HAL
open science

A model-driven approach for embedded system prototyping and design

Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa, Dominique Rieu

To cite this version:

Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa, Dominique Rieu. A model-driven approach for embedded system prototyping and design. IEEE International Symposium on Rapid System Prototyping (RSP'14) (part of ESWEEK'14), Oct 2014, New Delhi, India. pp.23 - 29, 10.1109/RSP.2014.6966688 . hal-01457187

HAL Id: hal-01457187
https://hal.science/hal-01457187v1

Submitted on 10 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Model-Driven Approach for Embedded System Prototyping and Design

Nicolas Hili, Christian Fabre
Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LETI, MINATEC Campus, F-38054 Grenoble, France
Email: {nicolas.hili, christian.fabre1}@cea.fr

Sophie Dupuy-Chessa, Dominique Rieu
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
Email: {sophie.dupuy, dominique.rieu}@imag.fr

***Abstract*—Embedded System (ES) development complexity is increasing. This increase has several cumulative sources: some are directly related to constraints on the ES themselves (dependability, compute intensive, resource constraints) while other sources are related to the industrial context of their development (fast prototyping, early validation, parallelization of developments). Although several Model-Driven Engineering (MDE) processes have been proposed for ES development, most of them are not completely formalized. This has several drawbacks that prevent their use in prototyping where iterations need to be short and focused. Incomplete formalized processes tend to be sidestepped in these situations where quick results are expected to be obtained with limited effort.**

In this paper we propose a MDE-based process for ES development. This process precisely defines the development tasks and their impact on the models throughout development. In particular we define iterations *width* and *depth* for the process that allow for a fined-grained and consistent planning of developments. The short and well defined iterations characterized by the process reduce the gap between rapid prototyping, ad-hoc methods and regular development processes.

Keywords*—*Embedded System, Process, Iteration, Prototyping

I. Introduction

Over the increasing complexity of Embedded System (ES) development, more and more developers turn to Model-Driven Engineering (MDE) to foster ES design [1], [2]. MDE promotes the use of models to abstract all or part of the system being considered. It permits to visualize ES through different viewpoints and extract essential properties to its modeling. Additionally, MDE provides techniques to perform from executable models a number of static analyses, performance analyses, simulations and so on. It also offers a way to separate design from implementation which is particularly well appreciated when designing ES composed of an application hosted by a platform.

Complexity of ES design is emphasized by the needs of early error detection and regular feedbacks to guaranty that the system is conform to its specifications. One of existing techniques to ensure specification validation is system prototyping [3]. Prototypes permits to assess the fulfillment of the specifications while giving a concrete view of the system under development and the design process as well [4], [5]. However, prototyping could be incompatible with model-based methodology where time spent to create the models at several abstraction levels might be considerable. Several aspects of MDE contribute to reduce the gap between model-based and prototyping-based methodologies. The use of model transformation permits to transform abstract models to more concrete ones which ensures the following of the process in a continuous way and the use of code generation techniques ensures to generate high quality and robustness application code and reduce time spent to debug and validate the produced software and hardware code.

Despite the acceptance of MDE for ES design, few of ES designers exploit it to formalize and automate ES processes. They are often defined *ad-hoc* and barely formalized in terms of models. Formalizing them could plan and foster their executions, with high confidence and help to reduce the gap between MDE and prototyping. This paper presents a model-based process for embedded systems called $\langle \mathsf{HOE} \rangle^2$ [6]. This process is formalized at very fine-grain thanks to activity and behavior diagrams so it exhibits development tasks and their impact on the models produced during the development. The fine-grained definition of the process allows a project manager to organize quick iterations to enhance the prototyping.

This paper is structured as follow. Section II is aimed to give a comprehensive view of the state of the art; section III introduces $\langle \mathsf{HOE} \rangle^2$, a model-based process for ES development while section IV shows how it is possible to reduce the gap between MDE and prototyping; section V presents the benefits in terms of project monitoring and development planning; we present the development of an embedded system case study in section VI to show the impact on a prototyping-based development compared to a regular one; finally, we summarize our results and give future directions in section VII.

II. State of the Art

This section is structured through two axes. The first one addresses the process modeling languages an their specificities. The second axe addresses model-based approaches for embedded system design.

A. Process Modeling Languages

In software engineering, a number of languages were proposed to model processes [7], [8]. The undoubtedly well-known is the Unified Modeling Language (UML) [9]. UML provides a few concepts that can be used to model processes. Those are assembled inside the *Activities* package. It proposes several concepts like *Activity*, *ActivityNode*, *flows* (*ControlFlow* and *ObjectFlow*) to model product-oriented and

activity-oriented flows. Among another well known processes, we can cite Business Process Model and Notation (BPMN) Metamodel [10] and Software & Systems Process Engineering Meta-model (SPEM) [11]. BPMN is a process modeling language focusing on a clear graphical notation understandable by all the participants of a process, from the analyst, to the designer, including end users. SPEM is a process modeling language based on a MOF 2.0-based metamodel that essentially focuses on software systems processes.

