The Making Of User-Interface Designer
A Proprietary DSM Tool

Laurent Safa

EMIT Middleware Laboratory
Matsushita Electric Works, Ltd.
1048, Kadoma, Osaka 571-8686, Japan
+81-6-6908-6752
safa at mail dot mew dot co dot jp

Abstract. This paper relates the construction of UI Designer, a proprietary tool that enables up to 5-fold productivity enhancements by synthesizing finished software products out of domain-specific models. We start by presenting our development context and rationales for using Domain-Specific Modeling (DSM). Next we describe our approach to define domain-specific notations with an emphasis on pseudo-realistic modeling and dynamic modeling. We conclude with a discussion on the topics of tool applicability and tool maintainability.

1. Introduction

As the world turns digital and connected, home controllers provide more features like energy saving [1], intrusion detection, and earthquake alerts. These new features require full-fledged user-interfaces with menu navigation, data visualization and system configuration functions. Product systems being composed of devices under the responsibility of several development groups, the volume of integration issues rises with the increased software complexity.

Our company decided to address these issues by building an application framework to enforce best coding practices and reuse of selected components. Because we were still exploring the market needs for embedded user-interfaces, the framework was voluntary limited to well-understood problems to avoid impeding the creativity of practitioners.

The framework programming interface resulted in domain-specific configuration files and generation of application skeletons. Addition of business logic and visual designs were left to the programmers.

2. Problem

The application framework proved effective in suppressing the behavioral inconsistencies between devices, which mechanically eased system integrations. However, practitioners reported new problems directly related to the use of that framework:

– Lack of documentation and good tool support for the framework's proprietary APIs
– Error-prone concept mapping based on name-matching across multiple files
– Uncontrolled insertion of business logic code into the generated skeletons – this was error-prone and defeated the purpose of guaranteed quality expected from automation
– Additional procedures to build the software

After several products were built with the framework, the company gained a good understanding of its needs for embedded user interfaces. It was then decided to enhance the framework by adding a visual modeling tool. This tool would provide for full code generation to systematically reduce human errors.

3. Investigation

The study of existing user-interfaces revealed patterns in the visual design of pages (layout, proportions, type and quantity of information), the types of pages that could be linked together, and the application source code itself. All these patterns result from business-specific guidelines defined by the design group.

Strong similarities across requirement documents confirmed the existence of a company-specific notation. This notation uses realistic snapshots to represent pages, arrows and lines of various width and color for the navigation paths, and overlaid annotations to specify special behaviors. Size and complexity are managed by splitting the requirements into groups of functionally related pages.

We finally found that 60% to 80% of existing user-interfaces is defined by the five most frequent page patterns.

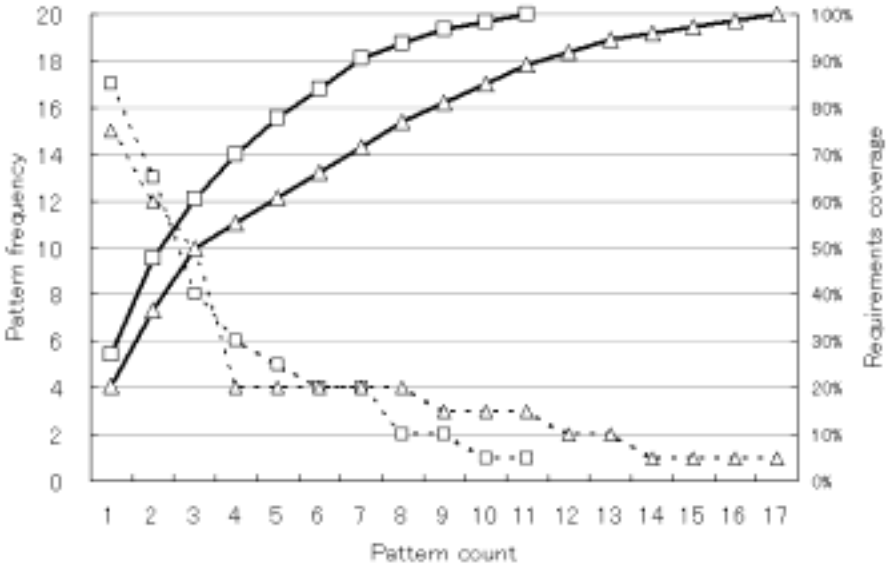

Fig. 1. Patterns and requirements coverage

As a result of this investigation, the domain was deemed favorable to visual modeling and we engaged into the tool development phase.

4. Formalization

We formalized existing development guidelines and patterns of usage into page standards, with one page standard per pattern.

For example, a menu page is formalized into a standard pattern of 13 elements ranging from decorative icons to information texts and navigational buttons.

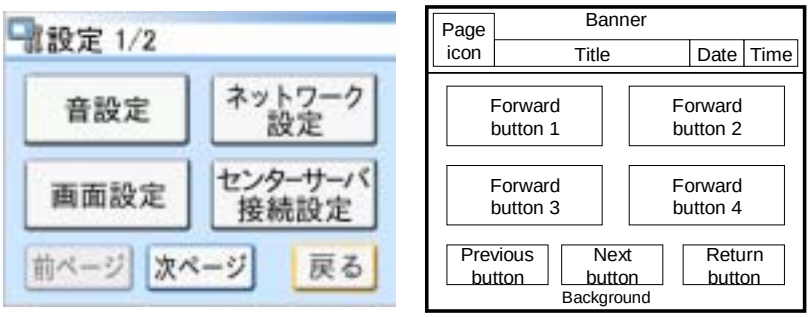

Fig. 2. The menu pattern

Another recurrent pattern is the configuration page which includes an interaction with the underlying machine.

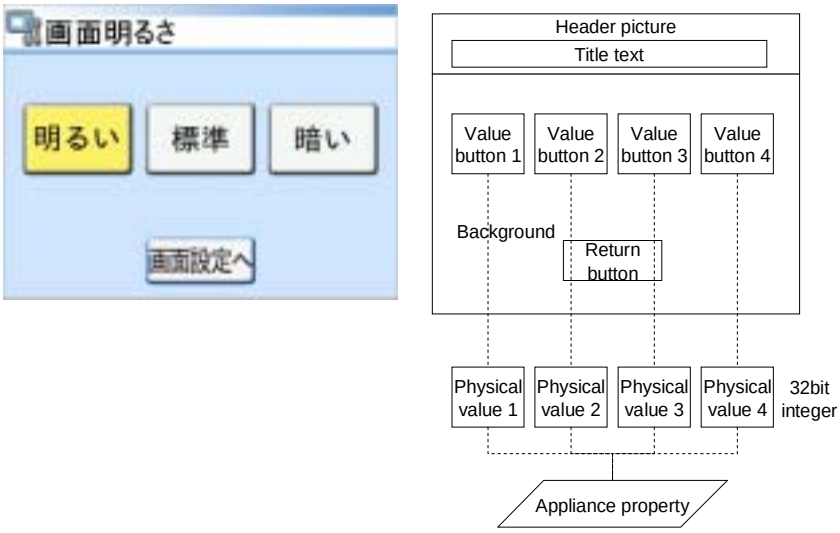

Fig. 3. The configuration pattern

We also formalized behavioral standards like the one used to split strongly related commands across sibling menu pages linked with *previous* and *next* buttons.

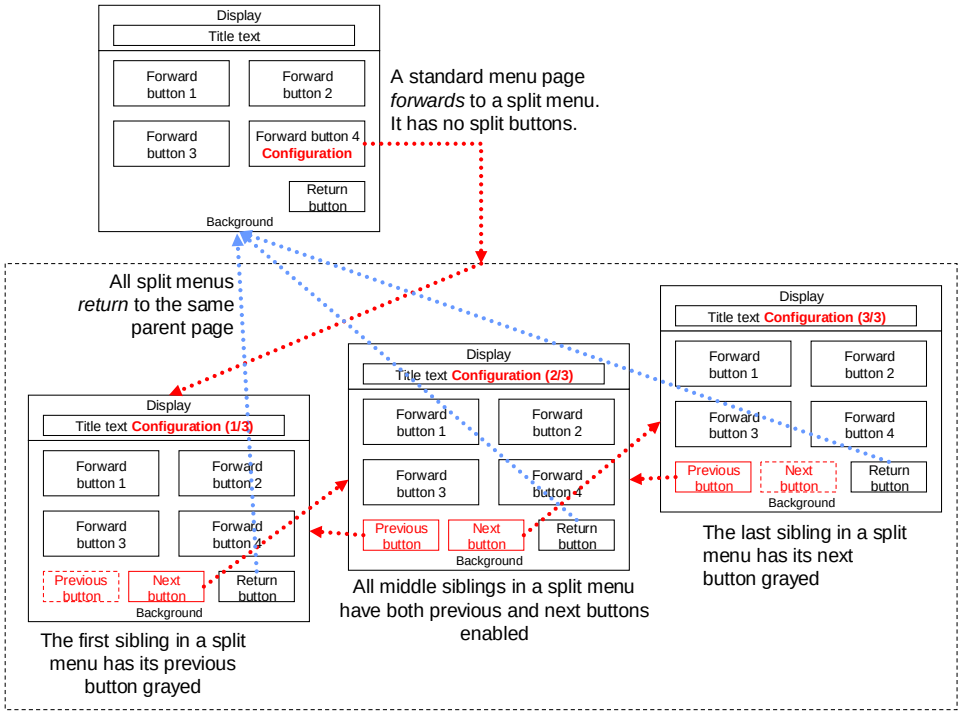

Fig. 4. The menu behavior pattern

Having clarified page standards and behavioral standards we were ready to realize *full product synthesis* from visual models.

