Improving MDD Productivity with Software Factories

Benoît Langlois
Thales Research & Technology
RD 128
91767 Palaiseau, France
33(1) 69.41.60.28
benoit.langlois@thalesgroup.com

Jean Barata
Thales Research & Technology
RD 128
91767 Palaiseau, France
33(1) 69.41.60.32
jean.barata@thalesgroup.com

Daniel Exertier
Thales Research & Technology
RD 128
91767 Palaiseau, France
33(1) 69.41.60.42
daniel.exertier@thalesgroup.com

ABSTRACT
Productivity improvement is a main issue in the context of large-scale developments, where produced software needs to meet quality criteria, both on budget and schedule. This paper studies the introduction of the software factories technique to automate and improve production of complex software systems in the context of Model-Driven Development (MDD), which is still evolving (standards, technologies and tools). To validate this technique, a case study presents a progressive paradigm shift from a traditional model-driven development toward software factories usage. The result is the definition of a technical foundation improving productivity that can be enriched by techniques and practices enabling intensive production of software systems with software factories.

Keywords
MDD, SOFTWARE FACTORIES, DSL, PRODUCT-LINE.

1. INTRODUCTION
To produce high quality software both on budget and schedule, companies usually face productivity improvement issue. This is particularly true in the context of large-scale systems. This paper focuses on the introduction of the software factories technique to improve productivity in the MDD context, and on the paradigm shift from a manually written set of modeling tools to an intensive production of modeling tools with software factories. To appreciate this progression, this paper presents a case study of modeling assistance HMIs (Human-Machine Interaction) production with four successive development strategies: manual development, manual development with framework, automated development with software factories, and automated development with software factories and DSL (Domain-Specific Language). The comparison between production times reveals the most productive and profitable strategy. But beyond software factories adoption, the stake is to determine a set of techniques and practices to continue to substantially improve the productivity of complex software systems developed and maintained for many years. This encompasses technical aspects, such as the definition of software factory architecture, but also human aspects.

Section 2 introduces the case study of HMI production in its context. Section 3 presents the four strategies of HMI production and section 4 analyses production time in order to determine the most efficient solution. Section 5 exploits this productivity analysis to recommend techniques and practices for complex software system development. Section 6 presents further work and section 7 concludes.

2. Context of the Case Study
The Architecture and Engineering Department (AED) of the THALES Software Research Group aims at putting the MDA® vision at work. In this perspective, one activity is the development of a system-engineering modeling tool called MDSysE [8][14]. MDSysE, standing for Model-Driven System Engineering, is currently being used by a set of THALES business units for building complex systems with a model-driven approach, such as air-traffic management system. From a product-line viewpoint, the MDSysE baseline can be derived into several variants, potentially one by business unit project. MDSysE offers a set of integrated modeling tools: modeling assistance HMIs, model checking, model refinement and model views generation (diagrams, files, documentation). To reach that goal, MDSysE complexity led to the development of core technology tools. A first category of tools is a set of core MDD tools on top of which end-user tools are built. For instance, a traceability tool originally required for refinement implementation in MDSysE can now be reused by other modeling tools. A second category of tools is a software factory tooling, mainly with MDSoFa [13], dedicated to MDD.

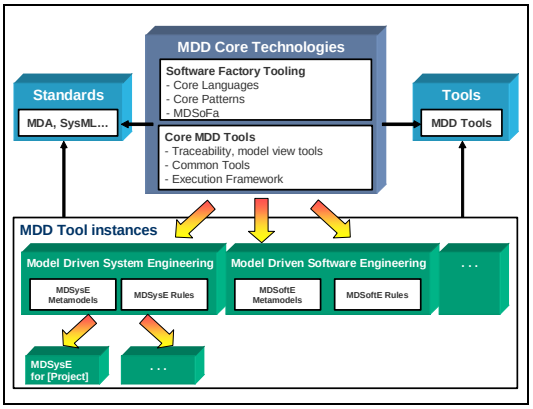

Figure 1. MDD Tool Architecture

2.1 MDSoFa, a Software Factory Tool
MDSoFa, standing for Model-driven Software Factory, is a software factory environment based on the generative technique to produce languages, frameworks and tools in series. Four core technologies participate in the MDSoFa foundation. 1) Languages are described with MOF-level metamodels [15], and a mapping notation allows expressing correspondence between languages, *e.g.* MOF to UML, DSL to DSL mappings. 2) A rule-based language allows expressing patterns: a rule is composed of (i) a

specification part with a rule context and conditions, (ii) an implementation part to express rule actions. The specification part is graphical for readiness whereas the implementation part is textual for efficiency. 3) A template-based language, for the rule implementation part, allows code expansion of template-based expressions with language, mapping, and rule information. 4) To avoid monolithic production, production results are separated by concerns, *e.g.* separating model management from model checking concern.