A number of model processes was proposed to address industrial concerns related to rapid development and prototyping. The Spiral model [12] and the Rapid Application Development (RAD) [13] are such process models. They focus on short iterations and prototyping development. Prototyping helps to reduce risk and improve quality of the desired system. Another approaches for rapid development are Rational Unified Process (Rational Unified Process) [14] and Unified Software Development Process (USDP) [15]. They are usecase-driven processes where usecases are prioritized and define iterations in order to cover the highest risks first. Each iteration ends with the production of prototypes.

B. Model-Based Embedded System Design & Prototyping

A number of system and embedded system design methods address prototyping issue though the use of models and incremental processes. ACCORD/UML [4], [16] is an embedded system design method applied to the automotive area. It relies on an extended version of the UML language based on the definition of an UML profile to enable real-time embedded system development. The ACCORD/UML process focuses on three phases: analysis, design and implementation, it is defined as iterative and continuous with possible backwards. The approach promotes an iterative development to build a system by increments. In [17], the authors give an overview of the ACCORD/UML using the modeling artifacts defined in the SPEM profile. They define three participants, among them the *prototyper*. A prototyping phase allows the prototyper to build a prototype model. While the process is rather textually described, their is no real use of executable process modeling language to enact the process, establish a project monitoring and reduce the gap between MDE and prototyping.

MOPCOM [18], [19] is a co-development method for system on chip design. It is based on the Model-Driven Architecture (MDA) techniques, especially on the separation between platform independent and specific models. The MOPCOM language is based on both MARTE [20] and SysML [21] languages. On process side, MOPCOM proposes a top-down flow through three abstraction levels. At the end, software and hardware synthetizable code are generated. We can compare the MOPCOM process to the Y life cycle from Capretz [22]. More precisely, the process is built upon three successive Y life cycles enabling parallelism. However, the process is pretty unclear and imprecise, no participant is identified, neither the possibility of iterations, requirement-guided development, etc.

Behavior, Interaction, Priority (BIP) [23] addresses the design of system based on model and component oriented approach. It permits to build hierarchical and composite models in which each atomic component is considered in terms of behavior and interactions with other components. The formal language ensures that the design and assembly of atomic components is correct-by-construction. BIP embeds a product and activity-oriented process. In [23], the process is illustrated by a flow diagram with input and output models, iterations and activities, but is not formalized in an executable process modeling language.

Embedded system model-based approaches usually propose flows that cover all or part of the life-cycle of the system under consideration. Those flows are composed of many steps involving modeling at different abstraction levels and occasionally propose consistent rules to go down throughout the development process. Some of them explicitly propose prototyping activities, and process automation through consistent transformation rules to go down throughout the development process and thus foster the prototyping. However, none of them proposes a clear formalization, preventing a method user to know when the model under development at one abstraction level is ripened enough to initialize downstream activities on the flow. Such formalization would leverage the vagueness and imprecision around processes and would allow a project manager to have a big picture of project planing and task organization adopt a prototyping strategy.

III. The $\langle \mathsf{HOE} \rangle^2$ Approach

In this section, we present the $\langle \mathsf{HOE} \rangle^2$ approach. $\langle \mathsf{HOE} \rangle^2$ is a model-based approach previously proposed in [6] and stands for *Highly Heterogeneous Object-Oriented Efficient Engineering*. $\langle \mathsf{HOE} \rangle^2$ embeds both a collaborative process and a common language for application and platform based on useful concepts of *objects*, *association*, *state machines* and *message passing*. Fig. 1 gives a partial view of the $\langle \mathsf{HOE} \rangle^2$ process. Not discussed here, but the $\langle \mathsf{HOE} \rangle^2$ process is platform-based design [24] as the platform design flow is composed of the same four phases as the application design one. This aspect allows designers to iterate over several stacked platforms, giving to the process its *fractal* nature [25].

A. Requirement Phase

The *requirement* phase permits to formalize informal *specifications* of the system. These specifications describe functional as well as technical requirements and serve as inputs to the two development flows: functional requirements for the application development flow and technical requirements for the platform one. During this phase, the requirement is formalized in terms of actor and usecases organized according to their causality (i.e. their importance regarding the system under development – possible values are *primary* or *secondary*) and of scenarios which are prioritized through their nature (*nominal* or *error*) and their importance. Each scenario represents the smallest-grain of the requirement definition. All the further activities will endeavor to fulfill the requirements by keeping consistency and compliancy with the scenarios of the requirement model.

