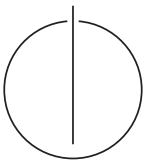
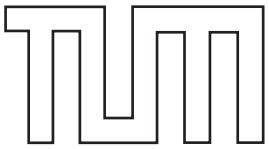


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



SPES 2020 Deliverable D1.4.A-7/8

Distributed Modeling in Context of the Functional and Logical View

Software Plattform Embedded Systems 2020

Author: Markus Herrmannsdörfer
Thomas Kofler
Version: 1.0
Date: January 3, 2011
Status: Released

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

Weitere Produktinformationen

Erzeugung	<Autor des Dokuments>
Mitwirkend	<Co-Autoren des Dokuments>

Änderungsverzeichnis

Änderung			Geänderte Kapitel	Beschreibung der Änderung	Autor	Zustand
Nr.	Datum	Version				
1	29.9.10	0.1	Alle	Initiale Produkterstellung	T. Kofler, M Herrmanns-dörfer	
2	13.10.10	0.2	Alle	Erweiterung	H. Eckardt	
3	20.12.10	0.9	Alle	Verbesserung	T. Kofler, M. Herrmanns-dörfer	

Abstract

Since model-based development promises to increase productivity, more and more embedded systems are developed by specifying models rather than writing code. When an embedded system is specified on a sufficient level of detail, the code can be automatically generated from these models. To manage the complexity, multi-functional embedded systems are modelled on different layers of abstraction. The functional view decomposes the embedded system into functions that define the behaviour observable by the user of the embedded system. The logical view decomposes the embedded system into components that define the platform-independent implementation of the functions.

Even when using model-based development and abstraction layers to reduce the complexity, multi-functional embedded systems become so complex that several developers are necessary to specify the models. To be able to reduce the time to market, these developers have to work concurrently on the models of the embedded system. However, concurrent modification of the models may lead to conflicts, since the developers may have to modify the same parts of the models to fulfil their tasks. Appropriate methods and techniques are thus required to either avoid conflicts at all or to resolve them. In this paper, we discuss different methods and techniques to enable distributed modelling with a special focus on the connection between the functional view and the logical view.

Inhalt

1 The Transition from the Functional to the Logical View 5
1.1 Functional View 5
1.2 Logical View 6
1.3 The Connection between the Logical and Functional View 6
2 Techniques 8
2.1 Parallel and Distributed Development 8
2.2 Developer Access to Models 8
2.3 Locking Strategies 9
2.4 Access to Models and Locking Strategies 10
2.5 Conflict Granularity 11
2.6 Change Tracking 12
3 Distributed Modelling in Context of the Functional and Logical View 14
3.1 Problems when working on two different layers 14
3.2 Distributed Development Process 15
4 Summary 17
5 References 18

1 The Transition from the Functional to the Logical View

To manage the complexity of multi-functional embedded systems, they are developed along different layers of abstraction [1]. Each abstraction layer concretizes the models of the embedded system by adding more detail to the previous, more abstract abstraction layer. To reduce the complexity of model-based development, each layer of abstraction focuses only on one aspect of the embedded system. We distinguish 3 layers of abstraction [2], starting with the most abstract layer: The *functional view* decomposes the embedded system according to the functions that are observable by the user of the embedded system. The *logical view* defines the software architecture by means of software components and maps the functions onto software components which provide a platform-independent implementation of the embedded system. The *technical view* defines the hardware architecture by means of hardware components and maps the software components onto hardware components which define the hardware platform of the embedded system. In this paper, we focus on the transition from the functional to the logical view.

1.1 Functional View

Figure 1 illustrates the metamodel of the *functional view* as a UML class diagram. The central construct of the functional view is a *Function* which defines a part of the behaviour of the embedded system that can be observed at the system boundary. Consequently, a function does not contain information about how the function is implemented in the embedded system. Each function defines a *SyntacticInterface* which consists of a number of typed *Port*s that are either *InputPort*s or *OutputPort*s. However, since the functional view does not specify the internal implementation, the functions do not define the ports themselves, but rather refer to ports of the interface at the boundary of the embedded system. We distinguish two kinds of *functions: AtomicFunction*s and *CombinedFunction*s. Whereas combined functions are decomposed of at least two sub functions, atomic functions cannot be decomposed. Atomic functions provide a *Specification* to define their behaviour observed at the system boundary. Combined functions can define *Dependencie*s between at least two of their sub functions. In a nutshell, atomic and combined functions allow the developers to decompose the functional view into a hierarchy of functions.

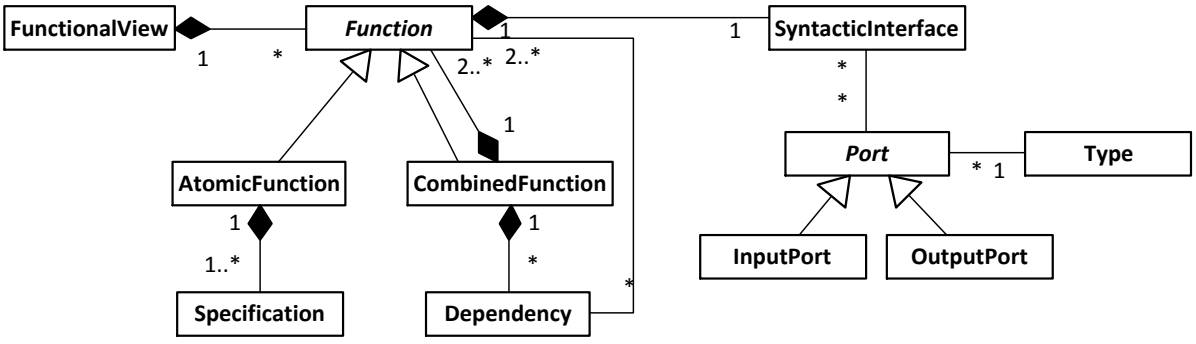


Figure 1 Metamodel of the Functional View [2]

1.2 Logical View

Figure 2 illustrates the metamodel of the logical view as a UML class diagram. The central construct of the logical view is a *Component* which defines a part of the internal implementation of the embedded system. Whereas the behaviour defined by functions can be underspecified, the behaviour defined by components needs to be complete. Similar to the functional view, each component defines a *SyntacticInterface* which consists of a number of typed *Port*s which are either *InputPort*s or *OutputPort*s. However, on the logical view, each component defines its own ports and thus the behaviour of the component is not necessarily directly observable at the system boundary. Since ports are connected by *Channel*s specifying flow of data, the behaviour of the component may only be indirectly observable at the system boundary. Similar to the functional view, we distinguish to kinds of components: *AtomicComponent*s and *CompositeComponent*s. Whereas composite components are decomposed of at least two sub components, atomic components cannot be decomposed. Atomic components provide a *Specification* to define their internal implementation. Composite components define a number of *Channel*s which connect the ports of the sub components with each other and with the ports of the composite component. Each channel connects a source to a target port. A source port may be an input port of the composite component or an output port of a sub component. A target port may be an input port of a sub component or an output port of the composite component.

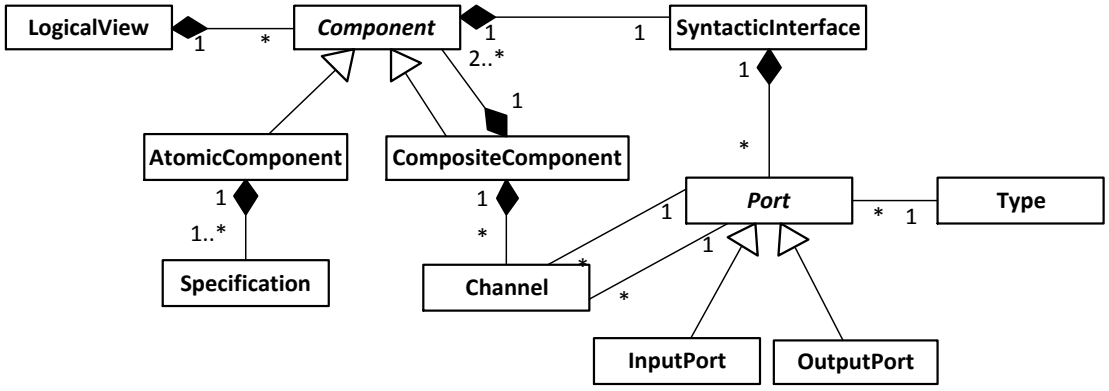


Figure 2 Metamodel of the Logical View [2]

1.3 The Connection between the Logical and Functional View

The focus of this paper is not on the transition from the functional view to the logical view. We assume that there is a rather complete version of both the function hierarchy and the component hierarchy. The task that is in the focus of this paper is to concurrently develop the implementation of the components on the logical view. The developers get tasks like to implement a certain feature in the logical view. Therefore, it is important to know the connection between the functional and the logical view. As we stated earlier, the functions from the functional view are mapped to components from the logical view. In general, there is an m-to-n mapping between functions and components [1]. A function can thus be mapped to several components, and a component can be the target for several functions. As only atomic functions and components have a specification, we focus on the mapping of atomic functions to atomic components. The combined functions and composite components serve as an organizing structure for the behaviour of the embedded system as well as to distribute the

development of parts of the embedded system to different sub contractors. When a developer is working on the implementation of a function on the logical view, he or she may run into a conflict with another developer, since the function of the other developer may also be mapped onto the same component. In the following, we explain the methods and techniques to avoid or resolve these conflicts. We also talk about the need for changing the syntactical interface of components or the component hierarchy in the course of implementation.

2 Techniques

In this section, we present a number of techniques to support parallel and distributed development of models.

2.1 Parallel and Distributed Development

When more than one developer works on a model, this is called a parallel development. Then problems of consistency arise which have to be resolved by synchronization methods. These do exist in conventional software development but have to be extended to model-based development. In this case specific problems arise, for example because groups of changes have to be handled instead of single changes of code lines.

Developers can work on one model at the same site (i.e. by direct server connection). When the system under development is growing, often additional teams are deployed at other sites. Then we speak of distributed development. Typically there is no common repository for the model in this case, the model has to be replicated to the development sites, either in parts or as a whole. This incurs additional synchronization problems which are discussed in the next section.

2.2 Developer Access to Models

When working with models in parallel, the connection between an arbitrary client and a server is a major subject of interest. The server which is used as an intelligent data store (e.g. repository) contains the model in an arbitrary form of representation as a whole. There are two major kinds of connectivity between a client and a server: online and offline connectivity. Connectivity is an important factor when developers are working simultaneously on the same model. