

Representing Repairs in Configuration Interfaces: A Look at Industrial Practices

Tony Leclercq, Maxime Cordy, Bruno Dumas, Patrick Heymans

University of Namur, Belgium

first.last@unamur.be

ABSTRACT

Configurators are widespread applications where users can tailor products (i.e. goods or services) to their needs by selecting options and setting parameter values. Constraints over these options exist to avoid building invalid products. Thus, when the user attempts to combine incompatible options, the configurator should raise an error and help the user *repair* her configuration, that is, change the selected options to obtain a valid product. In this paper, we observe how 54 configurators from different industries handle this repair mechanism. We show that in a majority of cases, the configuration interfaces exhibit bad practices that impede an effective usage of repair, thereby impoverishing user experience.

ACM Classification Keywords

H.5.2. User Interfaces: Graphical user interfaces (GUI), Standardization, Configurator, Smart system

INTRODUCTION

Today's companies increasingly rely on mass customisation to provide their customers with products (i.e. goods or services) that meet their particular needs, while still achieving economies of scale [12]. The customisation of a product is typically performed via a software application named configurator [6]. This application often comprises a graphical user interface where users enter their needs and preferences by selecting options and setting parameter values. Configurators have been developed for many years and are more and more frequently used, be it as back-office tools in companies [2, 8, 11, 7] or as public tools on the web. The ever-growing size of Cyledge's Configurator Database [5] witnesses the ongoing interest towards configurators. Given their critical role in today's businesses, these must be effective at guiding users and absolutely reliable.

An essential functionality of a configurator is to guarantee that the configuration created by its user is *valid*. Indeed, invalid configurations must be avoided as they lead to inappropriate products, be it for technical, marketing or legal reasons. The

configurator achieves this by, notably, making unavailable new options that are incompatible with the options already selected. This functionality, named *propagation*, drastically simplifies the work of the user. The downside lies in the risk for the user to be refused options that she really wants, due to previous choices that were of lesser importance. In this case, she must change her configuration to make the desired option available. This can be really difficult, as the incompatibility between the desired option and the current configuration can result from the interaction of many constraints between many options.

A prerequisite is to explain the reason why the option is not available. This is a common functionality required in configurators, that classifies them amongst the range of explainable software systems. Feedback explanations already help the customer resolve the incompatibility problem, but they are far from sufficient. This motivated the need for *configuration repair* [3, 9, 15, 4], an automated method to determine what must be changed in the configuration to accept a forbidden option. Computing configuration repair is not an easy task: consider, e.g. that to select an option o_1 , an option o_2 must be unselected, but doing that requires to select a third option o_3 . Still, efficient solutions exist, e.g. [16]. Together with validity checking, propagation and explanation, configuration repair is essential for a smooth user experience [13].

A question yet unaddressed is how to properly integrate repairs in the graphical user interface of configurators [14, 1, 10, 4]. Major difficulties include the possibly high number of changes to perform, and the potential inability of the repair algorithm to determine all the changes to make in one execution step. Our long-term objective is to fill this gap by providing HCI guidelines for representing repairs, thereby contributing to build a consistent body of knowledge dedicated to engineering configurators, as envisioned by Abbasi et al [1].

This paper constitutes the first step of our work. More precisely, we analyse how 54 configurators, chosen across 10 industries, represent repairs in their interface. Thereby, we aim at identifying bad practices in current configurators, which would in turn justify the need for precise guidelines specific to configuration repair. Our observations show that 46 configurators do not handle repair properly. This is due not only to their inability to compute repairs in every problematic case, but also to bad integrations within the configuration UI. This motivates the need for standard guidelines for handling repair within any configuration interface.

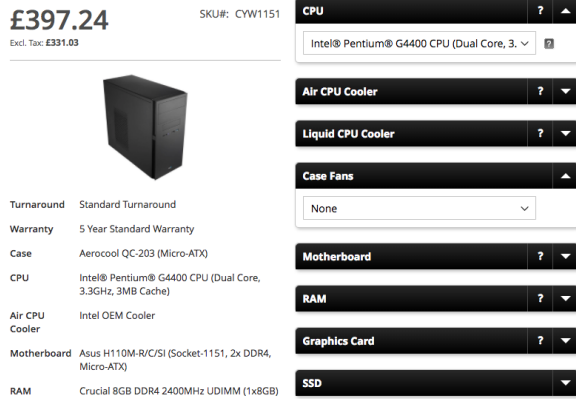


Figure 1. An example of PC configurator

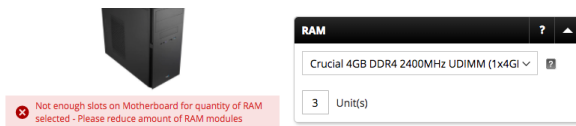


Figure 2. Explanation after a detected configuration invalidity

A FEW EXAMPLES OF REPAIR IN CONFIGURATORS

We first illustrate through intuitive examples what lies behind a configurator and the concept of configuration repair. Figure 1 shows an overview of a PC configurator.¹ The left part of the screen gives a summary of the current configuration, while the right one depicts the parts of the PC to configure (such as CPU, Coolers, and motherboard). Not all combinations are valid, though. For instance, once the customer has selected a motherboard, obviously the quantity of RAM modules cannot exceed the number of slots on the motherboard; see Figure 2. In this case, the configurator raises an error and explains that the selected quantity of RAM modules is inappropriate. It also suggests to reduce this quantity to make the configuration feasible. Thanks to this explanation, the user knows how to repair her configuration. This mechanism, however, leads to two pitfalls. First, the customer has to fix the configuration manually, and is therefore forced to check how many slots are available on the selected motherboard. Second, the configurator omits the alternative of selecting another motherboard equipped with more slots. This example is an instance of what we name *manual repair*.

Figure 3 illustrates another case problematic to the user. We observe that some Power Supply Units (PSUs) are greyed because they are not compatible with the current configuration. This impedes the user to select these inappropriate power PSUs. However, no explanation is given and no automated means of changing the configuration to accept these PSUs is provided. While this might seem unimportant in this specific PC example, it can be really problematic when numerous options are provided to the user.

Consider instead car configurators, arguably the most popular kind of configurator [5]. Cars are known to give access to a high number of options and accessories. Their configurators,

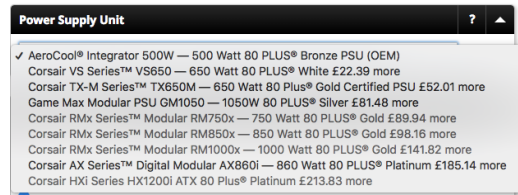


Figure 3. Propagation without explanation or repair

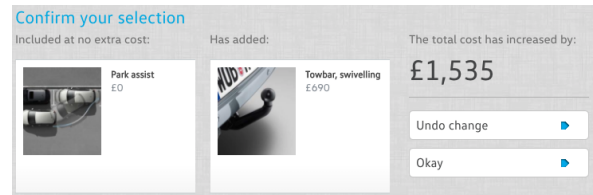


Figure 4. Configuration repair following a selection

especially in the western industry², provide a lot of customisation possibilities: customers can combine any trim with the many available options. It is therefore unrealistic not to provide the customer with a repair functionality showing her how the configuration must change to accept a desired option. Figure 4 illustrates this situation occurring in the UK Volkswagen configurator. The rear-view camera option requires either the standard park assist or its trailer variant. Accordingly, after selecting rear-view camera, the customer is asked to select one of the two park assist options.

BENCHMARKING CURRENT PRACTICES IN INDUSTRY

The previous section already highlights multiple difficulties that users may encounter. To better assess the rate of occurrence of these issues, we carried out an evaluation of the current practices in industry. More precisely, we selected 54 configurators taken from different industries: cars (28), computers (5), audio systems (5), electrical appliances (5), boats (6), drones (1), smart altimeters(1), trucks (1), motorbikes (1), and garden rooms (1). Car configurators are particularly interesting because they typically provide a high number of options and constraints, which even more justifies the need for explanations and repair mechanisms. For each configurator, we check whether 11 specific bad practices occur when repairs are triggered. We consider three categories of bad practices depending on the type of repair mechanism that is provided.

1. **No repair.** This first category comprises bad practices that make it impossible to compute repairs. These practices are:
 - (a) **Undetected errors.** The configurator does not check for inconsistencies, allowing users to build configurations that are not valid.
 - (b) **Unexplained errors.** Configuration errors are raised as soon as they are produced, but no explanation is given to the user.
 - (c) **Invisible propagations.** Unavailable options are hidden. The user is unaware of them and thus cannot ask for a repair.

²We indeed observed that in their asian counterparts, configurators often limit the customer's choice to a smaller number of models with predefined options.

