This presentation describes a possible overall workflow in system modeling of software intense systems, that aims to avoid complexity and reduce cost, based on constraints in existing modeling tooling. The ideas presented are based on experience from the introduction of model based systems engineering at several large Ericsson system organizations.
Presentation Contents

- Context
- Background
- Workflow
- Conclusions and Experience
Context
Presentation Context

- Large software intense system
  - Possibly system of systems
  - Possibly hundreds of developers
- Subsequent releases with feature growth
  - Not the first release of a completely new system
- Some level of established architecture
  - Still possible to evolve
- Limited use of model simulations
This section will describe and give a background to a few concepts that will be used later in the presentation.
Traditionally there has also been a focus on doing models as *blueprints*, i.e. specifications of the system, that is handed over to the next phase in development. This is how it often works in the traditional water-fall approach. Another approach is to do models as *maps*, for the purpose of understanding the system, that possibly already is existing or is under development. A *map* is an abstraction of the reality, i.e. in this case the system. A *map* can have different scales (abstraction levels) and focus on different aspects. A map can have different purposes. But it is always the reality that is the "truth", the *map* is a way to find yourself around in the reality. So if we mainly are doing modeling for the purpose of understanding and documenting the final system, we can produce *map* kind of models. It is very easy that *blueprint*, or specification, kind of models must be complete and thus tend to be over-specified, whereas the *map* kind of models can be kept on a good-enough level. The fact that a *map* kind of model also can be produced in parallel with the evolving implementation, that it is supposed to describe, makes it beneficial.
Traditionally there is a distinction between analysis models vs. design models. The analysis model is being produced with the purpose of understanding and defining what the system shall do (the problem domain) vs. a design model with the purpose of understanding and detailing how the system shall do it (the solution domain). The what aspect of an analysis model shall not be confused with a black box or requirement model. An analysis model is also used for describing what the system shall do internally. The analysis model is refined, or elaborated, into the design model and you often end up with a decoupled analysis model that is too costly to maintain and it thus it often ends up degenerating. The analysis models can then not be trusted, and often it is just thrown away since it does not reflect the actual system.

Traditionally there has also been a clear distinction between translation vs. elaboration. The description about analysis models vs. design models describes the elaboration approach where the source, analysis model, and the target, design model, very easy can get inconsistent. The other approach is translation, where the analysis model is automatically translated, or transformed, into the design model. Then the consistency can be assured since the target model, i.e. the design model, is always based on and generated from the source model, i.e. the analysis model. This approach means that the analysis model must be detailed enough regarding what the system shall do to be able to automatically translate it into how this shall be
done. Producing such a detailed analysis model can be a daunting task for very large and complex systems. With this approach the analysis model is the *blueprint*, or the *specification*. Compare with PIM (Platform Independent Model) and PSM (Platform Specific Model) as defined by OMG’s MDA (Model Driven Architecture).
If we see that a map kind of model has its strong benefits, especially with agile and lean ways of working, how can we still utilize the translation, or transformation, approach and achieve the same kind of benefits like consistency between the source and the transformation output? Since the map kind of model shall reflect the reality we can use the reality, i.e. the implementation or other detailed models closer to implementation, as the source and transform it to a higher abstraction level suitable for use in the system model.

So we can see two distinct different kinds of transformations:
1. Transformation going from a higher abstraction level to a more detailed abstraction level, where details are added during the transformation step
2. Transformation going from a more detailed abstraction level to a higher abstraction level, where details are removed during the transformation step

It is often much easier to remove details, than adding them, so transformations of kind 2 has its benefits with respect to reduced complexity and cost of performing the transformation. The core aspect, i.e. consistency between the source and the target, is still achieved.

So we can still combine transformation with map kind of models. The only difference is that the transformation goes in the “wrong direction” compared to traditional model translation, or transformation, approaches.
Abstraction Levels and Patterns

Tie in to Bran Selic’s keynote presentation at MODPROD 2011, "Abstraction Patterns in Model-Based Engineering".

