-spot from the COCOMO II cost-model of dedicating up to 20 % of the software budget to the early analysis and to developing the right architecture.

Addressing the large amount of requirements generated in typical software-intensive systems, requirements are documented at different levels (or layers) of abstraction: The top layer defines why the system is build and what the owning organization hopes to achieve. This type is termed as *business or stakeholder requirements* [11] – an example would be to cut costs by reducing manual steps in a business process. Already at this level-of-abstraction the requirements need to be validated as soon as possible – is the requirement really necessary at the documented level? Seemingly inconsequential numerical targets can have profound effects on the technical solutions, [17] gives the example of 99% data completeness for scientific observations necessitating additional redundancies. The translation of the requirements into an executable specification allows exploring the effects of the requirements on the solution space early on. Vice versa, the virtual prototype transports operational properties of the real-world system back to the solution space potentially modifying or restricting the requirements.

The subsequent levels of detail produce additional layers of requirements where the whole system is defined in terms of an implementable solution. Each layer provides precise means of qualifying the solution and the requirements of a given layer are linked upwards to the next higher layer [18]. To that extend requirements are modeled as uniquely identifiable entities in the same way as all other elements of the prototype model. The resulting links from the different layers of abstraction form an important prerequisite for establishing formal traceability [19].

In SDD, like in the classic V-Model [20], the different types of requirements appear in the distinct development phases:

In the analysis stage very few high-level business requirements exist. They express the overall visions, goals and uses cases of the system under consideration. This initial specification is used to derive executable virtual prototypes for simulations of the system behavior. In light of the cost/benefit discussion above the virtual prototype aims in this stage at clarifying the overall requirements and system architecture – i.e. to identify the essential functionality and to avoid accidental complexity [21] in the overall system and its subsystem.

In the design stage the system architecture becomes more detailed, components emerge and their requirements are formulated. The executable specification helps in drafting accurate requirements and simulation runs yield the resulting dynamic behavior prior to the implementation of the system.

The implementation stage shifts the focus to the real prototype and the system under construction. In this stage the requirements are supposed to remain fixed and only minor adjustments need to be returned to the repository. The simulation model is an executable representation of the state-of-knowledge and is technically able to integrate a given component into the overall system – especially as long as the whole system is not yet available.

In the test stage, the high-level requirements are used for acceptance tests of the whole system. Usually the development of the virtual prototype precedes the development of the real system. In that case the already implemented components of the real systems are tested using Hardware-in-the-Loop tests. The simulation model provides the still missing ones and allows to test dynamic coupling effects even when not all components of the real system are available. Additionally all requirements in the central repository, which only apply to the real prototype are tested.

The emphasis on introducing an executable specification – e.g. as a simulation model – at the very beginning of the development process is important to connect the abstract requirements to the operational context. In the words of [17] the recommendations are to “raise [the] awareness of downstream complexity” and to “involve operations engineers early and often”. The discussion necessary to bring the initial set of abstract requirements to a set of executable specifications will automatically involve subject matter experts from all fields concerned and yield numerous reviews of requirements, design decisions and the architectural choices taken.

During the whole process all requirements are stored in a central repository. Initially, the repository is populated either manually or by importing them from external resources. As it is the case in any repository, additional meta-data is available to support the development and maintenance process, e.g. by providing information on authors, priorities, costs or authorization.

IV. Coupling the requirements to the system specification

Where the prior sections focused on the overall SDD process, this section explains the coupling between the requirements repository and the various systems supposedly implementing these requirements: The virtual and real prototypes, the simulation model and the real system under consideration. At the core of the SDD process is the availability of an executable system at any time during the whole development life-cycle. Together with the links between the executable specification and the requirements at any level-of-detail the validity and correctness of the executable specification is constantly assured by the attached requirements (see section IV-A). To that extent the current system state needs to be captured and compared with the requirements as detailed in section IV-B. In the end, validating the requirements necessitates a dedicated work-flow (see section IV-C) and the creation of dedicated test-functions (see section IV-D).

A. Requirements validation

“Treat English as Just Another Programming Language” [22] – requirements start with those people that are responsible for the system: product owners, marketing experts and domain experts whose domain is typically not the software industry.

The requirements may be gathered by specialists but start as a natural language document – fuzzy and open to multiple interpretations [23] – before they are translated into more formal notations.

Many well-documented methods exist to refine and to formalize requirements:

- Formal notations, e.g. the Z-Notation [24], Vienna Definition Language [25], Language of Temporal Ordering Specification (LOTOS) [26] and the B-Method [27].
- Cause-effect graphs [28] provide the relationship between input (causes) and expected output (effect) specified by the requirement [29]. Generators are able to derive test vectors from this model that are fed into the requirements model and into the system under test to compare the results. However, as the number of requirements grows, the size of the cause effect graphs becomes hard to handle.
- Computation tree logic (CTL) or linear temporal logic (LTL) are yet other ways to formalize requirements [30].

Naturally these methods rely on the manual task of translating the natural language into the formal notation chosen. A rigorous approach is rarely taken since the cost is typically only justified for critical systems i.e. ones in which potential financial or human loss would be catastrophic [18]. In addition, these methods are difficult to apply in the very early design stages when the requirements are at a very high level and still a subject to change. Therefore a different approach is necessary.

To address the size of the solution space in the early stages, SDD introduces configurable scenarios, called missions. Each mission is driving the virtual prototypes – the specification becomes executable via a set of parameters or even architectural choices. The dynamic system behavior – at the yet considerable level-of-abstraction – is obtained for a particular scenario by executing the mission as a simulation run. The set of all missions describes the solution space that is considered to adhere to the known requirements. At this point, detailed requirements for the initial subsystems and components are not yet settled or completely absent. Yet the simulations will already give boundaries for the subsystem behavior and the discussions with the subject matter experts will quickly refine the requirements – already within the context of the integrated system.

To validate the requirements, the authors have chosen a method similar to test oracles [31]. A test oracle is a predicate that determines whether a given test activity sequence is an acceptable behavior of the system under test [32]. In this context, a testing activity can be seen as a sequence of stimuli and response observations. To that extent the virtual prototypes are enhanced by dedicated test-function blocks representing test oracles: These functions are used to check if the model state matches the expected as defined by the requirements. Links between the requirements, the virtual prototype and the test-functions provide the traceability in the SDD approach.

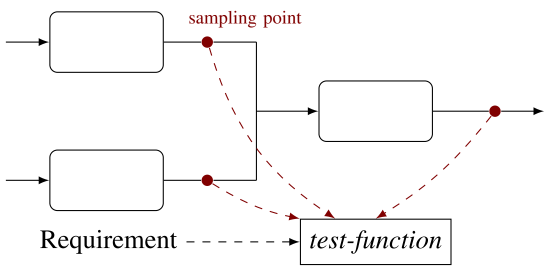

Figure 3. Sampling points are embedded into the simulation model to capture event streams

B. Capturing the system state

One advantage of the virtual prototype running as a simulation is the access to the whole system state – which is not feasible in many distributed systems in the real world. To capture the system state in a simulation run we embed sampling points in the model (see figure 3).

Depending on the placement of the sampling points, different information is recored: If placed on a connection between two or more components, the event flow of the discrete event simulation model is recorded along the chosen connection. Additionally, each component can be extended so that its local internal state can be sampled. In both cases, every sampling point produces a stream of data as the simulation run produces and processes events over the execution time. Data extraction is read-only, i.e. the semantic of the virtual prototype remains unchanged albeit at a minor performance hit.

The sampling data stream adds the event time and the component to the data sampled at the sampling point. Of course, the interpretation of a given sample value is model and domain specific. The tuples sampled in a simulation run form the stimuli or the responses of the test-functions used to validate the requirements. Any combination is conceivable: A test-function may use a single tuple, i.e. one value at a given simulation time, several tuples at the same or even at different times – opening the possibility to correlate information along the flow of a business process over time. The data processing can itself be performed either synchronously or asynchronously to the simulation run.

C. Validation work-flow

The SDD approach explores the solution space by maintaining different missions (see section IV-A) corresponding to different virtual prototypes. To validate the requirements all missions are executed as simulation runs, each run produces its sampling data stream to feed the test-functions embedded in the virtual prototype. For each mission the result is a simple boolean “pass” or “not-passed”, a detailed look at an individual simulation run of a given mission will in turn show the boolean result for each embedded test-function (see figure 3). The traceability of all requirements results from the links between various levels-of-abstraction and to the virtual prototype and its test-functions. As a result of executing all missions, it is possible to determine those missions that implement the

