SURROGATES-BASED PROTOTYPING

Els Du Bois
Faculty of Design Sciences
University of Antwerp
Belgium
e.dubois@tudelft.nl

Imre Horváth
Faculty of Industrial Design Engineering
Delft University of Technology
The Netherlands
i.horvath@tudelft.nl

ABSTRACT
The research is situated in the system development phase of interactive software products. In this detailed design phase, we found a need for fast testable prototyping to achieve qualitative change proposals on the system design. In this paper, we discuss a literature study on current software development and opportunities for building fast testable prototypes. Next, we discuss a new methodology called surrogates-based prototyping (SBP). The SBP methodology supports combining existing software products as surrogates to simulate all function sets. To test the methodology, we discuss an application of the methodology in a reference case. We concluded that SBP supports reasoning about the stakeholder-computer interaction on such a detailed level that all functions and usability aspects can be reconsidered for improvement before coding. SBP goes beyond the conventional concept of pure component-based design and avoids the problem of interfacing of heterogeneous components, using a platform-enabled approach.

KEYWORDS
Testable software prototyping, surrogate software, surrogates-based prototyping, component-based development, platform-based development

1. INTRODUCTION
The research described in this paper is part of a PhD research whose aim was to support and increase stakeholder involvement in the development process to achieve higher quality products and reduce late drastic changes [1]. We stated that software development is considered to be applying a reflective practice, in which the development is happening according to a phased implementation process. We investigated the three most critical phases of the development process (idea generation, concept development and detailed design), and accordingly, we split up the complete research into research cycles to investigate each. Our assumption was that each phase needs specific prototypes, generated with different and increasing functionality level, in order to be able to test and improve the to-be-developed software. This paper focuses on the third of the critical phases. In the two preceding research cycles, we investigated how stakeholder involvement can be beneficial and how it can be supported in the early phase of ideation and framework development, and in the phase of concept development, respectively [2].

The research cycle that is discussed in this paper is actually the detailed design (or system design) phase. Here, the concept is further developed from all aspects to make the innovation ready for manufacturing and commercialization [3]. Having the aim of achieving higher quality with complex interactive software products, we needed to pay attention to the importance of an all-aspect system development and testing before code production. The system design should integrate all cross-disciplinary conceptual and preliminary data into a complete digital product definition. Along the way to this destination, software design engineers must continually manage change and design complexity. They need to assess risks and balance trade-offs while rapidly delivering high quality designs that work reliably and offer customers value. Hence, software functionality and quality i.e. usability, understandability, learnability, operability, attractiveness and compliance of the software system, should be tested [4].
Prototypes of the detailed designs are needed to facilitate the achievement of the needed quality. Low- and medium-fidelity abstract prototypes were used in the earlier phases and these should be transferred into higher-fidelity testable (tangible) prototypes as a last validation step before the software is fully developed. A practical requirement is that the prototype should be realized fast and at low costs, but should also be feasible, detailed, integrative, and facilitating system testing. Regarding the development of complex interactive software systems, we experienced a lack of dedicated prototyping methodologies and means that fit to the characteristics of complex systems and that are meaningful in the detailed design phase. Pre-implementation prototyping of such software products and systems is complicated and does not seem to be fully solved by the conventional methodologies. However, there is a major opportunity to use prototyping to avoid the need for functional or structural modifications when production of software programming code is started.

In this research, we developed and tested a novel methodology, called surrogates-based prototyping that supports the development of high-fidelity rapid prototypes by a compositional approach. The methodology supports combining existing software products as surrogates to simulate all function sets. These cheap and fast testable prototypes can be used for the investigation of dependability, functional integrity, technical feasibility, accuracy, etc.

The research described in this paper is framed by the Design Inclusive Research methodology [5], which goes through an exploratory phase to develop a theory, uses design activities to generate an adaptable research means, and ends with a confirmative research phase. During the exploration, we investigated the literature on current software development and opportunities for building fast testable prototypes (Section 2). Furthermore, we developed a theory for the new surrogates-based prototyping methodology (Section 3). During the constructive (design) phase of the research cycle, we developed an application example of a reference case (Section 4). The paper ends with a discussion of the activities and the results of the confirmative research phase (Section 5) and with the overall conclusions (Section 6).

