Creating Formal Models from Informal Design Artefacts

Judy Bowen[a], Benjamin Weyers[b] and Bowen Liu[a]

[a]University of Waikato, New Zealand; [b]University of Trier, Germany

ABSTRACT

The use of robust software engineering processes is essential in the design and development of interactive systems. This ensures that software is both functionally correct and also usable in the required context. There are a variety of software engineering techniques that can be used to consider different aspects of a system under construction, and at different stages in the development process. There is, however, a natural tension between formal approaches (typically used in the domain of safety-critical systems to consider functional correctness) and informal design approaches, which focus on users and user requirements in a manner which is accessible by stakeholders. In this paper we present two new approaches which enable a tighter coupling of informal design requirements with formal models. We present examples of these two approaches and discuss the benefits that transformations between the informal and formal provide. We also discuss the current limitations of such work along with recommendations about how these might be addressed.

KEYWORDS

Interactive system design; model transformations; Formal methods; informal design; Petri nets; BPMN; behaviour-driven design; Gherkin; Cucumber

1. Introduction

Interactive computer systems and software support real-time interactions between humans and computers. Such systems require human interactions for input and control and provide information to the human users in return. These interactions are facilitated by user interfaces which may consist of a wide range of input/output mechanisms (WIMP-based[1], touchscreen, speech, haptic etc.) (Preece et al., 1994). Interactive systems are now ubiquitous and appear in all parts of our daily life (mobile phones, ATM machines, smart-home devices, travel ticket kiosks) and as such, failures or errors with interactive systems likewise have the ability to impact all parts of our lives (Weyers, Burkolter, Luther, & Kluge, 2012a). When we build interactive systems, therefore, we should employ suitable software engineering techniques and principles to avoid such errors (Ameur, Bowen, Campos, Palanque, & Weyers, 2021). What we mean by 'suitable' may depend on the domain of the software being developed and its context of use. For a mobile-app which is primarily for entertainment the focus may be on data security, user satisfaction and engagement. As such, the engineering methods we employ will be those that best support these properties (testing, user-centred design

CONTACT J. Bowen. Email: jbowen@waikato.ac.nz

[1]Windows, Icons, Menus, Pointers

etc). If, however, we are building software to manage a nuclear power plant, which is a safety-critical system, then we have much stronger criteria and a wider range of consideration (Weyers et al., 2012a). As such we employ techniques such as formal specification and modelling, verification and validation, model-based testing etc. so that we can investigate safety properties using model-checking and formal proofs prior to implementation (Bolton, Bass, & Siminiceanu, 2012; J. Bowen & Reeves, 2013; Campos & Harrison, 2009). Irrespective of the methods we use, there is a natural division between the front-end (user, user interface, interactivity) of the system and the back-end (functionality), and we will typically also use different methods to consider these parts.

There is, therefore, a need for using formal models in interactive system engineering (particularly in safety-critical contexts) but formal models and notations are known to be complex and mainly only applicable by experts (J. P. Bowen & Stavridou, 1993a). For safety critical systems it is more likely that formal methods and robust engineering techniques will be followed (Fayollas et al., 2013; Harrison et al., 2019) and indeed these are mandated in some domains (Masci et al., 2013). However, it is often the case that the functional behaviour of the system is the target of these formal techniques and the user interface and interactive components do not receive the same attention. Further challenges arise from the fact that many interactive systems are also safety-critical and properties of the interface and interaction must also be investigated formally (J. Bowen & Reeves, 2013). When we develop safety-critical systems, it is possible that the business stakeholder who gives requirements knows very little about software development, therefore it is important to have a clear set of unambiguous requirements that both the stakeholders and software developers can understand. On the other hand, requirements can be very domain-specific which leads to a different set of problems. For example, the system developer might not understand how a medical infusion pump works or the context of the medical domain in which it is used. Thus the possibility of having miscommunication between business stakeholders and professional software developers is too high to ignore. In addition to the communication problem, consistency needs to be maintained throughout the whole implementation process, the developer needs to carefully follow the original requirements as the system grows bigger and more complicated.

User-centred design (UCD) methods aim to ensure that interactive systems are usable by their target users by involving users throughout the design process (Norman & Draper, 1986). There are many different methods that come under the umbrella of UCD (these are defined under ISO standard 9241-210:2019 [2]) and each involves a different amount of user involvement at different points in time within the process. The most inclusive is co-design (Bødker, Ehn, Sjögren, & Sundblad, 2000), where users directly contribute to design throughout, but UCD can also be much more lightweight and perhaps involve stakeholders and end-users at the requirements gathering stage only. As such, UCD ensures that we integrate the user and user requirements into the development processes and enables them to give feedback early on. This, however, raises further challenges, as users are usually not experts in applying, or understanding formal modelling methods. Thus, our focus in this paper is on taking informal user-centred design processes, particularly those that relate to the gathering and expressing of user requirements, and using them as the basis for more formal models. This allows us to take advantage of the benefits and strengths of the informal processes, which are tailored to capture the needs of users in a way that is easily understood by all stake-

[2] https://www.iso.org/standard/77520.html

holders (such as qualitative methods in UCD (Lack, 2007)). We can then use these well-constructed descriptions to support the generation of formal models, which are more suited to robust software engineering processes. In this way we seek to integrate the informal approaches with the formal in order to receive the benefits that both can provide. Our work contributes to the body of knowledge in the domain of engineering interactive systems, specifically we propose two new methods that can be adopted within this context to support the development of robust and safe interactive systems. Our approaches seek to capture the richness that exists in UCD approaches and design artefacts (which can be any artefact created and used in the design process, e.g. prototypes, use-cases, scenarios etc.) and use transformation approaches to enable their incorporation into more formal approaches, such as specification and model-checking.

Therefore we started with the following high-level research question in this work:

- What methods can we use to integrate informal software design methods with more formal processes?

Our focus on specific parts of the design process then led to two more specific questions:

- How do we support end-users in providing requirements in a structured manner such that they can be integrated into a formal development process using Petri-nets?
- How can we ensure that user-centred requirements expressed in behavioural specifications are consistent with formal specifications?

To address these, this paper describes two new approaches which make use of informal descriptions and specifications by transforming them into formal notations. Figure 1 provides an overview of the processes we describe. The thin arrows indicate existing standard methods: stakeholders provide information (which may include user interviews) which enables the development of requirements; requirements are used as the basis for formal models and behavioural specifications. Thick arrows indicate our two new methods: 1. a subset of the stakeholders (end-users) are interviewed with the data transformed to BPMN models; 1. the BPMN models can be transformed to formal models (Petri nets), which can then be validated and verified as well as used for implementation of the interactive system; 2. behavioural specifications are used to derive FOL predicates. 2. FOL predicates can be combined with existing formal models and used for model-checking and refinement. The dotted arrow suggests potential future work, where the BPMN model might be used to support the development of requirements.

In the next section we give a wider background to the domain of formal modelling and interactive systems more generally, and then discuss work related to requirements, model transformations and integrated modelling approaches. We then describe the first of our new methods, which addresses the problem of gathering informal requirements from users, via interviews, and giving these some formal structure. This method enables a transformation into Petri nets which provides the basis for formal reasoning and use in implementations. This is followed by section 4 where we describe our second method which looks at a transformation from behavioural specifications into predicate logic by way of pre and post conditions. We demonstrate several ways in which these formal representations can be used in conjunction with Z specifications. We next present a discussion of the outcomes of these two methods and then outline some limitations and discuss future work.

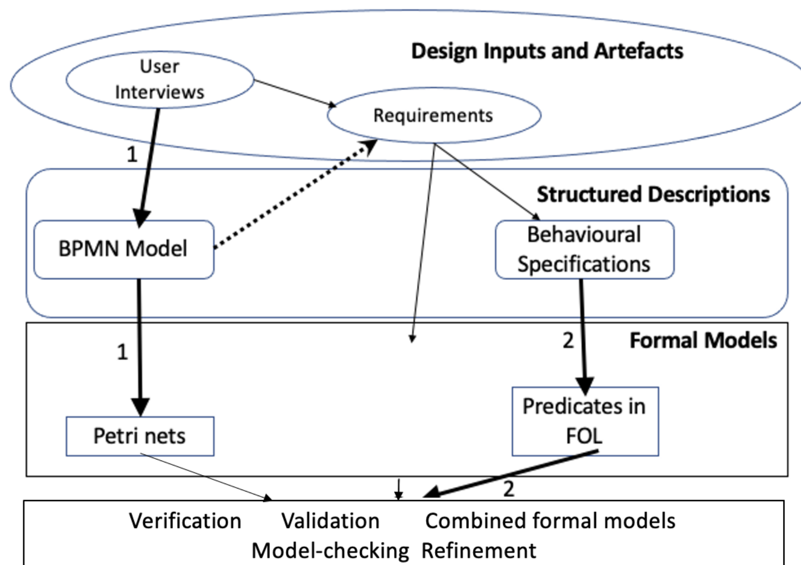

Figure 1. Process

2. Background and Related Work

2.1. Model-Driven Development

Software development is an increasingly complex task. The similarly increasing ubiquity of software has broadened the variety of end-users (who can no longer be assumed to be expert users) as well as extended the contexts of use. We do, of course, want all software that we build to be correct, robust and safe under all circumstances, but in the case of safety-critical software we also want to be able to formally reason about these properties. Formal methods are increasingly being used in such cases (Basile et al., 2018; Bonfanti, Gargantini, & Mashkoor, 2018) where the perceived overheads of the use of formal techniques is mitigated by the potential for deadly errors (Hatcliff et al., 2019). While testing can provide some reassurance that the systems being developed are error-free, it is limited by the skills and expertise of the testers and the testing approach used. It is also well-known that testing cannot be guaranteed to find all errors,

> "Program testing can be used to show the presence of bugs, but never to show their absence!" (Dijkstra, 1976)

. We therefore expect to use a combination of approaches and software engineering techniques, to ensure that we capture as many potential errors as possible. While our focus in this paper is on the use of formal methods, we assume that these are used alongside testing approaches and in some cases, such as model-driven testing, integrated with them (Utting & Legeard, 2006)).

The use of formal methods and robust software engineering processes gives us some assurance about correctness and robustness (J. P. Bowen & Stavridou, 1993b), but such rigorous design methods can exclude certain parts of the design and development process (particularly those relating to users, user interfaces and interactions). This may lead to problems further down the track. Safety-critical interactive systems are just as

likely to cause harm if users cannot understand how to use or interact correctly with them (Thimbleby, 2015) or if there are mismatches between the front-end and back-end functionalities (Turner, Bowen, & Reeves, 2020). The development of robust software must also consider that a user can successfully interact with, and use, the software and that it has been designed to meet their needs. However, such considerations are more usually (and more usefully) considered in informal user-centred design processes.