These technologies define the minimal set to produce in series with MDSoFa. MDSoFa is provided with core languages and patterns. A MDSoFa engine orchestrates the production: generation by concern, pattern-matching usage to apply rules, interpretation of template-based expressions in the implementation parts, packaging for deployment. In addition to these core technologies, product-line management aspects are addressed (variability is expressed with rules before and during the production). All these elements are organized in a MDSoFa product structure. A product is organized according to the following viewpoints: 1) language, 2) pattern, 3) variability, 4) deployment, and 5) production, the whereabouts of production results. A product can contain sub-products.

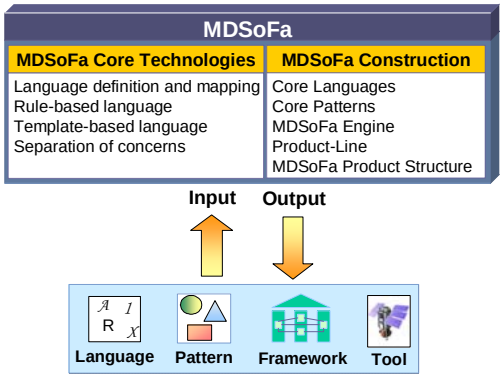

Figure 2. MDSoFa Organization

To position MDSysE and MDSoFa, MDSysE is a MDSoFa instance. MDSoFa takes as input MDSysE metamodels and patterns to produce the MDSysE instance. But a MDSysE instance is not a monolithic block; it is the composition of several MDSoFa instances, such as model checking or HMI for MDSysE, which are assembled to give the final MDSysE tool. In a recursive way, by using a bootstrap with core services, MDSoFa is also instance of itself.

MDSysE (2 years of development with MDSoFa) is used in real industrial projects, and is seen as a testing case for MDSoFa (18 months of development, with its associated methodology). Here are some metrics about MDSysE with MDSoFa: 180 metaclasses, 400 associations; 20 metamodels; 25.000 generated methods; one target modeling-platform and another one for prototyping; development in multi-user mode; three client THALES Business Units. The complete usage of the product-line will be operational soon. A success measure is that MDSysE could not be developed now without MDSoFa.

2.2 Automated HMIs Production

To evaluate the technical solutions used to improve the productivity of MDD products, we focus here on a sub-part of MDSoFa dedicated to the HMI production for user assistance. About 80% of MDSysE HMIs are specified and generated by MDSoFa, the other HMIs being manually produced due to their specificities.

One main function of these HMIs is to manage model element lifecycle. HMIs are characterized by: i) HMI content definition (fields, buttons, and icons) and presentation, ii) model element identification and navigation description in different metamodels, iii) model element lifecycle of the related model elements (model element creation / update / deletion can imply creation / update / deletion of other ones), iv) transaction management, v) model checking to have consistent metamodels, and vi) user facilities. HMI generation is complex because there is no systemic relationship between the model elements involved in an HMI. Moreover, every HMI has its own content. In this kind of situation, the first approach of the development teams is to develop HMIs manually. With a continuously increasing number of similar HMIs, this would mobilize too many resources regarding the other key tool functionalities to be developed. As explained later, the solution is to develop a specification language capturing HMI description. The interest is twofold: (i) with the adoption of a common and systematic notation, it opens the way to automation, and (ii) with the problem abstraction, it avoids pointless implementation details treated later during the HMI production.

3. Case Study Analysis

Four strategies have been successively experimented, one building upon the other, in four stages by the HMI development team: 1) manual development; 2) manual development with a framework; 3) automated development with MDSoFa using the rule-based formalism; and 4) automated development with DSLs easing the production of rules.

Table 1. Fixed and Unit Cost Metrics

Strategies	FC	UC
1. Manual	0	5
2. Framework	10	1
3. Rules	13	0.33
4. DSL	16	0.125

The following subsections describe each strategy along its associated metrics to measure productivity improvement. As for the metrics, we identify (i) the fixed cost (FC) to produce at least one HMI, (ii) the unit cost (UC) per HMI, (iii) the variable cost (VC) equals to (UC x number of HMIs to be developed), and (iv) the global cost (GC) equals to (FC + VC). FC and UC metrics are summed up in table 1, and are used as input to elaborate production times and profit points metrics presented in tables 2 and 3. The figures are the result of the development of 13 wizards, each containing an average of 4 HMIs, that is to say 52 HMIs. Each FC figure reflects the development time of the solution from scratch; the difference between two FC figures being explained by the refactoring time toward the new solution. UC figures are figures for the development time of HMIs with an average complexity (neither simple nor complex). Besides, designers and developers are considered as being familiar with the used techniques.