2. EXPLORING THE STAGE

2.1. Overview of the trends of software development

In order to properly evaluate the current situation, we needed to increase not only our awareness of the existing trends in the software industry, but also our technical knowledge concerning the current means and methods. During the examination of the four most dominant software development paradigms (function-oriented development, object-oriented, component-oriented, and service oriented development) [6], we observed the trends of increasing evolvability and the increasing complexity of software products [7-9]. To respond to these trends, a need emerged to split the software concept into manageable parts.

The concept of separation into concerns is a matter to handle this complexity [10]. Software concerns are combinations of functionalities, which are logical, structural and from user perspective separable. Reacting upon these evolving needs of both flexibility and complexity, an important trend in software development (SD) is the shift of the conceptual resources. Several paradigms emerged in the last fifty years. The concepts proposed by these

![Figure 1 Evolving software paradigms](image-url)
programming paradigms try to support developers in the process of improving separation of concerns in a different manner [9,11-13].

More details on the evolving paradigms can be found in Figure 1. We conclude that these paradigms evolved over time to achieve more flexibility and from processing-based to utility-based SD [14]. The most recent paradigms adopt a rather pragmatic approach that believes business system development is an incremental process [15], so changes are inescapable aspects of software design and are expected to occur in every stage. The evolution of programming is tightly coupled with reuse in two important ways: (i) by reusing ever larger grained programming constructs from ones and zeroes to assembly statements, subroutines, modules, classes, frameworks, etc. [16], and (ii) the language is evolved to be closer to human language, more domain focused, and therefore easier to use [17].

For the time being, the majority in industry still uses conventional (function-oriented and object-oriented) software development [18]. However, considering the growing flexibility and complexity that must be dealt with [19], the conventional methods are not ideal [20]. The conventional software development methods are also inadequate due to their required knowledge and complexity of programming languages and the appearance of bugs, which increase the time, efforts and costs [21]. The identification of a kind of in-between stage between abstract and fully developed software, needs a method for fast and low cost prototyping, but which is feasible, detailed, integrative, and can be used for system testing. Besides, the methodology must also support the conversion of previous generated and received information (of abstract prototyping) which was given by the different stakeholders to increase the software quality.

Consequently, we focus on the non-conventional paradigms (component-based and service-oriented software development) to see how prototyping can be conducted in the present and in the future. We have to mention that we did not focus on service-oriented software development further for the reason that it was still in its infancy at the beginning of the PhD research project. Instead, we concentrated our attention on component-based development which was already more wide-spread and dynamically growing.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Characteristics of CB SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach</td>
<td>use of pre-built components</td>
</tr>
<tr>
<td>Target</td>
<td>from large, rigid systems, which are not easily modified to smaller, more portable, independent, and flexible systems</td>
</tr>
<tr>
<td>Arguments [7]</td>
<td>reuse, portability, flexibility implicit: it saves time and money, minimize bad builds, and fatal errors, minimize need for key personnel</td>
</tr>
<tr>
<td>Component [22]</td>
<td>= piece of executable software with interface commercial, open-source, or in-house developed context independent</td>
</tr>
<tr>
<td>Developers</td>
<td>two types of developers: components developers, and applications assemblers</td>
</tr>
<tr>
<td>Process [23]</td>
<td>component design and testing component’s use by application builders component search, satisfying requirements combining components in a frame interaction building</td>
</tr>
<tr>
<td>Auxiliary demands [24]</td>
<td>component interactions interaction rules</td>
</tr>
<tr>
<td>Problems in implementation</td>
<td>architectural interface mismatches interoperability incompatibilities [22]</td>
</tr>
<tr>
<td>Prototyping opportunities</td>
<td>fast development opportunities existing pre-built components</td>
</tr>
</tbody>
</table>

2.2. Component-based prototyping and testing

In component-based software development (CB SD), a compositional approach, similar to those realized in the hardware products industry [25,26], is used instead of generative building. The advantages of this reuse-based development are lower costs, faster delivery & increased quality [19]. An overview of the characteristics of CB SD can be found in Table 1.

Although the CB SD is rather similar to manufacturing goods, ensuring the quality of component-based systems is much more difficult than is the case with manufacturing goods [29], as the raw material (software components) may be of uncertain quality and their uses and behavior may be only partially known, hindering the effectiveness of possible quality assessment processes [24]. In CB SD, two parallel software development tracks could
be identified [30]; equivalently there are also two separate testing actions for validation and verification: (i) Component testing, and (ii) Application testing [28]. An overview of these testing approaches can be found in Table 2.