The need for robust engineering techniques and formal methods for safety-critical interactive systems is well understood (Weyers, Bowen, Dix, & Palanque, 2017). While we may wish to ensure the correctness of all interactive systems, we are pragmatic about the time and effort required to produce the formal artefacts needed to undertake tasks, such as model-checking or specification and refinement. As such, methods that enable the use of more lightweight user-centred approaches in conjunction with formal methods, for example as seen in work such as (J. Bowen & Reeves, 2008, 2009; Sousa, Campos, Alves, & Harrison, 2014), and via the types of transformations we suggest in this paper, provides benefits beyond just safety-critical systems. Formal methods do not always fit naturally with user-centred design principles and techniques, although these are equally crucial for developing usable systems that meet the users' expectations.

Providing transformations between informal and formal models enables us to have two very different views of the same system. Creating models of systems can highlight or uncover unexpected properties, as we view the same problem from different perspectives and at different levels of abstraction. So, having several models increases the benefit without increasing the workload correspondingly, as transformations can be (at least partially) automated. In addition, the benefits of having several different models of the same system which can target different parts and/or be combined at different stages in the development process have also been highlighted (J. Bowen & Reeves, 2017a).

The 'gap' that exists between informal development artefacts such as stakeholder and user-requirements, design prototypes etc. and formal software development processes is a problem that has been addressed in many different ways, particularly in the domain of interactive systems (J. Bowen & Reeves, 2008; Dix, 1991; Weyers et al., 2017). In our work here we particularly focus on the problem of the use of natural language in informal design processes (such as those seen in requirements documents or user-based processes such as interview techniques) and consider how the ambiguity and free-form of such a natural language process might be captured within a formal model. The benefit of this is that it allows us to retain the richness and user-centred focus of the original method and at the same time incorporate the information gathered in this way into formal software engineering processes that rely on syntactically correct (and constrained) modelling notations.

Finding ways to translate, or re-express, informal design artefacts as formal models is not done with the intention of replacing the informal. Rather it gives us the additional benefits that a formal model can provide, whilst at the same time retaining the original artefacts which are typically useful in other parts of the design process, communicating with stakeholders for example. The work we describe here adds to the existing body of work in the domain of engineering interactive systems, particularly where integration with formal methods is considered, for example (J. Bowen & Reeves, 2017b; Harrison, Masci, & Campos, 2018; Prates, Palanque, Weyers, Bowen, & Dix, 2017; Weyers, Burkolter, Luther, & Kluge, 2012b). While we aim at relating user-centred approaches and artefacts to formalisms seen in other work, primarily that on Petri nets (Jaidka, Reeves, & Bowen, 2019; Navarre, Palanque, Coppers, Luyten, &

Vanacken, 2019; Stückrath & Weyers, 2014) and Z (J. Bowen & Reeves, 2008; Turner et al., 2020) we specifically focus here on behavioural specifications and deriving models from users interviews. Our contribution, therefore, is to extend the breadth of the existing body of work and provide additional entry points.

2.2. Behaviour-Driven Development

In 2003, Agiledox, which some consider the predecessor of behaviour-driven languages and frameworks such as Gherkin and Cucumber (discussed later) was created by Stevenson (Stevenson, n.d.). Agiledox is a tool that automatically generates simple documentation from the method names in JUnit test cases, providing a way of tying together different parts of the development process (in this case using the testing procedures to assist with user documentation). Later, North introduced Behaviour Driven Development (BDD) (North, 2006), an agile software development process that built on the ideas of Test Driven Development (Beck, 2002). BDD combines the practices, techniques and principles of Test Driven Development, Acceptance Test Driven Development (*Acceptance Test Driven Development (ATDD)*, n.d.) and Domain-Driven Design (*What is Domain-driven design*, 2007). The key concept of BDD is that it supports "...implementing an application by describing its behaviour from the perspective of its stakeholders" (North, 2009).

Cucumber (Rose, Wynne, & Hellesoy, 2015) is a software testing framework that can create and run automated acceptance tests written in a BDD style. Cucumber uses its own language, Gherkin, which describes expected software behaviours using natural language. Business stakeholders can, therefore, easily understand the BDD specification and can work with developers to ensure the software that is delivered meets the user requirements. The connection between stakeholders, designers and developers is enhanced through the combination of easily-understood scenarios and software test generation. When developers apply BDD, they are still writing tests, but these "tests" are explained as behaviours of applications, which is more user-focused. Because the behaviour specifications are written in descriptive plain language[3], BDD utilises language that non-technical stakeholders can understand (what behaviour they expect from the application). BDD aims at strengthening the understanding of user requirements by describing the expected behaviours of the system, and so can be used to help developers better understand these requirements. This focus on behaviours from the user's perspective is, of course, common in user-centred design methods, however in BDD this is tied directly to software outcomes via the ability to generate tests from the behaviour specifications. As such, it already provides one way of generating something more formal (in this case tests) from something informal (natural-language descriptions of user requirements). The structure of the behavioural specification and the defined syntax means that it acts as an intermediary between the natural language of the requirements and the formal specifications. Our goal is to extend this to support the generation and validation of formal specifications.

Carter and Gardner introduced a development approach called BHive (Carter & Gardner, 2018), which combines BDD with the B-method (a tool-supported formal method (Abrial, 1988)). The authors outline three classes of failure that occur in the development of complex software systems: failure to deliver; catastrophic failure and failure to maintain. They propose that the combination of Agile and Formal methods they propose will help to avoid delivery risks while maintaining high standards

[3]which in our examples is English but many different languages can be used

of correctness, particularly relevant for safety-critical systems. BHive generates a B-machine based on Cucumber features by translating the specified scenarios. The B-machine captures the expected behaviour of the system under development for the developer team, thus this model can be verified and used to build new tests. Although this approach was prototyped using Python's "Behave" module (Rice, Jones, Bittner, & Engel, 2014), BHive is generalisable to any BDD tool.

While it is typical to use structured natural-language for BDD specifications, for example the Gherkin syntax[4], Lübke and Lessen described an approach where they used Business Process Model and Notation (BPMN) as the specification model (Lubke & van Lessen, 2016). These models were then fed into a generator along with content mappings and assertion mappings. The generator creates executable test suites automatically based on these BDD artefacts in a similar manner to the Cucumber tool. This is an example of how the BDD process can be made more flexible through the use of different 'front-end' models. While our end-point is a formal notation rather than tests, the use of BPMN in this manner might enable us to tie together our two different formal approaches and provides another layer of transformations between models.

Other research seeks to enhance the Cucumber scenarios through a conversion approach more similar to our own, where the target is a more formal set of models. For example Colombo et al. introduced an approach to combine related scenarios into models which can be consumed by Model-Based Testing tools (Colombo, Micallef, & Scerri, 2014). While this research was restricted to web-applications, the approach adopts specific conventions when writing Cucumber Scenarios which then allows a QuickCheck model (Claessen & Hughes, 2000) to be constructed. This use of conventions for writing and structuring the specification is something we adopt in our approach. A more general approach aimed more closely at formal modelling and validation is presented by Snook et al. (Snook et al., 2018). Starting with a (proven) consistent model based on manually written scenarios they then use Event-B/iUMLB (Event-B is a formal method for system-level modelling and analysis; iUMLB is a Graphical front-end to a collection of diagrammatic editors for Event-B) to verify it against manually written Cucumber scenarios. The scenarios generated are also used for acceptance testing of the verified model and illustrated by Cucumber for the Event-B/iUML-B tool. This enables Gherkin scenarios to be executed directly in Event-B and provides a Gherkin syntax to validate iUML-B class diagrams.

Khanal and Bowen proposed an approach that combines interactive system models and behavioural specifications to automatically generate test stubs (J. Bowen & Khanal, 2018). Given that Cucumber scenarios can be automatically converted into test stubs, the authors developed an automatic process that transforms Presentation Models created by the PIMed tool (J. Bowen & Gyde, 2015) into Cucumber scenarios by applying transformation conventions. Although the transformed language of the scenarios is more abstract, the initial meaning of the models is proven to be preserved. Furthermore, Khanal extended the PIMed tool so it can automatically generate limited Presentation Models from Cucumber Scenarios. While this transformation requires restrictions on the syntax, it does, however, demonstrate the practicality of converting between more formal models and behavioural specifications.

[4] https://cucumber.io/docs/gherkin/reference/

2.3. Integrated Design Processes

From existing research we can recognise the benefits of integrating informal design methods with more formal processes. Rather than seeking to replace either of these processes, we rather propose two new methods which enable such an integration. As we have shown in figure 1 our methods fit into existing user centred practice in order to increase options to work with different types of models/notations within different parts of the design and development process. We now describe our two approaches in detail and outline their uses.

3. From User Interviews to Petri nets

As described above, a major challenge in the creation of interactive systems is the active involvement of users (Abras, Maloney-Krichmar, Preece, et al., 2004; Fischer, Peine, & Östlund, 2020). Various approaches may by suitable: Starting with interviews (Preece et al., 1994), the development of personas (Preece et al., 1994) or methods like crowd-sourcing (Alonso & Baeza-Yates, 2011; Zuccon et al., 2013) to gather information directly or indirectly from the target audience. A special class of information represents processes in everyday working situations, such as in an office environment (Langel, Law, Wehrt, & Weyers, 2018; Zielasko et al., 2017) or in production (Borisov, Weyers, & Kluge, 2018). The first approach we describe here is based on user interviews as one standard technique in UCD, specifically in earlier stages of the design.

For a subsequent formalisation of described interaction processes and functionality in interviews e.g. to create functional elements such as interaction logic as defined in (Weyers et al., 2012a), various types of domain specific modelling approaches exists, which offer more or less formal structures and vary in their complexity and associated learning curve (Weyers et al., 2017). For instance, Petri nets (Reisig, 2012) are formally specified and can be used for system simulation (Kummer et al., 2000) and therefore to describe executable components of an interactive system. Additionally, a significant variety of tools exist for the validation of Petri net-based models due to certain requirements or characteristics (e.g., (Van der Aalst, 1997)). Nevertheless, the creation of Petri net-based models is a critical process as it puts high demands on the necessary pre-knowledge of the developer and user involved in the process, which contradicts with our assumption that users do not have expert knowledge.