3.1 Manual Development of HMI

This strategy is the traditional method of development where HMIs are manually written. They fulfill the six kinds of HMI functions described above. Their objective is to assist users in their modeling tasks, avoiding manual and tedious modeling work. The main weakness is development and maintenance time: every HMI development is unique and evolutions are costly. There is no initial cost (FC = 0) but there is an average of 5 days of development per HMI (UC = 5).

3.2 Manual Development with Framework

The next strategy is to identify HMI common parts and generalize them into a framework. With this framework usage, only HMI-specific query and transformation operations and HMI content description remain to be manually coded. The interest is to write and maintain only specific HMI code. Compared to the previous strategy, the development time spent is divided by 5. After writing the HMI framework (here, FC = 10 days), the main weakness is to write and maintain the code for each HMI (UC = 1).

3.3 Automation with Rules

This strategy consists in abstracting all HMI descriptions (metamodel navigation, HMI content, constraints) in a declarative way with rules. As mentioned in subsection 2.1, a rule is described by a context, a set of conditions, and a set of actions. i) The context identifies the used model elements and can contain descriptions in a dedicated language to bring relevant information in a considered domain. For instance, for the HMI description, a reduced textual language allows declaring HMI content, presentation, and constraints. In Figure 3, the XML-like notation allows declaring a "Define Inheritance" label button. ii) Conditions are constraints written in a dedicated language, *e.g.* OCL. iii) While the context and conditions represent the rule specification part, actions constitute the implementation part. Here, we have adopted a textual notation with a template-based language giving a flexible code production. The code is expanded in using any metamodel, mapping, rule, and deployment information.

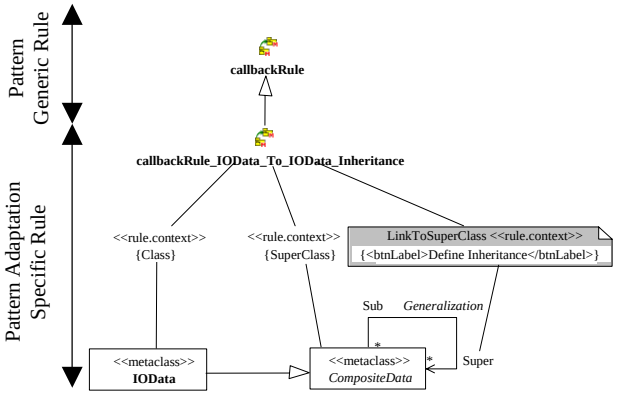

Figure 3. Example of HMI specification rule

The interest of this rule-based approach is to clearly differentiate what is reusable from what is domain-specific. Patterns (generic rules) have been introduced to capitalize on reusable descriptions, and mainly on the action part. Pattern adaptations (specific rules) derived from patterns, specify the context, conditions, and possibly new actions. Figure 3 depicts these two levels: the "callbackRule_IOData_To_IOData_Inheritance" specific rule, which declares here the context to open an HMI for an IOData inheritance, inherits from a "callbackRule" pattern containing the recurring solution to open a new HMI. The context is identified by the "rule.context" link from the rule to the involved model elements. In practice, when designing a new HMI with this rule strategy, the designer has only to declare pattern adaptations with specific rules. No code is to be written. During the production, MDSoFa consumes specific rules which give sufficient descriptions to specialize patterns to the context. Template-based code in the action part of the pattern is transformed into a contextual code.

The paradigm shift implies a developer activity shift from development (of code) to design (of rules). The main advantage is that the reliability of the generated code replaces the error-prone programming activity. Another interest is maintenance: when common HMI behavior changes (located in the patterns), specifications are not affected; a simple re-generation is needed. However, the main drawback is that declaring specification rules may in turn become a nuisance.

An initial task is to write the patterns containing the generic HMI behavior (completely from scratch, FC = 13 days) or to proceed to a refactoring of the framework defined in the previous strategy (FC = 10 days + 3 days of refactoring). The specification time for each HMI with specification rule is divided by 3 (UC = 0.33).

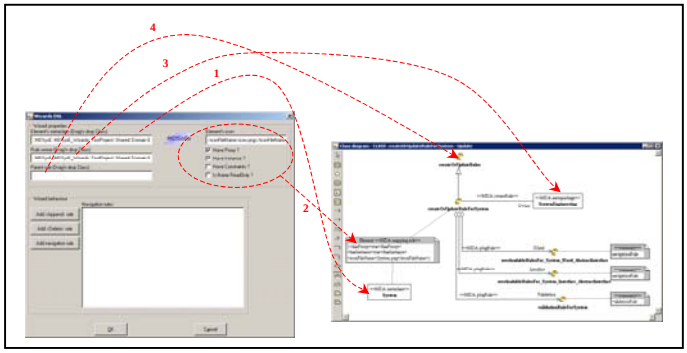

Figure 4. DSL to Rule Specification mapping

3.4 Automation with DSLs

The last strategy is to ease the specification rule description with DSLs. A DSL captures all and only necessarily criteria to produce and update the specific rules as described in the previous section. The strength of this strategy is to reduce again the specification time (UC = 0.125), with more safety for the designer to express the specification. Moreover, dialog with the designer is simplified because internal machinery descriptions disappear and are replaced with human-comprehensive words. However, an initial task is needed to code the DSL HMI based on the previous strategy (FC = 13 days for FC for the rule strategy + 3 days to write the DSL HMI).