B. Analysis Phase

During the first phase, a black box system (an application for the first flow and a platform for the second one) is defined in terms of requirements. During the *analysis* phase, this system is opened and detailed in terms of *communicating objects* and *object behaviors*. Behaviors are captured with state

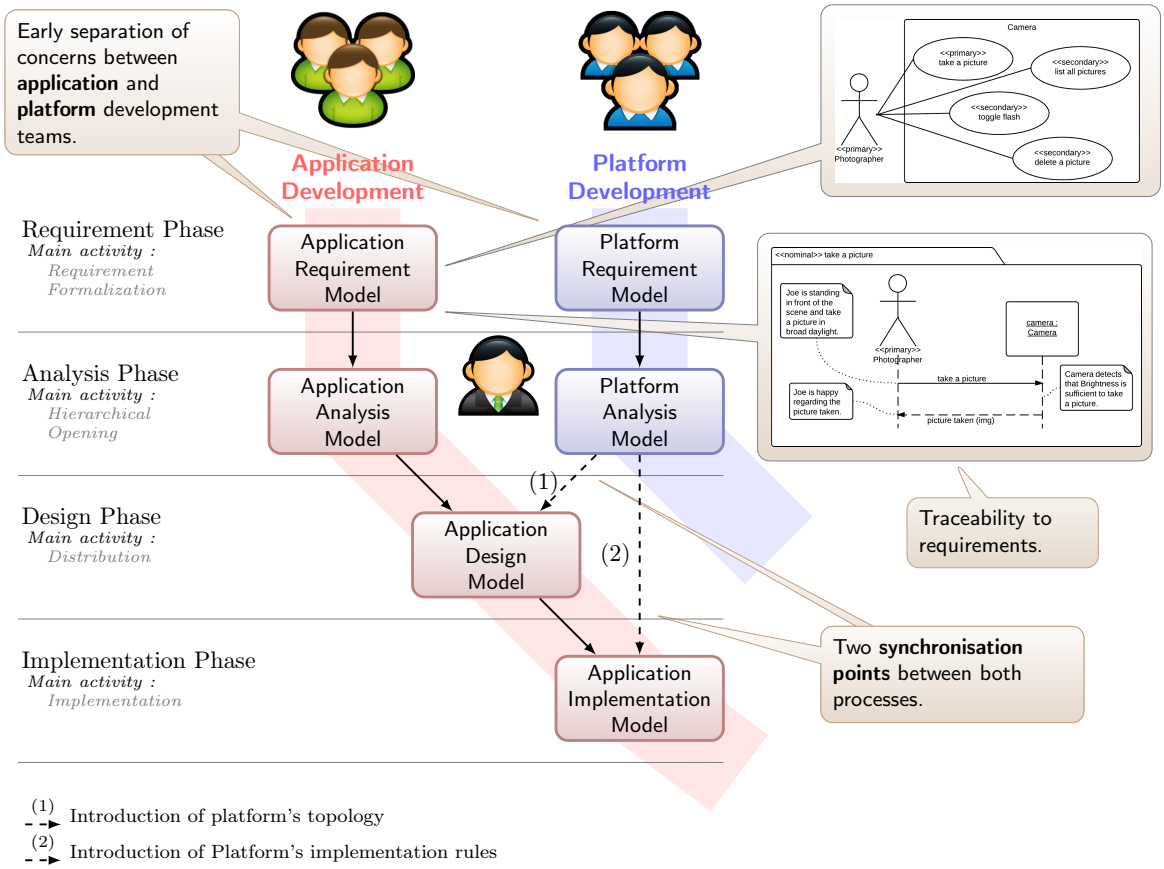

Fig. 1. ⟨HOE⟩^{2}: a Collaborative Top-Down Process for Embedded System Design

machines. Both development teams can perform the *hierarchical opening* activity in a concurrent way. Each opening activity aims at fulfilling the requirement formalized in the above requirement phase.

C. Design Phase

During the *design* phase, a first topology of the platform is introduced in the application development flow. This topology is limited to the definition of the *world* – or *execution domain* – of the platform. The *distribution* activity involves distributing the application objects over the worlds of the platform. Again, compliancy from the requirements must be ensured during the *distribution* activity.

D. Implementation Phase

The *implementation* phase is the last step before code generation. During this phase, a complete description of the platform feeds the application development flow. It embeds implementation rules to define how objects are concretely implemented on the platform. The *implementation* activity involves defining which implementation rules should be used for each object.

The ⟨HOE⟩^{2} process is distinguishable from other processes and development life cycles by its many aspects. Two concurrent flows allow developers to design the application and the platform that hosts it. Both flows are independent and concurrent but the ⟨HOE⟩^{2} process defines two synchronization points to gradually introduce the platform model inside the application. The *platform analysis* model defines the platform in two steps and offers a smooth implementation of the application on the platform by defining 1) where the object should be located and (*design* phase) and 2) in which way they are concretely implemented (*implementation* phase). This smooth implementation permits a rapid feedback from the application development to the platform one.

In ⟨HOE⟩^{2}, organization of activities is guided by the formalized requirements during the first phase. That means that all the tasks producing new elements on a model must be performed in accordance of one or more scenarios. This aspect addresses prototyping in the sens of a project manager can organize quick iterations to cover only partial widths of the requirements and go down through the four phases in order to design prototypes with minimal effort.

Tool Support

The ⟨HOE⟩^{2} process is partially supported by a dedicated tool named *CanHOE2*, a standalone product based on Eclipse. It addresses several collaborative aspects like model exchange and synchronization, and uses Git as a backend.