3. Technology Choice

Resources allocated to tool development were scarce as most of funding was tunneled to new product developments with low-risk, well-established methods. That situation led us to search for a *modeling tool toolkit* with the aim of building robust tools with minimal coding. Our principal selection criteria were:

- Ability to define the metamodel and visual editor without programming. In addition to coping with limited resources, this criterion had also to do with ethic: as we engaged in a road to reduce application programming errors by way of automation, we thought we had to show the example by minimizing our very own coding.
- Powerful features to scan application models and generate source code, make files and other textual artifacts
- Any feature that facilitates close-loop interaction with practitioners whom work will be impacted by the tool deployment
- Support and training services
- And above all, a stable technology that would allow us to focus on the problems at hand: notation, design and automation.

Our choice went finally to the GOPPRR method [2] and MERL scripting language as provided by MetaEdit+®.

4. User-Interface Designer

Tool construction with GOPPRR and MERL being largely described in the literature [3], we propose to skip this step and present instead some key aspects of our tool.

6.1 On-Site-Configurable Modeling Tool

The following illustration represents a typical work session of User-Interface Designer.

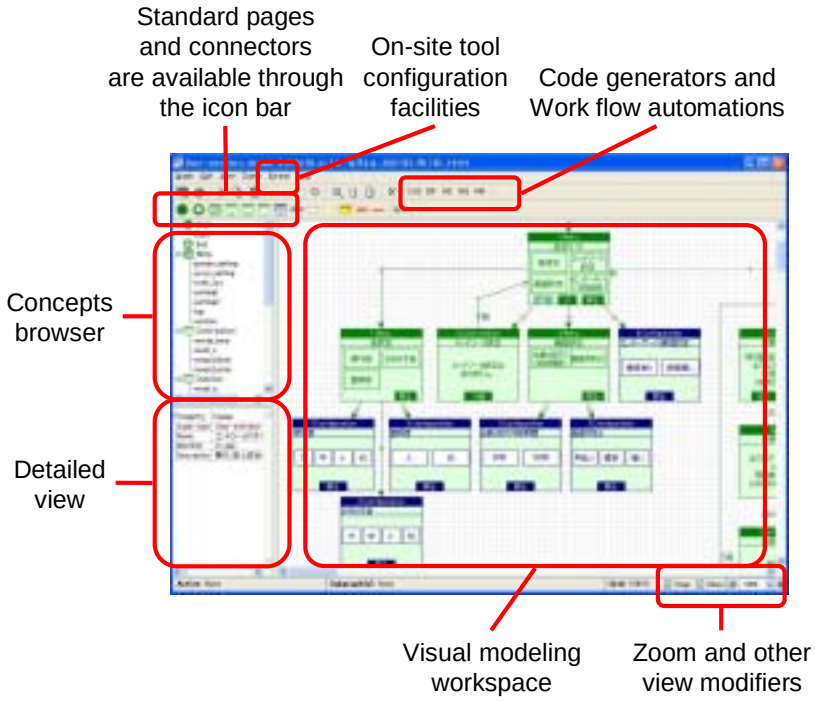

Fig. 5. UI Designer

In addition to features commonly found in visual editors like concepts browsing or icon bars, MetaEdit+® offers *on-site tool configuration facilities*. These allow for modifying the metamodel, the visual notation and the generators at the end-user site, right in the middle of a modeling session, making it possible to experiment user suggestions on-the-fly, together with the user, without involving complex programming language neither time-consuming compilation cycles.

This feature significantly helps put users at ease with the modeling tool as it is perceived to be open to changes in an elegant and easy manner.

6.2 Sub-Model Inclusion

We provide a visual statement to *include* sub-models. The goal is to help manage complexity, and to facilitate the reuse of standard models like sound configuration or firmware update. This matches the established practice of grouping functionally-related pages in requirement sections. Each sub-model corresponds to one requirement section, and each *include* statement corresponds to a reference/reuse statement in the requirements documentation.

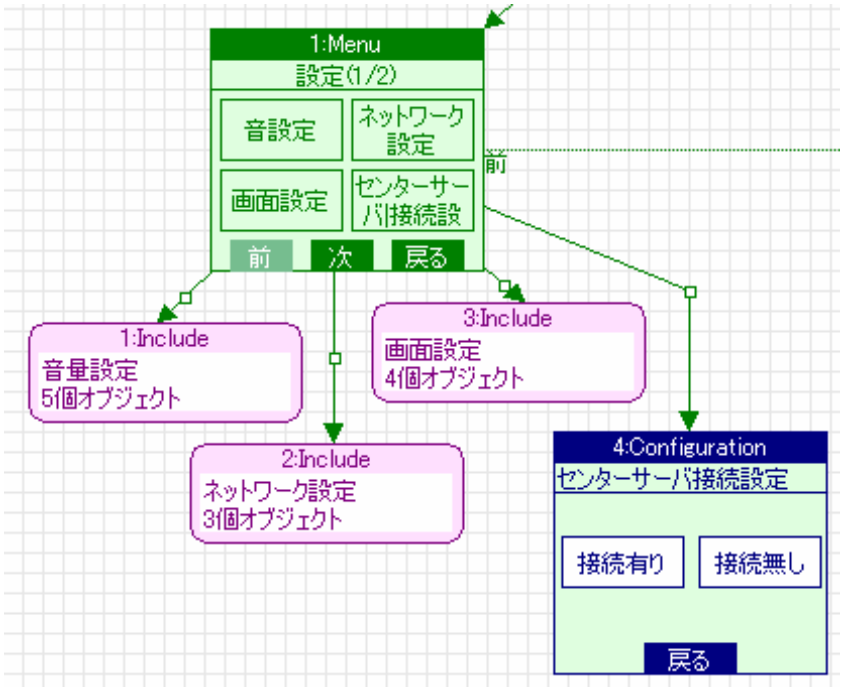

Fig. 6. Model inclusion

6.3 Pseudo-Realistic Modeling

As found during the investigation, user-interfaces used to be specified in term of realistic page snapshots. Keeping the model as close as possible to the final product was required by stakeholders to get diverse practitioners involved in the product construction process. We comply with this requirement by providing visual symbols that mimic the visual design of product pages.

For example, the symbol of *menu* pages (Fig. 7) changes according to the number of forward pages, whether it is in a split menu chain with *previous* and *next* buttons, and whether it has a parent page.

Fig. 7. Pseudo-realistic menu symbol

Similarly the symbol for *configuration* pages (Fig. 8) faithfully shows buttons which size is inversely proportional to the number of choices presented to the end-user.

Fig. 8. Pseudo-realistic configuration symbol

Although pseudo-realistic symbols are not exact reproductions of the final pages, these convey valuable information to the tool user: a quick look at the *menu* symbol (Fig. 7) tells the leftmost menu has room for more buttons, while the rightmost is full.

6.4 Dynamic Modeling

Symbols change dynamically based on their context in the model, which provides the modeler with immediate feed-back to his editing actions.

Figure 9 represents a model before and after a link is established between two pages. As illustrated in the rightmost snapshot, the menu symbol gains one forward button, and the configuration symbol gains a return button. Removing the link brings the model back to the initial state.

Before the link

After the link

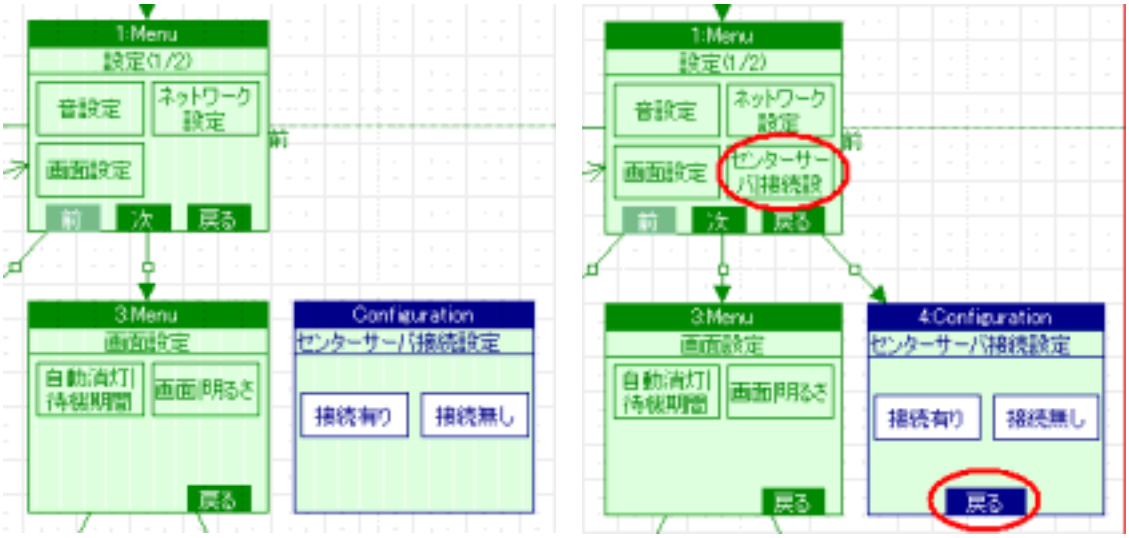

Fig. 9. Dynamic symbols

6.5 Code Generation and Productivity Increase

The first release of the tool supported the five most frequent page patterns (*menu*, *configuration*, *question*, *confirmation*, *splash*), allowing to model and generate 60% to 80% of typical applications.