- Abstraction Pattern
  - The process of selective reduction of information
  - Can also be called Refinement Pattern in the opposite direction
- Examples of abstraction patterns from Bran Selic’s paper “A Short Catalogue of Abstraction Patterns for Model-Based Software Engineering”
  - Black Box
  - Cable
  - Summary Message

Successive levels of abstraction can be compared to successive levels of problem domains and solution domains. The solution domain for a higher level problem domain, becomes a new problem domain on a lower abstraction level.
Workflow
Opportunities with Agile Teams

• Go from producing **specifications** to producing **descriptions**
• Models can go from being **blueprints** to be **maps**
  • Keep the map (model) consistent with the reality (code)
• Allow ourselves to first produce detailed models/code in cross functional agile teams
  • Then update higher level abstraction model for the purpose of documenting and understanding the complex system
Core Principles

• Produce a *map* kind of system model
  • Describes the system, not specifies it
• Maintain **one** system model
  • Not separate analysis and design models
• Transformation in the "wrong" direction
  • Reduced complexity

With this background we can now state some core principles that the system modeling will be based on:

• We shall produce a system model which is the *map* kind of model, i.e. the system model shall be authored during or slightly after the implementation.
• We will only maintain **one** system model, not separate analysis and design models.
• We want to reduce the complexity of transformations, i.e. we will transform going from a more detailed abstraction level to a higher abstraction level, simply removing details. This is strongly connected to the first principle that we want to produce a *map* kind of model.
Scaling agile way of working requires that at least two levels of backlogs, i.e. the backlogs for agile teams themselves, but also a backlog of features on a higher level. Compare with Scaled Agile Framework, SAFe with the team level and the program level. At Ericsson these two levels can be referred to as Product Development for the team level respectively Continuous Analysis for the program level. The agile teams are referred to as XFT, Cross Functional Team, since they consist of all the roles and competencies needed, e.g. system engineers, software engineers, programmers, testers and so on, for the team to take an end-to-end-responsibility.
Since we do not want to maintain a separate analysis model, we want to utilize the single system model as a design base for doing pre-studies in Continuous Analysis. Product Development continuously produces new versions of the system model describing the current state of the system, and its implementation. When doing a pre-study in Continuous Analysis it is beneficial if the pre-study documentation as input uses the latest available version of the product documentation in the system model.

With this input we can use different approaches to describe the proposed changes to the system:

- Perform the actual changes in the system model.
- Describe the changes in the model using notes and other similar annotation techniques using the same tool as the modeling is using.

All approaches is made under the assumption that all pre-study and analysis work is done on a dead-end branch meaning that the information is not supposed to be (directly) merged into the product documentation on the main branch. The information is implicitly transferred, via the XFT and the activity branch where the XFT make updates to the product documentation. Even though this can be seen as a drawback, there are some benefits:

- Since everyone knows that the work performed within the pre-study in
Continuous Analysis on the dead-end branch will not survive, there is a force to keep the amount of work down in the early pre-study phases.

- We can allow many-to-many-relations between dead-end branches in Continuous Analysis vs the branches in Product Development.
- We can freely describe several different solution alternatives on the same branch, new diagrams that only are drawn during analysis can be made.
- Technical issues related to how models are handled in modeling tools, with unique identifiers used for keeping track of the formal model references, can be reduced and completely avoided.

The main reason for working on a dead-end branch is technical, and based on how (current) modeling tools are working. Since parts of the model shall be transformed from the implementation, which does not yet exist, we will get into severe complexity issues if we were to allow that certain model elements, e.g. representing interfaces, initially were created by hand during Continuous Analysis, and then later should be created from, and kept synchronized, from another source.

By simply stating that the work is done on a dead-end branch lots of complexity can be avoided, and the drawbacks with that some information has to be re-entered into the model in Product Development, should instead be seen as benefits, as listed above. For example: If we know that the work is done on a dead-end branch we can actually allow the automatically transformation output that has been produced in Product Development to be manually changed in Continuous Analysis, to indicate and propose coming changes. These proposed changes can be on an even higher abstraction level than the automated transformation output if wanted.
During Product Development the XFT teams updates the system model on activity branches, keeping the work with a new feature in isolation on the branch. The Definition of Done for User Stories states the product documentation in the system model shall be updated, reviewed and integrated and merged into the main branch. This means that during Product Development the system model continuously evolves with new user stories and features being documented. A new design base (baseline) of the system model is produced for each product development branch (also referred to as activity or feature) branch being merged into the main branch. This design base is then used for new product development branches with the purpose of documenting additional new user stories and features.

During Product Development the system model shall describe and document the system on higher abstraction level for the understanding of the system. The model produced can be seen as a map of the implementation of the system, with a suitable scale (abstraction level) to only include the most relevant parts.
If we now try to visualize the overall information flow within and between Continuous Analysis and Product Development we get the following picture. The main purpose is to show the information flow, i.e. that arrows indicated in which direction information flows.

Let's take a closer look at the different parts of this figure:

- **A1.** Is the System Model on the main branch in ClearCase/Git. This is the product documentation that is continuously maintained and updated.
- **A2.** Is the System Model on a dead-end branch in ClearCase/Git.
- **A3.** Is the System Model on an merge-able branch in ClearCase/Git.
- **A4.** Is the Implementation Model, or the Code. The Implementation Model could potentially be made in lots of different ways and made in lots of different formats. The important aspect is that the source format is formally defined in some machine readable form. Very often the Implementation Model can be a UML model, e.g. as a UML-RT or xtUML model, or some interface definition language, e.g. IDL. Internally within the Implementation Model the translation approach (as discussed earlier) can very well be used, where code is being generated from a higher abstraction level to detailed code. But this is not of importance for the overall way of working with the system model which should be on an even higher abstraction level.

Even though it is not shown the figure, one should consider that A1 will only exist in one instance, whereas A2 and A3 will exist in multiple instances, one for each branch in Continuous Analysis respectively Product Development. Conceptually A4 can also be seen as one instance, even though it is not relevant for this discussion the internal details how the
implementation model or the code is treated on branches in the version control system being used.

The different flows of information are:

- **I1.** When a new pre-study is started a design base (specified by a baseline) of the system model is picked. The design base for the branch can be updated multiple times by doing a rebase. For example: A pre-study will propose some updates to document A and B. The currently available design base only has document/package A updated, but document/package B is still being updated in Product Development. Then the pre-study can still be started and make the proposed updates to document/package A, then later when document/package B has been made available on the main branch, do a rebase to bring in new design base and continue doing the proposed changes to document/package B. Information flow I1 can thus be taken one, or optionally several times.

- **I2.** When the pre-study in Continuous Analysis is finished some pre-study documentation is produced, possibly extracted automatically from the model. The documentation is read and manually transformed and elaborated in Product Development. Both the system model on a product development branch as well as the implementation models are updated to add further details and implementation.

- **I3.** When a product development branch is created a design base of the system model on the main branch is used. Whenever a new design base is made available (as a consequence of another product development branch being merged back to main) a rebase/uplift is made to bring in the new design base. Information flow I3 is thus taken once each time a new design base is made available.

- **I4.** When the product development branch is finished, i.e. Definition of Done shall be reached, the product development branch is merged back to main. To keep things simple, the merge back to main is made once, i.e. a final merge. When the merge has been done, the branch is closed and is no longer used. If work needs to continue, based on this branch, a new branch is created with the design base (baseline) that the final merge of this branch produced. Information flow I4 is thus only taken once (per product development activity branch).

- **I5 and I6.** The XFT continuously evolves the implementation model (or code) and the documentation in the system model. Preferably this should be made iterative to avoid having a large chunk having to go through I6 as a final “big bang” to reach Definition of Done. Information flow I5 and I6 is thus taken multiple times during the work on one product development activity branch.

- **I7.** Whenever some part of the implementation model (or code) that is used as a source of some automatic transformation, e.g. the interfaces between system classes/subsystems, the automatic transformation to a representation in the system model is triggered. Information I7 is thus taken each time one or more transformation sources, e.g. interface products, are being delivered to continuous integration. It shall be noted that due to the technicalities in tooling, the information flow in practice goes to A1 on the main branch first, and then a rebase/uplift to a new design base using information flow I3 brings in the
transformation output (in form of so called TRO, Transformation Output, documents) and makes them available to be used on the product development branch.

The most fundamental information flows are I2 where dead-end branches which is not formally merged when finished, and instead a “traditional” document, which is read, elaborated and transformed manually by humans, into the system and implementation model, is produced, and I7 where the automatic transformation is made in the “wrong” direction than in traditional transformations.
If we take a look at the information flows in the previous slides in form of branches in ClearCase/Git we get the following picture. The same identifications as in the previous slide is used.
Abstraction Pattern Example

- Transform detailed interfaces to higher level abstraction in system model
  - Keep sequence diagrams consistent with interfaces in detailed models/code
  - Abstract away details like type information, extraneous data classes, attributes and pointers, and transform directly into parameters for visualization in sequence diagrams!
Concrete example of one abstraction pattern related to interfaces and UML-RT protocols in specific.
Conclusions and Experience
Conclusions

- Utilize the opportunity to transform in the “wrong” direction
  - Use the detailed representation as the source
  - Simplified transformation
- Consistency between higher level abstraction and detailed model/code can still be achieved
- Always transform detailed source representation
  - Do no use source representation directly
  - Source representation can then change without impacting system model
Experience

- Hard to get automatic transformation in place
  - Interface base line harder to maintain
  - Potentially heterogeneous use of version control systems
  - Several different source formats
- When the system matures and feature growth reduces
  - No need to have automated transformations
  - The transformation output can simply be maintained manually
Changing perspectives™

Peter Cigéhn
Lead Software Architect
Tieto
peter.cigehn@tieto.com