¹See <http://www.dinopc.com>

- (d) **Immutable propagations.** Unavailable options are greyed out but the user is impeded to select them.
 - (e) **Unresolved repair.** When the user attempts to make an invalid choice or to change a propagated option, the configurator proposes her to reset her configuration, thereby losing any previous progress.
2. **Manual repair.** Manual repairs display messages explaining users how to change their configuration to achieve their goal. Related bad practices include:
 - (a) **Misleading:** The repair message is ambiguous and does not precisely pinpoint the options to change.
 - (b) **Misplaced:** The repair message is not well situated, resulting in low odds of being detected by users.
 - (c) **Understated:** Too little emphasis is given to the message, running the risk for it to be unnoticed.
 - (d) **Incomplete:** The message suggests only one solution although there exist others.
 3. **Assisted repair.** Bad practices also occur when an automated repair mechanism directly proposes solutions to the user (see Figure 4). These are:
 - (a) **Unknown effects:** The configurator proposes to modify the configuration, but does not explain which options would be changed.
 - (b) **Cascading repair:** The configurator is able to detect the option o_1 to change to achieve the user’s goal. However, if changing o_2 requires changing another option o_2 , the configurator alerts the user only after she accepted to change o_2 . This process is repeated as many times as the total number of options to change.

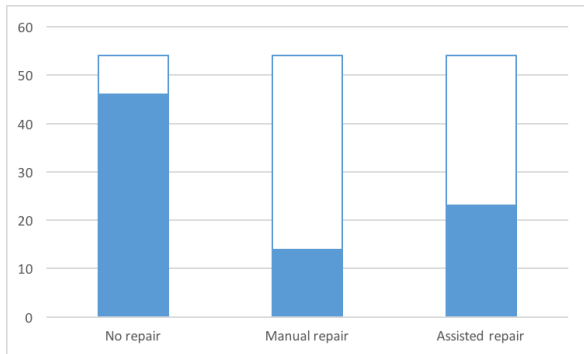


Figure 5. Number of configurators that support repair mechanisms

Results. As a first step, we measure how many configurators exhibit cases of manual repair, assisted repair, and no repair at all (see Figure 5). It turns out that 46 configurators do not provide a repair each they should. Manual repair occurs in 14 configurators, while assisted repair does in 23. These numbers also reveal that the way repair is handled is not always consistent within a given configurator. Indeed, it happens that for some errors the configurator provides no repair, while for others it provides manual or assisted repair. This is symptomatic of configurators that are developed as any software and do not rely on a rule-based engine to perform logical computations [4].

Looking at the no-repair cases (see Figure 6), we find out that the most common reasons for the absence of repair are the

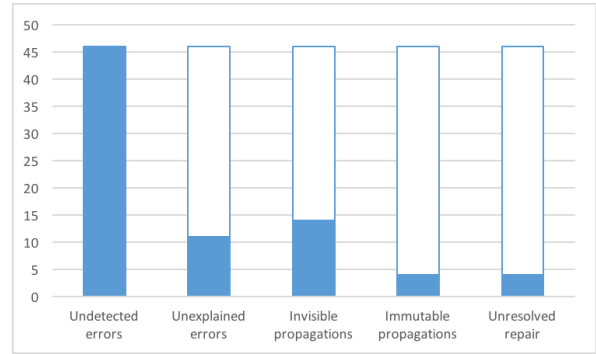


Figure 6. Reasons for lack of repair

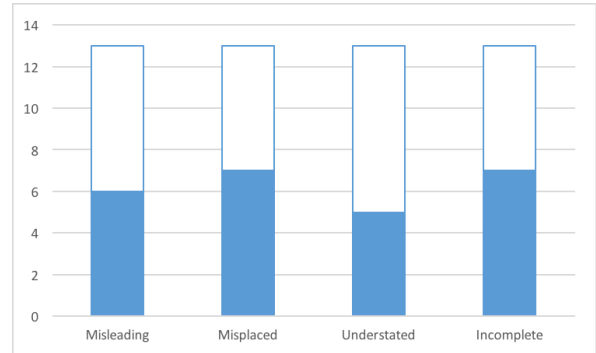


Figure 7. Issues encountered in manual repair

undetected errors. Indeed, 46 configurators are not able to detect all the inconsistencies that may occur in a configuration, which is quite surprising given the importance of this must-have functionality. When errors are raised, they are too often unexplained (11 cases). As for propagations, 14 configurators make them invisible while four do not allow one to modify a propagated value. Finally, four configurators do not support repair and only allow the user to completely reset the configuration.

Figure 7 reports issues observed in the 14 occurrences of manual repair. The four kinds of bad practices occur almost equally, with six occurrences of misleading explanations, seven of misplaced explanations, five of understated, while seven configurators propose incomplete solutions. In total, only four configurators implement manual repair properly (i.e. without exemplifying an identified bad practice).