Assuming a similar level of coding complexity across the range of patterns, this translates into productivity increases of factor 3 to 5 for the software design, implementation and configuration phases. We expect further gains in the tasks of documentation and product testing, but these have not been adequately measured yet.

Although we see no possible improvement by way of DSM for the highly creative product design task, initial results suggest that the activities of design instantiation and design/code integration benefit from the same 3- to 5-fold productivity improvement.

5. Applicability

7.1 Tool Cost/Benefit Ratio

We found that each page standard took us around 3 man-days to implement the metamodel, the visual notation and the code generation. When compared to an estimated development cost of 0.5 man-day per page, we assumed that any page that occurred 6 times or more was worth metamodeling.

7.2 Need For Escape Semantics

Beside recurrent patterns there exist a number of low frequency pages that are not worth automating with the tool. These include original welcome pages, product-specific data visualization pages, or first-of-a-kind pages.

Fig. 10. Low-frequency patterns happen to contain high-value element

This problem corresponds to the gap between tool and practices [4]. To cope with this issue we need *escape semantics* that provide for semi-controlled and free modeling of new kinds of pages.

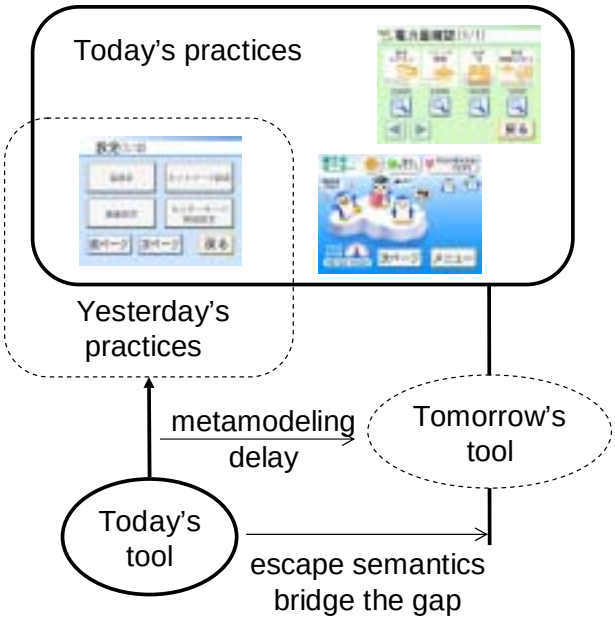

Fig. 11. Metamodeling delay

7.3 Sample Escape Semantics: QVGA Joker

We provide *escape semantics* by way of Joker symbols. The goal is to let modelers specify new product features still unsupported by the tool, and to allow for integrating these with currently supported page patterns like *menu* and *configuration*. Fig. 12 illustrates a simple Joker template (note how we use a different color scheme to distinguish the Joker symbol - red - from menus and configurations - green).