Thus, direct use of Petri nets (which also offer a visual model 'front-end' language) in UCD may be not applicable. This leads to the need to find or develop alternative description formats, which on the one hand are able to 'extract' the specific information from the user as needed for the design process and on the other are still structured enough such that an engineering process can direclty profit from this information.

In a current research project, we are focusing on the development of a persuasive interactive system that supports a user to change unwanted habits in everyday working context (Langel et al., 2018; Law, Wehrt, Sonnentag, & Weyers, 2017). In this project, we use an interview-based approach supported by BPMN as a visual modelling language to keep the gap between the gathered qualitative data and the model-based description small (Law et al., 2017; Law, Wehrt, Sonnentag, & Weyers, 2022). In a second step, the gathered BPMN models can be used to generate Petri net-based models algorithmically (Law et al., 2017), such that the generated model can directly be used in the system's implementation (Langel et al., 2018) or for further

validation or verification. In order to extend the advantages provided by this algorithmic transformation of BPMN models into Petri net-based descriptions we show here how we can use BPMN as a visual modeling support in (informal) nearly unstructured user interviews. This provides a two-step transformation of informal interviews to BPMN models, and then to Petri nets (which is implemented algorithmically). We present here an overview and first insights into this endeavour based on a previously developed example (Langel et al., 2018; Law et al., 2017) and discuss future research directions based on identified limitations.

In the following subsections we first present a more detailed discussion of the overall system design of a persuasive system using Petri net-based descriptions of habits (based on the example from (Langel et al., 2018)) in Section 3.1. In Section 3.2, we will discuss the informal gathering of BPMN models (which is further supported by a structured transformation method as presented in Law et al. (2022)) followed by an overview of the algorithmic transformation and processing of BPMN models to finally generating a Petri net (using the algorithmic approach from Law et al. (2017)). Each subsection is concluded by a discussion of limitations this approach shows, which includes potential next steps which are then discussed further in Section 5.

3.1. Petri net-based models for persuasive systems

In (Langel et al., 2018), we presented a first prototype of a persuasive system supporting the change of unwanted habits in a work context. In this prototype, we used virtual reality to create a virtual office environment in which a person is able to execute basic office tasks, such as sorting, editing and printing documents. Considering given work processes and information about the user such as head direction, we triggered the user via a virtual smartphone supporting him/her to show a wanted behaviour and thus to prevent the unwanted habit.

For the use of Petri nets in the implementation of persuasive systems, we designed the system architecture as shown in Figure 2 on the left. Renew (Kummer et al., 2000) is a simulator for reference nets, a specific type of Petri net. The main feature of this specific variety of Petri nets is that transitions can be synchronously bound to the execution of methods in Java code. For more details on reference nets, we would like to refer the reader to (Kummer, 2001).

Renew is encapsulated into a server such that various clients can communicate with the Petri net as control instance in the architecture. Clients may be sensors and any kind of digital system, which is capable of communicating with the user.

As mentioned above, we used a virtual environment as shown in the right of Figure 2 to implement a first prototype of this architecture. This system enabled us not only to simulate a working environment and related working tasks but also sensors e.g. to detect where the user is looking at or what she/he is currently working on. Considering such a setup in a physical environment would render a very high complexity in sensor technology as well as data processing, which we wanted to keep for a later stage of this research endeavour.

This prototype showed that it is possible to make use of Petri nets in such a context, but also identified various challenges. On the one hand, the implementation of virtual environments, which enable realistic work processes is highly challenging and restricts the potential variety of processes to those that are controllable through interaction concepts currently available for virtual reality systems. On the other hand, data simply accessible in VR, such as viewing direction, is extremely complex to gather in real

contexts, not only because of limitations in hardware and computational power but also in terms of intrusiveness as well as data protection.

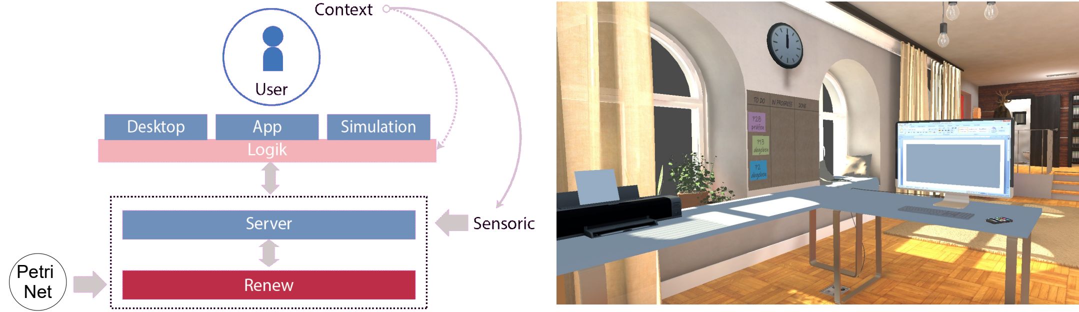

Figure 2. System design of the prototype. On the left, the architecture is shown, which includes the simulator Renew. On the right, the virtual office environment is shown.

3.2. Modeling and Transformation

For the use of Petri nets in the implementation of persuasive systems, the nets have to be created in the first place. With the requirement to generate user-specific models to consider the individuality of work habits (Sonnentag, Wehrt, Weyers, & Law, 2022) and that replicate the user's thinking (in this project the individual habit) (Oinas-Kukkonen & Harjumaa, 2008) as precisely as possible, we developed a modelling strategy which includes the user. Thus, in a first step (as presented in Figure 3) the goal is to gather a BPMN model in the context of an informal interview, where the actual modelling of the BPMN model is supported by the interviewer in terms of help and supervision to generate a syntactically correct BPMN model.

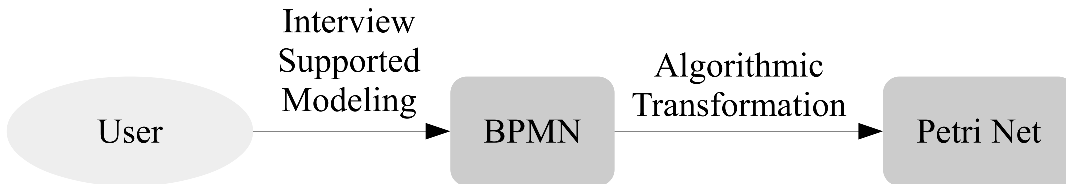

Figure 3. Generation process of formal Petri nets from informally (interview-based) gathered BPMN descriptions of work habits.

This approach has been tested in an informal pilot study with office workers as participants, which has been previously described in (Law et al., 2017). Therefore, we first pre-tested the use of BPMN as a modeling language for non-experts, here in the sense of persons without pre-knowledge of modeling in the context of software development or business processes. The major observation for the BPMN modelling was that using the full set of BPMN elements would overwhelm and disturb the users unnecessarily. Thus, we ended up with a reduced number of BPMN nodes as follows: task, event (used as interrupting and non-interrupting event), as well as the two gateway (and/xor) nodes for splitting and merging processes (Law et al., 2017). Further (structuring) elements, such as swim lanes were omitted completely. More information on the semantics of BPMN can be found here (Dijkman, Dumas, & Ouyang, 2008; White, 2004).

In the pilot study, we asked office workers to identify unwanted habits in a first step followed by a description of an alternative wanted habit. For description purpose, we offered a magnetic whiteboard on which the study participants were able to model a BPMN description of their habits with support by the interviewer. The participants were provided with magnetic markers representing the previously mentioned sub-set of BPMN nodes. Additionally, participants were able to add written notes to their models using whiteboard markers. After a short introduction to the modeling approach using BPMN including examples, the interviewers asked the participants to image a specific situation in their everyday work life in which they showed an unwanted or dysfunctional habit. After modeling it using the previously described material, they were asked to imagine a wanted alternative behaviour and also describe this in the same way. In a second part, participants were asked to repeat these habits aurally, thus in a classic interview style. The sequence of BPMN modeling and aural interviews has been randomized between participants. The interviews took between 20 and 60 minutes each.

We were able to gather models from these interviews but still made the observation that without the help from the interviewers no syntactically correct models would have been created despite the reduced number of BPMN nodes.

Another approach to gather BPMN models from interviews has been presented by Law et al. (2022). In this method, BPMN models are not created by the actual users under the supervision of a modeling expert but via a structured three-step process in which interviews are first segmented then the generated segments get classified and finally transformed into BPMN (sub-)nets in a third step.

Based on the BPMN models developed from this user-centric input, we can then apply an algorithmic transformation from BPMN to Petri nets (previously described in (Law et al., 2017)). In this approach, we defined certain Petri net "patterns" for each type of node. The algorithm generates a full net out of a given BPMN model in a two step approach. In the first step, for each node the corresponding pattern is generated and parameterized due to the information given in the BPMN model (e.g. considering a task's name). In the second step, the various sub-nets are connected via transitions and according edges forming the final Petri net. Figure 4 presents a transformation example, where at the top a BPMN example is visible and at the bottom the corresponding Petri net. By colored boxes, the corresponding sub-nets generated out of the BPMN model is indicated.

3.3. Summary

Thus, we have shown that it is possible to generate formal models (the Petri nets) from an informal (interview-based) design process. We presented a scenario for the use of a persuasive system in an office-like work environment using virtual reality. By means of this scenario, we presented an approach in a second step in which we first gather BPMN models as a semi-formal description of work-related habits. This gathering process happened in an interview in which the interviewer guided the interviewee to create an actual BPMN model describing a previously identified habit. In a second step, we showed how such a BPMN model can be transformed into Petri net-based models algorithmically.

