COMPUTING SCIENCE

Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems

Fuyuki Ishikawa, Alexander Romanovsky, Elena Troubitsyna

TECHNICAL REPORT SERIES

No. CS-TR-1503  November 2016
Abstract.

Model-Driven Development (MDD) and Formal Methods (FM) put the emphasis on the use of models through the process of system development. Though MDD traditionally rely on graphical model representations, while FM uses mathematical notations, they pursue the same goal of ensuring dependability and robustness of the software intensive systems. The workshop looks into how to enhance the use of MDD by the power of FM for assurance and dependability. The focus is on discussing the interplay between the use of FM approaches and the MDD techniques, and on ensuring a smooth integration of the two viewpoints. The programme of the workshop consists of two invited talks, one invited industrial presentation and six technical talks to be given by the experts from all over the world.


Bibliographical details

Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems

Fuyuki Ishikawa, Alexander Romanovsky, Elena Troubitsyna

NEWCASTLE UNIVERSITY


Abstract

Model-Driven Development (MDD) and Formal Methods (FM) put the emphasis on the use of models through the process of system development. Though MDD traditionally rely on graphical model representations, while FM uses mathematical notations, they pursue the same goal of ensuring dependability and robustness of the software intensive systems. The workshop looks into how to enhance the use of MDD by the power of FM for assurance and dependability. The focus is on discussing the interplay between the use of FM approaches and the MDD techniques, and on ensuring a smooth integration of the two viewpoints.

The programme of the workshop consists of two invited talks, one invited industrial presentation and six technical talks to be given by the experts from all over the world.

About the authors

Fuyuki Ishikawa is an Associate Professor at the National Institute of Informatics (NII) in Tokyo, Japan. He received the PhD degree in information science and technology from the University of Tokyo in Japan, in 2007. He is a visiting associate professor at the University of Electro-Communications in Japan. His research interests include service-oriented computing and software engineering. He has served as the leader of 6 funded projects and published more than 100 papers.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software and Reliability, Newcastle University. He is the leader of the Dependability Group at the School of Computing Science. His main research interests are system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring and verification of fault tolerance. He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, University of Newcastle upon Tyne. Alexander is now the Principle
Investigator of the TrAmS-2 EPSRC/UK platform grant on Trustworthy Ambient Systems (2012-16) and of the EPSRC/RSSB research project SafeCap on Overcoming the Railway Capacity Challenges without Undermining Rail Network Safety (2011-14), and the Co-investigator of the EPSRC PRiME program grant (2013-18) and of the FP7 COMPASS Integrated Project (2011-14).

Elena Troubitsyna is an Associate Professor at Aabo Akademi University, Finland. Her research interests include dependable and resilient software-intensive systems, formal modelling and verification methods, model-driven engineering for dependability and resilience, safety-critical and fault tolerant systems and cyber security. She is now the project coordinator of the three projects: CoRA on Continuous Resilience Assurance of Complex Software-Intensive Systems, Open-CPS on Open Integrated Framework for Accelerating Development of Resilient CPS and CyberTrust.

**Suggested keywords**

Model-Driven Development, Formal Methods, assurance, dependability, system engineering
FM&MDD Workshop at ICFEM 2016

Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems

November 14, 2016
Tokyo, Japan

Organisers:

Fuyuki Ishikawa (NII, Japan)
Alexander Romanovsky (Newcastle University, UK)
Elena Troubitsyna (Aabo Akademi, Finland)
Welcome to the FM&MDD workshop in Tokyo!

Model-Driven Development (MDD) and Formal Methods (FM) put the emphasis on the use of models through the process of system development. Though MDD traditionally rely on graphical model representations, while FM uses mathematical notations, they pursue the same goal of ensuring dependability and robustness of the software intensive systems.

The workshop looks into how to enhance the use of MDD by the power of FM for assurance and dependability. The focus is on discussing the interplay between the use of FM approaches and the MDD techniques, and on ensuring a smooth integration of the two viewpoints.

The programme of the workshop consists of two invited talks, one invited industrial presentation and six technical talks to be given by the experts from all over the world.

We hope you will enjoy the event,

Fuyuki Ishikawa, Alexander Romanovsky and Elena Troubitsyna
Programme of the FM&MDD Workshop

10.00-10.05 – Introduction from the Chairs

10.05-11.05 – Invited talk 1. Mark Lawford. Stupid Tool Tricks for Smart Model Based Design

11.05-11.20 – coffee

11.20-12.50 – Technical talks
- Quanqi Ye, Guangdong Bai, Naipeng Dong and Jin Song Dong. ProRef: An Automatic Authentication Protocol Refinement Tool for Extracting Formal Models
- Tom Bienmüller, Tino Teige, Andreas Eggers and Matthias Stasch. Modeling Requirements for Quantitative Consistency Analysis and Automatic Test Case Generation
- Ahmed Al-Brashdi, Michael Butler, Abdolbaghi Rezazadeh and Colin Snook. Tool Support for Model-Based Database Design with Event-B

12.50-13.50 - lunch


14.50-15.50 – Technical talks
- Manuel Toews, Marie-Christine Jakobs and Felix Pauck. PAndA^2: Analyzing Permission Use and Interplay in Android Apps (Tool Paper)
- Rajiv Murali, Andrew Ireland and Gudmund Grov. A Formal Approach to Use Case Driven Testing

15.50- 16.20 – coffee

16.20 – 16.50 – Invited industrial presentation. Daichi Mizuguchi. A trial application of B method for an embedded device by constructing B model via UML

16.50-17.40 – Technical talk
- Alexei Iliasov. Safety Kernel for Control Systems Design

17.40 - Close
Workshop papers and abstracts of the talks

1. Mark Lawford. Stupid Tool Tricks for Smart Model Based Design (Invited talk)
2. Kenji Tei. Assured and Correct Dynamic Update of Controllers (Invited talk)
3. Daichi Mizuguchi. A trial application of B method for an embedded device by constructing B model via UML (Invited industrial presentation)
5. Tom Bienmüller, Tino Teige, Andreas Eggers and Matthias Stasch. Modeling Requirements for Quantitative Consistency Analysis and Automatic Test Case Generation
7. Manuel Toews, Marie-Christine Jakobs and Felix Pauck. PAndA^2: Analyzing Permission Use and Interplay in Android Apps (Tool Paper)
8. Rajiv Murali, Andrew Ireland and Gudmund Grov. A Formal Approach to Use Case Driven Testing
9. Alexei Iliasov. Safety kernel for control systems design
Stupid Tool Tricks for Smart Model Based Design

(Invited Talk)

Mark Lawford
McMaster Centre for Software Certification
McMaster University
Canada

Abstract. Formal methods tools can be used to detect and prevent errors so researchers assume that industry will use them. We are often frustrated when we see industrial projects where tools could have been used to detect or prevent errors in the final product. Researchers often fail to realize that there is a significant gap between a potentially useful tool and its use in a standards compliant, commercially viable, development process. In this talk I take a look at seemingly mundane industrial requirements - qualification (certification) of tools for use in standards compliant development process for general safety (IEC 61508), Automotive (ISO 26262) and Avionics (DO-178C), Model Based Design coding guidelines compliance, standards compliance documentation generation and integration with existing industry partner development processes. For each of these topics I show how “stupid tool tricks” can be used to not only increase adoption of academic methods and tools, but also lead to interesting research questions with industry relevant results.
Assured and Correct Dynamic Update of Controllers

(Invited Talk)

Kenji Tei

National Institute of Informatics
Japan

Abstract. In many application domains, continuous operation is a desirable attribute for software-intensive systems. As the environment or system requirements change, so the system should change and adapt without stopping or unduly disturbing its operation. There is, therefore, a need for sound engineering techniques that can cope with dynamic change. In this keynote, I will address the problem of dynamic update of controllers in reactive systems when the specification (environment assumptions, requirements and interface) of the current system changes. I will present a general approach to specifying correctness criteria for dynamic update and a technique for automatically computing a controller that handles the transition from the old to the new specification, assuring that the system will reach a state in which such a transition can correctly occur. Indeed, using controller synthesis I will show how to automatically build a controller that guarantees both progress towards update and safe update. Seven case studies have been implemented to validate the approach.
A Trial Application of B Method for an Embedded Device by Constructing B Model via UML

(Invited Industrial Presentation)

Daichi Mizuguchi
Atelier Corporation
Japan

Abstract. We are trying to apply the B method for the development of embedded software for a safety-critical sensor device. We report our methodology, in which B models are constructed based on UML, and our outcomes.
ProRef: An Automatic Authentication Protocol Refinement Tool for Extracting Formal Models

Quanqi Ye\textsuperscript{1}, Naipeng Dong\textsuperscript{2}, Guangdong Bai\textsuperscript{3}, and Jin Song Dong\textsuperscript{2}

\textsuperscript{1} NUS Graduate School for Integrative Sciences and Engineering, National University of Singapore
\textsuperscript{2} School of Computing, National University of Singapore
\textsuperscript{3} Singapore Institute of Technology

Abstract. To ensure the correctness of the authentication protocol implemented in a web application, formal analysis is needed. A key part in the formal analysis is to abstract a formal model of the protocol from implementation. To do so, a common method is to record the network traffics of the authentication process and infer the formal model from the traffics. However, the protocol directly derived from the traffics often has many redundant parameters and cookies in the messages, an analyst often needs to manually perform fuzzing to the protocol implementation to derive a refined model. The fuzzing process may require tens of thousands of re-sending and comparing messages, and thus it is labor intensive and consumes the analyst too much time and effort. Therefore, we propose ProRef—a tool to automatically refine the raw network communication messages for web-based authentication protocol implementations. This tool aims to help security analysts derive refined models of underlying protocols in web applications and to facilitate the formal verification of the protocol implementations.

1 Introduction

Security has become an increasingly important concern in web applications. Ensuring security of web applications has become necessary and important. We focus on a widely used functionality in most web applications—the authentication. Due to its preciseness, formal analysis is a promising way to verify whether an implemented authentication protocol satisfies security properties [1].

Intuitively, one can record the communicated messages when logging into a web application, consider the messages as the authentication protocol, and perform the formal analysis. However, there are often many redundant parameters in the recorded network messages whose existence does not affect the result of the authentication, for example, the cookies identifying the screen resolution, the version of the client being used, the version of the operating system, etc. These cookies and parameters may be meaningful to the entire web application, but they are meaningless to the authentication. Keeping these redundant cookies and parameters in the formal model makes the formal model unnecessarily complicated and may cause timeout during verification.