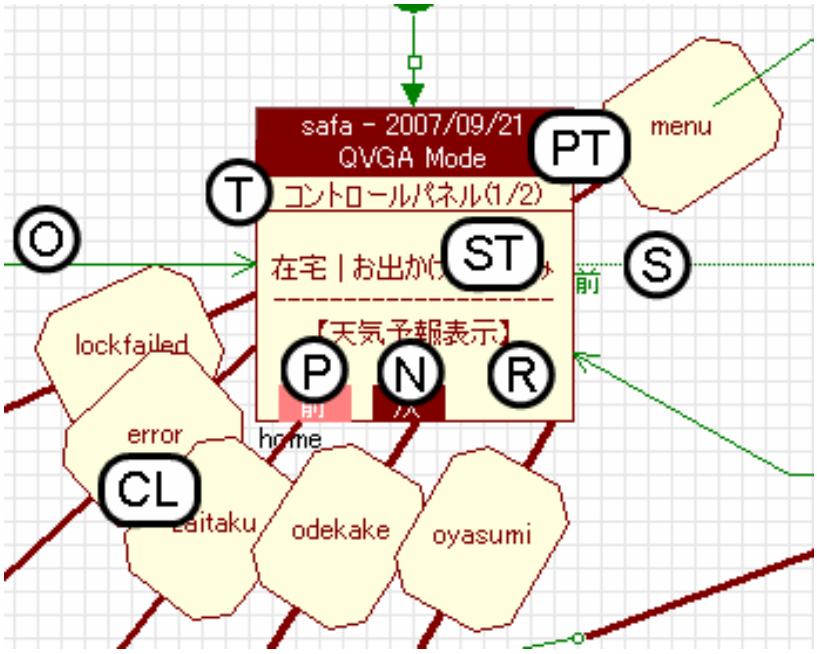

Fig. 12. An instance of *escape semantics*

We recognize several standard elements that are often reused by low-frequency patterns: title (T), previous button (P), next button (N), return button (R), link to sibling pages (S), and link from other pages (O). This corresponds to semi-controlled modeling, with enough information to enable the generation of mock-up pages that integrate into working products.

Free modeling is provided in the form of simple text (ST), conditional links (CL), and the ability to define/propose a new page type (PT). In the present case the modeler specified a special front page with two sections. The first section contains three buttons to change the home operation mode in just one touch, and the second section contains a widget to display weather outlook.

7.4 Tooling Limit

Based on either requirements coverage or usage frequency, we decide a limit over which not invest time and money into tooling, but instead focus resources on the development of new features with unique value. This *tooling limit* is set arbitrarily and periodically reviewed based on domain knowledge and then-current perception of market needs.

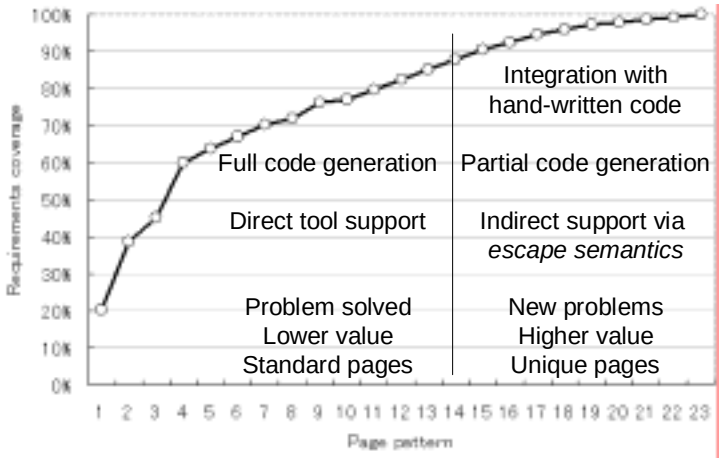

Fig. 13. Tooling limit

7.5 Sample Model Of A Real-World Application

Fig. 14 illustrates the model of one user-interface application in Lifinity® Control Panel [5] when written with five standard page patterns (*menu*, *configuration*, *question*, *confirmation*, and *splash*) and one Joker symbol for *escape semantics*.

Out of a total of 45 pages, 27 pages can be modeled with the five standard patterns, and 18 pages require the use of Joker symbol. In that case UI Designer covers 60% of the requirements.

A close look at the requirements would reveal two patterns P1 and P2 which, if supported by UI Designer, would allow to generate 7 more pages, raising the requirements coverage to 34/45 = 75%.