IV. Model-Driven Engineering and Prototyping

To reduce the gap between MDE and prototyping, an informal definition of the process as illustrated in fig. 1 is

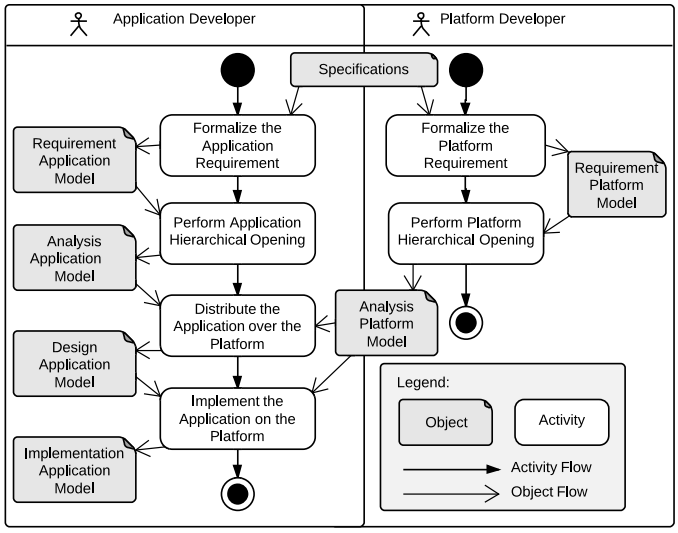

Fig. 2. $\langle$HOE$\rangle^2$ Process Activity Diagram

not sufficient and tends to be sidestepped. Our proposition is based on the adaptation of the UML2 metamodel to give a stronger definition of the modeled requirement-driven process. Other works also deal with requirements [21], [26].

We propose in this section an *activity* package to model activities, tasks and their associated input and output models. Those models need to be defined at the *M2* layer (metamodel) of the OMG pyramid so they can be tightly coupled with the $\langle$HOE$\rangle^2$ product models. To do so, we propose an *Activity* package as an extension of the UML2 activity metamodel.

Fig. 2 models the $\langle$HOE$\rangle^2$ process in terms of *activities* and input / output *objects*. The first input of the process is the informal *specifications* of the system.

The *Activity* package we propose in this section is part of the $\langle$*HOE*$\rangle^2$ *project management* three-layers metamodel illustrated in fig. 8. The two bottom layers are not addressed in this section and will be the subjects of section V. In the activity package, we model the four $\langle$HOE$\rangle^2$ process main activities and detail them into a number of dedicated tasks. For each phase, we list used and produced models, give a state representation of them, detail the requirement to initialize each phase and decompose the activity into small-grained tasks. We further formalize each task in terms of input and output model elements and of pre and postconditions. This section is structured in three parts. First part gives a dynamic vision of the $\langle$HOE$\rangle^2$ models throughout the execution of the process while second and third parts focus on the description of the $\langle$HOE$\rangle^2$ activity package that enables to bridge the gap between prototyping and regular development processes.

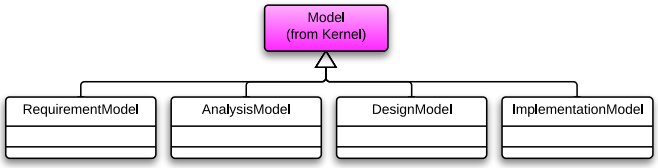

Fig. 3. The $\langle$HOE$\rangle^2$ Four Models

A. The four $\langle$*HOE*$\rangle^2$ models

We propose in this part both a structural and a behavioral modeling of the $\langle$HOE$\rangle^2$ models. Fig. 3 depicts the taxonomy of $\langle$HOE$\rangle^2$ models produced throughout development. Each of them specializes the Model UML meta-class.

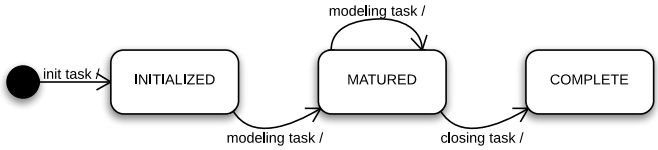

Fig. 4. Common $\langle$HOE$\rangle^2$ model lifecycle

Fig. 4 is a state machine that models the common lifecycle of all $\langle$HOE$\rangle^2$ models. First, the model is INITIALIZED. In the INITIALIZED state, the model is built from upstream models. After *initialization*, a first *modeling* activity has to be performed. After the activity is performed, the model under development evolves into the MATURED state. This intermediate state means that the model is not fully complete, but it reaches a sufficient degree of ripeness to initialize the next phase. In the case of the *implementation* model, code generation can be performed from it. Finally, a model is marked as COMPLETE during the *closing* activity. This activity is performed by the project manager when he or she considers the model under development is complete. Except for the first one, each model may be considered as complete when each modeling activity performed on it contributes to satisfy all the requirements formalized in the upon *requirement model*.