However, it is hard for the security analyst to know which parameters are critical to the authentication result. Especially, the protocol designer, the developer who implements the protocol and the security analyst of the protocol implementation are often not the same person. Therefore, the analyst often needs
to perform black-box fuzzing[^4] to identify the redundant parameters. The fuzzing procedure and algorithm are as follows:

1. For every message in the protocol, we first parse and extract the parameters in its URL string, cookie headers and the data posted.
2. Next, we remove one of the parameters parsed in the previous step and send the new message with one parameter removed to the server.
3. The response from the server is compared with the original response (when no parameter is removed) to determine if the message with a parameter removed successfully triggers the same response from the server. If the answer is yes, then the removed parameter causes no effect to the result of the authentication. Otherwise, the removed parameter is indeed critical to the successful authentication and thus cannot be removed.
4. After the process iterates on all the parameters, the fuzzing for that message is finished and the same procedure continues with the next message.

This procedure continues until all the redundant cookies and parameters in all messages are removed. Manual fuzzing is labor intensive and would consume too much time of the analyst to derive a refined protocol model.

Therefore, we propose ProRef, an automatic tool for security analyst to refine the raw protocol under analysis. Given a raw network message in the protocol, ProRef can automatically check which parameters and cookies are redundant and automatically generate the refined version of the message. We make ProRef as general as possible so that it can be applied to different authentication protocols of different websites. We have tested it using Facebook and Instagram and it can successfully produce the refined authentication protocols for both of the websites. Although more experiments are needed in the future development of the tool, we believe using this tool in analyzing the protocol would bring great convenience to the security analysts.

2 Running Example

In this section, we introduce ProRef with a running example, the Instagram[^5] login authentication protocol.

When ProRef is started up, a window pops up for displaying the captured traffics. Then the security analyst opens the browser and logs in to Instagram. Meanwhile, ProRef automatically records the messages exchanged between the browser and the remote Instagram servers. Fig. 1a shows the interface of ProRef with the recorded messages. Each message is a string of text with an ID at the beginning. The messages are organized in a tree structure in the TreeView tab showing the triggering relationships of the messages. The parent message triggers the children messages.

[^4]: Fuzz here is a terminology meaning that the security analyst handcrafts requests and sends them to the server to test the protocol.
[^5]: Instagram is a popular online photo sharing social network. It can be visited from www.instagram.com. 
When the security analyst wants to fuzz a message, she can double click on the message in the TreeView, and the fuzzing view interface in Fig. 1b pops up. Then the security analyst double clicks at the space at Fuzzing view to start fuzzing the message. When the fuzzing process is completed, the result message is automatically saved in a plain text file. Inside that file, the original request and the refined request are both saved in order to show which parameters and cookies are removed.

Based on the domain knowledge of security analysis, the login authentication protocol for Instagram involves three critical requests (the messages that drive the protocol flow). The raw messages of the three requests are shown as in Fig. 2a. After the security analyst clicks on the three critical requests to fuzz them one by one using ProRef, the refined login protocol is shown in Fig. 2b where many redundant cookies are removed. The red boxes in the figures show the differences between the raw protocol and the refined protocol. For the requests (messages sent from User to Ins), it is shown in Fig. 2 that parameters “mid”, “ig_pr” and “ig_vw” in the second request, and “csrf_token”, “mid”, “ig_pr”, “ig_vw” and
“ds_user_id” in the third request are refined. For the responses (messages sent from Ins to User), it is shown in Fig. 2 that the parameter “mid” is attached in the set-cookie command in the second and third response because our tool identifies that this parameter does not cast effect on the success authentication of user and thus removes it from the request. However, Instagram server detects the parameter “mid” is removed and thus resets it in the response.

3 Functionality

ProRef is an application layer proxy that can automatically refine authentication protocols for web applications. It mainly contains the following functionality:

- **Organizing messages**: ProRef organizes all the communicated messages in a tree view representing their triggering relations. Each message contains a request and its corresponding response. The response to a request in one message often contains triggers to requests in other messages. For example, a response containing an HTML file often triggers the browser to request and fetch images and Javascripts from the server. Therefore, the messages can be organized in a tree based on their triggering relations. This tree contains the protocol flow as well as other redundant requests (e.g., fetching images). Such a tree provides security analyst a visualized way of viewing the protocol.

- **Fuzzing messages**: ProRef automatically identifies the redundant parameters and cookies in a message and automatically removes them to produce a refined version of the message. This is the main functionality of ProRef, which saves the security analyst much time of fuzzing the protocol.

- **Storing the refined messages**: After the automatic refinement process, ProRef stores the refined messages in plain text. ProRef outputs both the refined and the original version of the messages, so that the security analyst can see which parameters and cookies are redundant and have been removed.

In summary, ProRef is designed to help security analyst refine the raw network communication messages to recover the refined protocol in order to perform formal analysis. It helps the security analyst to perform the repetitious fuzzing process which saves the analyst much time and effort.

4 Design

The structure of ProRef is shown in Fig. 3. ProRef serves as a proxy between the application under analysis and the Internet. It captures all the messages exchanged between the application and the remote servers that the application communicates with. ProRef contains three components: network, UI and core.

The network component has two units: proxy and message container. The proxy unit intercepts the network traffics. It is built upon Fiddler [2], a web debugging proxy. Fiddler provides the library which contains the main functionality of a proxy and it can handle HTTPS [3] traffics of applications. The message container unit stores all the network messages for analysis.

In the core components, there are four main units: the request generator, the requester, the response compare unit that is built upon a third-part library and the logger. Once the security analyst selects a message to fuzz, the message
is sent to the request generator. The request generator generates an HTTP(S) request with one parameter or cookie removed. The newly generated request is sent to the requester unit and it is sent to the communication partner from there. After being sent, the request goes through the proxy unit and is recorded for the purpose of comparing with the original request. On receiving the response, the requester unit compares it with the original response to determine whether the removed parameter is redundant or not. For two responses, the requester mainly compares their headers, cookies and HTML pages. For the headers and cookies, the requester can directly compare their strings. If the request header or cookie in the new response is different from that in the original response, it means that the new request cannot proceed to the next step of the protocol, and thus the new request fails the authentication, i.e., the modified parameter is not redundant and cannot be removed. As for HTML pages, the differences, caused by the dynamic content such as the news update in user’s social network homepage, in the HTML script string do not imply the failure of authentication. To determine whether the differences between the HTML page in the new response and that in the original response lead to authentication failure, we first use a third-party library named HtmlDiff \[4\] to show all the differences in the HTML pages, and then calculate the percentage of the difference. We consider the authentication fails if the difference exceeds certain percentage. Currently, we set the threshold value to 5\%, i.e., more than 5\% would lead to failure of comparison. The setting of the threshold percentage is empirical and there is not a ground standard for setting the threshold. More experiments are possible to derive a better value for the threshold. The above process in the requester unit iterates until no parameter or cookie can be removed, i.e., the refined message is obtained. Finally, the refined version of the message is written to the disk by the logger unit.

![Fig. 3: Structure of ProRef](image)

The UI component consists of two units: the tree view unit and the fuzzing view unit. The tree view unit displays all the messages in a tree view showing the triggering relations between the messages. For the triggering relations, first, if the request in a message has referer header \[5\], then the message is a child of the nearest earlier message that contains the request to the same URL in the referer header; second, if the response of a message is of status 302 redirection \[5\], then the message is the parent of the nearest later message that contains the request to the redirected URL. The proxy unit gives every message an unique number as its identity. The ordering of the identity numbers captures the chronological
sequences of the appearance of the messages. Smaller numbers mean that the messages appear earlier and larger ones indicate that the messages appear later. The fuzzing view shows information on how ProRef works during fuzzing process.

5 Limitation and Comparison

ProRef is open-source and free to download. The full installation instruction and pre-requisite environment of ProRef are available online [6].

Limitations. The limitations of ProRef are as follows:

- Currently the tree view is not 100% correct. Some of the requests are triggered by Javascript on certain events and the referer headers of those requests can be optionally deleted since referer headers may leak private information [7]. In this case, the parent-child relation between two messages is missing. Thus, the child message becomes the root of a new tree. Therefore, there may exist the situation that there are more than one root in the tree view. However, this problem will not affect the refinement functionality of ProRef as it does not rely on the parent-children relationships to perform the fuzzing. It only affects the security analysts as they might be confused as they can not find the parents of these messages without headers. One possible solution to address this problem is to introduce data dependency to determine the triggering relations.

- Security analyst needs to manually select the critical messages for the protocol. Due to the previous limitation, we did not implement the feature of automatically selecting critical messages for the protocol. We leave this part as our future work.

- Not all the messages are repeatable. For example, if the login authentication protocol implements an anti-bot system such as CAPTCHA [8], then the messages cannot be repeated automatically. To solve this limitation, more sophisticated strategy such as pattern recognition needs to be applied.

Comparison. Many protocol fuzzers have been developed. Most of them serve different purposes from the one proposed in this paper. Most of the prior developed tools aim at automatically detecting bugs and vulnerabilities of the web applications or protocols. They use various strategies to generate the inputs which are used to test the web applications or protocols. One of such tools is AspFuzz [9]. Similar to our tool, AspFuzz targets at the application layer protocol. To fuzz a web application, it automatically generates anomalous messages by twisting the format and the order of the messages exchanged in the protocol. However, this tool aims at automatically detecting bugs and vulnerabilities instead of refining the protocol messages to derive a refined protocol. Sudhodanan et al. propose a framework that is built upon OWASP ZAP to automatically test the web applications for vulnerabilities [10]. First, security experts summarize attack patterns based on literature and the known attacks, then, by configuring and recording the normal traffics from the application under fuzz/test, their tool automatically infers which attack pattern can be applied and lastly the tool automatically performs the fuzzing/testing to the application. Unlike our tool, the traces their
tool records are used to perform the inference to match the attack patterns and the fuzzings/testings are mainly to confirm the success of the attacks. While our tool records the traces for comparing with the responses of the fuzzing messages and our tool performs the fuzzing to uncover the refined protocol.

Other tools are mainly proxy debugger applications like Burp Suit [11] and Fiddler [2]. Burpsuit is not an open-source software like ours and its main functionality is to record application layer traffics and it allows the user to manually repeat or modify the messages. It can also serves as a web application scanner and intruder. Our tool is not doing the same thing as Burp Suit. Although our tool also needs to record the network traffics and perform fuzzing to web applications, our tool mainly focuses on refining the protocol. Fiddler, similar to Burp Suit, can help developers to test their web applications and it supports manually fuzzing of an protocol. ProRef is built upon Fiddler and provides further functionality – automatically fuzzing to refine the messages.

6 Conclusion
A refined authentication protocol without redundant parameters and cookies in messages is needed for building the formal model of the protocol. To derive a refined version, the security analysts need to manually fuzz the protocol, which is a tedious and time consuming work. We propose ProRef which can help the security analysts to automate the fuzzing process and generate the refined messages in the protocol. This tool can save the researchers and security analysts much time and effort from manually fuzzing protocol. ProRef is open-sourced with the instructions for installation, demo usage and case studies available online [6].