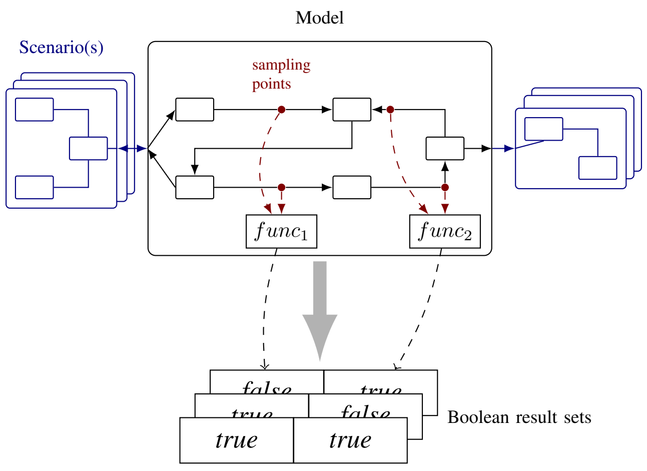

Figure 4. Requirement validation: Different scenarios are used to run the model. The test-functions create an output value for each simulation run, which alltogether build the boolean result sets.

Table I
INTERPRETING THE RESULTS OF THE REQUIREMENTS VALIDATION WORK-FLOW

Test-Function Results			Overall Rating
All	test-functions in every mission are evaluated to	*true*	fulfilled
Some		*true*	partially fulfilled
All		*false*	violated

requirements successfully. If some requirements in the central repository are not validated – either explicitly as "not-passed" or implicitly when no test-function is available, the traceability allows the automatic high-lighting of these requirements in the repository.

The validation work-flow starts with executing – probably in parallel – all available missions of the virtual prototypes and the subsequent collection of the test results. During the simulation runs, the embedded test-functions are constantly triggered by the events passing through the simulation model resulting in a sampling data stream that is captured and evaluated to return the boolean result set of the used test-functions (see figure 4). Eventually when all simulation runs are finished, all result sets are aggregated allowing to identify valid virtual prototypes and potential problems and their origin by following the traceable links from the test results via the test-functions and the simulation model components back to the individual requirements.

Table I summarizes the possible outcomes at a global level: All test-functions of all missions could return a positive result, some might return a negative result or all of them fail. This global overview concerns the overall solution space since the various missions are equivalent to different possible solutions – it is expected that as the knowledge about the system under consideration progresses more and more potential solutions will fail the added requirements and constraints. The design stage therefore aims at retaining at least one valid virtual prototype where all requirements are successfully verified by test-functions. This approach naturally leads to a iterative spiral-model [33] where the negative test-results will start the search for an incrementally improved solution and the refinement or alteration of the requirements responsible for the negative test result.

The validations work-flow can of course be automated in large parts: New simulation runs are automatically created and executed, the results of the test-functions are mirrored to the requirements repository. The search for better virtual prototypes or adjusted requirements remains a human task involving the domain experts as well as the technical experts. Once a change to the existing requirements is identified and submitted, the traceability automatically yields all virtual prototypes and test-functions impacted by the change.

D. Implementing a test-function

The SDD approach takes the ideas of test-driven development (TDD) to the very early design stages: The requirements undergo testing prior and in parallel to the system implementation using test oracles as test-functions [34]. As in the TDD case, testing is not the aim of the SDD rather the "driven [...] focuses on how TDD leads analysis, design, and programming decisions" [35]. Of course, technically test-functions need to be implemented to verify the requirements through the correct behavior of the virtual prototype: An obvious implementation is to compare the input events (stimuli) of a particular component in the simulation model with output created (responses) – basically a simple unit test of a component. However, often a single deterministic outcome is not sufficient to determine the success of a test-function. Rather the test-function is used to explore the boundaries of the specification, the statistical distribution of events or correlates information from different components of the virtual prototype at the same time or over time periods.

To that extent, test-functions are again source code potentially with (read-only) access to the whole simulation run and a private data store to retain and correlate data over the runtime. As the complexity of the test-function increases, the risk of program errors increases as well. To mitigate this risk as set of predefined, configurable and proven test-functions is provided ready-for-use. The set may consist of functions, to test whether a value is bound to a specific interval as well as functions to express boolean conditions in the form *if* ... *then* ... *else*. With this staring point mathematical relationships are straightforward to implement.

Test-functions are implemented like the other model components using the same levels of abstractions such as nested sub-components, if necessary. The only difference is, that they cannot influence the model semantics or impose any side effects.

V. APPLICATION: SIMULATING THE GERMAN AUTOMATIC TOLL SYSTEM

We have applied the SDD approach – in parts – to the ongoing development of the German automatic toll system, a large-scale autonomous toll system [36] operated by Toll Collect GmbH. The toll system collects the tolls for heavy-goods vehicles (HGVs) driving on federal motorways – at present it is the largest system of its kind in operation, collecting more than 4.6 bn € annually predominantly automatically using the more than 1 000 000 on-board-units (OBUs) deployed at present.