Regarding the 23 configurators that provide assisted repair (see Figure 8), only one of them suffers from the cascading-repair problem. The problem of unknown effects is, however, more frequent with 11 occurrences. Overall, half of these configurators handle assisted repair properly, although only eight of them provide this functionality each time it is needed. This low number corroborates our previous statement that managing configuration repair is a laborious task that cross-cuts the whole configurator.

Surprisingly, the choice between manual and assisted repair highly depends on the considered industry. Car configurators rely more often on assisted repair than on manual repair

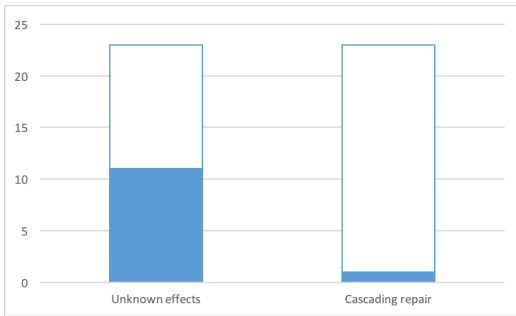


Figure 8. Issues encountered in assisted repair

(19 cases against 2). On the contrary, manual repair is more common in computer and electrical appliance configurators, whereas the audio configurators provide no repair at all. One possible explanation is that assisted repair is more important in customer-oriented configurators, while the ones aimed at domain experts can settle for manual repair. Nevertheless, this observation justifies the need for studying both types of repair.

CONCLUSION

Although repair is essential for a smooth user experience, our study reveals that it is rarely properly and thoroughly supported. This is indeed a difficult functionality to implement, notably on the back-end side [9]. However, HCI guidelines are also needed [10], especially as there is no de-facto standard across multiple industries [14]. Given the bad practices that proliferate amongst the configurators we studied, designing appropriate HCIs for repair will constitute an important focus of our future research work.

ACKNOWLEDGEMENT

This work was partly supported by the European Commission (FEDER IDEES/CO-INNOVATION) and the Wallonia-Brussels Federation under the ARC programme.

REFERENCES

- Ebrahim Khalil Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, and Patrick Heymans. 2013. The Anatomy of a Sales Configurator: An Empirical Study of 111 Cases. In *CAiSE'13*. Springer-Verlag, Berlin, Heidelberg, 162–177.
- Liliana Ardissono, Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, Ralph Schäfer, and Markus Zanker. 2002. A Framework for Rapid Development of Advanced Web-based Configurator Applications. In *ECAI'02*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 618–622.
- Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *ASE '10*. ACM, New York, NY, USA, 73–82.
- Maxime Cordy and Patrick Heymans. 2018. Engineering Configurators for the Retail Industry: Experience Report and Challenges Ahead (to appear). In *SAC '18*. ACM.
- Cyledge. 2017. Configurator Database. (2017). Retrieved July 24, 2017 from <http://www.configurator-database.com>
- Xuegong Ding. 2008. Product Configuration on the Semantic Web Using Multi-Agent. In *ICNSC '08*. 304–309.
- Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen. 2014. *Knowledge-based Configuration: From Research to Business Cases* (1 ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Gerhard Fleischanderl, Gerhard E. Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. 1998. Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems* 13, 4 (July 1998), 59–68.
- Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A User Survey of Configuration Challenges in Linux and eCos. In *VaMoS '12*. ACM, 149–155.
- Tony Leclercq, Jean-Marc Davril, Maxime Cordy, and Patrick Heymans. 2016. Beyond De-Facto Standards for Designing Human-Computer Interactions in Configurators. In *EnCHIReS@EICS 2016, Bruxelles, Belgium*. 40–43.
- Deborah L. McGuinness and Jon R. Wright. 1998. Conceptual Modelling for Configuration: A Description Logic-based Approach. *Artif. Intell. Eng. Des. Anal. Manuf.* 12, 4 (Sept. 1998), 333–344.
- B.J. Pine and S. Davis. 1999. *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press.
- Rick Rabiser, Paul Grünbacher, and Martin Lehofer. 2012. A Qualitative Study on User Guidance Capabilities in Product Configuration Tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 110–119. DOI: <http://dx.doi.org/10.1145/2351676.2351693>
- C. Streichbier, P. Blazek, and F. Faltin. 2009. Are De-Facto Standards a Useful Guide for Designing Human-Computer Interaction Processes? The Case of User Interface Design for Web Based B2C Product Configurators. In *HICSS '09*. IEEE Computer Society, Washington, DC, USA, 1–7.
- J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. 2008. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *SPLC '08*. IEEE Computer Society, Washington, DC, USA, 225–234.
- Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. 2012. Generating Range Fixes for Software Configuration. In *ICSE '12*. IEEE Press, Piscataway, NJ, USA, 58–68.