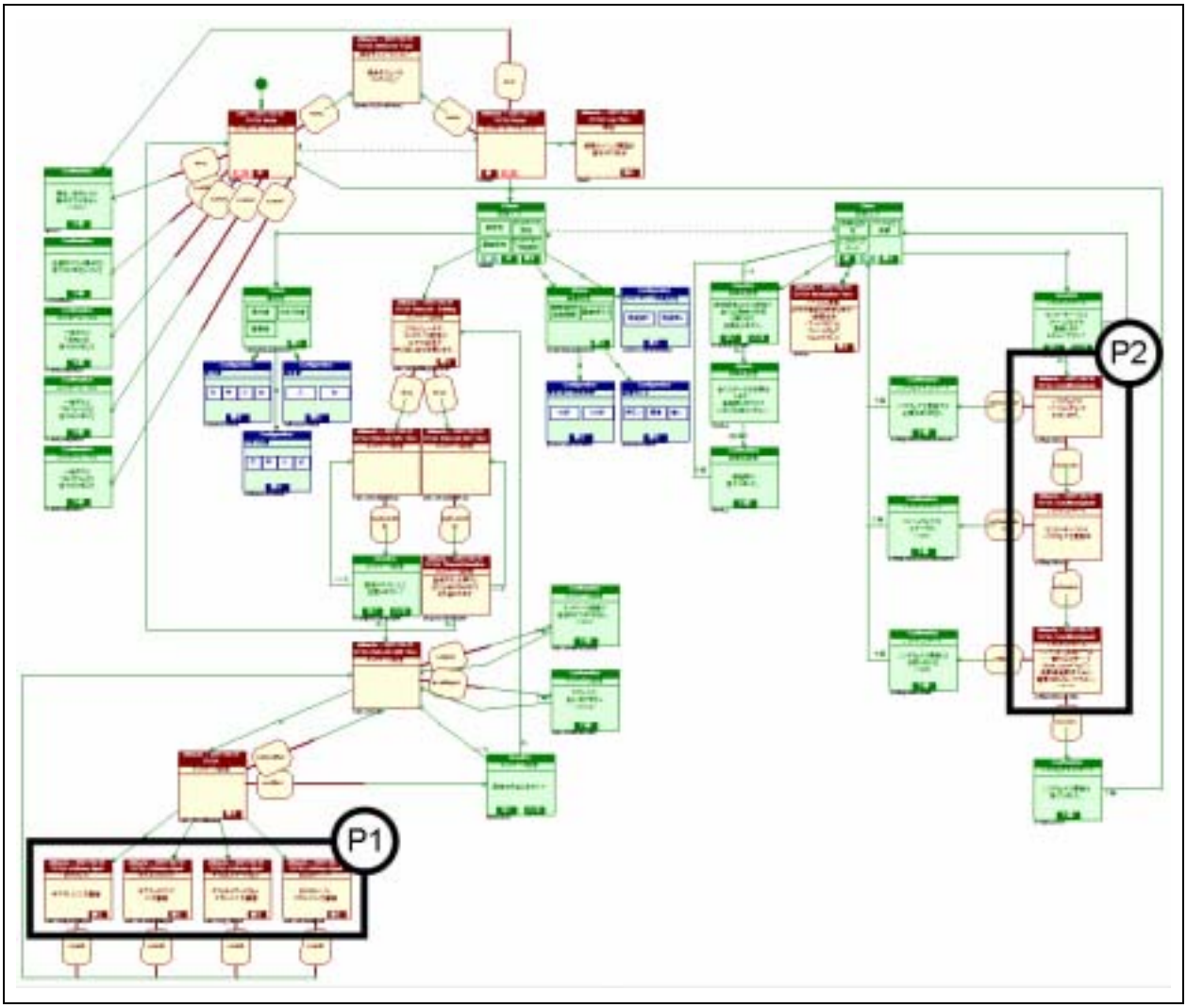

Fig. 14. A legacy application modeled with five patterns and one Joker symbol.

6. Maintainability

Beside technical concerns, we wanted to develop a solution that is maintainable, meaning it should be cheap to evolve, and its ownership should be easily transferable to tiers. We took the opportunity of an undergraduate student joining the company for a six-month internship to conduct several experiments. We will call him *fresh-man*.

8.1 Simplicity

After one week in the company, fresh-man did not know anything about the product, the underlying framework, the programming guidelines, and the modeling tool. However, half-a-day training to the tool was sufficient to make him comfortable with modeling and synthesis of embedded user-interfaces of same quality as the ones hand-coded by the experts… but in a fraction of their time.

This experiment helped us demonstrate the capability of our domain-specific tool to empower basic employees into experts within a given application domain and process area.

8.2 Adaptiveness

Fresh-man then knew the tool and modeling language. To check the adaptiveness of our solution, we asked him to implement code generation for a completely different target platform.

Supported platform	Challenge platform
Embedded OS	No OS
CPU	microcontroller
MB RAM	KB RAM
QVGA screen	101x64 pixels
color	monochrome
touch panel	3-position knob (top, down, push)

After 3 days spent learning the GOPPRR metamodeling approach and MERL scripting language with MetaEdit+® standard documentation and sample DSLs, fresh-man could write a C source code generator with build/flash/run workflow automation included within 3 additional days.