Behavior modeling of $\langle$HOE$\rangle^2$ models brings some benefits. We clearly define when a phase can be initialized from upstream models. The intermediate MATURITY state enables a project manager to initiate the next phase before the model is completed. This facilitates the development of prototypes without covering the whole specified requirements. The second benefit results from the fact that state preconditions can be applied to automate the successive flow of activities with a high confidence in the consistency of it. While modeling activities need to be handworked, *initialization* and *completion* ones can be automated or at least assisted. To enable it, we define constraints as precondition for each initialization and completion activity. Additionally, we define for each initialization activity a set of transformation rules that can be executed to fill up the initialized model with data collected from upstream models. Code generation rules are also defined to produce code from the last model of the $\langle$HOE$\rangle^2$ flow.

B. Activities

To model the $\langle$HOE$\rangle^2$ activities, we chose to extend several concepts defined in the UML *Activities* package from the UML metamodel. Fig. 5 illustrates the fundamental concepts of the activity package at metamodel level (*M2*) we propose. An *activity* is a set of *tasks*: initialization, closing and modeling tasks. We extend the Activity and ActivityNode UML metaclass. A task may also have two constraints: a *precondition* and *postcondition*. The precondition is based on input model states. The postcondition defines into which states must evolve the output models. We chose to model an activity diagram at model level (*M1*) for each phase of the $\langle$HOE$\rangle^2$ process.

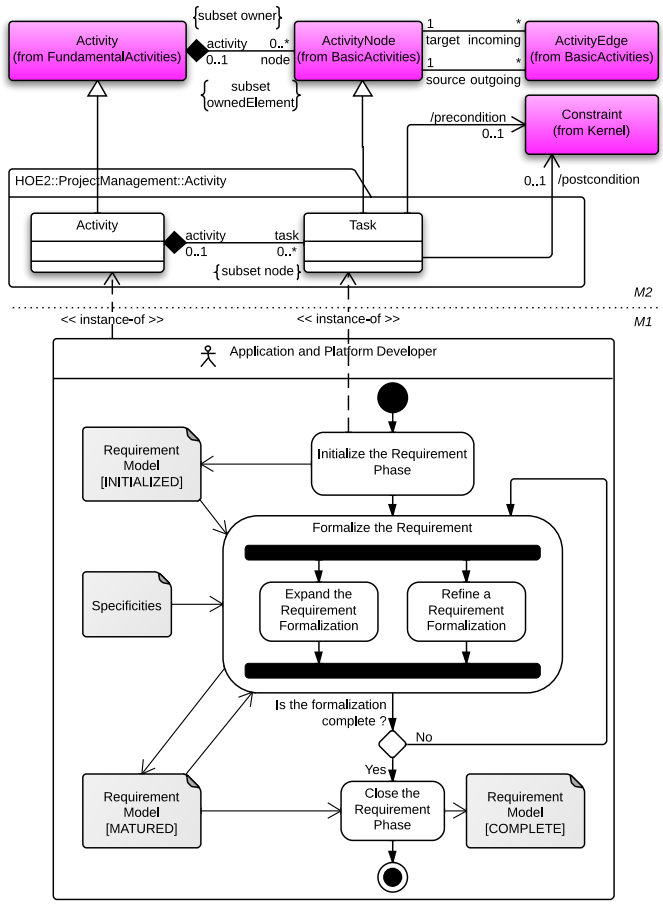

Fig. 5. $\langle$HOE$\rangle^2$ Activity Package & Modeling of the $\langle$HOE$\rangle^2$ Requirement Activity Diagram

C. Tasks

In $\langle$HOE$\rangle^2$, we define a *task* as an atomic modeling activity. Task can be manual, automated or assisted. In Fig. 5 a *task* may define two constraints, one precondition and one postcondition. Additionally, it references a set of input and output models as well as model elements.

Once we have modeled the four models and the four activities, we can further define a taxonomy of tasks for each activity. First, we model the activity with UML activity diagram. In Fig. 2, we detail the *requirement* activity by decomposing it in a number of tasks. Each task can accept inputs and produce outputs. For each input and output, we may define one or more states between square brackets. We decompose the *requirement* activity into three tasks (see Fig. 7).

task 1 (Initialization): First task *initializes* the phase. The output is an INITIALIZED requirement model. It only contains the system without any usecase or actor using the system.

task 2 (Formalization): Second task *formalizes* the requirement. it can be decomposed into two subtasks. The first subtask *expands* the current requirement formalization. It allows a developer to add new usecases, actors, and at least one nominal scenario per newly created usecase. When this subtask is realized for the first time, i.e. the model is INITIALIZED, the developer needs to define at least one primary actor executing one primary usecase, and the usecase must own at least one

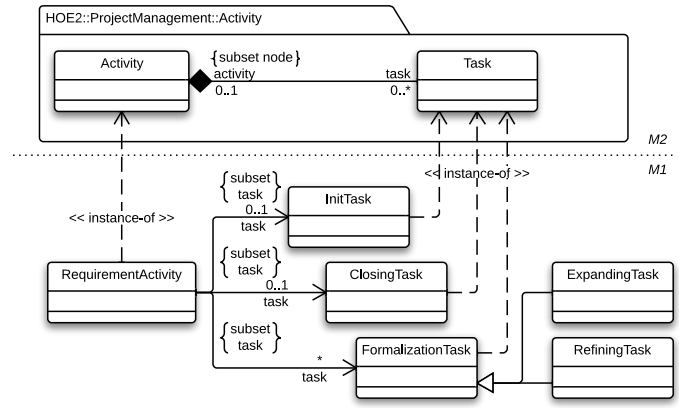

Fig. 6. Requirement tasks in the $\langle$HOE$\rangle^2$ Activity Package

nominal scenario. This is mandatory for the requirement model to evolve into the MATURED state. The second subtask *refines* a usecase by adding new nominal and error scenarios.

task 3 (Closing): Last task *closes* the phase by considering the model as *complete*. The output is a COMPLETE requirement model. It covers the formalization of the whole requirements.