The major benefit gained from this transformation is that the generated Petri net used later in an implementation is very close to the initial habit described by the user (see also next paragraph). Of course, the applied interview method including BPMN

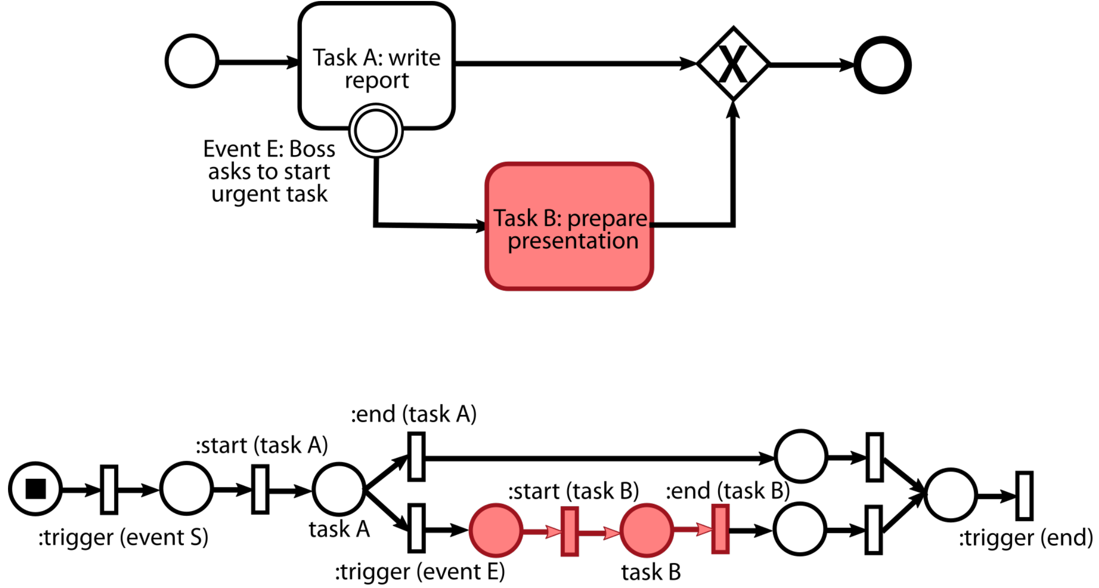

Figure 4. Transformation example of a BPMN model (top) into a Petri net (bottom). As an example, a BPMN node (task node) is highlighted together with its corresponding sub-net in the Petri net.

as a modeling language runs the risk to omit certain details of the user's mental model of the habit. However, this may be addressed by the approach described by Law et al. (2022) and shortly described above. Additionally, Petri net tools offer a broad variety of validation methods. This might render less relevant in this specific work focusing on the formal modeling of user's habits but gain relevance in case the habit related net gets integrated into a larger control structured (such as in case of Langel et al. (2018)). This may be also of high interest in case of safety critical systems which may be personalised using the presented approach presented by Weyers et al. (2012a). This uses formal rewriting approaches and showed a positive impact of personalisation of a user interface on the measured number of errors that occurred in a safety critical scenario (control of a nuclear power plant). Finally, Petri net as a formalism is quite close to the original modelling approach (BPMN) and still easy to read using its visual representation as node link diagram if needed.

In summary, the benefits gained from the use of Petri nets are

(1) direct use as part of the final implementation of a specified system or system part (here the habit as control element)
(2) validation and verification using a large number of tools focusing on Petri nets, and
(3) the opportunity to use the Petri net as (visually presented) model in classic model driven software development.

There are still some challenges to be addressed to enable this method to be used in an integrated model-driven approach, we discuss these further in section 5.

4. From Behavioural Specifications to Formal Models

While the first approach we have described above produces formal models (Petri nets) in full from the BPMN, our second approach takes a more modular perspective. We

describe how we can extract logical predicates from behavioural specifications, and use these in several ways in conjunction with formal specifications to demonstrate consistency between design artefacts (where here we consider the behavioural specification as the design artefact). This combines the flexibility and user-centred approaches embedded within behaviour-driven development with traditional formal methods.

Ensuring that the systems we build meet the expectations of our intended users requires a number of steps during the design and development process. We must gather, and fully understand, the requirements, which means including the users of the system in our design process as early as possible (as is typical in any user-centred design approach). We must also use design methods that ensure users and other stakeholders are kept in the loop. This means creating design artefacts that can be easily understood by non-technical people. Design methodologies such as the Agile approach (Beck et al., n.d.) use this as their central focus by setting short design iterations based around scenarios of use. Other design approaches may rely more heavily on prototyping and user evaluation. In this process our focus is on behaviour-driven development and formal methods.

4.1. Description of the approach

In this section we focus on the parts of the process depicted on the right-hand side of Figure 1. We assume that the business stakeholders and users provide the requirements which form the basis of the behaviour specification, written using the Gherkin syntax. As is typical in a user-centred approach this is done iteratively allowing for reflection and refining until a full set of features have been defined. The feature files are then used as the input to a parsing tool we have developed which converts the features into a set of first-order logic predicates which can be used:

(1) as the basis for the development of a formal specification,
(2) as a set of model-checking conditions which can be used with a formal model of the system
(3) as a subset (refinement) of a formal specification to ensure consistency

Each of these contributes to creating a closer relation between the user requirements and 'lightweight' user-centred specification of system behaviour with a formal specification that enables a more robust software engineering process required for safety and security.

4.2. Example Illustration

We next present a small example to illustrate the steps of the process and demonstrate each of the uses described above. The example is based around a simplified online banking system. For the purposes of demonstration we only present the parts of the system example which are needed, but these are a subset of what you would expect from a fully functional online banking system (including all of the necessary security and log in functions that are omitted here). Our simplified system allows users to do the following:

- set the type of a new account from 'new' to 'cheque' or 'savings'
- deposit money
- withdraw money

For security purposes the bank sets a limit on the maximum amounts that may be deposited or withdrawn in one transaction and also limits the number of transactions that may take place on any given day. Accounts may not be overdrawn, so withdrawals can not be made that are larger than the amount in the account.

A set of scenarios were created describing the user and business requirements. In a typical BDD approach the scenarios are developed during meetings between the developers and stakeholders. Starting with the requirements, the various stakeholders develop user stories and iteratively refine these into the specification. For our small example used here the scenarios were created by a researcher with reference to a set of business requirements based on a real-world online banking facility.

Scenarios are stored in *Feature* files, each *Feature* file is purely textual and consists of a high-level description of one expected function of the system (the feature), and scenarios which are concrete examples that describes situations of use. Scenarios illustrate the function with some user interactions and shows the results of these interactions. They use the Gherkin syntax which relies on keywords: “Given”, “When”, “Then”, “And”, or “But”, where “Given” describes the initial context, “When” describes the event/interaction and “Then” describes the expected outcome. “And” or “But” are optional and are used to replace “Given”, “When”, “Then” as required for conjunction. The rest of the text in the scenarios is natural language and comments can be included which appear preceded by a ′#′ character. Below we give some examples of scenarios for our banking example (the full behaviour specification can be found in (Liu, 2019)).

```
Feature: Depositing Money Into The Account
Banking operations of depositing money
into different kinds of accounts.
#Maximum deposit amount is 6
#Should make relevant change to the account
#balance when operations today don't exceed 5
Scenario: Successfully deposit some money into
any account
#Attempt to deposit money when operations today
#is smaller than 6
Given OperationToday is 4
And The current bank credit is 12
#amount < 6
#after operation, credit + amount = 16 < 21
When The amount deposited is 4
Then add amount to credit
And add 1 to OperationToday
#The credit should change
Then The credit should be 16
And OperationToday should be 5
```

The scenario above is just one example for the feature file, which would include other scenarios (such as depositing money into different types of accounts). Scenarios can define success (as above) and also be used to define failures, e.g., cases where deposits should be unsuccessful, as in the following:

```
Scenario: Unsuccessfully try to deposit some money
```

```
into any account
#Attempt to deposit money when credit is too large
#and exceeds maximum
Given OperationToday is 4
And The current bank credit is 12
#amount > 6
When The amount deposited is 10
#after operation, credit should not change
Then The credit should be 12
#if operation fails do not count it
And OperationToday should be 4
```

The scenario describes an unsuccessful deposit because one of the constraints (the maximum amount that can be credited in one action) is violated. This use of negative scenarios is common when writing scenarios for use with Cucumber, although not seen in original descriptions of how to write behavioural specifications.

The scenarios combined in a feature file can be interpreted by the Cucumber tool which generates test stubs from the scenarios (using regular expressions to replace constants). As part of the parsing process, Cucumber ignores any comments in the file and also strips out the keywords "Given, Then, When" etc. and logically conjoins the expressions. So, the first example above would initially be parsed by Cucumber into something that resembles:

$$\begin{aligned}&\textit{OperationToday is } 4 \wedge \textit{The current bank credit is } 12 \wedge\\&\textit{The amount deposited is } 4 \wedge \textit{add amount to credit} \wedge\\&\textit{add } 1 \textit{ to OperationToday} \wedge\\&\textit{The credit should be } 16 \wedge \textit{OperationToday should be } 5\end{aligned}$$

Effectively it treats the components of the scenario as a predicate, which is, of course, exactly what we want to do. We do have some restrictions on our process however, as in order to construct correctly formed first-order logic predicates we cannot deal with the full expressiveness of any natural language that may be used descriptively within the feature files. To deal with this we define some restrictions that the scenarios must follow in order to be used successfully within our process. Firstly we categorise the steps that a scenario contains as one of the following:

(1) Declaration
(2) Calculation
(3) Action

For each category we then define rules of how they should be constructed:
(i) *Declaration* - when a variable is declared as belonging to, being equal to, or in the state of some other value or variable, the step should be written in the form of *"A is/are/should be B"*
For example:

```
Given The current bank credit is 15
```

(ii) *Calculation* - only simple algebra calculation is allowed in the steps. The calculation can be addition, in the form of *"add A to B" or "A is added to B"*, or subtraction, in the form of *"subtract A from B" or "A is subtracted from B"*. For example:

```
And add amount to credit
```

Table 1. Converting keywords.

Keyword	Logical Symbol	Text
Given	$\exists$	E
When	$\wedge$	A
Then	$\Rightarrow$	Imp
And, But	$\wedge$	A

(iii) *Action* - to describe that an action is performed on a subject, the step should be written in the form of *"A makes/make/do/does on/to/from/by/with B"*, depending on whether A is plural or singular, and what the action is. There can only be one object and one subject and these must consist of single words (so if you have an object that is called 'account type' this becomes 'accountType'). For example:

```
When Paul does switch accountType to cheque
```

In addition, when writing steps, each should only have one statement (where they are multiple steps they can simply be split into separate statements). Following these syntax rules creates scenarios that are more tightly structured (and therefore easier to parse) but does not remove expressivity, as any scenario can be reformatted to follow these rules whilst preserving the original meaning.

4.3. Converting the Feature Files

We developed a text parser (which we refer to as the 'converter') that reads a feature file and processes it by transforming each step of a scenario into a predicate in first order logic. Each scenario, therefore, produces a group of (related) predicates. The first step is to convert the keywords (Given, When, Then, And, But) into text descriptors for logical operands and operators as shown in table 1.

We then separate declarations from calculations using the text required by our syntax rules, so steps containing "add", "is added", "subtract" or "is subtracted" are calculations, whereas those containing "is", "are" or "should" are declarations. Declarations are then constructed using the assignment operator "=" and calculations using a predicate convention of "add(A,B)" or "subtract(A,B)". We follow the same convention for actions, where we take the name of the action as the predicate and apply it to the subject and object, so "A does someAction to B" becomes "someAction(A,B)". Finally we consider start and end states and use the convention of priming names to indicate the end state. So, in terms of equality for example, "A does not change" becomes "$A = A'$" and for inequality we simply use "A' != B".