A. Challenges of the development process

As a typical system-of-systems the toll system consists of a multitude of sub-systems for the various domain-specific tasks. Wherever possible, sub-systems are designed around existing commercial off-the-shelf applications and very few are custom-developed (the most notable one is the hard- and software of the OBU). Most often the development and operations of the sub-systems is outsourced to technology partners – at least the system specification and later on the system integration remain as a core competency [37].

The common software or system development practices suffice to address most aspects of the liability-critical system: Following a V-Model approach, requirements are documented prior to the system design and implementation, all of which create test cases for the subsequent verification in different stages. However, the more than 1 000 000 OBUs deployed in HGVs pose a particular challenge. They form a 'distributed system", i.e. "one in which the failure of a computer you didn't even know existed can render your own computer unusable" [38]. In addition the OBU behavior depends on the user interaction which is in large parts not known due to technical restrictions and data privacy protection.

To address these challenges posed by the OBU-fleet, additional test stages are added using tens of OBUs in a lab environment, hundreds and up to a few thousand OBUs in dedicated test fleets. Yet these tests are still at a scale below 1:100 and occur only at the end of the development once the software change has been implemented and tested in unit and component tests. Scaling based on past experience from the real-world system is of course possible when the changes are minor and the operational context remains unchanged.

Major changes to the software of the toll system cannot rely on past experience: Even recourse to expert advice is known to be problematic – experts tend to be over-confident [39], a well-known cognitive bias that needs to be mitigated by the system design process. Besides, given time, the cost-of-operations dominates over the initial system development costs. I.e. the validation of the requirements – *did we build the right system?* – can only be answered in the context of the daily operations of the whole system at a scale of 1:1.

B. Adding simulations to the development process

To overcome the challenges mentioned above we developed a simulation model of the automatic toll system that incorporates the most important processes – collecting tolls and providing updates to the fleet – at a scale of 1:1 (figure 5, upper part) and a model for the temporal behavior of the user interaction (figure 5, lower part, for details see [40], [41] and references therein). Having this executable specification of the

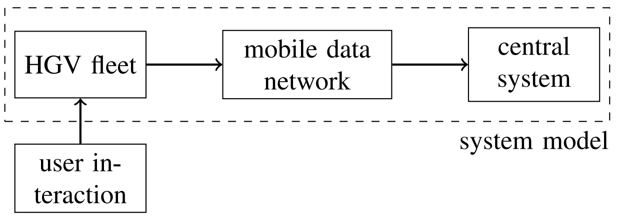

Figure 5. The simulation model includes a model of the technical system (above, dashed) and a model of the user interaction (below).

automatic toll system we derive missions corresponding to the system in operations either at present or in the near future. Simulation runs based on these missions predict the dynamic operational behavior over weeks or months, e.g. the propagation of software updates across the fleet. Where possible, the predictions of the simulation runs are compared with data observed in the real-world system and the parameterization is calibrated accordingly.

This realistic, microscopic simulation model of the real-world toll system is the starting point to change the software development process to *simulation driven*: As the software development starts, the virtual prototype of the existing toll system is forked to reflect the proposed changes. In that way the proposed system is accompanied from the very beginning with a simulation model: The very early design stages start with an executable specification that transports much of the existing operational context to the newly drafted requirements. Design decisions are from the start *driven* by the simulation results where the simulation takes into account the system operations at a 1:1 scale. Consequently the initial draft of the new requirements – typically a document using natural language descriptions – quickly becomes more precise and the discussions are anchored in the real-world operational context.

VI. Conclusion

A realistic simulation model of a software-intensive system-of-systems is the natural extension of the test-driven development approach: The development process is at any time driven by the results *as observed in the real-world operational context*. The core of this idea is to create an executable specification of the known requirements in every development phase and to trace changed requirements from the beginning with a focus on the real-world effects. In this article we have outlined our approach – Simulation Driven Design – and briefly mentioned the case of the automatic German toll system. There the effects of proposed changes are from the beginning measured against the (simulated) effects in the integrated system at a scale of 1:1. The effect of SDD is twofold: Simulation runs predict the effects of a proposed change and creating the virtual prototype drives the development process with the focus on the systems' operational context.

Acknowledgment

One of the authors was funded by the German Federal Ministry of Education and Research (BMBF), grant "SimDesign, 01|S14026B".