2.3. Concluding the exploration

Regarding the necessity for a fully testable high fidelity prototype, we interpreted the findings from the literature study and concluded the following. The most important conclusion is that reusability has been an important concern for industry as research means for software testing [17]. Next to additional advantages such as lower cost and time, there is just no significant advantage to develop software from scratch if similar utilities were readily available in other existing software packages, and can be reused in the new software product. However, there are also some technically lacking issues [31]. The main challenge of compositional SD is interaction between the different components. Often a glue-code is needed between the components to initialize intelligible communication. The principles of non-conventional prototyping help to develop testable prototypes easily, in short time and with low cost. They allow offering a real-life experience for the testers to criticize and improve the functionality and utility of the software in development. Nevertheless, CB SD is only focused on the development of detailed final software products, without considering the opportunities for in-development prototypes, characterized by its limited functionalities and the aim to involve stakeholders in the development process in an earlier phase. In industry, however, there is a need and opportunity for such a software prototyping methodology that is in line with the component-based design approach.

3. SURROGATES-BASED PROTOTYPING

In this paper, a methodology for prototyping with surrogate software is proposed. Surrogates-based prototyping (SBP) is a methodology that is based on the following assumptions:

- The methodology should use the principles of component-based software development as enabler. This reduces the efforts and time needed for original code development in the prototyping phase, while it offers the opportunity for faster functionality and utility testing. Its major objective is to provide a relatively high-fidelity realization of the intended software functionality and support testing.
- We assume that surrogate software can be used as a means of simulating or prototyping different application parts or concerns and to simulate the function sets of the intended software product. We define surrogate software as existing commercial, in-house, or open source software with certain functionalities that are similar to or match specific function sets of the intended software.
- Our hypothesis has been that functionally testable software prototypes can be created with purposeful combination of surrogate software. We assume that the advantage of using these surrogates as components is that only a minimal amount of programming is necessary and functionality and usability testing can be conducted earlier. To ensure a working system, these surrogates must communicate with each other. This might bring up a problem of interfacing.
- In order to be efficient, we state that the methodology should capitalize on simplification possibilities offered by functional and structural similarities, extent of behavioral influence, and abstraction opportunities of sub-systems and components.
- Considering the availability of software products in the market, we can assume that there are enough software surrogates available on which to base the building the prototype.

<table>
<thead>
<tr>
<th>Table 2 Component-based testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component testing</td>
</tr>
<tr>
<td>Integration testing</td>
</tr>
<tr>
<td>System/ functionality testing</td>
</tr>
<tr>
<td>Acceptance / utility testing</td>
</tr>
</tbody>
</table>
• To handle complexity in software development, literature identified concerns to split the projected software in manageable parts. This can be achieved by functional decomposition into sets of functions. These function sets should be the base to identify the different useful surrogates to prototype the software product.

• Generally speaking, software can be built in two manners: using a generative approach or by applying compositional construction. Surrogate software can be used directly (compositional) or using the functionalities of the surrogate as a programming language (generative). We use surrogates to realize rapid high fidelity prototyping, because we assume that composition work takes less time than generative software building, since these components only need interfaces to be built.

3.1. Theory and realization of surrogates-based prototyping

Based on the abovementioned assumptions, we developed a comprehensive and detailed methodology of surrogates-based prototyping. In this Section, we deepen the theory of SBP. According to our interpretation, to discuss all aspects of the theory of the proposed SBP methodology, we had to investigate: (i) the underpinning theory (in Section 3.2), which explains the principles of the procedural execution, the method selection, and the criteria and way of testing, and (ii) the implementation aspects (in Section 3.3), which include the procedural aspects and the methods and techniques.

3.2. Underpinning theory

The underpinning theory contains those ideas on which the theory of the SBP is built. To summarize, the main argumentation for using the surrogates-based prototyping methodology is that it enables to develop a high fidelity prototype for functional testing in a minimal time. To achieve this, the theory must answer following questions:

What time is needed?