Consider the following feature file:

```
Feature: Operation on the account
Banking operations such as depositing (adding)
or withdrawing (taking) money from an account.
Or switching account types such as from new
to cheque.

Scenario: Unsuccessfully withdraw some money
from account with insufficient credit
Given Operationtoday is 4
And current bank credit is 4
When The amount withdrawn is 5
Then The credit is not changed
```

```
And credit should be 4
And Operationtoday should be 4

Scenario: Successfully switch Accounttype from
new to cheque
Given Accounttype is new
And Operationtoday is 0
When Paul does switch AccountType to cheque
Then add 1 to Operationtoday
Then Accounttype should be cheque
And Operationtoday should be 1
```

Each scenario is parsed into text with a corresponding logical representation as follows:

```
Scenario: Unsuccessfully withdraw some money from
account with insufficient credit
E Operationtoday = 4
A credit = 4
A amount = 5
Imp credit' = credit
A Operationtoday = 5
```

which equates to

$$\begin{aligned}&\mathit{Scenario} : \mathit{Unsuccessfully\ withdraw\ some\ money\ from}\\&\mathit{account\ with\ insufficient\ credit}\\&\exists\, \mathit{Operationtoday} = 4 \wedge \mathit{credit} = 4\\&\wedge\, \mathit{amount} = 5\\&\Rightarrow \mathit{credit}' = \mathit{credit} \wedge \mathit{Operationtoday} = 5\end{aligned}$$

```
Scenario: Successfully switch Accounttype from
new to cheque
E Accounttype = new
A Operationtoday = 0
A switchAccountType(Paul, cheque)
Imp Add(1,Operationtoday)
A Operationtoday' = 1
```

which is

$$\begin{aligned}&\mathit{Scenario} : \mathit{Successfully\ switch\ Accounttype\ from}\\&\mathit{new\ to\ cheque}\\&\exists\, \mathit{Accounttype} = \mathit{new} \wedge \mathit{Operationtoday} = 0\\&\wedge\, \mathit{switchAccountType}(\mathit{Paul}, \mathit{cheque})\\&\Rightarrow \mathit{Add}(1, \mathit{Operationtoday}) \wedge \mathit{Operationtoday}' = 1\end{aligned}$$

Note that because “current bank credit” consists of three separate words, this gets parsed to just “credit” (the first words are discarded), we can prevent this if we need to by using the rule that requires multi-word subjects to be combined, as in “currentBankCredit”.

We used our parser to convert the full set of feature files for the banking exam-

ple (after first editing them to match the new grammar rules described above) and successfully generated a complete set of predicates for the example. As a 'proof-of-concept' test for the parser, we also downloaded ten arbitrary Cucumber feature files from (*Cucumber and Scenario Outline*, n.d.; *Cucumber Scenario Outline*, n.d.; *Writing scenarios with Gherkin syntax*, n.d.) and followed the same two step process: first converting the text in the files to meet our grammar rules and then running the edited files through the parser. We compared the resulting predicate sets to the original specifications and were satisfied that there was no information lost or converted incorrectly during the process. While this does not constitute 'proof' that any arbitrary set of feature files can be successfully converted it gives us some confidence that our process will be generalisable, we discuss this further in Section 4.5.

4.4. Using the Predicates

As we have described, we can use the predicates in three different ways: as the basis for developing a formal specification; to support model-checking of a formal specification; as a form of refinement to ensure consistency between user requirements and a formal specification. We demonstrate each of these next.

4.4.1. Forming the Basis of a Formal Specification

Writing formal specifications is not a trivial task. Once we have learned the formal notation (or notations) that we wish to use and gained experience in abstracting the state and behaviours of a system in a suitable manner, we still face the challenge of ensuring that our specification fully describes all of the requirements correctly. One way of achieving this is by using the features from the behavioural specification as the basis for the formal specification. Having produced a set of predicates in the manner described above, we can now use these to guide the construction of a partial specification (the specification produced in this way will always be partial as it only addresses the user specified features rather than the full requirements, we discuss this further later). We do not suggest that this is a process that can be automated by further transforming the predicates into a specification, but rather they can be used as the 'building blocks' of a specification.

In our examples we use the Z specification language, but the same approach can be used with any similar declarative specification languages. A typical Z specification consists of declarations and definitions of required types, a description of the system being modelled in terms of its observable components (the observations) and descriptions of operations that can be performed on the system. Z uses a notation consisting of schemas which can be visually constructed into schema boxes which are collections of state observations and values. The system is described in a state schema as follows:

Behaviours are then described using operation schemas:

```
OperationName
ΔStateSchema
Inputs?
Outputs!
-----------
Predicates
```

The top part of the operation schema describes the observations that may be altered by the operation, the Δ symbol denoting all of the observations of the named StateSchema in their initial state as well as their after state (where the $'$ symbol is used to decorate after state names for clarity). If there are any inputs to, or outputs from the operation these are listed using the ? and ! decorations as a suffix to their name. The predicates define the pre-conditions for the operation as well as the post-conditions.

Assume we are starting with no specification at all, we will show how we can use the predicates generated from the feature files to form the basis of such a specification. Each feature file contains scenarios relating to a behaviour (for example a set of scenarios for the DepositMoney feature will include both positive/successful scenarios as well as negative/failure scenarios). The predicates derived, therefore, describe both the variables and values of interest (which will become the observations and inform their types) as well as the conditions (which will become the pre- and post-conditions). Although this process could be partially automated, that is not the intention here. The aim is to use the predicates as a guide in order to assist with the creation of the specification, which will in turn increase the confidence of those writing the specifications.

Consider again the following two scenarios from the Deposit Money feature file.

```
Feature: Deposit Money Into The Account
#Should make relevant change to the account
#balance
#When operations today don't exceed 5
#when amount to deposit doesn't exceed 6

Scenario: Successfully deposit some money into
any account
#Attempt to deposit money when operations today
#is smaller than 6
Given OperationToday is 4
And The current bank credit is 12
#amount < 6
#after operation, credit + amount = 16 < 21
When The amount is 4
Then add amount to credit
And add 1 to OperationToday
#The credit should change
Then The credit should be 16
And OperationToday should be 5

Scenario: Unsuccessfully try to deposit some money
into any account
#Attempt to deposit money when credit is too large
#and exceeds maximum
```

```
Given OperationToday is 4
And The current bank credit is 12
#amount > 6
When The amount is 10
#after operation, credit should not change
Then The credit does not change
#if operation fails do not count it
And OperationToday does not change
```

These are transformed into the following sets of predicates:

$$\begin{aligned}
&\exists\, operationToday = 4 \land credit = 12\\
&\land\ amount = 4\\
&\Rightarrow credit' = amount + credit\\
&\land\ operationToday' = 1 + operationToday\\
&\land\ credit' = 16
\end{aligned}$$

$$\begin{aligned}
&\exists\, operationToday = 4 \land credit = 12 \land amount = 10\\
&\Rightarrow credit' = credit\\
&\land\ operationToday' = operationToday
\end{aligned}$$

The first step in building the Z specification is to look at all of the variables and values from the predicates to determine their types. So for this example we have:

$$\begin{aligned}
&operationToday = 4\\
&credit = 12\\
&amount = 4\\
&credit' = amount + credit\\
&operationToday' = 1 + operationToday\\
&credit' = 16\\
&amount = 10
\end{aligned}$$

As these are all numeric, we can trivially assign them Z's inbuilt number type, which we used to declare the observations and their types as follows:

$$\begin{aligned}
&operationToday : \mathbb{N}\\
&credit : \mathbb{N}\\
&amount : \mathbb{N}\\
&operationToday' : \mathbb{N}\\
&credit' : \mathbb{N}
\end{aligned}$$

If we have values that are not numeric we will define a free type, which is declared and enumerated in Z using:

$$NEWTYPE ::= value1 \mid value2$$

So the a free type for account types would be declared as follows:

$$ACCOUNTYPE ::= New \mid Cheque \mid Deposit$$

Next we consider those variables that appear in both a primed and unprimed state (e.g. credit and operationToday) as these are observations of state that are acted upon by an operation (described by the feature file) so we can add these to a default system schema

$$
\begin{array}{|l}
\hline
\textit{BankSystem} \\
\textit{credit} : \mathbb{N} \\
\textit{operationToday} : \mathbb{N} \\
\hline
\end{array}
$$

Any remaining variables will be considered as inputs or outputs to the operations. Now we can start to define the operation schemas, the feature file contains both positive and negative scenarios (in our example above only one of each, but typically there may be more). We will use any comments from the feature file which define conditions that determine the positive and negative conditions, which here are:
#Should make relevant change to the account balance
#When operations today don't exceed 5
#when amount to deposit doesn't exceed 6
These can be used as the pre-conditions for the two operation schemas (successful deposit and unsuccessful deposit) and the rest of the predicates can be used for the post-conditions:

$$
\begin{array}{|l}
\hline
\textit{SuccessfulDeposit} \\
\Delta \textit{BankSystem} \\
\textit{amount?} : \mathbb{N} \\
\hline
\textit{operationToday} \leq 5 \\
\textit{amount?} \leq 6 \\
\textit{credit}' = \textit{credit} + \textit{amount} \\
\textit{operationToday}' = \textit{operationToday} + 1 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{UnsuccessfulDeposit} \\
\Delta \textit{BankSystem} \\
\textit{amount?} : \mathbb{N} \\
\hline
\textit{amount?} > 6 \lor \textit{operationToday} > 5 \\
\textit{credit}' = \textit{credit} \\
\textit{operationToday}` = \textit{operationToday} \\
\hline
\end{array}
$$

We then create a total operation for *DepositMoney*

$$\textit{DepositMoney} \mathrel{\widehat{=}} \textit{SuccessfulDeposit} \lor \textit{UnsuccessfulDeposit}$$

As we continue to build up operation schemas from the various feature files, we may begin to see repetitions of pre-conditions, for example

$$\textit{operationToday} <= 5$$

will appear in all of the operations. These conditions can then be lifted out of the

operations to become invariants of the system state schema, as they are always required to be true:

$$
\begin{array}{|l}
\hline
\textit{BankSystem} \\
\textit{credit} : \mathbb{N} \\
\textit{operationToday} : \mathbb{N} \\
\hline
\textit{operationToday} <= 5 \\
\hline
\end{array}
$$

Once we have completed all of the operations from the predicates of each of the feature files, we will have a partial specification which can then be used as the basis for further development. In this way, if we do not have experience in writing such specifications we are supported by the transformed behavioural specification. This also gives us more confidence that the formal specification meets the requirements.