[1] B. Boehm, “A view of 20th and 21st century software engineering”, in Proceedings of the 28th international conference on Software engineering, ACM, 2006, pp. 12–29. DOI: 10.1145/1134285.1134288.

[2] R. L. Glass, “The standish report: Does it really describe a software crisis?”, Communications of the ACM, vol. 49, no. 8, pp. 15–16, 2006. DOI : 10.1145/1145287.1145301.

[3] A. Sikora and R. Drechsler, Software-Engineering und Hardware-Design – eine systematische Einführung. Fachbuchverlag Leipzig, 2002, ISBN: 978-3446218611.

[4] W. Ecker, W. Müller, and R. Dömer, Hardware-dependent Software. Netherlands: Springer, 2009, ISBN: 978-1-4020-9435-4. DOI: 10.1007/978-1-4020-9436-1.

[5] G. Piccoli and B. Ives, “IT-dependent strategic initiatives and sustained competitive advantage: A review and synthesis of the literature”, MIS Quarterly, vol. 29, no. 4, pp. 747–776, 2005, ISSN: 0276-7783.

[6] T. Baumann, “Simulation-driven design of distributed systems”, SAE International, SAE Technical Paper, 2011, pp. 1–7. DOI: 10.4271/2011-01-0458.

[7] D. Gelperin and B. Hetzel, “The growth of software testing.”, Communications of the ACM, vol. 31, no. 6, pp. 687–695, 1988, ISSN: 00010782.

[8] K. Beck, Test-driven development: by example. Addison-Wesley Professional, 2003.

[9] T. Baumann, Automatisierung der frühen Entwurfsphasen verteilter Systeme. Saarbrücken, Germany: Südwestdeutscher Verlag für Hochschulschriften, 2009, ISBN : 978-3-8381-1266-4.

[10] I. Sommerville, Software Engineering, 9th edition. Boston: Addison-Wesley Longman, 2010, ISBN: 978-0137053469.

[11] J. Beatty and K. Wiegers, Software Requirements, 3rd. Redmond: Microsoft Press, 2013, ISBN: 978-0735679665.

[12] M. Glinz, “On non-functional requirements”, in 15th IEEE International Requirements Engineering Conference, (Delhi), Oct. 2007, pp. 21–26, ISBN: 978-0-7695-2935-6. DOI: 10.1109/RE.2007.45.

[13] I. Jacobson, G. Booch, and J. Rumbaugh, The Unified Software Development Process. Reading, MA: Addison Wesley, 1999, ISBN: 978-0201571691.

[14] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing”, IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11–33, 2004, ISSN: 1545-5971. DOI : 10.1109/TDSC.2004.2.

[15] B. Pfitzinger, T. Baumann, D. Macos, and T. Jestädt, “Using parameter optimization to calibrate a model of user interaction”, in Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, M. P. M. Ganzha L. Maciaszek, Ed., ser. Annals of Computer Science and Information Systems, vol. 2, IEEE, Sep. 2014, pp. 1111–1116, ISBN: 978-83-60810-58-3. DOI: 10.15439/2014F123.

[16] H. Balzert, Lehrbuch der Softwaretechnik: Basiskonzepte und Require ments Engineering. Springer, 2008, ISBN: 978-3-8274-2247-7. DOI: 10.1007/978-3-8274-2247-7.

[17] D. L. Dvorak, “NASA study on flight software complexity”, 6th AIAA InfotechAerospace Conference, Seattle, Washington, Apr. 9, 2009. DOI: 10.2514/6.2009-1882.

[18] E. Hull, K. Jackson, and J. Dick, Requirements Engineering, 3rd ed. London: Springer, 2011. DOI: 10.1007/978-1-84996-405-0.

[19] O. Gotel, J. Cleland-Huang, J. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, J. Maletic, and P. Mäder, “Traceability fundamentals”, in Software and Systems Traceability, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds., London: Springer, 2012, pp. 3–22, ISBN: 978-1-4471-2238-8. DOI : 10.1007/978-1-4471-2239-5 1.

[20] Bundesstelle für Informationstechnik, Zusammenarbeit mit IT-Organisation und Betrieb, [accessed 09-Jul-2014], V-Modell XT Bund, Bundesministerium des Innern, Sep. 20, 2013. [Online]. Available: http://gsb.download.bva.bund.de/BIT/V-Modell_XT_Bund/V-Modell%20XT%20Bund %20HTML/f4a3125029a3017.html.