As shown in Figure 4, there is a continuity between the rule and DSL strategies because a DSL is mapped into one or more rules. It may be considered relevant to directly have a [DSL to code] transformation and not a [DSL to Rule] transformation followed

by a [Rule to code] transformation. In fact, the rule language, which is a lower level DSL, is a pivot language aiming at capitalizing and assembling patterns with uniformity.

4. Case Study Productivity Analysis

An issue for a project leader is to apply the best development strategy at a given time, in given project conditions (due time, quality criteria, reliability of the solution, technology, team development experience). Regarding the above presented strategies, it is not straightforward to determine the best strategy for a given project. A major criterion is the number of similar assets to be developed, here HMIs, but it may not be sufficient. The two following tables list the production time for each strategy (Table 2) and compare strategies to deduce the profit point when one strategy becomes more profitable than the others (Table 3). For instance, if the development team has to develop just one HMI, strategy 1 (manual development) is the most profitable; starting from three HMIs, strategies 2 (manual development with a framework) or 3 (automated development with rule) becomes more profitable; starting from fourteen HMIs, strategy 4 (DSL) is always the best strategy.

Table 2. Production Time by Strategy

	VC (= FC + nb of HMIs x UC)						
Nb of HMIs	0 (FC)	1	3	5	7	10	15
1. Manual	0	5.00	15.00	20.00	35.00	50.00	75.00
2.Framework	10	11.00	13.00	15.00	17.00	20.00	25.00
3. Rules	13	13.33	14.00	14.67	15.33	16.33	18.00
4. DSL	16	16.125	16.375	16.625	16.875	17.250	17.875

Table 3. Production Time Comparison and Profit Point

	PP = Profit Point is when Rate (i) / (j) ≥ 1						
	Rate (i) / (j) = time for strategy i / time for strategy j						
nb of HMIs	1	3	5	7	10	15	PP
(1) / (2)	0.45	1.15	1.67	2.06	2.50	3.00	3
(1) / (3)	0.38	1.07	1.70	2.28	3.06	4.17	3
(1) / (4)	0.31	0.92	1.50	2.07	2.90	4.2	4
(2) / (3)	0.83	0.93	1.02	1.11	1.22	1.39	5
(2) / (4)	0.68	0.79	0.90	1.01	1.16	1.40	7
(3) / (4)	0.83	0.85	0.88	0.91	0.95	0.95	14

As mentioned above, this case study has been tested on 52 HMIs, and required 150 specification rules (an average of 2.88 rules per HMI). According to Table 2, this gives: 260 days for strategy 1, 62 days for strategy 2, 30 days for strategy 3, and 23 days for strategy 4. So, the software factories technique (strategies 3 and even more strategy 4) is the most efficient and profitable strategy for development and maintenance of MDSysE HMIs. The ROI[1] (Return On Investment) is even higher in case the number of HMIs increases.

Table 2 lists the production costs for HMIs produced in series. However, we must be aware that this does not reflect the complete production cost. HMIs constitute altogether one MDSysE product, called the MDSysE Wizard product. A production has to differentiate 1) the unit cost of one asset, here an HMI, and 2) the product cost, *i.e.* the production of all HMIs contained in the MDSysE Wizard product. The product cost is the sum of (i) VC (the sum of the cost unit of each asset) and all activities related to build a product (requirements management, analysis, testing, integration, documentation; tooling aspect, etc.), and (ii) the tailoring cost of each derived MDSysE Wizard product. This product cost study is not covered by this paper. On this subject, the reader can refer to [5].

[1] ROI compares cost savings on cost of investment. When the production cost, related to the cost of investment, decreases, the ROI increases.

5. Techniques and Practices

Deducing general conclusions on productivity and ROI improvement only from this kind of case study would be premature. For this reason, this section extends this productivity improvement study in four points covering technical as well as non-technical aspects of the production process: (i) software factories objectives must be consistent with the project strategy to improve the productivity; (ii) productivity measures must be reliable in order to optimize the production; (iii) assets produced and consumed during the production process must be organized for efficient and long time developments, which implies a clear software factory asset architecture; (iv) and at last, project teams must improve their software factories awareness and practice in order to obtain a better production maturity level.

5.1 Consistent Productivity Objectives

Development with software factories is integrated in a consistent project strategy: product strategy, synergy between customer expectations, technology, standard strategy, architecture strategies, legacy systems, and handcrafted practices. Then, technological objectives of software of factories must meet project objectives.

The main objectives of software factories are threefold. (i) *Industrialization*. Software factories aim at industrializing software production by reducing software complexity and favoring software description (abstract raw or complicated software aspects, reduce tedious and rote tasks). As shown in the case study, rule-based formalism and DSL are examples of means toward easing problem expression; irrelevant implementation details are masked. Industrialization is the result of systematic production from abstract information of assets, such as code generation or automated production of different kinds of assets (models, model transformations, model views, configuration files, documentation). The maximization of industrialization consists in automatically building as far as possible a complete modeling chain from requirements down to the packaging of the produced assets. (ii) *Capitalization*. Software factories are a vector of capitalization, especially for large-scale developments implying multiple pieces of sharp domain expertise (business, technical, or process). The stake is to capture and reuse expertise expressed with patterns in different contexts during software mass-production. In the case study, the rule-based strategy has shown an example of clear separation between what is reusable from what is specific; the interest is the HMI patterns can be reused in contexts different from MDSysE. (iii) *Flexibility*. Developers have to produce multiple products, on schedule and budget, and face the specification, methodology, standard and technology evolutions. The stake is twofold: produce variant products from a

same product family, and have durable assets. Regarding the productivity, software factories are useful to rebuild assets when patterns or specification parts change and to have ever up-to-date product versions.

When relying on software factory techniques, the project must be managed so that its technical objectives and allocated resources are consistent with these software factory objectives: a) For the production process maturity, task organization is product-oriented, process activities must use techniques and practices in line with the software factories objectives, while architecture activities must define clear software factory chains and clear asset architectures; b) Project management must promote software factory awareness among the project team. For instance, project members can make the paradigm shift progressively with, at first, simple applications, and then, over the months, begin to develop catalogs of patterns, languages and tools consumed and produced by the software factories tools.

5.2 Productivity Measure

The interest of a productivity measure is to appreciate the efficiency of a production process and to determine how to improve it. The metric formulas used in the case study are inspired from economics. The issue is to find when the minimal cost is reached to realize the optimum profit. But the cost function is project-specific and depends on the kind of produced assets. For instance, the initial cost in building a model refinement framework is obviously different from an HMI framework; a learning team is less productive than an experimented one. Moreover, these metric formulas address only a kind of productivity measure. For instance, they are unsuitable for the performance analysis of pattern used during the production, because the way to realize the measure and their objectives differ. On this purpose, it becomes useful to elaborate a flexible catalog of analysis criteria meeting the different kinds of measure. The interest is to possess criteria to investigate and identify the points in the production process where productivity can be improved.

5.2.1 Quality catalog for production optimization

Even if software factories are identified as a solution to improve software productivity, it remains essential to ever go further in productivity improvement: for instance, what are the assets (languages, frameworks, and tools), the patterns, the manual and automated tasks reducing productivity?; what would be the cost and the profit to automate a manual task?; is the production scheduling optimal? These kinds of questions should emerge, especially when unexpected facts are encountered at production time. For instance, efficient patterns for small models may become unusable for large ones; over successive productions, dedicated languages may become inconsistent; growing volume of generated code may worsen performance of the user tools. In other words, development teams may face problems of scalability, consistency, and performance, *i.e.* problems of software quality. To address this issue, common criteria must be defined to identify optimization points. In this perspective, the stake is to establish a quality catalog for software factories, capitalizing project experiences.

To organize this catalog, we propose to adopt the three-level organization defined in the Quality of Services UML profile [17]: category, characteristic, and dimension. A category is used to classify characteristics by interest groups. A characteristic represents a non-functional aspect, such as performance, or scalability. A dimension is a quantifiable criterion of a characteristic, such as asset production time.

Table 4. Example of Quality Catalog

Category	Characteristics	Dimension
Pattern	Throughput	Mean Time of generation
Asset	Availability	Mean Time To Rebuild Asset

The table above exemplifies a reduced catalog of two characteristics that can be put in relation, Throughput and Availability. For the first category, Pattern, the “Mean Time of generation” dimension of the Throughput characteristics determines how much time is necessary to apply a pattern in a given context. This allows determining the most time-consuming patterns, which need improvement. For the second category, Asset, the “Mean Time To Rebuild Asset” dimension on the Availability characteristics indicates the mean time to rebuild an asset. Crossing pattern throughput and asset availability characteristics can determine why an asset generation is long; for instance, investigations can reveal that a pattern needs improvement because it does not respect scalability expectations, or that a metamodel needs to be redesigned, or that it is due to the MDD platform configuration. Other characteristics should be: coherence, latency, efficiency, demand, dependability, scalability [17]; portability (adaptability, installability, replaceability), maintainability (analyzability, changeability, testability, stability) [12].