Fig. 6 depicts the five modeled requirements tasks from the *activity* package. Fig. 7 focuses on the definition of the *expanding* task. It precisely references which models and model elements serve as input of the task and which ones should be created during the task. For this purpose, we subset the four references depicted in 5 between the *Task* and both UML *Model* and *Classifier* meta-classes. In the case of the expanding task, as the system is created during the *initialization* task, it cannot be recreated during the *expanding* one.

```
self.system > 0
```

Listing 1. Precondition: the model is in INITIALIZED or MATURED State

```
self.actor->filter (a : Actor | a.causality = 'primary')->
    size () > 0
self.system.usecase->filter (u : Usecase | u.causality = '
    primary')->size () > 0
```

Listing 2. Postcondition: the model is in MATURED State

Additionally, the expanding tasks must verify preconditions and postconditions. Listing 1 and 2 are example of pre and post conditions. The precondition checks if the model is in *initialized* state, by verifying whether the model is composed of a system. No other verification is performed since the model can either be in INITIALIZED as well as MATURED to perform this task. The post condition checks if the system is *matured*.

By refining this work for each task, we get a complete description of the process. This contributes in rising confidence in the consistency of the flow and accelerating the development by preventing a developer to deviate from the process.

V. Project Characteristics & Monitoring

This section focuses on project monitoring concerns. Fig. 8 depicts the $\langle$HOE$\rangle^2$ the *project management* metamodel built

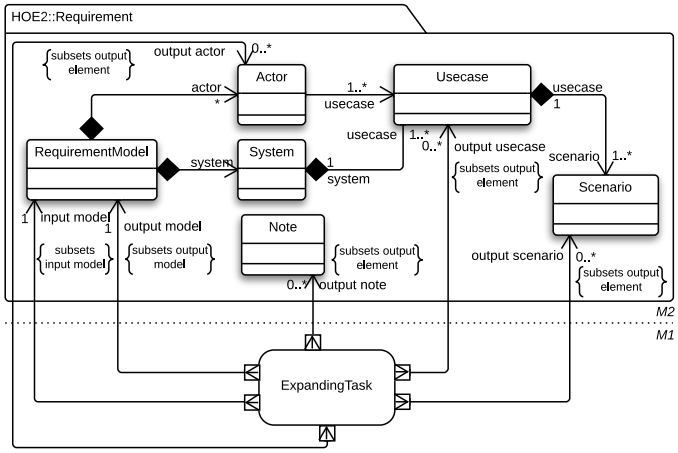

Fig. 7. Expanding task in the ⟨HOE⟩^{2} Activity Package

on three package. The bottom package represents the *participant* to the process. In ⟨HOE⟩^{2}, we identified two participants, the *project manager* and the *developer*. The *project manager* is in charge of creating new tasks and assigning them to a *developer*. The *developer*'s activity consists of realizing tasks assigned by the *project manager*. In ⟨HOE⟩^{2}, the developer can be either the application or the platform developer. The *Project* package contains concepts for project monitoring concerns. We present two concepts. *Project* to precisely characterize the project aimed at designing the system under study and *Iteration* to allow the project manager in charge of the *project* to organize tasks into a set of ordered iterations.

The ability of organizing tasks into iterations confers benefits for project monitoring. It is easy to create a consistent planning of developments. The left side of Fig. 9 shows such an organization. Horizontal axe illustrates the requirement width. It symbolizes the ordered set of scenarios that need to be satisfied during the four phases of the ⟨HOE⟩^{2} process. Vertical axe illustrates the process depth, i.e. the four phases of the ⟨HOE⟩^{2} process. Each square can be colored with proper color related to the iteration. The right side of Fig. 9 shows another task organization that enhances prototyping. Both sides of Fig. 9 illustrate the tradeoff project manager can make to either favor prototyping with quick vertical iterations or large