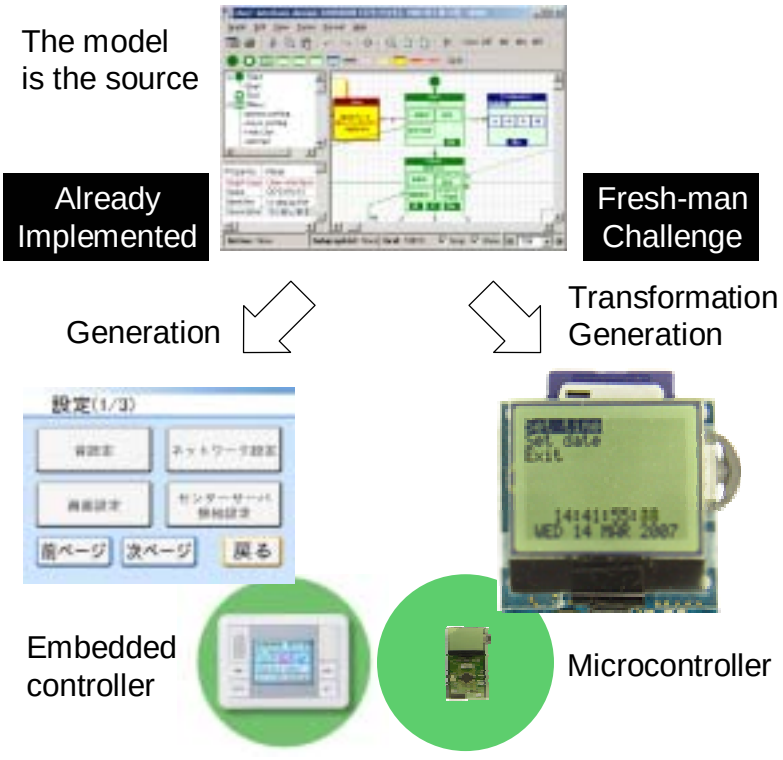

Fig. 15. Adapting the tool to a new target platform

8.3 Ownership Transfer

After three months in office fresh-man was happily participating to the tool's core development. We believe this was made possible by our choice of minimalist coding on-top of stable and well-documented GOPPRR and MERL technologies.

7. Conclusion

This paper reported the construction of UI Designer, a proprietary tool to model and synthesize embedded user-interfaces for the Japanese home automation market, with a 3- to 5-fold productivity increase when compared to standard development methods.

We presented several features found to be critical facilitators for the tool acceptance: pseudo-realistic modeling, dynamic modeling, and on-site tool configuration. We also introduced the concepts of tooling limit and escape semantics. These serve to manage the gap between the features provided by the tool and actual product development needs. Finally we investigated the problem of tool maintainability with several experiments employing a fresh-man. We verified that our tool is useable and maintainable by a practitioner with standard skills widely available on the job market.

Overall, the construction of UI Designer as reported in this paper ended being a milestone in our company. The ability to synthesize product quality software and to fully automate the software construction process from the visual model down to the target acted like a trigger that opened the doors to deployment on new product developments.

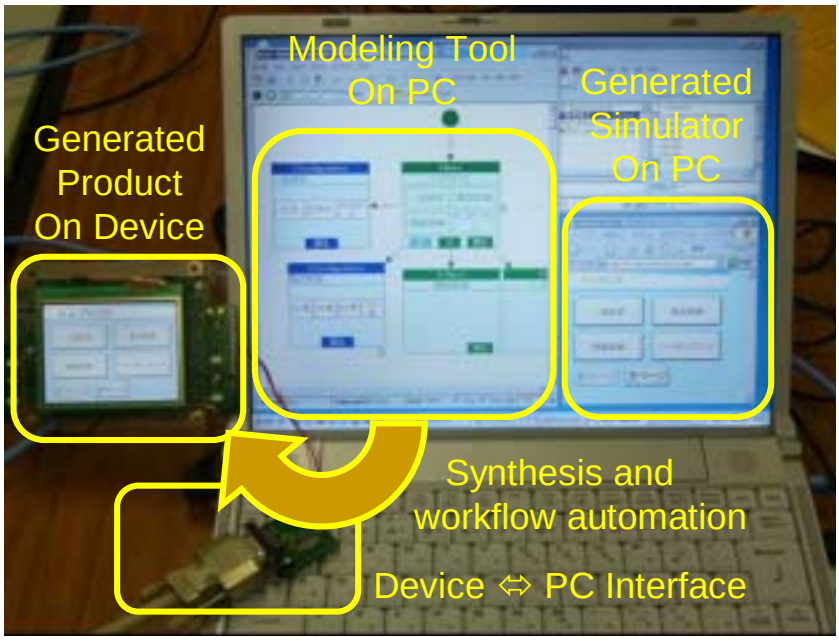

Fig. 14. First occurrence of complete synthesis and workflow automation for the embedded user-interfaces

8. References

1. Web news about MEW adding energy saving features to Lifinity®, http://plusd.itmedia.co.jp/lifestyle/articles/0705/15/news073.html
2. Kelly, S., Lyytinen, K., and Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment. In: Proceedings of CAiSE'96, 8th Intl. Conference on Advanced Information Systems Engineering, Lecture Notes in Computer Science 1080, Springer-Verlag, pp. 1-21, 1996.
3. Kelly, S., and Pohjonen, R.: Interactive Television Applications using MetaEdit+. MDD-TIF07.
4. Safa, L.: The Practice of Deploying DSM - Report from a Japanese Appliance Maker Trenches. DSM'06
5. Lifinity® product home-page http://biz.national.jp/Ebox/kahs/index.html
6. MEW Corporate report: Lifinity® Home Network System, http://www.mew.co.jp/e/corp/ir/annual/2007repo/pdfs/05.pdf