4.4.2. Model Checking

The second approach we present for making use of the generated predicates is to support model-checking. Model-checking is an automated process whereby a given model (typically as a finite-state automata) is checked for desirable (or undesirable) properties by examination of its state-space. There are many different modelling notations and tools that can be used for this purpose, typically they differ in the types of models they can describe and/or the nature of the properties that can be checked. For example, checking for temporal properties requires a different approach from static invariant-checking. Once a notation has been chosen and a model has been successfully built and validated, we still have the challenge of coming up with suitable conditions to check. For safety-critical systems we may rely on safety properties (in the manner of (J. Bowen & Reeves, 2013)), but we show here how we can explicitly take properties defined as part of the user requirements in feature files and use the generated predicates as the model-checking conditions. Again we use the Z specification language, along with the ProB model-checker (which has a plug-in for Z). Again, any other suitable language/model-checker could equally be used. We assume that a Z specification has already been created (rather than using the predicates to guide its development as above). In order to ensure that our model meets all of the user requirements (defined in the feature files) we can use the predicates that have been created. Figure 5 shows an example specification loaded into the ProB model-checker. Based on our set of predicates (assuming we have a full set generated from all of the feature files) we can identify common patterns. For example

$$
\begin{array}{l}
\textit{operationToday} > 6 \land \\
\quad \textit{operationToday}' = \textit{operationToday}
\end{array}
$$

This suggests that a property we should check of the model is that this condition holds true in all states. Depending on the choice of model and model-checker this might be done in a variety of ways. In Z (using ProB) we can either create an invariant on the model, and then model-check to ensure it holds. Figure 6 includes an invariant schema (left hand side) which gives a condition on the system that:

$$\textit{OperationToday} \leq \textit{OperationLimit}$$

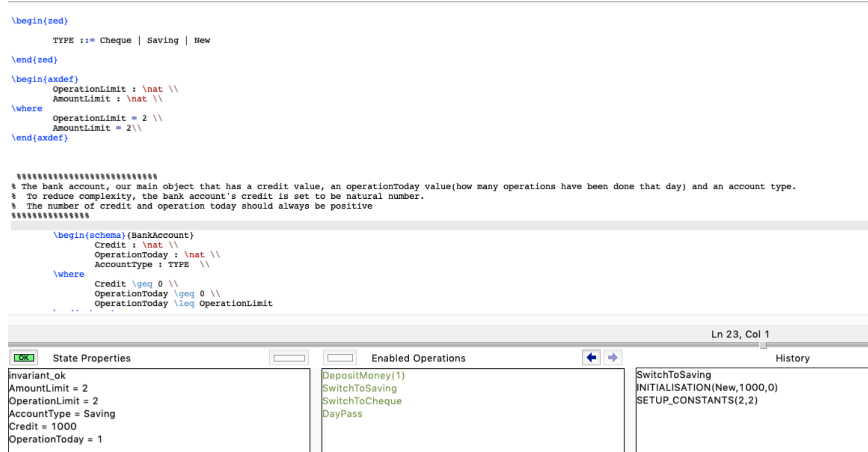

Figure 5. ProB Model Checker with Bank Example Specification Loaded

and this will be checked against every state in the model when the "Find Invariant Violations" checkbox (right hand side of Figure 6 is checked. We could also use the inbuilt temporal logic checker and directly convert the predicate into LTL (linear temporal logic). Figure 6 shows the Invariant checker after we have added an invariant to the specification which constrains the maximum value of operationToday in all states. In Figure 7 we have created a linear temporal logic formula which requires the DepositMoney operation not to be enabled if the value of operationToday exceeds the maximum allowed (OperationLimit) (we could perform similar checks for all other operations to ensure they are similarly not enabled under this condition). Other

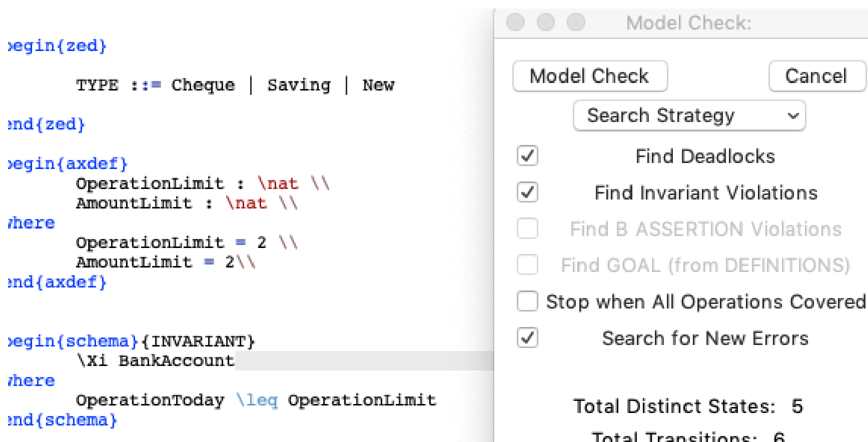

Figure 6. Invariant Checking

conditions may be identified from the predicate set in a similar manner and checked accordingly, for example the balance of an account must always be greater than 0, which is something we can check for in the model.

Again, the intention here is not to develop a fully comprehensive model-checking strategy from the predicates, but rather use them to assist the process by linking the user-requirements from the scenarios to the formal process of specification and

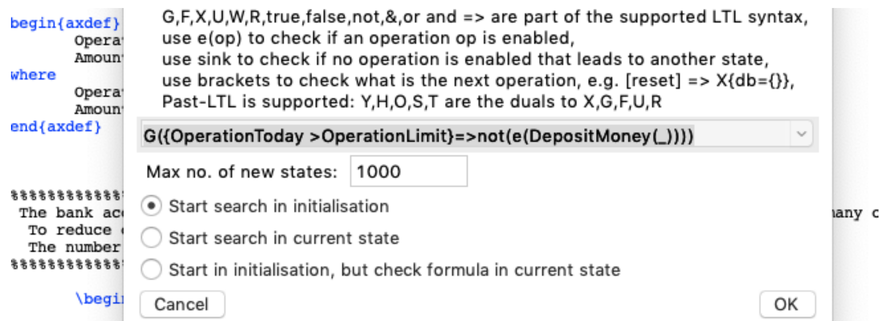

Figure 7. Using LTL to Model Check

model-checking. This ultimately contributes to our larger goal of enabling the user-centred design artefacts and formal specifications to be used together within the design process.

4.4.3. Refinement

The final example we give of using the predicates is as a form of refinement. The goal here is to demonstrate formally that the two specifications (behavioural and Z) are consistent. Refinement and refinement algebra are used to transform formal specifications (which are typically abstract) into more concrete representations (Derrick & Boiten, 2001). While the ultimate goal may be an implemented system, refinement is equally well-used in considering specifications of the system at different levels of abstraction. In this way it enables us to understand whether or not two representations of the same system can be considered equivalent (or consistent).

We want to show that the predicates derived from the specification are a subset (or partial refinement) of our larger specification. This then guarantees that the properties from the user features are satisfied in the specification - i.e. they are consistent. We may need to deal with inconsistencies in naming (as the predicates are derived from the scenarios and produced by different parts of the development team, and as such there is no guarantee they will use the same terminology). If we are going to perform a formal proof of refinement we might do this by creating a retrieve relation similar to those used in data refinement (Derrick & Boiten, 2001) where we take two state spaces - usually an abstract and a concrete space but in this example the state space from the predicates and that of the Z specification - and then relate the observations. However, in a less formal process we might just rename the observations of the predicates to match those of the formal specification. For each of the predicates we then show that they are implied by the specification. This is similar to the model-checking process above, but rather than identifying specific properties from the predicates as we did with the model-check process, here we must ensure that the full set of predicates are implied by the specification to guarantee consistency.

4.5. Summary

In this section we have demonstrated an approach that converts feature files into first-order logic predicates and shown several ways that these can be used. This demonstrates the applicability of using model transformations designed to provide a tighter

coupling between informal and user-centred processes with more formal software engineering processes. The process provides both a way of supporting non-experts in creating formal specifications (by deriving what we might call specification snippets from the automatically created predicates) and a way of using behavioural specifications and formal specifications together for model-checking or refinement. The first part of the process, whereby the predicates are generated from the behavioural specification, can be automated as long as the behavioural specification is structured according the the given rules. The remainder of the process is currently manual, as it requires that the language and naming conventions of the formal specification match those of the predicates, we discuss this further in section 5.

5. Discussion

In this work we have presented two new methods which answer the research question - What methods can we use to integrate informal software design methods with more formal processes? Our first new method also addresses the first sub-question - How do we support end-users in providing requirements in a structured manner such that they can be integrated into a formal development process using Petri-nets? While our second new method address sub-question two - How can we ensure that user-centred requirements expressed in behavioural specifications are consistent with formal specifications? The examples we have used to demonstrate the new methods show both the applicability and the benefits of these approaches. However, despite these benefits, there are also challenges that still need to be addressed, these relate to each of the methods individually, as well as to the overall concept of increasing the types of models and methods used within an integrated software development approach.

In our first method, we identified that the full set of BPMN nodes is too complex, and thus the resulting restriction of nodes is necessary, which also results in a restriction in the expressiveness of the language itself. Together with the limited ability to abstract from specific context and process (such as work habits) to model-based descriptions, it is not clear how accurate the generated models actually are. Thus, we are currently investigating this in ongoing work in which we compare thinking-aloud protocols to BPMN models. A further challenge is the automatic mapping of informal node inscriptions into the Petri net. Here, manual intervention by the system designer is needed such that events triggered in the physical context, or by the user, are mapped on the generated transitions in the Petri net. This problem may be addressed in future work by using knowledge bases or other types of models (e.g. using machine learning-based processing of natural language) to map inscriptions in the BPMN model to process-related information as used in the persuasive system.

Both of our methods involve the use of model transformations and one of the things we need to be mindful of when performing transformations between models is loss of information. This is particularly true when dealing with models and artefacts that focus on differing aspects of a system or present information from differing perspectives. For example, in our second method, the behavioural models are described from the perspective of users and use and as such there will typically be information that we do not include in the formal models. We must be clear about what is excluded, and ensure that we retain all of the original models to be used in their own parts of the development process. In this way, our methods should be seen as adding to the development process(by adding new models) rather than reducing or removing steps (by combining existing information).