References
Modeling Requirements for
Quantitative Consistency Analysis and
Automatic Test Case Generation

Tom Bienmüller, Tino Teige, Andreas Eggers, and Matthias Stasch
BTC Embedded Systems AG, Gerhard-Stalling-Straße 19, 26135 Oldenburg, Germany
{bienmueller, teige, eggers, stasch}@btc-es.de, http://www.btc-es.de

Abstract. We present improvements of software development processes based on formalized functional requirements. Fundamental basis is a graphical formalism called simplified universal pattern allowing users to model requirements using the intrinsic nature of functional requirements specifying a trigger/action relation. The underlying graphical formalism enables to generate additional benefits to existing processes. In this paper we focus on two techniques: quantitative requirement consistency analysis and automatic test case generation for functional requirements.

1 Introduction

Applying formal methods in software development processes requires formalized temporal functional requirements being available. Even though prominent formalisms such as LTL or CTL [5] exist, enabling non-formal-methods-experts to use them is still a major hurdle. Engineers need to be trained, stakeholders like quality managers demand to understand what has been expressed, and available formal requirements need to be revised after a new iteration cycle of the design has been initiated. Latest when it comes to real software production, non-functional requirements such as readability, understandability, maintainability become very important not only for the design being developed, but also for its formal functional requirement specification. Moreover, the return on investment for formal specifications may be doubted: does it pay off when we spend time and money for establishing the needed skills and changing our running processes?

Making formal methods applicable in production therefore means, first, to bridge the gap between traditional requirements engineering and formalization, and, second, to add value to existing processes which clearly overcomes return on investment doubts.

In this paper, we address both of these key fundamentals: we describe the graphical simplified universal pattern formalism to model functional requirements in a natural, intuitive, and declarative way. Then we put a focus on two benefits: automatic requirement consistency checking and automatic test case generation.

⋆ This work has been supported by the ITEA3 project 14014 ASSUME.
2 Tom Bienmüller, Tino Teige, Andreas Eggers, and Matthias Stasch

Fig. 1. Example of a simplified universal pattern.

generation for functional requirements. We finally present initial experiments to prove the concept of our approach which is implemented in the commercial product suite BTC EmbeddedPlatform® for specification, testing, and verification of requirements for Simulink® and TargetLink® models as well as production code.

2 Modeling Functional Requirements

In the past, several graphical formalization languages have been proposed. Symbolic timing diagrams [10, 14] or life sequence charts (LSCs) [2] are prominent examples. Such formalisms offer tremendous expressiveness but this expressiveness comes along with needed expert knowledge to apply them. Approaches like those presented in [1] (graphical pattern templates) or [3] (textual pattern templates) limit expressiveness to gain better readability, but also benefit from reduced complexity for tableaux generation for formal verification. Though the latter two approaches use terms and notions to better understand the formalism, none of them inherently bases on the nature of an informal functional requirement which impose another hurdle to requirement engineers.

With over 20 years of experience in the field of formal specifications we believe it is mandatory to directly relate the formalism to the intrinsic composition of informal functional requirements. Furthermore, we believe it is important to not overwhelm end users with exorbitant expressiveness, which will also lead to better performance when reusing the artifacts later within formal techniques such as formal verification or consistency analysis. Many of our customers’ functional requirements for system components can be described using a simple trigger/action relation, independent if they are expressing progress, ordering, or invariant requirements. The graphical language simplified universal pattern (SUP) proposed in this paper follows this path of building a formalism on top of the trigger/action relation. Note that approaches like LSCs or textual patterns could either complement or enhance the approach described here. LSCs are well suited to describe interactions between integrated components, while textual patterns are charming as these are easily accessible to humans. Chronologically, SUP is a further development from the graphical pattern templates [1] which have been used in previous BTC-ES’ formal verification tools.

In the next subsection, we briefly recall the fundamental ingredients of that graphical formalism. More details can be found in [13] and [12].

1 Product information can be found at http://www.btc-es.de/ under “Products”.
2.1 Simplified Universal Pattern

By long-standing experience and cooperation with engineers from prestigious automobile and aircraft manufacturers and suppliers, description languages for formalizing natural language requirements should be as intuitive as possible, easy to understand, and preferably presented in a graphical way such that formalization of human-readable to machine-readable requirements becomes a common engineering task without being very prone to errors. The simplified universal pattern (SUP) approach is based on the observation that the vast majority of real-life safety-critical requirements for components can be expressed by temporal trigger/action relationships like in the textual requirement “If the driver up or passenger up switch is pressed then the window has to start moving up in less than 50 ms”. An SUP explicitly introduces artifacts like trigger and action to close the gap between human intuition of a requirement and its formalized description, i.e. artifacts an requirements engineer talks about are directly reflected in the specification formalism, as shown in Fig. 1. We remark that a trigger or an action itself is not limited to be instantaneous but can have a temporal extent.

The semantics of an SUP is defined by runs, i.e. by (finite) executions of the system under test which are observed by an SUP. More precisely, a trigger or action is started by a run $r$ at step $i$ by consuming its start event from $r$ at step $i$ and successfully passed at step $j \geq i$ by accepting its end event at step $j$ within the specified time interval, while its condition must hold in between, i.e. for all steps $k$ with $i < k < j$. A trigger or action fails during processing if its condition became false or its end event was not observed in the time interval. An SUP is fulfilled by a run $r$ if its trigger and action are successfully passed by $r$ and their temporal relation is met. An SUP is violated by a run $r$ if its trigger is successfully passed by $r$ but the action does not start in the specified time interval or the action fails after entering it.

For a small example, consider the SUP from Fig. 1. One possible SUP run is shown in Fig. 2: in step 1 the expression of the trigger condition $\text{Sa1\_driver\_up}$ holds as $\text{Sa1\_driver\_up}$ is true, and thus the trigger is passed. The SUP is then ready to observe the action which happens immediately as $\text{Sa1\_move\_up}$ is also true in step 1. The SUP is fulfilled and waits for a new trigger. The next trigger is consumed in step 3 due to $\text{Sa1\_passenger\_up}$. Since the expression of the action condition $\text{Sa1\_move\_up}$ does not hold in the following 5 steps/50 ms (where one step corresponds to 10 ms), the SUP is violated in step 8.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
<th>Step 0</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
<th>Step 5</th>
<th>Step 6</th>
<th>Step 7</th>
<th>Step 8</th>
<th>Step 9</th>
<th>Step 10</th>
<th>Violated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sa1_driver_up</td>
<td>Input</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sa1_passenger_up</td>
<td>Input</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sa1_move_up</td>
<td>Output</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2. Tabular representation of an SUP run.
2.2 Requirement Coverage

Explicitly revealing the intrinsic artifacts of an informal requirement through a formal SUP enables to define intuitive and accurately measurable coverage metrics for requirements. A commonly used informal coverage notion says that “there shall be a single test case linked to a requirement which verifies it”. Whether the linked test case is actually doing that is not obvious. It requires a human review and confirmation. With coverage notions built on top of the SUP formalism, the above informal coverage metrics becomes clearly defined and measurable: executing the linked test case needs to generate a run which completely traverses, i.e. fulfills, the SUP once, in particular successfully passes the action end event.
Even though we are free to define arbitrary coverage notions based on SUP artifacts, we propose to use metrics first which only refer to the trigger part of a requirement, thus yielding the notion of trigger coverage. Coverage of the action part is left out but could be easily added. The proposed coverage metrics differ in the degree of exhaustion a trigger needs to be covered in order to reach the test exit criterion. Furthermore, we first restrict the coverage to fulfilling runs only, i.e. an SUP needs to be fulfilled by a run to induce coverage at all.

Once coverage (Once) intuitively corresponds to a metrics “one test for each requirement” and is achieved if there exists a run fulfilling the SUP.

The following more sophisticated coverage notions are defined based on a variant of multiple condition coverage (MCC). MCC is defined on all the atomic conditions occurring in an expression to be covered plus all their possible combinations. As the focus for requirement coverage is on fulfilling runs only, we restrict MCC to a subset we call satisfying MCC (sMCC), containing all combinations of conditions for which the overall expression evaluates to true.\footnote{In contrast to MCC, which induces $2^n$ coverage goals for an expression consisting of $n$ atomic boolean conditions, sMCC induces $0 \leq i \leq 2^n$ goals.}

Trigger event coverage (TEC) is based on the sMCC coverage for the boolean expressions of the trigger start and end events of an SUP, while trigger condition coverage (TCC) is focussed on sMCC coverage goals of the trigger condition. Trigger event/condition coverage (TE/CC) combines both TEC and TCC.

For a coverage metric $C \in \{\text{Once}, \text{TEC}, \text{TCC}, \text{TE/CC}\}$, an SUP $S$, and a set $R$ of runs, we define the coverage measure $C(S,R) \in [0,1] \subset \mathbb{R}$ as follows. If $C = \text{Once}$ then $C(S,R) = 1$ if there is a run $r \in R$ that fulfills the SUP $S$, $C(S,R) = 0$ otherwise. If $C \neq \text{Once}$ then $C(S,R) = c/g$ where $c$ is the number of $C$-goals for $S$ covered by runs in $R$ while $g$ is the total number of $C$-goals for $S$. An SUP $S$ is called fully $C$ covered by a set $S$ of runs iff $C(S,R) = 1$.

An illustration of above mentioned coverage metrics is given in Fig. 3. We remark that an analogous definition is conceivable to establish the notion of action coverage and moreover a combined trigger/action coverage.

### 3 Requirement Consistency and Test Case Generation

“Front loading” becomes more and more important. Hence, the quality of requirement specifications is obviously of tremendous relevance: the higher the quality of the requirements, the lower the probability of late iterations due to inconsistencies and incompletenesses of these requirements and their derivate artifacts. With formalized requirements using the SUP formalism and its contained requirement artifacts we can support front loading processes by applying dedicated requirement consistency analysis techniques on top of formalized requirements. Roughly, the set of runs induced by each SUP can be brought into relation to figure out, e.g., that requirements are contradicting. Then, the intersection of runs would be empty. Additionally, as we have detailed knowledge about each of the requirements, we even can give more quantitative information about a common implementation of those requirements. Though there might be
runs that fulfill all requirements, parts of the requirements might be contradicting. This could be made visible based on the coverage notions from Sec. 2.2.

3.1 Quantitative Requirement Consistency

In the literature, topics like requirements consistency, completeness and correctness are quite rarely addressed in a formal sense. Survey papers like [4] show different informal but intuitive interpretations of these terms in different domains. In context of formal specifications the term inconsistency frequently refers to the fact that requirements are in conflict s.t. no valid system run exists, cf. [6]. The authors of [7] go some steps further and also take into account “whether timing bounds of real-time requirements may be in conflict” leading to the notion of rt-inconsistency. As requirements are often modeled by means of pre- and post-conditions as in SUP, the term of vacuity [8] even considers conflicting pre-conditions. In the following, we propose a new consistency notion called basic SUP consistency combining the ideas of consistency and non-vacuity but further incorporates a quantitative measure, namely by means of requirement coverage from Sec. 2.2.

Let be given a set $S = \{S_1, \ldots, S_n\}$ of SUPs. We say that some SUP $S_i$ is basically SUP consistent for a run $r$ wrt the remaining SUPs $S \setminus \{S_i\}$ iff $S_i$ is fulfilled by $r$ and the remaining SUPs $S \setminus \{S_i\}$ are not violated by $r$. We further call $S_i$ basically SUP consistent for a set $R$ of runs wrt $S \setminus \{S_i\}$, a coverage metric $C$, and a coverage threshold $\theta$ iff $S_i$ is basically SUP consistent for each $r \in R$ wrt $S \setminus \{S_i\}$ and the coverage measure $C(S_i, R)$ according to metric $C$ for the SUP $S_i$ wrt the set $R$ meets the coverage threshold $\theta$, i.e. $C(S_i, R) \geq \theta$. We finally define that a set $S$ of SUPs is basically SUP consistent wrt a coverage metric $C$ and a coverage threshold $\theta$ iff for each SUP $S \in S$ there is a set $R$ of runs s.t. $S$ is basically SUP consistent for $R$ wrt $S \setminus \{S\}$, $C$, and $\theta$.

3.2 Automatic Test Case Generation

The SUP coverage definition straightforward leads to automatic generation of test cases for functional formal requirements. Generating a functional test from an SUP reduces to a proof task for a model checker claiming that an SUP can not be traversed completely while taking the coverage metric into account. If a corresponding counter witness exists, then this run fulfills the SUP and therefore can be viewed as a functional test for the corresponding requirement. We remark that within BTC EmbeddedPlatform® we rely on model checkers based on SAT, SMT, and BDDs, cf. [11] and [9].

For this type of automatic test case creation, different interesting application scenarios exist. These scenarios differ in the definition of the “surrounding” of a formal requirement for which test cases shall be generated. Theoretically, we need to have a definition of runs available which describe the behavior of the system under test. An obvious source of runs is the system under test itself: a test generation takes only those runs into account which are induced by its (operational) implementation. The advantage is, that the generated test cases
will be functionally reasonable. The drawback is that test cases are generated from the system which shall be independently tested. Another option is to specify the “surrounding” by a set of formal requirements constraining the set of runs to a reasonable size – giving the advantage that the approach is independent of the system under test and can therefore be initiated in parallel to an implementation process. Here, the drawback is, that one is required to provide a “reasonable amount” of formal requirements in order to obtain the desired functional tests.

4 Example

The main goal of any consistency analysis is to find out early if requirements contradict with each other and hence to avoid that no controller can be built that satisfies them all. In the one extreme, one could look for a tool which checks if requirements contradict each other in every situation. This is probably the easiest analysis that can be performed, but also leads to the weakest consistency notion, since there could still be many situations in which more subtle contradictions exist. The other extreme is a scenario where there would be no contradiction for all situations (i.e. for every input or parameter combination in every temporal order). This analysis and methodology is the most costly, as it requires both the most computation complexity but also requirement refinement effort upon detected inconsistencies.

We therefore strive for an analysis laying between these two extremes. We propose to analyze formalized requirements encoded as SUP by means of basic SUP consistency from Sec. 3, which facilitates a quantitative aspect of requirements consistency. As a nice side effect, this approach generates functional test cases for requirements under consideration as mentioned in Sec. 3.2. For reasons of space, we use the following syntax for SUPs throughout this section:

\[
\begin{align*}
&\text{trigger} \quad \delta_{\text{trigger}} \quad \Delta \quad \text{action} \quad \delta_{\text{action}}
\end{align*}
\]

with \( \text{trigger} \) being the trigger condition, \( \delta_{\text{trigger}} \) the optional trigger duration, \( \text{action} \) the action condition with its optional duration \( \delta_{\text{action}} \), and \( \Delta \) the “local scope”, i.e. the duration between trigger and action. A slightly extended form of the requirement from Fig. 1 is thus written as follows:

\[
\begin{align*}
\text{driver-up} \ || \ \text{passenger-up} \quad \rightarrow \quad \text{move-up, move-down} \quad \text{[0,50] ms, 50 ms}
\end{align*}
\]

We define a second requirement that shall enforce that the window moves down for 50 ms at most 10 ms after an obstacle is detected:

\[
\begin{align*}
\text{detection-obstacle} \quad \rightarrow \quad \text{move-down} \quad \text{[0,10] ms, 50 ms}
\end{align*}
\]
Fig. 4. Results of basic SUP consistency analysis and test case generation.

The goal of our tool-supported analysis is now to check basic SUP consistency of these two requirements to prove whether a correct implementation for them is possible or not. In order to model the environmental effects of the actuators move_up and move_down which might reveal a potential inconsistency of the requirements, we need to encode relevant behavior of the environment. The window moves up when the move_up output is set to true and the end-stop position is not yet reached:

\[
\text{move}_\text{up} \land \left( \text{window} \land \text{position} < 0.4 \right) \rightarrow \left[ 10, 10 \right] \text{ms} \quad \text{where last}(x) \text{ denotes the value of } x \text{ at the last sample point. In our example, the sample time is set to 10 ms. It moves down when the move}_\text{down} \text{ output is set to true while above the bottom:}
\]

\[
\text{move}_\text{down} \land \left( \text{window} \land \text{position} > 0 \right) \rightarrow \left[ 10, 10 \right] \text{ms} \quad \text{where last}(\text{window} \land \text{position}) - 0.05
\]

It does not move at the top, bottom, or when no actuator output is set:

\[
\text{(!move}_\text{up} \land \neg \text{move}_\text{down}) \lor \left( \text{move}_\text{down} \land \left( \text{window} \land \text{position} \leq 0 \right) \right) \lor \left( \text{move}_\text{up} \land \left( \text{window} \land \text{position} \geq 0.4 \right) \right) \rightarrow \left[ 10, 10 \right] \text{ms} \quad \text{where last}(\text{window} \land \text{position})
\]

4.1 Automatic Consistency Analysis and Test Case Generation

We have implemented an automatic check for basic SUP consistency in combination with automatic test case generation within BTC EmbeddedPlatform®. In our example, we are particularly interested in basic SUP consistency of the SUPs S1 and S2. As the triggers are instantaneous, we chose as coverage metric \( C = \text{TEC}. \)

Note that for instantaneous triggers, the expressions of the start and end events are equal. This implies that the derived coverage goals are the same for both events but are nevertheless reported separately.

3 Note that for instantaneous triggers, the expressions of the start and end events are equal. This implies that the derived coverage goals are the same for both events but are nevertheless reported separately.
4.2 Towards Controller Integration

The above example showed that basic SUP consistency defines a reasonable notion of requirement consistency. One obtains the information that a controller implementation exists for an analyzed set of requirements. Complementary, functional tests measurably covering requirements are automatically generated.

On the other hand, the example also shows limitations of basic SUP consistency. Even though the specified requirements are proved to be basically SUP consistent with 100% requirement coverage for the single requirements, there could be still inconsistencies in which could lead to problems when implementing a controller strategy for those requirements. In the above case, no controller exists which is able to handle some window-up request (driver_up or passenger_up) together with some obstacle detection (detection_obstacle). Please note that this type of inconsistency is an intuitive one and requires human validation — no automatism can judge whether it describes a relevant scenario, i.e. whether a controller implementation needs to deal with the mentioned inconsistent cases. It could be guaranteed by the integration of the controller that detected input inconsistencies can not occur. Here, computed inconsistencies would be irrelevant for a subsequent controller design and could be ignored.

It would be beneficial to have another notion of consistency available which is able to reveal potential inconsistencies based on specific input valuations as mentioned above. From a controller’s perspective, we want to know whether constraints to its integration exist. To enable this, we reuse the notion of basic SUP consistency. The only addition is to derive a dedicated SUP which combines all triggers of the defined functional SUPs in order to express a context-free integration of the controller to be implemented. This additional SUP is called integrity SUP and is defined as follows:

\[
\text{trigger}_1 \parallel \ldots \parallel \text{trigger}_n \quad \Longrightarrow \quad \delta_{\text{action}}
\]

Applying test case generation for this integrity SUP requires to check all sMCC-combinations of all the SUP triggers of interest. If for some combination no run exists then this shows either demands on a controller’s integration or an unwanted inconsistency has been detected. We need to remark that the current definition of an integrity SUP is only applicable if the triggers of the SUPs of interest are instantaneous. Moreover, the time duration \(\delta_{\text{action}}\) need by given which currently is a manual task. In future work, we will thoroughly investigate the notion as well as the automatic derivation of integrity SUPs.

When considering the SUPs S1 and S2 of interest. Then, we can derive the integrity SUP S3:

\[
\text{(passenger_up} \parallel \text{driver_up}) \parallel \text{(detection_obstacle)} \quad \Longrightarrow \quad \frac{1}{100 \text{ ms}} \quad (S3)
\]

The result of the automatic consistency analysis for S3 is given in Fig. 5. It actually turned out that 6 of the 14 coverage goal are unreachable and thus
conflicts in behavior of the requirements are revealed. Due to the fact that the expressions of the trigger start and event event are equal (and thus their coverage goals), there are three conflicting situations remaining, namely exactly these with some window-up request together with some obstacle detection, more precisely (1) passenger\textsubscript{up} \&\& \neg driver\textsubscript{up} \&\& detection\textsubscript{obstacle}, (2) \neg passenger\textsubscript{up} \&\& driver\textsubscript{up} \&\& detection\textsubscript{obstacle}, and (3) passenger\textsubscript{up} \&\& driver\textsubscript{up} \&\& detection\textsubscript{obstacle}. Based on this information, one can see that the simultaneous occurrence of passenger\textsubscript{up} or driver\textsubscript{up} with an obstacle detection causes situations in which no control strategy can satisfy both requirements. Based on this information, some kind of refinement can be performed, e.g. by relaxing the first requirement to enforce window movement only in case no obstacle is detected.