Focusing on the time-aspect, we must notice that two kinds of time can be identified: (i) time to find the surrogates, and (ii) time to build the prototype. Nonetheless, they are not necessary inversely related; moreover they can even be complementary. This leads to the fact that a high finding time can be together with a high building time, if lots of interfaces are needed. This time constraint is determined by two parameters. The optimal number of surrogates (S) and functions (F) must be found in order to be able to minimize the time: Opt T (F, S). Consequently, the minimal time requests an optimization problem of both the surrogates and the functions of the in-development software. Another consideration regarding the time is that the total time needed for the development and execution of the SBP must be minimal, because SBP promises to reduce time using a compositional approach compared to other generative prototyping approaches that can be used for functional and utility testing.

What resolution is needed?

The optimal number of surrogates is as low as possible. In order to be efficient, SBP capitalizes on simplification possibilities offered by functional and structural similarities, the extent of behavioral influence, and the abstraction opportunities of sub-systems and components.

Extreme topologies are varying between low number of surrogates, which results in a rough topology, and a high number of surrogates resulting in a fine topology. In Table 3, an overview is given of the advantages and disadvantages of using a rough and fine topology. A rough topology represents a compositional development, while fine topology represents a generative approach. We assumed that a composite approach is less time intensive than a generative, and as the main goal is to minimize the time, an optimal topology must be found.

Table 3 Comparing rough and fine topology

<table>
<thead>
<tr>
<th>Rough topology</th>
<th>Fine topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fewer components (+)</td>
<td>More time must be invested to find surrogates for all sets (-)</td>
</tr>
<tr>
<td>Fewer interaction (+)</td>
<td>Larger number of interfaces (-)</td>
</tr>
<tr>
<td>Hard to find components (-)</td>
<td>Articulated coverage (+)</td>
</tr>
<tr>
<td>More extra code generation might be needed (-)</td>
<td>Lower risk of no coverage (+)</td>
</tr>
</tbody>
</table>

What is the basis of surrogates-based prototyping?

Considering the quantity of available surrogates, we identified two approaches to build the prototype: pure component-based design or platform-enabled component-based design. We define a platform as a software on which other software modules are build
and which they use to operate. Considering the platform-enabled design, a platform framework, must be chosen that will serve as basis for the surrogating components, plugins, extensions or modules. The novelty of using a platform-enabled approach compared to pure component-based is that it enables the reduction, or even elimination, of the most important weakness of component-based design, namely the interactions between the different components. Hence, the platform can serve as an underlying surrogate that provides the interactions among the components.

What type of surrogates should be used?

In addition to the chosen basis, two main groups of surrogate software can be identified based on their deployability and affordances: (i) mono-functional software, (ii) multi-functional software, such as Matlab and Visual Studio, which can accomplish multiple functionalities, and software packages or suites such as Microsoft office or Adobe CS, which have a collection of various software programs.

How to deal with the multilevel prototyping?

The feasibility of component-based design depends on two key conditions: composability and compositionality [32]. Composability expresses that component properties are not changing as a result of their interactions with other components within the system. It is a measure of the degree to which components can be assembled in various combinations to satisfy specific user requirements. Compositionality determines if synergic system-level properties can be established by local properties of components. A SBP is compositional if its emergent behavior may be derived from the behavior of its constituent components. Lack of compositionality results in systems that do not behave well outside a small operational envelope.

3.3. Implementation aspects

Procedural aspects

The process of how, according to our research activities, surrogates-based prototyping can be conducted, is proposed in this Section. We identified three main phases: (i) identification and selection of the surrogates, (ii) construction or design of the prototype, and finally (iii) the prototype is used for functionality and utility testing. In Figure 2, a schematic overview is given of the identified steps that are to be completed in each and every phase of SBP. In this Figure also the used tools and methods were mentioned. More information on the specific tools and methods that need to be used in the process can be found in the next Section.

Methods and techniques

As shown in Figure 2, which serves as a transition figure of the procedure to the methods, it can be seen that the process of SBP involves the application of different methods and techniques to support the implementation. In chronological order, methods of: functional decomposition, resource selection, matching the software affordances, optimized mapping, checking the function compliance, matching the interfaces, functionality testing, utility testing, and correspondence validation must be applied. More information on the applied methods and techniques can be found in the explanation of the

Figure 2  Overview of the process of SBP and the applied methods
4. APPLICATION IN A REFERENCE CASE

As a first step towards empirical testing in concrete application cases we demonstrate, using an example, how the methodology can be applied in a practical case. The purpose of this example is (i) to demonstrate that the methodology of SBP can be applied with benefits, and (ii) to show in an example how the methodology is applied. In this example, we will provide information about how the methodology has been used in a concrete application case, which involves the identification and selection, and the construction of the surrogates.