[21] F. J. Brooks, “No silver bullet: Essence and accidents of software engineering”, IEEE Software, vol. 20, no. 4, pp. 10–19, Apr. 1987, ISSN : 0018-9162. DOI: 10.1109/MC.1987.1663532.

[22] D. Thomas and A. Hunt, The Pragmatic Programmer: From journeyman to master. Boston, MA: Addison-Wesley Professional, 1999, ISBN : 978-0201616224.

[23] H. Sneed, “Testing against natural language requirements”, in QSIC ’07. Seventh International Conference on Quality Software, 2007, Oct. 2007, pp. 380–387. DOI : 10.1109/QSIC.2007.4385524.

[24] J. M. Spivey, The Z Notation: A Reference Manual. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989, ISBN : 0-13-983768-X.

[25] C. B. Jones, Systematic Software Development Using VDM, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990, ISBN: 0-13-880733-7.

[26] ISO, “ISO 8807:1989, information processing systems – open systems interconnection – LOTOS: A formal description technique based on the temporal ordering of observational behaviour”, International Organization for Standardization, Geneva, Switzerland, ISO 8807, Sep. 1989.

[27] J.-R. Abrial, The B-book: Assigning Programs to Meanings. New York, NY, USA: Cambridge University Press, 1996, ISBN : 0-521-49619-5.

[28] G. J. Myers and C. Sandler, The Art of Software Testing. Hoboken, NJ: John Wiley & Sons, 2004, ISBN: 0471469122.

[29] C.-C. Lee and J. Friedman, “Requirements modeling and automated requirements-based test generation”, SAE Int. J. Aerosp., vol. 6, pp. 607–615, Sep. 2013. DOI: 10.4271/2013-01-2237.

[30] E. M. Clarke Jr., O. Grumberg, and D. A. Peled, Model Checking. Cambridge, MA, USA: MIT Press, 1999, ISBN : 0-262-03270-8.

[31] W. Howden, “Functional program testing”, IEEE Transactions on Software Engineering, vol. SE-6, no. 2, pp. 162–169, Mar. 1980, ISSN: 0098-5589. DOI : 10.1109/TSE.1980.230467.

[32] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey”, IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 507–525, May 2015, ISSN: 0098-5589. DOI : 10.1109/TSE.2014.2372785.

[33] C. Larman and V. R. Basili, “Iterative and incremental development: A brief history”, Computer, vol. 36, no. 6, pp. 47–56, Jun. 11, 2003, ISSN: 0018-9162. DOI : 10.1109/MC.2003.1204375.

[34] S. G. Alawneh and D. K. Peters, “Using test oracles and formal specifications with test-driven development.”, International Journal of Software Engineering & Knowledge Engineering, vol. 23, no. 3, pp. 361–385, 2013, ISSN: 02181940. DOI: 10.1142/S0218194013500113.

[35] D. Janzen and H. Saiedian, “Test-Driven Development: Concepts, taxonomy, and future direction”, Computer, vol. 38, no. 9, pp. 43–50, Sep. 2005, ISSN: 0018-9162. DOI : 10.1109/MC.2005.314.

[36] CEN, ISO/TS 17575-1:2010 Electronic fee collection - Application interface definition for autonomous systems - part 1: Charging. Geneva, Switzerland: CEN, 2010.

[37] M. Hobday, A. Davies, and A. Prencipe, “Systems integration: A core capability of the modern corporation”, Industrial and corporate change, vol. 14, no. 6, pp. 1109–1143, Dec. 2005. DOI: 10.1093/icc/dth080.

[38] L. Lamport, Distribution, e-mail message, [accessed 16-Nov-2014], May 1987. [Online]. Available: http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt.

[39] D. Griffin and A. Tversky, “The weighing of evidence and the determinants of confidence”, Cognitive Psychology, vol. 24, no. 3, pp. 411–435, 1992. DOI : 10.1016/0010-0285(92)90013-R.

[40] B. Pfitzinger, T. Baumann, D. Macos, and T. Jestädt, “Using simulations to study the efficiency of update control protocols”, in 2014 47th Hawaii International Conference on System Sciences (HICSS), Jan. 2014, pp. 5154–5161. DOI: 10.1109/HICSS.2014.634.

[41] ——, “Modeling regional reliability of 2G, 3G, and 4G mobile data networks and its effect on the German automatic tolling system”, in 2015 48th Hawaii International Conference on System Sciences (HICSS), Jan. 2015, pp. 5439–5445. DOI: 10.1109/HICSS.2015.640.