5 Conclusions and Future Work

In this paper we presented improvements of software development processes based on formalized functional requirements. Fundamental basis is a graphical formalism called simplified universal pattern (SUP), which enables users to model requirements using the intrinsic nature of functional requirements specifying a trigger/action relation. The underlying graphical formalism enables to generate additional benefit to existing processes. In this paper we focused on two methods, namely requirement consistency analysis and automatic test case generation for functional requirements, and proved the concept of these techniques by an example. In particular, when it comes to consistency analysis we motivated the necessity of introducing quantitative measures for being able to rate also controller integration demands.

In the current status of the technology we see several extension links in the future. Besides equipping consistency analyses with appropriate debugging facilities, we think about adding further quantitative measurements such as amount of runs fulfilling an SUP. We also need to consider completeness analyses of requirements, giving evidence whether “enough” requirements have been specified. By working closely together with industrial users we will collect the required feedback and needs and will let the tool evolve with respect to these needs.
Acknowledgments

The authors are very grateful to the anonymous reviewers for their helpful comments to improve the quality of the paper.

References

Tool Support for Model-Based Database Design with Event-B

Ahmed Al-Brashdi, Michael Butler, Abdolbaghi Rezazadeh, and Colin Snook

University of Southampton, Southampton, UK
{azabigl4,mjb,ra3,cfs}@ecs.soton.ac.uk

Abstract. UML-B provides a graphical notation for Event-B that enables formal development in a UML style. UB2DB is a tool that translates UML-B models to relational database implementations in SQL. The UB2DB tool is implemented as a plugin for Rodin, an extensible toolkit for Event-B. This paper presents the current version of UB2DB that translates the main components of UML-B class diagrams to SQL code. The generated SQL code defines a database and provides procedures that manipulate it. The UB2DB tool exploits the Eclipse Modeling Framework (EMF) to realise the required model transformation. The current tool provides the basis for a more comprehensive tool that will provide support for a broader range of UML-B features and support a variety of database components and constraints.

Keywords: Event-B, UML-B, Database design, Model-driven design

1 Introduction

The design of database systems is an important field in software engineering, and therefore it requires a verifiable and rigorous design approach. Event-B is a formal method for rigorous specification and verification of digital systems [1]. It is supported by an open platform called Rodin [2]. UML-B is a graphical notation for formal modeling in Event-B that is based on UML [10]. A tool, called iUML-B, is provided which supports building UML-B diagrams in Rodin and is integrated in an Event-B machine or context. The iUML-B tool is based on the Eclipse Modelling Framework (EMF) for Event-B [11]. The Eclipse Modelling Framework (EMF) is a framework for tools development which provides modeling and code generation facilities [12].

The UB2DB (UML-B to Database) tool enables developers to rigorously model their database or database intensive application in UML-B for verification and translate the model to SQL code. SQL is a relational database definition and manipulation language [7]. The UB2DB provides an automatic generation of SQL code from a model defined in UML-B in the Rodin platform. The generated SQL will create the relational database structure using create and alter commands and provide procedures that populate and manipulate the data.

Developers will model a database system in UML-B and verify it in the Rodin platform to detect any inconsistency or ambiguity. Then they will use UB2DB to translate the verified model to SQL code.
When modelling databases in UML-B, different refinement levels might be introduced so that the model of database can be built gradually. In the first abstraction level, an abstract view of the database is modelled and can be translated to SQL. In every refinement level, a more concrete view of the database is modelled which adds more details to the preceding one.

UB2DB translates directly from UML-B and not from Event-B. Since UML-B uses class diagrams, it is more aligned with the database as class diagrams are commonly used to describe databases. Developers will benefit from being able to generate Event-B from UML-B using the existing UML-B tool, allowing them to apply formal analysis to their UML-B models. While the UML-B class diagram clearly adds attributes to each class directly, Event-B lists all variables together without clear distinction of each class's attribute. This makes it easier and more straightforward to translate from UML-B to database than from Event-B to database, and without compromising the model verification.

2 Translation process and approach

To translate from UML-B to a relational database, we designed our own meta-model for relational databases in EMF. Figure 1 shows a part of the defined meta-model. Each database is composed of tables, and each table may have many attributes. Each element has a name associated with it. Along with the name, attributes have types and constraints that specify if an attribute is not null, unique or has a default value.

The first step in UB2DB is to translate the EMF representation of UML-B to the EMF representation of the database such as translating a class in UML-B to a table, or a class diagram to a database. A second step is to generate the textual SQL code from the database EMF representation. For each UML-B model, there are two translations done, one to SQL by UB2DB, and another to Event-B by UML-B as in Figure 2. These two translations are separate from each other.

Fig. 1. Part of defined database meta-model

Each element in UML-B such as class diagram or class has a unique name. Classes may have associations between them that relate each class to another. UB2DB generates the SQL statements that create a database whose name is given by the class diagram name, and generates a table for each class in the
model. The associations between classes are translated to relations between tables. Each class attribute in the UML-B model will result in an attribute in the corresponding table. Each component such as association or attribute is translated into a separate statement in the generated SQL. This way, dealing with refinement will be easier as adding a new attribute for a class in a refined model will correspond to adding only one SQL statement that adds that attribute to the table instead of going through the whole process of creating a table again.

For each attribute in the model, there are some defined properties like total function, injective function and initial value. These properties are translated into constraints in the generated SQL. A class invariant might be added to constrain an attribute value such as $a \in 1..100$. Such an invariant is translated to a constraint in the generated SQL.

If an association between two classes is set as functional, it will be translated to an attribute of one of the tables. If the association is non functional (n:n association), it will be translated to a separate table with references to both classes/tables. An example of that is the association member_pod between Member class and Pod class in Figure 3. The UB2DB tool will generate a new table called member_pod with two references, one to Member table and another to Pod table. The following SQL statement is generated automatically by UB2DB which creates a table with the association name, member_pod. The table has two attributes: member_id and pod_id that reference Member and Pod tables.

```sql
CREATE TABLE member_pod(
    Member_id INT,
    Pod_id INT,
    PRIMARY KEY (Member_id, Pod_id),
    FOREIGN KEY (Member_id) REFERENCES Member(Member_id),
    FOREIGN KEY (Pod_id) REFERENCES Pod(Pod_id)
);
```

UML-B provides three kinds of events; constructor, destructor and normal. A constructor event should be selected for events that aim to create an instance of a class. Destructor is used for the opposite. For other operations, the normal
event is selected where it adds a guard automatically to check that the instance to select or update is an element of that class set. Each constructor event in UML-B is translated by UB2DB into a *procedure* with the *insert into table* statement in SQL. The procedure takes all class attributes and associations as parameters for the insertion. Destructor events are translated into procedure with the *delete from table* statement in SQL. Normal events are translated into procedures with an *update table* statement if the event has an override operator, or to a *select from* statement if the event does not have an action.

UB2DB generates SQL code in which class invariants are translated to constraints in the generated code. Event guards are also maintained by the translation to ensure correct implementations of the UML-B models. Also, the generated code ensures the atomicity of an event by translating all actions to one atomic transaction.

The UB2DB translation is implemented using a generic EMF translation plugin which is provided by University of Southampton. Translation and rules are contributed using the Eclipse extension mechanism. For each component in the database meta-model such as table or attribute, there is a rule defined by a Java class to translate or map UML-B to it. Each rule in UB2DB has *fire* and *dependencyOk* methods. The *fire* method does the mapping between UML-B elements to database elements. In the translation process, dependencies must be checked by the *dependencyOk* method before proceeding to the translation. A table is dependant on a database which means it cannot be generated before the database, and an attribute is dependant on a table. This also ensures an ordering of the translation of different components.

### 3 Case study and evaluation

Two cases were built to study various components and relations of database systems and to help identify good practice in modelling databases in Event-B with levels of refinements. The first case study is a student enrollment and registration system while the other is a car sharing system. After having these two cases modelled in UML-B, we ran UB2DB on different abstraction levels to generate the database for them.

Starting from an abstract model, as in Figure 3 for the car sharing case study, where classes have associations but no attributes, the tool generates the tables with one attribute as a key for each table. Then the associations are added to the source tables as attributes that references the target table. As the same classes will appear in another refinement level, the *create table if not exists* command will create a table only if it does not already exist.

Further refinement of the model might include adding attributes to different classes as in Figure 4 for the student enrollment case study. Another refinement could add more detail to the model by introducing new classes and associate them to classes in the abstract model such as the *Booking* class for car sharing in Figure 5. Attributes and relations added in later refinements are translated to the *alter table* command so that we can build on the previous generated database.
without rebuilding it. The *alter* command can be used to modify the structure of a table by adding, modifying or deleting attributes or relations.

Fig. 3. Abstract model for car sharing case study

Fig. 4. Refinement by adding attributes in Program and Student classes

Fig. 5. New classes in refinement model

The generated SQL code was successfully imported in the database management system and all the supported database structures and constraints were successfully generated. This includes generating intermediate tables and assigning different constraints such as primary key, foreign key, not null, uniqueness, default value and basic check constraints. Events were translated to procedures for constructors, destructor or normal events. For any constructor event, the tool generated a procedure with a name as the event name and took the class
attributes and associations as parameters for the procedure. Destructor events were translated to procedures with one parameter corresponding to a key for the record to be deleted. Normal events were translated to either update or select procedures. However, UB2DB does not yet deal with complex queries in normal events in UML-B.

4 Related Work

Much work has been done in the area of formalizing databases or translating formal methods to database applications. Barros in [4] translates Z notation to relational databases with support for different operations and transactions. In [8], Khalafinejad and Mirian-Hosseinabadi present a method for translating Z notation to SQL and the Delphi programming language with no tool implementation.

Laleau and Mammar in [9] present a tool that refines a UML specification into a B model and then to a database application. While their work is close to ours, they do not translate to update and read operations or deal with transaction management. Our work will provide extra features than theirs such as normalizations, design patterns, database security and translation to database views.

Davies et al. in [6] use a model-driven approach to automatically generate object-oriented databases with an extended version of B method and Object Constraint Language [14]. We are interested mainly in the relational model of database design.

Wang and Wahls in [13] developed a Rodin plug-in that translates Event-B to Java and JDBC code to create and query a database. While, to the best of our knowledge, this is the only work that translates Event-B to database applications, it has some limitations. The results in [5] identify major performance issues as well as the issues with preserving database integrity as in [3].

5 Conclusion and future work

UB2DB is a tool that translate UML-B models to relational databases by generating SQL statements that build the database and structure its tables and relations. The UML-B model is translated by the UML-B tool to Event-B for verification. UB2DB provides support to translate different components in UML-B model into code that can be easily imported in MySQL database and reserves the constraints such as not null and unique. It also provides support for events that create new instances of classes, delete an existing one, update its attributes or select from one or more classes.

In future, full support for events will be provided which might translate one event in UML-B to different statements in one procedures such as delete and insert when moving a record from one class to another. As the current implementation of the tool translate an abstract level without looking to preceding
abstraction level, the future plan is to make the tool look for the the specification of a model and all of its preceding abstractions. The tool will extend the support for class invariants. The future work includes looking at preserving normalization when modelling in UML-B and defining design patterns for database systems and supporting them by our tool.