In order to assess the example, some information of the in-development software should be known. This example is about a design support software tool to support product designers in making decisions on smart energy saving using ubiquitous controllers [33]. The example is a database management system, which is to be implemented as an internet application. The tool is being built around a case-base and assists the designers by providing information and means for estimating functional characteristics of information appliances and ubiquitous technologies. The development of such a tool is desirable since industrial design engineers are challenged by the time available for conceptualization and should pay attention to a multitude of aspects in designing sustainable products.

In the previous development step, the functional operation of the software tool was discussed with its stakeholders using modular abstract prototyping as methodology [34]. And suggestions for functional improvements were applied. In the continuation of the development process the objective was to test the operation of the software relatively early to avoid overseen mistakes. The SBP should be built with the objective of testing the software operations. We wanted to know how it worked, e.g., how fast it was, what time was needed for computation and what amount of data was needed. Therefore, functional and performance testing have been done by using the SBP in testing contexts. In the following Subsections we will use the demonstrative example to show how to go through the process of identification, selection and construction of a surrogates-based prototype.

4.1. Identification and selection of surrogates

Step 1: Functional decomposition

Identifying the software functions was the first step needed for the selection of surrogates. As a basis for this an interaction diagram was used, containing a detailed story of how designers’ reasoning happens. The software is driven by the designers thinking, and not algorithm oriented. This means that the software is assisting the designer in his thinking process by providing the necessary information and by guiding him to identify a solution. And these operations should happen as the designers want it. Next, based on this detailed story, a functional decomposition hierarchy, as shown in Figure 3, was made. We identified a three-level hierarchical decomposition of function sets based on functionalities and on use-relations. In addition to the interaction between the designers’ thinking process and the software tool support, we also need to consider the interaction and functions for the other stakeholders, as these functions also have to be inserted into the system.
The second important stakeholder to consider is the knowledge engineer. By reasoning on their needed interactions, we also entered the functions of how the knowledge engineers can insert the information into the system.

Step 2: Resource selection

The identification of potential surrogates was a hard task due to the unlimited amount of software products available. Consequently, first we had to select the best resource. Therefore, we started to look for software surrogates both in the more-traditional component-based domain and in the novel platform-based domains. In the end, we decided to rely on the platform-based approach, because it offers the benefit of taking care of composability by establishing associations among the modules considered. That is, no interactions among the components need to be established because all functional interactions are managed by and through the platform. Another decision was made to choose a multi-functional approach, i.e., to choose just one platform for the entire prototype, which could be argued by pragmatic reasons.

Step 3: Matching of affordances

In the third step of the process of selecting surrogates we identified the different platform tools and their affordances. We limited our search directly to the most popular web application frameworks to make our search manageable.

In the end, Drupal was chosen as platform for this platform-based surrogate software prototyping. In short, Drupal is similar to a Lego kit, for which almost 20000 building blocks – in the form of contributed modules – are available to create an online web application, whether that is a news site, an online store, a social network, blog, wiki, or something else altogether [35]. Having this significant number of modules, they can state that only the really hard 5% needs to be coded from scratch. Drupal could be used in the context of SBP as a platform that enables the surrogating components, called modules, to work together and to facilitate the interaction among them. The use of this platform shortens the development time by enabling automatic interaction between the modules compared to when they have to be adjusted manually. It provides the composability of the system.

Step 4: Optimization mapping

Afterwards, all Drupal modules had to be overseen to select the appropriate module surrogates for the identified function sets. All available information on the internet, such as forums, blogs, and you tube tutorials were used to judge the affordances of different modules for the specific case. To simplify the process, possible surrogates were identified and immediately the decision criteria were used to refine the rough selection. Obviously, the functional relevance was the main criteria for surrogates, but also the adaptability and Resource Dependencies were important in the selection process.