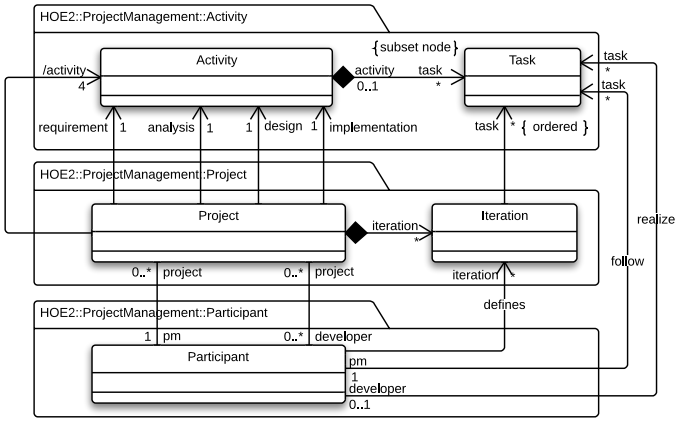

Fig. 8. ⟨HOE⟩^{2} Project Management Metamodel

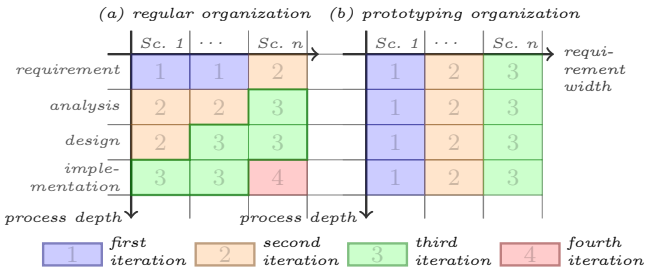

Fig. 9. Consistent Planning of Developments: (a) regular organization and (b) organization enhancing prototyping

TABLE I. FACE TRACKER EMBEDDED SYSTEM USECASES

Id	Name	Causality	Description
1	Subscribe to information flux	Primary	The actor subscribes to the system information flux. The system is able to notify him of any presence or face detection.
2	Switch to automatic mode	Secondary	The actor switches the system to automatic mode.
3	Switch to manual mode	Secondary	The actor switches the system to manual mode.
4	Orientate the camera	Secondary	The actor manually orientates the camera

development system with spread ones.

VI. A FACE TRACKER EMBEDDED SYSTEM CASE STUDY

In this section, we present a *Face Tracker* embedded system. It is embedded on a *Pan & Tilt Camera* platform which includes a number of processors, a Passive InfraRed (PIR) sensor and a bracket on which the camera is attached. Two servomotors are used to orientate the bracket in two axes.

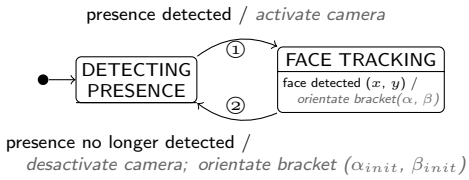

Fig. 10. Case Study: The Face Tracker Behavior

An abstract view of the Face Tracker system behavior is provided Fig. 10. Initially, the bracket and hence the camera, are in an initial position and the camera is inactive. When the PIR sensor detects a presence – See transition ① – the camera becomes active. The video stream is sent to an algorithm that performs face detection. When a face is detected, the system computes the Cartesian coordinates of the face on the picture. Those coordinates are then translated into spatial coordinates and the system orientates the bracket to center the face on the pictures. If the face moves out from the picture, the system stays in the same position. When the PIR sensor does no longer detect a presence – See transition ② – the camera becomes inactive and the system returns to its initial position.

Tab. I and II show fragment of lists of usecase and scenario formalizing the requirements. A strategy favoring prototyping can be chosen by selecting appropriate scenarios and develop through the depth of the process once, and planning another scenario fulfillment during a second iterations. For instance, a project manager can ignore scenarios about switching between

TABLE II. FACE TRACKER EMBEDDED SYSTEM SCENARIOS

Id	Name	CU	Nature	Description
1-1	Presence & face detected notification	1	nominal	The actor subscribes to the system information flux. Both presence and face are detected. The system notifies the presence and the position of the face.
1-2	Presence detected notification	1	nominal	The actor subscribes to the system information flux. Only the presence is detected. The system notifies the presence and the position of the face.
...				
2-1	Automatic mode switching	2	nominal	The actor switches the system from manual mode to automatic one.
2-2	Automatic mode switching	2	error	The actor tries to switch to automatic mode but the system is already in this mode.
...				
4-1	Camera orientation	4	nominal	The system is in manual mode, so the author can orientate the camera.
4-2	Camera orientation	4	error	The system is in automatic mode, so the author cannot orientate the camera.

two modes and prioritize usecases related to face detection and recognition in two first iterations. During the third iteration, it can choose usecase related to camera orientation and finally, the last iteration for manual / automatic mode switching.

VII. CONCLUSION AND FUTURE WORKS

In this paper we present the $\langle \mathsf{HOE} \rangle^2$ process and its formalization language. We show how, by adapting the UML activity metamodel, we are able to define a very small-grained set of activities and tasks with clear inputs and outputs. This allows us to provide concepts for project characterization and monitoring – possibly automatic. Our contribution shows benefits in terms of organization of tasks and produces consistent planning by means of explicit dependencies across tasks. We think that this is not only applicable to regular development but also to prototyping, where shorter cycles and efforts are expected to produce very targeted results. The dedicated tool *CanHOE2* currently supports only the first phase of the process.

Future work will address the formalization of remaining activities and tasks of $\langle \mathsf{HOE} \rangle^2$, the modeling transformations rules for those that can be automated and the extension of *CanHOE2* for their support.

Acknowledgements: This work was partially funded by the EU Artemis JU and the French *Ministère de l'Économie, du Redressement productif et du Numérique* for COPCAMS under GA 332 913. Christian Fabre would like to thanks Bernard Rygaert for introducing him to requirements-driven iteration scheduling for MDE.

REFERENCES

[1] M. Broy, "The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems," *Computer*, vol. 39, no. 10, pp. 72–80, Oct. 2006.

[2] ——, "Seamless model driven systems engineering based on formal models," in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science, K. Breitman and A. Cavalcanti, Eds. Springer Berlin Heidelberg, Dec. 2009, vol. 5885, pp. 1–19.

[3] F. Kordon and Luqi, "An introduction to rapid system prototyping," *Software Engineering, IEEE Transactions on*, vol. 28, no. 9, pp. 817–821, Sep. 2002.

[4] P. Tessier, S. Gerard, C. Mraidha, F. Terrier, and J. M. Geib, "A component-based methodology for embedded system prototyping," in *Rapid Systems Prototyping, 2003. Proceedings. 14th IEEE International Workshop on*, Jun. 2003, pp. 9–15.

[5] J. Hugues, M. Perrotin, and T. Tsiodras, "Using mde for the rapid prototyping of space critical systems," in *19th IEEE/IFIP International Symposium on Rapid System Prototyping*, Jun. 2008, pp. 10–16.

[6] N. Hili, C. Fabre, S. Dupuy-Chessa, and S. Malfoy, "Efficient Embedded System Development: A Workbench for an Integrated Methodology," in *ERTS2 2012, Toulouse, France*, Feb. 2012.

[7] B. List and B. Korherr, "An evaluation of conceptual business process modelling languages," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06. New York, NY, USA: ACM, 2006.

[8] R. Ellner, S. Al-Hilank, J. Drexler, M. Jung, D. Kips, and M. Philippsen, "eSPEM – a SPEM Extension for Enactable Behavior Modeling," in *Proceedings of the 6th European Conference on Modelling Foundations and Applications*, ser. ECMFA'10, 2010.

[9] Object Management Group, *Unified Modeling Language Superstructure 2.4.1*, May 2010.

[10] OMG, *Business Process Model and Notation*, OMG Std., Rev. 2.0, Jan. 2011.

[11] ——, *Software & Systems Process Engineering Meta-Model Specification*, OMG Std., Rev. 2.0, Apr. 2008.

[12] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, vol. 21, no. 5, pp. 61 –72, May 1988.

[13] J. Martin, *Rapid Application Development*. Indianapolis, IN, USA: Macmillan Publishing Co., Inc., 1991.

[14] Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Unified Software Development Process*. Addison Wesley, Feb. 1999.

[15] IBM, "Rational Unified Process: Best Practices for Software Development Teams," Dec. 2003.

[16] T. H. Phan, S. Gérard, and F. Terrier, "Languages for system specification," C. Grimm, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2004, ch. Real-time System Modeling with ACCORD/UML Methodology: Illustration Through an Automotive Case Study.

[17] S. Gérard, F. Terrier, and Y. Tanguy, "Using the model paradigm for real-time systems development: Accord/uml," in *in OOIS'02-MDSD. 2002*. Springer, 2002, pp. 260–269.

[18] A. Koudri, D. Aulagnier, D. Vojtisek, P. Soulard, C. Moy, J. Champeau, J. Vidal, and J.-C. Le Lann, "Using MARTE in a Co-Design Methodology," in *Proceedings of MARTE Workshop at DATE*, Munich, Allemagne, Mar. 2008.

[19] D. Aulagnier, A. Koudri, S. Lecomte, P. Soulard, J. Champeau, J. Vidal, G. Perrouin, and P. Leray, "SoC/SoPC development using MDD and MARTE profile," in *Model Driven Engineering for Distributed Real-time Embedded Systems*, J.-P. Babau, M. Blay-Fornarino, J. Champeau, S. Gèrard, S. Robert, and A. Sabetta, Eds. ISTE, Mar. 2009.

[20] Object Management Group, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems v1.1*, Jun. 2011.

[21] ——, *Systems Modeling Language Specification v1.3*, Jun. 2012.

[22] Luiz Fernando Capretz, "Y: A New Component-Based Software Life Cycle Model," *Journal Of Computer Science*, vol. 1, p. 1, 2005.

[23] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *Software, IEEE*, vol. 28, no. 3, pp. 41–48, Apr. 2011.

[24] A. Sangiovanni-Vincentelli, "Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design," *Proceedings of the IEEE*, vol. 95, pp. 467–506, Mar. 2007.

[25] N. Hili, C. Fabre, S. Dupuy-Chessa, D. Rieu, and I. Llopard, "Model-Based platform composition for embedded system design," in *IEEE 8th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (IEEE MCSoC-14)*, Aizu-Wakamatsu, Japan, Sep. 2014.

[26] D. Blouin, E. Senn, and S. Turki, "Defining an annex language to the architecture analysis and design language for requirements engineering activities support," in *Model-Driven Requirements Engineering Workshop (MoDRE), 2011*, Aug 2011, pp. 11–20.