References

PAndA²: Analyzing Permission Use and Interplay in Android Apps (Tool Paper)*

Marie-Christine Jakobs, Felix Pauck and Manuel Töws
Paderborn University, Germany
{marie.christine.jakobs,manuel.toews}@upb.de

Abstract. We present PAndA², an extendable, static analysis tool for Android apps which examines permission related security threats like overprivilege, existence of permission redelegation and permission flows. PAndA² comes along with a textual and graphical visualization of the analysis result and even supports the comparison of analysis results for different android app versions.

1 Introduction

Android uses permissions to protect access to sensitive data. To gain access to sensitive data, e.g., via API calls, apps must declare in the manifest which permissions they want to use. During install time the user then grants the declared permissions to the app. For a safe and secure app execution, the app’s permission declarations should follow the least privilege principle. On the one hand, for each permission protected access the required permissions must be declared to avoid abnormal termination. On the other hand, an overprivileged app declaring permissions which are not required can e.g. be exploited by malware. While some apps already do not adhere to least privilege [6], least privilege is not enough. Although not declaring the permission, apps may get access to sensitive data via redelegation [15]. Moreover, the set of used permissions may not be precise enough to decide if an app does not leak sensitive data. Thus, one also requires knowledge about information flows between permission protected statements.

Existing static analysis tools typically examine one of these three security aspects while our tool provides individual analyses for all three aspects. Tools like Stowaway [9], PermissionCheckTool [12], COPES [4], and PermitMe [5] examine least privilege by identification of the required permissions of an app and comparison with the declared ones. Thus, permission are classified into three categories: REQUIRED, UNUSED, and MISSING. We think that such a classification is not sufficient. Permission requirement must always be decidable and the classification does not describe access via redelegation. First, we add two further classifications MAYBE_MISSING and MAYBE_REQUIRED. Furthermore, we extend the classification with an access type, direct, indirect, or both if we

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901).
look at inter-app permission usage. To the best of our knowledge, we are the first who examine for an app and a given app set which permission are used indirectly, i.e., via redelegation. So far, it is studied if apps are vulnerable to or indeed execute a permission protected statement for another app, see e.g. [15,11]. Many tools, e.g. [14,8,2,13,9], exist which analyze information flow. Often, they consider predefined or configurable sets of sources and sinks, sometimes a subset of permission protected statements. Only BlueSeal [8], which computes flow permissions, considers permissions as sources and sinks. Unlike us, BlueSeal groups permissions into permission domains.

In contrast to many of the mentioned tools, our tool also provides a graphical representation of the result. To support the developer in its learning process, but do not bother an (inexperienced) app user with too many details, the user can select a more detailed view on the result or filter the result for certain aspects. To better grasp the impact of a change/update of an app our tool can load a previous analysis result and compare it with the analysis result of an updated version of the app.

2 PAndA$^2$

PAndA$^2$ (Paderborn Android App Analysis) is a static analysis framework for android apps provided as apk files. It is built on top of the SOOT analysis framework [10] which we use to built the intermediate representation of an app. Currently, it supports three security analyses, see Section 2.1 for more details.

Architecture. Figure 1 shows the overall architecture of PAndA$^2$. Generally, it consists of three parts reflecting the three phases each analysis runs through. In the first phase, the enhancement phase, SOOT is used to get the intermediate

---

Fig. 1. PAndA$^2$ architecture overview

---

1 Available at [http://pga3.foellix.de/?cat=downloads](http://pga3.foellix.de/?cat=downloads)
Fig. 2. The graphical analysis result of an inter-app permission usage analysis is displayed in comparison to a previous result.

...
result to certain aspects of interest, e.g., a subset of the permission groups or a subset of permissions like the subset of dangerous permissions. Figure 2 shows the graphical analysis result of an inter-app permission usage analysis with detail level COMPONENT but excluding permission group MAYBE_MISSING.

**Result Reuse.** A saved analysis result can easily be reused later, e.g. it can be redisplayed without rerunning the analysis or compared with a more recent result (for an updated version of the same app). Figure 2 shows the screenshot of a comparison which aims at illustrating the difference between the two results and, thus, highlighting what has changed. To support reuse of results, PAndA^2 can load and save analysis results. We use Kryo [1], an alternative serialization API for Java, to save and load our analysis result. A loaded analysis result can be processed as if obtained after analysis, e.g., it can still be filtered.

### 2.1 Available Analyses Plug-Ins

PAndA^2 is shipped with three different security analyses. One of these analyses, is a typical flow-sensitive taint analysis restricted to the analysis of a single app. Our taint analysis follows the approach of FlowDroid [2] to compute a single entry for the taint analysis and uses the approach of Hammer [7] to detect the tainted flows. In contrast to many existing taint analyses [14,2,13], our taint analysis considers all statements protected by permissions to be potential sources and sinks.

The other two analyses examine the permission usage in a flow-insensitive manner. One only considers the permission usage within the app itself, being able to examine least privilege. Assisted by such a permission usage analysis a developer can easily adapt its permission declaration s.t. it adheres to the least privilege principle. The other also analyzes permission usage of an app across app boundaries within a given app set, being able to detect redelegation threats.

To describe the permission usage of an app, both analyses group permissions declared or used by the analyzed app into categories. Next to the three standard categories REQUIRED (declared and used), UNUSED (declared, but never used) and MISSING (not declared, but used), these analyses consider two further categories MAYBE_REQUIRED and MAYBE_MISSING. A permission is added to one of these categories if it is neither added to REQUIRED nor MISSING, it is declared, not declared, respectively, and the app contains an intent which may target another app with unknown permission usage or the app specifies an intent filter, i.e., the app can be called by another app. We consider the latter case because an app should ensure that it has at least all permissions of the calling app to exclude redelegation. For permission usage across app boundaries, we use an even more fine granular categorization. The standard categorization classes are extended by a tag, DIRECT, INDIRECT, or DIRECT & INDIRECT, describing if a permission is only relevant within the analyzed app itself, if the permission is indirectly accessed through another app in the environmental app set in the sense of redelegation, or both.
Table 1. Real-world app results for permission usage analysis

<table>
<thead>
<tr>
<th>App</th>
<th>REQUIRED</th>
<th>MAYBE</th>
<th>UNUSED</th>
<th>MAYBE</th>
<th>MISSING</th>
<th>MISSING</th>
<th>filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe Acrobat Reader</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>170</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Barcode Scanner</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>162</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ES File Explorer</td>
<td>10</td>
<td>9</td>
<td>0</td>
<td>151</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Google Photos</td>
<td>15</td>
<td>5</td>
<td>0</td>
<td>143</td>
<td>12</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Instagram</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>155</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Tiny Flashlight</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>162</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>WhatsApp Messenger</td>
<td>25</td>
<td>7</td>
<td>0</td>
<td>140</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

2.2 Limitations

Of course PAndA\textsuperscript{2} suffers the same limitations any static analysis has. Furthermore, we currently do not support user-defined permissions and ignore native code of third parties. Moreover, we assume that components and especially apps exchange information via intent-based communication. Exchange via shared resources, memory or files is currently not taken into account. Additionally, we cannot handle all types of intent creation, e.g., an analysis may fail to identify the possible target of an intent. However, our tool will show a message to the user if intents cannot be handled properly.

3 Evaluation

We tested PAndA\textsuperscript{2}’s functionality, including all three security analyses, with a set of manually created apps that are delivered with PAndA\textsuperscript{2}. To study the practical usability, we applied PAndA\textsuperscript{2}’s analyses to a set of real-world apps from the most downloaded apps in Google Play\textsuperscript{2}. The goal of our selection is to cover a large variety of apps with different purposes. We did not compare our tool with the previously mentioned tools Stowaway, PermissionCheckTool, and COPES because they are outdated and only support Android API levels below 22 (the level we consider for our tool). Furthermore, a comparison with PScout, which we use to detect permission usage, is not very meaningful. During evaluation we recognized that the taint analysis still suffers from some stability and performance problems. Thus, in the following we only present some preliminary results for the permission usage analysis which was executed on an Intel i7 2600 @ 3.4 GHz with 8 GB of RAM. Typically the analysis only took several seconds except for WhatsApp, the largest app, in which our analysis suffered from the huge memory consumption. Table I shows the detected numbers of permissions per permission group. Note that on purpose we do not present precision and recall metrics since it would be very laborious to manually infer the correct permission usage from the respective Dalvik executable.

First of all, we observe that none of the analyzed apps is overprivileged (column UNUSED always 0). However, for all apps we found missing permissions.

\textsuperscript{2}https://play.google.com
Some of these permissions have protection level normal and need not be declared. The last column shows the number of missing permissions excluding those with protection level normal. Some missing permissions often remain. However, this does not mean that the app is unsafe. If certain statements are called in a specific context no permissions may be required, but PScout did not distinguish between calling contexts. Furthermore, all apps are calling other apps. Since the permission usage analysis does not know which apps are called and which permissions are required to call them, the analysis always outputs a high number of permissions in the MAYBE_MISSING category.


References

A Formal Approach to Use Case Driven Testing

Rajiv Murali, Andrew Ireland, and Gudmund Grov
School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh, UK

Abstract. Traditionally, use case specifications are documented informally outside the UML model, often in a text document, with no structure or traceability to other UML modelling elements, e.g. actors, subject, dependency relationships, etc. This often results in inconsistencies between the UML model and the use case specification, and introduces a barrier to more automated methods for analysis. In this paper, we describe an approach that extends the UML model to enable use cases to contain a structured use case specification. This extension is then used to support methods to: (1) generate activity diagrams to visualise the behaviour of the use cases; and (2) include our previous work on formalising the use case specification with Event-B to support the generation of test cases. An example of an Anti-lock Braking System (ABS) is used to describe our approach. An implementation of this approach is provided via a plug-in, UseCasePro, for the UML modelling tool Papyrus.

1 Introduction

UML use cases [1] are a popular technique used to define and communicate the behavioural requirements for software-intensive systems. They appear in two complementary forms: (1) a use case diagram that provides an easy to understand illustration of use case modelling elements, i.e. use cases, actors, dependency relationships, etc.; and (2) an accompanying textual use case specification that details the behaviour and constraints for each use case. Unlike the use case diagram that have their notations and semantics defined in [2], the use case specification follow a loosely defined template [3] and is documented informally outside the UML model. This limits the use case specification to only a review-based analysis and permits inconsistencies between the use case specifications and the UML model.