While it is easy to argue that the types of methods we propose, which enable integration of informal user-centred work with formal specifications and formal methods provide benefits, we are mindful of the difficulty faced in gaining acceptance of new methods by software developers. This can be particularly true when proposing the inclusion of more formal methods which can be seen as adding unnecessary time and cost (J. P. Bowen & Stavridou, n.d.). Even though our methods are intended to reduce the overhead of producing formal models directly, by using informal user-centred design approaches as the basis for their development, the full benefits are achieved when this is integrated into formal approaches more generally (see the bottom section of figure 1). For safety-critical interactive systems (medical devices, avionics etc.) it is more accepted that formal methods are needed (Fayollas et al., 2013; Masci et al., 2013), we intend that our work here guides their use more generally in interactive system design and development.

Formal methods and specification-based testing rely on having models that correctly describe requirements, and this is often considered the most crucial aspect in software engineering processes. Because the requirements are not formal, the 'gap' that exists between these and the formal models is a common source of errors (Stocks & Carrington, 1996). Our motivation in developing processes that brings these closer together is to continue to add new methods to the software engineering tools that can be used to minimise this.

Our two new processes have different entry points within the software development process. The first starts at the requirements stage and uses firstly a structured method to gather user information from interviews to construct BMPN models. These can then be automatically transformed to Petri Nets, which in turn can be be used within simulation approaches and bound to Java code execution. There is, therefore, more automation currently in this process which is beneficial in encouraging uptake and use (as it does not increase workload). In addition, the Petri nets that are produced can be used within standard formal methods processes (e.g. verification and validation) and in this respect both of our processes have the same end-point as they contribute to the set of formal models that can be used within such formal processes. Our second approach is more manual. While the creation of the predicates from the behaviour specification is automated, the use of the predicates in the three ways described requires manual intervention, as it requires human interpretation to map the language of the predicates to that of the formal specification. We consider this again under future work.

5.1. Future Work

Considering the previous discussion and the presentation of the two methods, we now outline aspects for future work.

First, considering the BPMN to Petri net approach, the first aspect to be addressed might be a closer look into the actual quality of the created models. This will face the issue that it will be hard to identify a ground truth, thus, answering the question what is the "correct" habit model. However, this could be compensated by implementing a study design in which participants are asked to model a given habit (not their own). A second aspect could consider to extend the actual approach with further data sources, such as the previously mentioned aural interview data. Finally, the method should be investigated in terms of its applicability in other domains than the design and implementation of persuasive systems. This might be of specific interest in terms of the design and implementation of safety critical systems considering the Petri net's

potential for being used for formal verification.

The use of the predicates created from the behavioural specifications is mostly a manual process. While this is largely necessitated by the use of natural language in the specification and the likelihood that naming conventions and terminology in the formal specification will be different, there are opportunities to explore further automation. In (Silva, Hak, & Winckler, 2017), for example, Silva et al. use ontologies to map actions from behavioural specifications to widgets in user interface descriptions. While their work focusses on sets of widgets which are more constrained than the text we encounter in our process, this is still an area that might be considered in future work. Additionally, while we have described a mechanism for using relations between the predicates and formal specification to support refinement, there are many possibilities for extending this. For example, creating retrieve relations between the two specification that are then used throughout the development process to ensure consistency is maintained would be a valuable addition to the process.

Finally, while in this work we may have particular reasons for wanting to derive formal models from the informal, going the other way (from the formal to the informal) may be equally important in other situations. We have not discussed this here but it remains an open topic for future work.

6. Conclusion

The work presented in this paper demonstrates different ways of bringing together informal design and user artefacts and more formal engineering representations, through a process of transformation. These approaches enable us to create a tighter coupling between different parts of the software design and development processes, recognising that the different methods used in each of these have their own intrinsic value and are worth preserving. In addition to the benefits we have shown that the tighter coupling of informal and formal methods can provide, it has also been shown previously J. Bowen and Reeves (2017b) that having different views of the same system (by way of different models) can elicit problems or hidden aspects that otherwise may not be obvious in one single model. As such, the type of work we present here supports a variety of important use cases, and the examples presented here are by no means exhaustive, rather they serve as motivation for what can be achieved through the use of such transformations.

The two processes we present can be used to enhance formal software development processes by providing a way of integrating less formal (but equally important) design artefacts. As such, they contribute to the body of knowledge for software engineering and equip software engineers with practical methods for interactive system design.

7. Acknowledgments

We would like to thank Sabine Sonnentag, Wilken Wehrt, & Yuen C. Law for their contribution and work within our collaborative research project, their valuable advice, comments and support. Additionally, we would like to thank Florian Langel for supporting the implementation of the VR scenario used for showing the applicability of Petri nets in the implementation of persuasive systems. This work is supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under Grant No.: 318151256.

References

Abras, C., Maloney-Krichmar, D., Preece, J., et al. (2004). User-centered design. *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*, *37*(4), 445–456.

Abrial, J. (1988). The B tool (abstract). In R. E. Bloomfield, L. S. Marshall, & R. B. Jones (Eds.), *VDM '88, VDM - the way ahead, 2nd vdm-europe symposium, dublin, ireland, september 11-16, 1988, proceedings* (Vol. 328, pp. 86–87). Springer.

Acceptance test driven development (atdd). (n.d.). (https://www.agilealliance.org/glossary/atdd/ Last accessed 17 March 2019)

Alonso, O., & Baeza-Yates, R. (2011). Design and implementation of relevance assessments using crowdsourcing. In *European conference on information retrieval* (pp. 153–164).

Ameur, Y. A., Bowen, J., Campos, J. C., Palanque, P. A., & Weyers, B. (2021). Heterogeneous models and modelling approaches for engineering of interactive systems. *Interact. Comput.*, *33*(1), 1–2.

Basile, D., ter Beek, M. H., Fantechi, A., Gnesi, S., Mazzanti, F., Piattino, A., ... Ferrari, A. (2018). On the industrial uptake of formal methods in the railway domain. In C. A. Furia & K. Winter (Eds.), *Integrated formal methods* (pp. 20–29). Cham: Springer International Publishing.

Beck, K. (2002). *Test driven development: By example.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Beck, K., Beedle, M., Van Bennekum, A., A.and Cockburn, Cunningham, W., Fowler, M., & Grenning, J. (n.d.). *Agile manifesto, 2001.* (from http://www.agilemanifesto.org, Last accessed September 2019)

Bødker, S., Ehn, P., Sjögren, D., & Sundblad, Y. (2000). Cooperative design perspectives on 20 years with "the scandinavian it design model". In *Proceedings of the first nordic conference on human-computer interaction.* United States: Association for Computing Machinery. (Invited Keynote; null ; Conference date: 23-10-2000 Through 25-10-2000)

Bolton, M. L., Bass, E. J., & Siminiceanu, R. I. (2012, nov). Generating phenotypical erroneous human behavior to evaluate human-automation interaction using model checking. *Int. J. Hum.-Comput. Stud.*, *70*(11), 888–906.

Bonfanti, S., Gargantini, A., & Mashkoor, A. (2018). A systematic literature review of the use of formal methods in medical software systems. *Journal of Software: Evolution and Process*, *30*(5), e1943.

Borisov, N., Weyers, B., & Kluge, A. (2018). Designing a human machine interface for quality assurance in car manufacturing: An attempt to address the "functionality versus user experience contradiction" in professional production environments. *Advances in Human-Computer Interaction*.

Bowen, J., & Gyde, S. (2015). *PIMed.* (An editor for presentation models and presentation interaction models, https://sourceforge.net/projects/pims1/, Last accessed 05 April 2021)

Bowen, J., & Khanal, S. (2018). Test stub generation from interaction and behavioural models. In *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems, EICS 2018, paris, france, june 19-22, 2018* (pp. 7:1–7:6). ACM.

Bowen, J., & Reeves, S. (2008). Formal models for user interface design artefacts. *Innovations in Systems and Software Engineering*, *4*(2), 125-141.

Bowen, J., & Reeves, S. (2009). Ui-design driven model-based testing. *ECEASST*, *22*.

Bowen, J., & Reeves, S. (2013). Modelling safety properties of interactive medical systems. In *ACM SIGCHI symposium on engineering interactive computing systems, eics'13, london, united kingdom - june 24 - 27, 2013* (pp. 91–100).

Bowen, J., & Reeves, S. (2017a). Combining models for interactive system modelling. In B. Weyers, J. Bowen, A. J. Dix, & P. A. Palanque (Eds.), *The handbook of formal methods in human-computer interaction* (pp. 161–182). Springer International Publishing.

Bowen, J., & Reeves, S. (2017b). Combining models for interactive system modelling. In

B. Weyers, J. Bowen, A. J. Dix, & P. A. Palanque (Eds.), *The handbook of formal methods in human-computer interaction* (pp. 161–182). Springer International Publishing.

Bowen, J. P., & Stavridou, V. (n.d.). Safety-critical systems, formal methods and standards. *Softw. Eng. J.*, *8*, 189-209.

Bowen, J. P., & Stavridou, V. (1993a). The industrial take-up of formal methods in safety-critical and other areas: A perspective. In J. Woodcock & P. G. Larsen (Eds.), *FME '93: Industrial-strength formal methods, first international symposium of formal methods europe, odense, denmark, april 19-23, 1993, proceedings* (Vol. 670, pp. 183–195). Springer.

Bowen, J. P., & Stavridou, V. (1993b). Safety-critical systems, formal methods and standards. *Softw. Eng. J.*, *8*(4), 189–209.

Campos, J. C., & Harrison, M. D. (2009). Interaction engineering using the ivy tool. In *Proceedings of the 1st acm sigchi symposium on engineering interactive computing systems* (p. 35–44). New York, NY, USA: Association for Computing Machinery.

Carter, J. D., & Gardner, W. B. (2018). Bhive: Behavior-driven development meets b-method. In S. H. Rubin & T. Bouabana-Tebibel (Eds.), *Quality software through reuse and integration* (pp. 232–255). Cham: Springer International Publishing.

Claessen, K., & Hughes, J. (2000). Quickcheck: a lightweight tool for random testing of haskell programs. In M. Odersky & P. Wadler (Eds.), *Proceedings of the fifth ACM SIGPLAN international conference on functional programming (ICFP '00), montreal, canada, september 18-21, 2000.* (pp. 268–279). ACM.

Colombo, C., Micallef, M., & Scerri, M. (2014). Verifying web applications: From business level specifications to automated model-based testing. In H. Schlingloff & A. K. Petrenko (Eds.), *Proceedings ninth workshop on model-based testing, MBT 2014, grenoble, france, 6 april 2014.* (Vol. 141, pp. 14–28).