4.2. Construction of the SBP

Step 1: Checking the functional compliance

Next, the highest functional coverage with the least amount of gaps had to be found. In Table 4, the compliance matrix is shown in which the different possible surrogate modules are matched with the software functions. We must notice that the different modules in the Drupal platform are not related to each other in a hierarchical structure as the function structure is. The Drupal modules form a holonic system in which different autonomous developed software components (modules) are related to each other as they are required by some and or require other modules. The relationship between the main modules is heterarchical, while sub-modules are hierarchically connected to its main modules.

Step 2: Interface matching between surrogates
The holonic structure of the Drupal system creates the interconnected elements that take care of the composability. Consequently, the issue of interface matching will be on all levels covered by the underlying platform. This will save much time as no adapters must be found or coded.

Step 3: Deriving the software components

Following the principle of CBD, we developed the surrogates-based prototype in a bottom-up fashion. Towards this end, an important step was to convert the functional structure of the software into meaningful data schemes that aim to show the software from a content view. As shown in Figure 4, the main item in the different level schemes is the data that is defined, processed and converted in the software product. In Figure 4, the used modules are also shown for the different purposes. To give an impression of the surrogates-based prototype, two screenshots can be seen in Figures 5 and 6. Figure 5 shows a screenshot of how a new product case can be generated at the start of a new project. Figure 6 shows a screenshot of how the prototype can be used to search energy saving solutions.

Step 4: Constructing the interfaces

As explained in Step two, all interfaces were supported by the Drupal platform. So no extra effort was needed to construct extra interfaces between different components.

4.3. Testing of the surrogates-based prototyping

Step 1: Functionality testing

By choosing the approach of the platform-based SBP, we had the opportunity of using an extra module that executed a part of the functionality test. The Simpletest module of Drupal creates a virtual web browser and uses it to walk through the software in a series of tests, in a comparable way to what we would do if we were doing it by hand [21]. However, because the software was not algorithm oriented, but designers-driven, no useful results were achieved from the automatic tests. Therefore manual tests were executed by the developers. To do so, we took the interaction diagram that was used in the beginning of this phase and went through the system to see if the designers thinking process was indeed efficiently supported by the mentioned software actions and to see if the software reacts according to the designers logic. In conclusion, a large cohesion was found.

**Figure 4** Data level schemes
between the designers’ thinking process and the software actions.

We concluded that the Drupal prototype successfully performed the intended functionalities with an acceptable performance level. However following adjustments should be made, before finalizing the complete system:

- The available information must be represented in a more visual way (using schemes, visual representations of products in charts, use scenarios, pictures, images…) because designers’ thinking is very visual. Applied in the software, we can improve the graphical aspect in the representation of products, by visualizing the use scenarios on timelines, by visualizing the energy consumption in charts, by showing the link between the functions and the saving solutions, and by graphically showing the effects of applied energy savings.

- We also discovered that there must be a better differentiation between the necessary conditions and sufficiency constraints, because designers must be informed about what aspects are necessary and what aspects give additional information that might be useful to detect constraints in the new design.

- In the step where designers will apply the selected solutions to their case, it is necessary to show the link between the selected solution and the function(s) for which the solution might be applicable. At the moment, this link is lost because all selected solutions will end up in one single list.

- Because products always have major and minor functions, it would be interesting to identify this difference in the software to remind designers about the impact of a saving solution on the usage aspects of the to-be-developed software.

- In addition to the pure algorithmic trade-off results, there should also a possibility to rank the solution kits according to their sufficiency level.

Presently, we only considered the functionality from the perspective of the main stakeholder, which is the product designer. However, additional testing should be carried out to consider the perspective of the other stakeholders as well, especially those of the knowledge engineers who have to insert their knowledge into the system. Here probably the ontology and terminology that need to be used in order to link and retrieve the data will be issues that need to be solved before going further with the realization of the software tool.

**Step 2: Usability testing**

In this application case, utility testing was not executed as we did not considered it to be crucial for
validating the goodness of the SBP methodology. Utility testing is rather time and resource consuming, and the results of this testing approach are always confounded by the used prototyping tools.

**Step 3: Correspondence validation**

We measured the conceptual distance between the intended system and the prototyped system. The distance could be considered as low since the intended functions can be executed as prescribed in the specifications. We concluded that the surrogates-based prototype is a good representation of the intended software tool and that it is a meaningful representative to test with on the functionality level. We could conclude this because all functions could be realized in the prototype and could be discussed on a detailed level with stakeholders in order to look for improvements.