In our previous work [4], we combined the informal use case specification with formal counter-parts to support an encoding to Event-B [5] that enabled an early access to formal analysis tools. In this paper, we extend the UML model to allow use cases to contain this use case specification, as seen in Figure 1 which aims to not only capture informal and formal counter-parts to support the automatic generation of Event-B, but also explicitly capture the relationships between the use case and other UML modelling elements, e.g. actors who play a role in the use case, dependencies to other use cases, etc. These relationships are used to support methods to automatically generate UML activity diagram to visualise the behaviour of the use case. Furthermore, the generated Event-B model for the use case is subject to the model checker ProB [6] that can be used to systematically identify traces, i.e. execution through the use case’s behaviour. These traces along with the structured use case specification are used to generate test cases represented via activity diagrams back in the UML model.
This paper is structured as follows: Section 2 provides background on UML use cases and Event-B; Section 3 describes the extension to the UML model to include a structured use case specification; Section 4 describes the automated analysis methods for use cases, i.e. visualisation and test case generation; Sections 5 and 6 describe the tool implementation of the approach and related work; finally, Section 7 describes the future work.

2 Background

2.1 UML Use Cases

Use Case Diagram. A use case diagram for an Anti-lock Braking System (ABS) is seen in Figure 2. It illustrates three key concepts of UML use cases: subject, actors and usecases. The subject represents the system under consideration, i.e. the ABS. An actor specifies a role played by an external entity that interacts with the subject, e.g. Driver. A use case specifies functionality performed by the subject in collaboration with one or more actors in order to yield some observable result that is of some value to those actors (or other stakeholders), e.g. PowerOn. The «extend» relationship allows for behaviour, e.g. PreventSkid, be added, conditionally, to the behaviour defined in one or more use cases, e.g. MonitorBraking. The extension takes place at one or more specific extension-points, e.g. ImminentWheelLockUp defined in the extended use case. The «include» relationship is used when there are common parts of the behaviour of two or more use cases, i.e. SystemTest.
Use Case Specification. Each use case can be further detailed in a use case specification [3] that capture a contract and a collection of flows. The use case specifications for use cases PowerOn and SystemTest are seen in Figure 3. The contract specifies pre-conditions, post-conditions and invariants that express statements that must hold before, during and after the execution of the use case, respectively. Each flow denotes a distinct sequence of steps that achieves the given post-conditions. Flows are organised into the basic (or “ideal”) flow and alternate flows that represent variations on the basic flow. In order to reduce the number of alternate flows, the basic flow can use simple branching [7] using conditional or loop statements, e.g. if, else if, else and while.

<table>
<thead>
<tr>
<th>PowerOn</th>
<th>Intent of use case is to safely power on ABS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contract</td>
<td>The pre-condition requires the ABS to be powered Off and not locked for technician. The post-condition requires the ABS to either be powered On or remain powered Off and be locked for technician in the event of a system failure being detected.</td>
</tr>
<tr>
<td>Basic Flow</td>
<td>Driver provides power to the vehicle. System test is performed. If system test reports pass (i.e. no component failures detected), then the ABS is powered On, else the ABS is locked for technician and the display switches on the ABS fail light.</td>
</tr>
<tr>
<td>SystemTest</td>
<td>Intent of use case is to check vehicle health.</td>
</tr>
<tr>
<td>Basic Flow</td>
<td>Health monitor performs a system test.</td>
</tr>
</tbody>
</table>

Fig. 3: Use case specifications for PowerOn and SystemTest.

2.2 Event-B

Event-B is a formal notation for modelling discrete systems. Event-B specifications are built using two elements: context and machine. A context defines the static aspects of an Event-B specification, i.e. sets, constants and axioms. The machine is the dynamic part that describes a state of variables that are first initialized. Then events can be executed to modify the state. An event, named evt, can be specified in the following form:

```
event evt = when P(v) then S(v) end
```

Here, the variables of the machine containing the event are denoted by v. P(v) is a predicate denoting the guards, and S(v) denotes the action that updates some variables. An action consists of a collection of assignments that modify the state simultaneously. An assignment has one of the following three forms:

- `x := E(v)` (deterministic assignment)
- `x := E(v)` (non-deterministic assignment from a set of values)
- `x := Q(v, x')` (non-deterministic assignment using a predicate)

Here, x are some variables, E(v) denotes an expression, and Q(v, x') a predicate. Refinement is used to represent the system in different abstraction levels and mathematical proofs are used to verify consistency between them. Event-B modelling is supported by an effective toolset Rodin [8]. The toolset also contains the model checker ProB [9] that can be used to systematically check an Event-B specification for a wide range of errors.
3 Structured Use Case Specification

UML profiles \[1\] provide a generic mechanism for extending the UML meta-model by defining custom stereotypes. In UML, a stereotype is a class and hence can have properties that represent structural features, e.g., attributes of a classifier. A profile uses a stereotype to create a new UML model element derived from existing ones (e.g., a use case) appended with properties (e.g., contract and flows) that are suitable for a specialised usage. In Figure 4 we introduce a profile UC-Specification that provides an extension to the UML meta-model by extending the UseCase meta-class to have a stereotype Specification. This stereotype Specification includes properties that allow it to contain a contract, a basic flow and a collection of alternate flows that reflects the template as discussed in Section 2.1. The contract is allowed to contain a collection of preconditions, postconditions and invariants, while the basic flow and the alternate flows are allowed to capture a collections of steps. The application of this stereotype to the use cases PowerOn and SystemTest from Figure 2 allows it to contain the specifications as seen in Figure 5a and 5b respectively, within the UML model.

Fig. 4: Profile UC-Specification that extends the UML meta-model.

The classes in the profile, e.g., Invariant, inherit abstract classes\[2\] that enable them to specify properties such as a label, an informal description, a formal predicate or an action statement. In our profile, this abstract basis enables the pre-, post-conditions and invariants to specify a label, an informal description and a predicate. For example, in Figure 5a the contract associated with the

\[1\] An extension in the UML profile is denoted by a line with a black arrow head.

\[2\] Abstract basis of profile: https://rajivmurali.github.io/UsecasePro/profile
PowerOn use case specification includes a pre-condition that specifies a unique label P_Pre_1, an informal description of the constraint where the ABS must be powered off, and a predicate abs_power = Off. The label ensures traceability when transforming the specification into other modelling elements (see Section 4) and the informal description preserves the ease of communication that is expected in use cases. The predicate is defined using the data model associated with the subject containing the use cases, i.e. the sets, elements, constants and variables. The profile UCSpecification allows the subject, e.g. the Anti-lock Braking System, to have the stereotype DataModel that enables it to capture a collection of sets, constants and variables, as seen in Figure 5c. A set can contain a collection of elements to denote an enumerated set, e.g. SWITCH is enumerated with the elements On and Off, as shown in Figure 5c.

<table>
<thead>
<tr>
<th>Contract</th>
<th>Label</th>
<th>Informal</th>
<th>Formal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-condition</td>
<td>P_Pre_1</td>
<td>ABS is powered off.</td>
<td>abs_power = Off</td>
</tr>
<tr>
<td>Pre-condition</td>
<td>P_Pre_2</td>
<td>ABS is not locked for technician.</td>
<td>abs_lock = Off</td>
</tr>
<tr>
<td>Post-condition</td>
<td>P_Post_2</td>
<td>ABS is powered on and the lock remains off or the ABS has been locked for technician and the power remains off.</td>
<td>(abs_power = On ∧ abs_lock = Off) ∨ (abs_lock = On ∧ abs_power = Off)</td>
</tr>
<tr>
<td>Invariant</td>
<td>P_Inv_1</td>
<td>ABS must never be powered On while being locked for technician.</td>
<td>¬(abs_power = On ∧ abs_lock = On)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic Flow</th>
<th>Label</th>
<th>Informal</th>
<th>Formal</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action P_1</td>
<td>Driver provides power to vehicle.</td>
<td>vehicle_power := On</td>
<td>Driver</td>
<td></td>
</tr>
<tr>
<td>Include Ref. P_2</td>
<td>«Include: SystemTest»</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If P_3</td>
<td>If system test reports safe then</td>
<td>systemtest = Pass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action P_3_1</td>
<td>ABS is powered on.</td>
<td>abs_power := On</td>
<td>ABS</td>
<td></td>
</tr>
<tr>
<td>Else P_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action P_4_1</td>
<td>ABS is locked for technician.</td>
<td>abs_lock := On</td>
<td>ABS</td>
<td></td>
</tr>
<tr>
<td>Action P_4_2</td>
<td>Display turns on ABS fail light.</td>
<td>display_absfaillight := On</td>
<td>Display</td>
<td></td>
</tr>
</tbody>
</table>

(a) Specification stereotype applied on use case PowerOn.

<table>
<thead>
<tr>
<th>Basic Flow</th>
<th>Label</th>
<th>Informal</th>
<th>Formal</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action ST_1</td>
<td>Health monitor performs system test.</td>
<td>systemtest ∈ {Pass, Fail}</td>
<td>Health Monitor</td>
<td></td>
</tr>
</tbody>
</table>

(b) Specification monitor stereotype applied on use case SystemTest.

<table>
<thead>
<tr>
<th>Data Model</th>
<th>Identifier</th>
<th>Elements/Type</th>
<th>Initialisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set SWITCH</td>
<td>(Elements) On, Off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set TEST</td>
<td>(Elements) Pass, Fail</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable abs_power</td>
<td>abs_power ∈ SWITCH</td>
<td>abs_power := Off</td>
<td></td>
</tr>
<tr>
<td>Variable abs_lock</td>
<td>abs_lock ∈ SWITCH</td>
<td>abs_lock := Off</td>
<td></td>
</tr>
<tr>
<td>Variable abs_systemtest</td>
<td>systemtest ∈ TEST</td>
<td>systemtest := Pass</td>
<td></td>
</tr>
<tr>
<td>Variable abs_faillight</td>
<td>display_absfaillight ∈ SWITCH</td>
<td>display_absfaillight := Off</td>
<td></td>
</tr>
<tr>
<td>Variable vehicle_power</td>
<td>vehicle_power ∈ SWITCH</td>
<td>vehicle_power := Off</td>
<td></td>
</tr>
</tbody>
</table>

(c) DataModel stereotype applied on subject Anti-Lock Braking System.

Fig. 5: UCSpecification profile applied on UML model of ABS.
The elements in the data model inherit the abstract basis that enable them to specify the following properties: sets and elements can specify an identifier and an informal description; constants specify an identifier, a type (predicate) and an informal description; and finally, a variable can specify an identifier, a type (predicate) and an initialisation (action).