From a process viewpoint, the main activities to put this quality catalog in practice are the following: 1) Define the quality catalog for software factories; 2) Apply it during successive productions; 3) If a problem occurs, find and analyze the defaults and then correct or mitigate them. This method requires regular statistics updates by auditing the time spent for production with tools and by the project team members to produce the expected assets: time for simple to complex language / framework / tool development, learning curve to create a new DSL or to learn an existing DSL, pattern development and refactoring time, or regression impact of pattern and correction time. Defaults and improvement analysis generally require several iterations, small iterations for pattern development, or large iterations for product development. Corrections are the way to elicit new techniques and best practices. Mitigation means are used when a default cannot be solved directly but by a work around. This productivity assessment with a quality catalog formalizes what is generally intuitive and guides the default tracking thanks to a characteristic / dimension structure. Capitalizing on this catalog is essential.

5.2.2 Applying analysis criteria

In possession of this catalog, it remains to know how to use it in the software factories context. The person responsible of the productivity analysis establishes formulas based on the quality catalog and conditions under which they can be applied. But this person needs to proceed with precaution. For their reliability, project productivity comparisons must always be based on constant formulas. For instance, productivity can be based on the complete development lifecycle time, from inception to deployment, so that productivity with and without software factories can be compared. Rates are falsely evaluated when sets

and kinds of measured activities vary. In the case study, FC and UC figures represent the HMI development time. In practice, analysis, design and development times are generally merged, and only a repetitive development is able to isolate the actual development time. On the other hand, reliability of the productivity figures depends also on the process maturity. For iterative development processes with software factories, during the learning phase, especially for developments with reflexivity, the separation between what is reusable and what is specific, between the different phases of architecture, analysis, design, or development, is not so obvious. Here again, only repetitive tasks ensure figure reliability. Besides, the objective of the usage of a measure must be clarified. For instance, in some cases, it may be useful to separate some activities, such as the specification activity from the generation activity. However, this measure may become too detailed: if it is useful for tool performance measure, it is therefore inappropriate for an asset production measure. If statistics and regular audits accurate measure, it always remains necessary to know how to use and interpret them.

5.3 Asset Architecture

After the measure, architecture is a key dimension to be investigated to industrialize production of complex software systems. The purpose of architecture is to (i) anticipate misconceptions as soon as possible to avoid growing costs during the development and maintenance phases, and (ii) enable efficient, durable and large-scale software developments. In this paper, we just focus on a central point of architecture: the asset architecture.

5.3.1 Asset architecture

Consumption / production relationships between assets define the backbone of the production and are the support to define key features of a software factory, such as building blocks and asset lifecycle. An asset architecture is driven by production objectives (produce final assets from intermediary ones), but also by constraints and expected qualities. Constraints are expressed by contracts, a set of pre- and post-conditions on assets. An example of pre-condition is “the asset x can be produced when its input assets are available”. Subsection 5.2.1 lists a set of software qualities. For meeting availability expectations, for instance, the number of assets and relationships between assets is minimal.

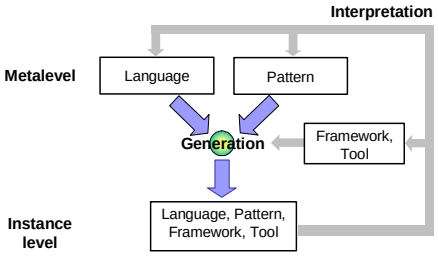

Figure 5. Asset Lifecycle

Regarding the asset lifecycle with software factories, output assets are instances of input assets. They are created or updated during the generation step. In input, languages respect syntax and semantics, *e.g.* they are expressed in MOF or with a DSL. The generation uses a set of patterns for the best reusability. Generation results are languages, patterns, frameworks, and tools. An input language has output languages for language mapping or language variants, such as variability of MDSysE metamodels. Patterns can be specialized automatically from patterns. For instance, for MDSysE, some MDSysE model checking patterns are produced from MDSysE metamodels and context-independent model checking patterns. A set of patterns can instantiate a framework which can be enriched and adapted by plug-ins. HMI production is an example of generated tool, executable in a modeling environment. MDSofa, instance of MDSoFa, is an example of tool generating tool.

A key point for managing complex chain of assets is uniformity: uniformity of language, pattern, but also of structure, service, protocol, and process. Uniformity eases reusability, integration, evolution, and avoids managing a large size of heterogeneous assets. Shared assets are enriched more quickly by diverse experiences and hence more tested. Moreover, this implies to find consensus on divergent solutions.