5. **CONFIRMATIVE RESEARCH ACTIVITIES**

In the confirmative research experiment, the application case was used to discuss the goodness of the SBP methodology. Empirical testing in concrete application cases is known to be the most effective way of testing methodologies, although it is a reasoning-with-consequences strategy [36, 37]. We applied this strategy, because in the current situation, no other theoretical or non-experimental strategies could be considered for confirmative validation testing. The SBP methodology has been applied and tested in the research project aiming at developing a prototype for functional testing of the reference case. SBP was used to improve (optimize) the functionality of the software before implementation.

To confirm the research, we interpreted the outcome, process and methods used from the SBP methodology. Criteria for goodness are used to support this reasoning with consequences process. In general, we were pleased with the efficiency and effect of the software prototype development and testing. The improvements received for the software tool could not be gathered without prototyping these functionalities and the SBP seemed to be a good method to develop a testable tangible prototype in a short time-range.

Regarding the validation of the SBP for the application case, we can conclude the following:

- The case is part of the intended application domain of the SBP methodology, as the case concerned a complex software tool for smart energy saving.
- The use of the SBP methodology helped to increase the logic consistency in the process.
- It also shortened the prototype development time and costs as the prototype development could be achieved by the existing development team.
- The level of testability of the SBP was very realistic and consequently the developers were satisfied.
- We could also conclude that the smart energy saving software was a representative full-covering case that could be used to explore the opportunities of SBP.

However, we also considered following limitations:

- We only developed one application case. This implies that we can only generalize the conclusions by reasoning with the consequences.
- We could not compare the development time and quality of the developed case with another case that is tested by using a coded prototype.
- Because in the reference case, all functions were simulated through existing Drupal modules, we do not have any experience with the use of a platform to support the interface with external surrogate components or generative build parts. Neither do we had to manually generate the programming code of specific parts of the prototype that could not be found.
- We experienced that the experience of the development team has a large influence on the quality of the prototype. We further considered the impact of experience in [38]

6. **CONCLUSIONS AND FUTURE RESEARCH**

Based on the findings, we could formulate the following propositions concerning the developed methodology:

- The methodology of SBP allows exploring the functional discrepancies exploration of the software since it supports the testing of the operations (functional realization, robustness and computational performance) of the software in a midterm phase of software development.
- An SBP implementation is successful if a high level of composability and compositionality is achieved. To achieve the matter, affordance matching of the surrogates with the functional sets must be conducted, and the latter can be achieved
through the establishment of the interfaces between the tools.

- The methodology of SBP goes beyond the conventional concept of pure component-based design and avoids the problem of interfacing of heterogeneous components, using a platform-enabled approach.
- The combinations of surrogates should be customized to replicate the functionality required by the final software. In turn, hi-fidelity predictions can be made.
- The success of SBP was amplified by the necessary conversion of the software view. A logically organization of the component-based flow of the system is needed to detail the functions sets and to find the best surrogates. However, for the construction of the SBP a content view was needed to show the data management using different schemes of levels.
- SBP allows reasoning about the stakeholder-computer interaction on such a detailed level that all functions and usability aspects can be reconsidered for improvement.
- The importance of SBP is in the possibility that all functions can be realized and tested before the final implementation. This is also of practical significance since the exact tool functions should be detailed before the development of the first testable prototype and many practical issues may emerge that must be solved before final realization.

Future work should be performed from two aspects. Firstly, an investigation must be carried out on how to efficiently optimize the design of the component-based surrogate prototype. Examination on how optimization can be achieved in Drupal to reduce the complexity, by decreasing the number of functions, the number of modules, and or the number of interfaces must be conducted. In addition, the relationship between these aspects must be defined, as it is not just linear. Secondly, future research work may also focus on the development of a solution for overviewing the available and continuous monitoring of the evolving surrogate software possibilities.

REFERENCES


[31] Aoyama, M., (xx), "New age of software development: how component-based software engineering changes the way of software development?".


[34] Du Bois, E., and Horváth, I., (2012), "Modular abstract prototyping as an instrument to demonstrate software tool concepts for multiple stakeholders", in Proceedings of the TMCE 2012, Karlsruhe, Germany.