The specification can contain one basic flow and a collection of alternate flows, each of which can contains a collection of steps. The profile enables the alternate flow to specify the properties deviate and rejoin. These are used to denote steps in the basic flow that the alternate flow deviates from and rejoins to, respectively. Note that if a rejoin step is not specified then it is expected that the alternate flow leads to the end of the use case. A full range of potential steps are described as follows:

**Action** an action is associated with a label and can have two parts; an informal description along with an optional formal representation. For example, the first step in the basic flow of the use case specification of PowerOn has a label \( P_1 \), an informal description of the action where the driver provides power to vehicle, and a corresponding formal assignment \( \text{vehicle\_power} := \text{On} \). The profile also allows the action to capture a reference to an actor in the UML model that plays a role in performing the action (e.g. Driver). If no role is specified, it is assumed that the subject associated to use case performs the action.

**Branch** is used to represent either an if, else if, else or while construct in the use case specification’s flow. It specifies a label (e.g. \( P_3 \)), informal description of a conditional statement (e.g. if system test has passed), a formal counterpart via a predicate (e.g. \( \text{systemtest} = \text{Pass} \)) and a collection of sub-steps (e.g. step \( P_3 \_1 \)). During the execution of flow, if the conditional statement is true, then the sub-steps of the branch are executed.

**Include Reference** allows the includes relationship defined in the use case diagram to be taken into account in the use case specification. That is, it specifies a step in the use case flow where the behaviour of the includes use case should be introduced. For example, the use case PowerOn includes SystemTest as seen in Figure 2. In the use case specification of PowerOn, step \( P_2 \) refers to the SystemTest use case via an includes relation.

**Extension Point Reference** a use case can contain a collection of extension-points that are targeted by extending use cases via the extends relationship. This step allows the location of the extension-point to be specified as a step in the use case specification’s flow. This is similar to the includes use case where the behaviour of the extending use case is introduced at this location in the flow. However, the extension-points are optional. That is, the step also captures a conditional statement that must be true in order for the behaviour of the extended use case to be executed.

The profile *UCSpecification* not only provides a structured specification for a use case but also provides a bridge between the conventional informal behavioural description an a formal counter-part.
4 Automated Analysis Methods

4.1 Visualising Use Cases

Activity diagrams provide an alternative to visualising the behaviour associated with a use case [7]. Within the activity diagram, the steps of the use case specification map onto action notes which are partitioned by the actor or subject that plays a role in it. Our extension of the use case specification to the UML model captures important associations between use case and other UML modelling elements that enables the automatic generation of an activity diagram. Figure 6 provides the activity diagram produced from the use case specification of PowerOn from Figure 5a. Labels are used consistently to ensure traceability between them. The following describes the relationship between the notation of the activity diagram and the use case specification:

**Activity Partition** represent the subject and actors that play a role in the action steps of the use case specification, e.g. Driver, ABS, etc.

**Initial Node** denotes the start of the use case’s flow where the constraints of the pre-conditions, e.g. P_Pre_1 and P_Pre_2, must hold.

**Action Node** represents an action step, e.g. P_1, from the use case specification.

**Decision and Merge Nodes** represent branching introduced by if, elseif, else, while steps, and alternate flows from the use case specification.

**Control Flows** are introduced between nodes to correspond to the sequence of steps in the use case specification.

**Final Node** represents the end of the use case’s flow where the constraint of the post-condition, e.g. P_Post_1, must hold.

Activity diagram explicitly illustrate branching in the use case specification. This is later used in the analysis to ensure that the test cases produced in Section 4.2 covers all possible paths of the use case.

![Activity diagram from use case specification of PowerOn from Figure 5a](image-url)

---

**Activity: PowerOn**

- **Driver**
  - Action Node: P_1: Driver provides power to vehicle.

- **Health Monitor**
  - Initial Node: ST_1: Health Monitor performs system test.

- **ABS**
  - Decision Node: P_4_Else: Display switches On ABS fail light.
  - Action Node: P_3_If: if system test is Pass then
  - P_3_1: ABS is powered On.
  - P_4_1: ABS is locked for technician.
  - P_4_2: Display switches On ABS fail light.

**Constraint**
- Pre-condition: (P_Pre_1) ABS is powered Off and not locked for technician
- Post-condition: (P_Post_1) ABS is powered On and not locked for technician or ABS is powered Off and locked for technician

**Activity Partition**
- Activity: PowerOn
- Display

---

Fig. 6: Activity diagram from use case specification of PowerOn from Figure 5a.
4.2 Generating Test Cases in UML

As previously mentioned, a use case specification detailed with both informal and formal counter-parts can be encoded in Event-B. Figure 7 illustrates a snippet of events from the concrete machine of the Event-B model produced for the use case specification of PowerOn from Figure 5a. These events model the basic flow in the use case specification. The encoding introduces an auxiliary control flow variable, $P\_flow$, to mediate the execution of these events to correspond to the sequence of steps in the use case’s flow, i.e. event $ST\_1$ is executed after $P\_1$ due to includes dependency SystemTest at step $P\_2$. The highlighted formulas indicate notations derived directly from the use case specification.

![Fig. 7: Snippet of events from final machine of Event-B model of PowerOn.](image)

Given an Event-B model corresponding to the use case, the Rodin provers can then be used in order to verify the flows against the given invariants. As discussed in Section 2.2, Event-B is also supported by the model checker ProB. This can be used to simulate the execution of events, i.e. steps in the use case specification, to search for invariant violations.

ProB can also automatically find sequences of events, i.e. traces, that lead to invariant violations. However, provided that there are no invariant violations this exhaustive check can be used to generate all possible execution paths in the use case’s flow (traces). Figure 8 provides the state space produced by ProB for the concrete machine of PowerOn. Each edge in the diagram represents an execution of an event in the machine and node represents a reachable state (containing values of all variables in that machine) of the Event-B model. INITIALSETION is a default event in the machine that initialises the variables to those provided in the data model from Figure 5c. This

![Fig. 8: ProB state space of PowerOn.](image)

---

3 PowerOn in Event-B: [https://rajivmurali.github.io/UsecasePro/ex/P0n.pdf](https://rajivmurali.github.io/UsecasePro/ex/P0n.pdf)
initialises to a state where the pre-conditions of the use case **PowerOn** hold. This state space produced by ProB contains the following two traces, T1 and T2:

$$T1: \text{INITIALISATION}; P_{-1}; ST_{-1}; P_{-3}_\text{If}; P_{-3}_1$$

$$T2: \text{INITIALISATION}; P_{-1}; ST_{-1}; P_{-4}_\text{Else}; P_{-4}_1; P_{-4}_2$$

The traces T1 and T2 are mapped to UML activities that represent test cases named, **PowerOn_TestCase_1** and **PowerOn_TestCase_2**, as seen in Figure 9, respectively. These test cases in relation to their traces are described as follows:

**Initial Node** represents the start of the test case. The constraint on it captures the values of the variables, e.g. \textit{abs\_power} and \textit{abs\_lock}, that are associated with the pre-conditions, e.g. \textit{P\_Pre\_1} and \textit{P\_Pre\_2}. Their values are derived from the state after the execution of the \textit{INITIALISATION} event.

**Action Node** represents a step in the test case. Each event execution in the trace that corresponds to an action step in the use case specifications, e.g. \textit{P\_1}, is introduced as an action node. These nodes capture the state of the variable, e.g. \textit{vehicle\_power = On}, modified by the action step.

**Final Node** represents the end of the test case. The variables, e.g. \textit{abs\_power} and \textit{abs\_lock}, associated with the post-condition, e.g. \textit{P\_Post\_1}, and their values found in the state at the end of the trace are introduced as constraints on the final node.

**Control Flows** are assigned in the order in which the nodes are introduced. This reflects the sequence of event executions in the trace.

Fig. 9: Test cases for use case **PowerOn** in the UML model.
5  Tool Development

Our approach is implemented via a plug-in, UsecasePro\textsuperscript{4}, for the eclipse-based UML modelling tool Papyrus\textsuperscript{10}. UsecasePro supports the automatic generation of the UML profile, UCSpecification (as discussed in Section 3), and its application to the UML model of a Papyrus project. Each use case and subject in the UML model is allowed to contain the stereotype Specification and DataModel, respectively. UsecasePro is a successor of UC-B\textsuperscript{9} by providing an editor to author and manage the contract and flows of the use case specification, as seen in Figure 10. The plug-in supports the automatic generation of activity diagrams from a specified use case in order to visualise its behaviour. It also supports the generation of an Event-B model from a formally specified use case. Traces generated by ProB on the Event-B model, via a DOT file, can be loaded into the plug-in to generate test cases represented as activities in UML model.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig10.png}
\caption{UML modelling tool Papyrus with UsecasePro plug-in.}
\end{figure}

6  Related Work

Whittle\textsuperscript{11} presents a precise way of specifying use cases based on a three-level modelling paradigm strongly influenced by UML. He provides a formal syntax and semantics of use case charts. Whittle’s use case charts are precisely and unambiguously defined, and can therefore be executed. Savic et al.\textsuperscript{12} have proposed SilabReq a domain specific language for use cases specifications. SilabReq helps to describe the use cases in clear and precise way through their custom meta-model. They support the visualisation of use cases in state machine diagrams. Sendall and Strohmeier\textsuperscript{13} described an approach that supplements use case with operation schemas. An operation schema is a declarative specification of a system operation written in OCL\textsuperscript{7}. In comparison to these works, our approach extends the standard UML model\textsuperscript{2}, via a profile, allowing use cases to contain a familiar template of a use case specification that can be detailed with both informal and formal counter-parts. The encoding to Event-B provides an early access to formal analysis tools that supports test case generation.

\textsuperscript{4} Demos of UsecasePro: \url{https://rajivmurali.github.io/UsecasePro/}
7 Summary and Future Work

In this paper, we described an extension to the UML model allowing use cases to contain a structured specification. We illustrated this using an ABS example. This specification elaborated the relationships between the use case and other UML modelling elements as well as consider a more rigorous approach to its specification via Event-B. Automated analysis methods to visualise the use case behaviour and generate test cases were discussed. We believe this approach assists in reducing errors.

For future work, we aim to further extend UseCasePro to include the notion of accident case [4] which can be used to explicitly state behaviour that lead the system to an unsafe state. This could help better integrate safety concerns early in the requirements engineering process for safety-critical systems. The Rodin toolset supports tools that provide animation of the Event-B model. The encoding of the use case specification in Event-B could be used to drive the animation of the system. This could help stakeholder to better understand the system being built at early stage in the development process.

References

Safety kernel for control systems design

Alexei Illiasov
Newcastle University
UK

Abstract. We consider an alternative approach to the design of control systems in Event-B. We propose to structure a system into a one or more safety kernels, encapsulating essential safety properties, and use the DSL-Kit extension to built control logic in an algorithmic notation with common control ow primitives and kernel actions as atomic statement. The result is then translated back into Event-B.