Cucumber and scenario outline. (n.d.). (https://www.baeldung.com/cucumber-scenario-outline,
Last accessed 18 June 2019)

Cucumber scenario outline. (n.d.). (https://www.tutorialspoint.com/cucumber/cucumber_scenario_outline.html,
Last accessed 18 June 2019)

Derrick, J., & Boiten, E. (2001). *Refinement in z and object-z: Foundations and advanced applications.* Springer.

Dijkman, R. M., Dumas, M., & Ouyang, C. (2008). Semantics and analysis of business process models in bpmn. *Information and Software Technology*, *50*, 1281-1294.

Dijkstra, E. (1976). *Report on a conference sponsored by the nato science committee, rome, italy* (J. N. Buxton & B. Randell, Eds.). Petrochelli Charter.

Dix, A. (1991). *Formal methods for interactive systems.* Academic Press.

Fayollas, C., Fabre, J., Palanque, P. A., Barboni, E., Navarre, D., & Deleris, Y. (2013). Interactive cockpits as critical applications: a model-based and a fault-tolerant approach. *Int. J. Crit. Comput. Based Syst.*, *4*(3), 202–226.

Fischer, B., Peine, A., & Östlund, B. (2020). The importance of user involvement: a systematic review of involving older users in technology design. *The Gerontologist*, *60*(7), e513–e523.

Harrison, M. D., Freitas, L., Drinnan, M., Campos, J. C., Masci, P., di Maria, C., & Whitaker, M. (2019). Formal techniques in the safety analysis of software components of a new dialysis machine. *Sci. Comput. Program.*, *175*, 17–34.

Harrison, M. D., Masci, P., & Campos, J. C. (2018). Formal modelling as a component of user centred design. In M. Mazzara, I. Ober, & G. Salaün (Eds.), *Software technologies: Applications and foundations - STAF 2018 collocated workshops, toulouse, france, june 25-29, 2018, revised selected papers* (Vol. 11176, pp. 274–289). Springer.

Hatcliff, J., Larson, B., Carpenter, T., Jones, P., Zhang, Y., & Jorgens, J. (2019, aug). The open pca pump project: An exemplar open source medical device as a community resource. *SIGBED Rev.*, *16*(2), 8–13.

Jaidka, S., Reeves, S., & Bowen, J. (2019). A coloured petri net approach to model and analyze safety-critical interactive systems. In *26th asia-pacific software engineering conference,*

APSEC 2019, putrajaya, malaysia, december 2-5, 2019 (pp. 347–354). IEEE.

Kummer, O. (2001). Introduction to petri nets and reference nets..

Kummer, O., Wienberg, F., Duvigneau, M., Köhler, M., Moldt, D., & Rölke, H. (2000). Renew–the reference net workshop. In *Tool demonstrations, 21st international conference on application and theory of petri nets, computer science department, aarhus university, aarhus, denmark* (pp. 87–89).

Lack, R. (2007). The importance of user-centered design: Exploring findings and methods. *Journal of Archival Organization*, *4*(1-2), 69–86.

Langel, F., Law, Y. C., Wehrt, W., & Weyers, B. (2018). A virtual reality framework to validate persuasive interactive systems to change work habits. *Mensch und Computer 2018-Workshopband*.

Law, Y. C., Wehrt, W., Sonnentag, S., & Weyers, B. (2017). Generation of information systems from process models to support intentional forgetting of work habits. In *Proceedings of the acm sigchi symposium on engineering interactive computing systems* (pp. 27–32).

Law, Y. C., Wehrt, W., Sonnentag, S., & Weyers, B. (2022). Obtaining semi-formal models from qualitative data: From interviews into bpmn models in user-centered design processes. *International Journal of Human–Computer Interaction*, *0*(0), 1-18. Retrieved from https://doi.org/10.1080/10447318.2022.2041899

Liu, B. (2019). *Using behavioural specifications to support model-checking* (Unpublished master's thesis). The University of Waikato, Available at https://hdl.handle.net/10289/13028.

Lubke, D., & van Lessen, T. (2016, sep). Modeling test cases in bpmn for behavior-driven development. *IEEE Software*, *33*(05), 15-21.

Masci, P., Ayoub, A., Curzon, P., Harrison, M. D., Lee, I., & Thimbleby, H. W. (2013). Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. In P. Forbrig, P. Dewan, M. Harrison, & K. Luyten (Eds.), *ACM SIGCHI symposium on engineering interactive computing systems, eics'13, london, united kingdom - june 24 - 27, 2013* (pp. 81–90). ACM.

Navarre, D., Palanque, P. A., Coppers, S., Luyten, K., & Vanacken, D. (2019). Fortune nets for fortunettes: Formal, petri nets-based, engineering of feedforward for GUI widgets. In E. Sekerinski et al. (Eds.), *Formal methods. FM 2019 international workshops - porto, portugal, october 7-11, 2019, revised selected papers, part I* (Vol. 12232, pp. 503–519). Springer.

Norman, D. A., & Draper, S. W. (1986). *User centered system design: new perspectives on human-computer interaction.* Lawrence Erlbaum Associates, Hillsdale.

North, D. (2006). *Introducing bdd.* (https://dannorth.net/introducing-bdd/, Last accessed 18 March 2019)

North, D. (2009). *How to sell bdd to the business.* (https://skillsmatter.com/skillscasts/923-how-to-sell-bdd-to-the-business#showModal?modal-signup-complete/, Last accessed 18 March 2019)

Oinas-Kukkonen, H., & Harjumaa, M. (2008). A systematic framework for designing and evaluating persuasive systems. In *International conference on persuasive technology* (pp. 164–176).

Prates, R. O., Palanque, P. A., Weyers, B., Bowen, J., & Dix, A. J. (2017). State of the art on formal methods for interactive systems. In B. Weyers, J. Bowen, A. J. Dix, & P. A. Palanque (Eds.), *The handbook of formal methods in human-computer interaction* (pp. 3–55). Springer International Publishing.

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., & Carey, T. (1994). *Human-computer interaction.* Addison-Wesley Longman Ltd.

Reisig, W. (2012). *Petri nets: an introduction* (Vol. 4). Springer Science & Business Media.

Rice, B., Jones, R., Bittner, P., & Engel, J. (2014). *Welcome to behave!* (https://behave.readthedocs.io/en/latest/ Last accessed 13 April 2022)

Rose, S., Wynne, M., & Hellesoy, A. (2015). *The cucumber for java book: Behaviour-driven development for testers and developers.* Pragmatic Bookshelf.

Silva, T. R., Hak, J., & Winckler, M. (2017). A formal ontology for describing interactive behaviors and supporting automated testing on user interfaces. *Int. J. Semantic Comput.*,

11(4), 513–540.

Snook, C. F., Hoang, T. S., Dghaym, D., Butler, M. J., Fischer, T., Schlick, R., & Wang, K. (2018). Behaviour-driven formal model development. In J. Sun & M. Sun (Eds.), *Formal methods and software engineering - 20th international conference on formal engineering methods, ICFEM 2018, gold coast, qld, australia, november 12-16, 2018, proceedings* (Vol. 11232, pp. 21–36). Springer.

Sonnentag, S., Wehrt, W., Weyers, B., & Law, Y. C. (2022). Conquering unwanted habits at the workplace: Day-level processes and longer term change in habit strength. *Journal of Applied Psychology*, *107*(5), 831.

Sousa, M., Campos, J. C., Alves, M. C. B., & Harrison, M. D. (2014). Formal verification of safety-critical user interfaces: a space system case study. In *2014 AAAI spring symposia, stanford university, palo alto, california, usa, march 24-26, 2014.* AAAI Press.

Stevenson, C. (n.d.). *Testdox.* (http://agiledox.sourceforge.net/index.html,
Last accessed 17 April 2019)

Stocks, P., & Carrington, D. A. (1996). A framework for specification-based testing. *IEEE Trans. Software Eng.*, *22*(11), 777–793.

Stückrath, J., & Weyers, B. (2014). Lattice-extended coloured petri net rewriting for adaptable user interface models. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, *67*.

Thimbleby, H. (2015). Safer user interfaces: A case study in improving number entry. *IEEE Trans. Software Eng.*, *41*(7), 711–729.

Turner, J., Bowen, J., & Reeves, S. (2020). Seqcheck: a model checking tool for interactive systems. In J. Bowen, J. Vanderdonckt, & M. Winckler (Eds.), *EICS '20: ACM SIGCHI symposium on engineering interactive computing systems, sophia antipolis, france, june 23-26, 2020* (pp. 7:1–7:6). ACM.

Utting, M., & Legeard, B. (2006). *Practical model-based testing - a tools approach.* Morgan and Kaufmann.

Van der Aalst, W. M. (1997). Verification of workflow nets. In *International conference on application and theory of petri nets* (pp. 407–426).

Weyers, B., Bowen, J., Dix, A., & Palanque, P. A. (Eds.). (2017). *The handbook of formal methods in human-computer interaction.* Springer International Publishing.

Weyers, B., Burkolter, D., Luther, W., & Kluge, A. (2012a). Formal modeling and reconfiguration of user interfaces for reduction of errors in failure handling of complex systems. *International Journal of Human-Computer Interaction*, *28*(10), 646–665.

Weyers, B., Burkolter, D., Luther, W., & Kluge, A. (2012b). Formal modeling and reconfiguration of user interfaces for reduction of errors in failure handling of complex systems. *Int. J. Hum. Comput. Interact.*, *28*(10), 646–665.

What is domain-driven design. (2007). (http://dddcommunity.org/learning-ddd/what_is_ddd/,
Last accessed 17 March 2019)

White, S. A. (2004). Introduction to BPMN..

Writing scenarios with gherkin syntax. (n.d.). (https://hiptest.com/docs/writing-scenarios-with-gherkin-syntax/,
Last accessed 10 June 2019)

Zielasko, D., Weyers, B., Bellgardt, M., Pick, S., Meibner, A., Vierjahn, T., & Kuhlen, T. W. (2017). Remain seated: towards fully-immersive desktop vr. In *2017 ieee 3rd workshop on everyday virtual reality (wevr)* (pp. 1–6).

Zuccon, G., Leelanupab, T., Whiting, S., Yilmaz, E., Jose, J. M., & Azzopardi, L. (2013). Crowdsourcing interactions: using crowdsourcing for evaluating interactive information retrieval systems. *Information retrieval*, *16*(2), 267–305.