Uniformity favors also reflexivity, with its advantages and its drawbacks. A special case is the factory retooling when a new version has side effects on itself: some updates have few impacts while others are critical, especially at the core level. For retooling, incremental validations on instances of the factory tool, than directly on the tool itself, prevent major failures. For instance, MDSysE has been a testing platform for MDSoFa. Moreover, experiences with MDSysE have been immediately at the disposal of MDSoFa because MDSoFa and MDSysE share the same execution framework and a common pattern library. For the execution framework, regressions are quickly visible; for the patterns, regressions are detected at the next generation of MDSoFa.

5.3.2 Complexity reduction by abstraction

Describe asset architecture becomes essential to manage complex chains of production. It ensures their consistency and optimization. Fortunately, software factories and final users do not see the successive asset transformations but only the final assets. Regarding the software factory environment, it has to provide the users with easy and efficient tools for industrializing software production. In this direction, we believe that high-level languages, addressed by DSLs, must abstract as far as possible language, framework, tool, behavior, and platform considerations, in order to reduce software complexity. This abstraction is captured by criteria giving sufficient information to deduce the solution during the asset instantiation, behavior production, assembly, and packaging steps. In the case study, strategy 4 is a simple example showing how productivity is improved by DSL, where internal representation, mechanisms, translation toward the rule-based formalism, and framework usage are completely hidden. If the goal is to reduce development and maintenance time, the issue is to maintain the integrity of the produced assets, altogether compliant both with the specification defined at the user level and the production description defined at the metalevel. At the instance level, typically when the user modifies one criterion, or at the metalevel when solution implementation changes (*e.g.* variation on metamodels, interfaces, algorithms), all produced asset instances must be maintained in consequence.

5.3.3 Scheduling

The asset architecture can be optimized whereas the production time remains long. This may come from a scheduling problem to build the expected assets. Actually, with exactly the same

elements, time production may completely change in function of the scheduling strategy. To take an analogy with classical developments, a C++ makefile generates only the necessary object and executable files. A makefile takes into account incremental file updates. The problem is the same for software factories but with a higher complexity. All along the development process, assets evolve at different rhythms, and, at a given time, all assets are not necessarily aligned. Then, to improve the production logic, it is necessary to: (i) optimize production branches where some assets can be built in parallel, while others are built in series; this means also clarifying conditions of synchronization; (ii) favor partial builds, especially when generations are long, and avoid useless asset rebuilds when assets are unchanged; (iii) have checking points before continuing the production process; for instance, for a product-line, validate the baseline before building the derived products; (iv) use configuration management to have consistent assets and avoid useless rebuilds.

5.4 Improving Team Maturity Level

Now, beyond technical aspects, human factor remains a determining criterion for industrializing with efficiency development of complex software systems. Project members using software factories must be convinced they can do their work better, quicker, and safer. Tool maturity is a first acceptation criterion. However, they must integrate this technology and be software factory aware. By analogy, the way of thinking, techniques and practices in Java are different from those in LISP. The way of using software factories, comprehension of related techniques and practices is central to manage developments with durability and efficiency. Project members can start with reduced developments to elicit for instance first patterns and DSLs, and to figure out production process and asset organization to be settled. The next stage opens the way of intensive, large-scale and durable developments meeting quality criteria with the foundation of core technology, asset architecture, process engineering and measure criteria. All along this period, a close coaching with an expert, implied in real situations, is irreplaceable to realize this paradigm shift. The team work must be also cooperative to better rationalize and share the same languages, techniques and practices. The team has to promote agility (understandability, accuracy, consistency, positive value, and simplicity) [1] and reactivity during the developments. The team must be also disciplined to keep all kinds of asset and processes managed.

6. Further Work

Two kinds of activities defined the future work of AED related to the software factories technique. First, for the MDD tools, instances of MDSoFa, the objective is to constantly increase the proportion of tools automatically built, as long as the ROI is insured. At the factory level with MDSoFa, efforts have to be focused on two points. (i) All product line aspects related to MDD development need to be taken into account. The different kinds of variability that must be addressed are: variability of MDD methodology definition and tools generation (*e.g.* MDSysE, MDSysE for Business Unit X, MDSysE for Project A, etc.), variability of infrastructure language (*e.g.* UML™1.5, UML™2.0, SysML™, etc.), variability of target modeling platform (*e.g.* Rational Software Architect, Objecteering/UML, iLogix Rhapsody). Regarding this last concern, MDSoFA targets at this stage two platforms, including one only for prototyping. A medium-term objective is to fully support production to an additional modeling platform. (ii) MDSoFa has been designed to address any MDD methodological approach, either of a relatively small or medium complexity (*e.g.* MDSysE), or based on a conceptual framework of a large scope (*e.g.* DoDAF). So, MDSoFA architecture must evolve to accept larger methodologies formalized with voluminous metamodels.

7. Conclusion

MDD projects continuously need to improve their productivity rates to produce software on budget and schedule meeting quality criteria. Only a paradigm shift with new generation of tools is able to meet this expectation. This paper has successively examined productivity of four strategies to produce HMIs: manual code production, framework usage, production with a software factory tool, MDSoFa, and production with a HMI DSL, revealing software factories usage with DSLs is the most efficient strategy. However, this statement is not sufficient: projects need techniques and practices to make this paradigm shift a success. Firstly, measure is a means to identify where, when and how productivity with the usage of software factories can be improved. In this perspective, we proposed the definition of a software factory quality catalog for productivity assessment that can be reused and enriched by different projects. Secondly, for large-scale developments, asset architecture description is central to define consistent and optimal asset lifecycles from the initial toward the final assets. For a better reusability and integration, we encouraged to build assets with uniformity. For improving productivity and guaranteeing integrity of assets, we recommend using high-level languages with DSLs. However, human dimension remains a key factor for software factories adoption. Using mature factory tools, project team members must be software factory aware. On that purpose, coaching by a software factory expert and sharing the same languages, techniques, practices, and values (agility for efficiency, and discipline for managed processes) are the best way to improve the maturity level of the project team.

Regarding the AED tooling, MDSoFa and MDSysE are the result of an extensive work that spread over a period of three years, mainly in the frame of MODELWARE which is co-funded by the European Commission under the “Information Society Technologies” Sixth Framework Programme. As such, MDSoFa has reached the quality of an advanced prototype, has proved its relevance and ROI, and is used to generate most of the MDSysE toolset used in several THALES real industrial cases. The next phase is for MDSoFa to reach an industrial quality level.

8. ACKNOWLEDGMENTS

We thank Serge Salicki, head of the Architecture and Engineering Department of the THALES Software Research Group, the members of the Architecture and Engineering Department, and especially Stéphane Bonnet.

9. REFERENCES

[1] Ambler, S., *Agile modelling, Effective Practices for eXtreme Programming and the Unified Process*, Wiley, 2002.

[2] Bass, L., Clements, P. Kazman, R., *Software architecture in practice*, SEI Series in Software Engineering, 1998

[3] Cook, S., and Kent, S. *The Tool Factory*, OOPSLA 2003 "Generative Techniques in the context of Model Driven Architecture" workshop. October 27, 2003.

[4] Clark, T., Evans, A., Sammut, P., Willans, J. *Applied Metamodelling. A foundation for Language Driven Development*. Version 0.1. Xactium, 2004.

[5] Clements, P.C., McGregor, J.D., and Cohen, S.G. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Technical Report CMU/SEI-2005-TR-003 ESC-TR-2005-03, Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, February, 2005.

[6] Czarnecki, K., and Eisenecker, U.W. *Generative Programming*, Addison-Wesley, 2000.

[7] Evans, E., *Domain-Driven Design, Tackling Complexity in the Heart of Software*, Addison-Wesley, 2004.

[8] Exertier, D., and Normand, V. MDSysE: A Model-Driven Systems Engineering Approach at Thales. Incose, 2-4 November, 2004.

[9] Greenfield, J., and Short, K. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*, OOPSLA 2003 "Generative Techniques in the context of Model Driven Architecture" workshop. October 27, 2003.

[10] Greenfield, J., Short, K., Cook, S., and Kent, S., *Software Factories, Assembling applications with Patterns, Models, Framework, and Tools*, Wiley, 2004.

[11] IEEE Architecture Working Group, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471-2000, IEEE, 2000.

[12] ISO/IEC TR 9126 (1991). International Organization for Standardization, Geneva. An international standard for quality factors

[13] Langlois, B., Exertier, D., *MDSoFa: a Model-Driven Software Factory*, OOPSLA 2004, MDSD Workshop. October 25, 2004.

[14] Normand, V., Exertier, D. *Model-Driven Systems Engineering: SysML & the MDSysE Approach at Thales*. Ecole d'été CEA-ENSIETA, Brest, France, September, 2004.

[15] OMG/RFP. *Meta Object Facility (MOF) 2.0 Core Specification*, OMG Adopted Specification, ptc/03-10-04, April 6th, 2003

[16] OMG/RFP. *MOF 2.0, Query / Views / Transformation*, Revised Submission, ad/2002-04-10, Version 1.0, 2004/04, QVT-Merge Group.

[17] OMG. *UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*. Final adopted specification, ptc/04-09-01. September 16, 2004.

[18] OMG. *Systems Modelling Language: SysML*. Version 0.3 (first draft). January 12, 2004.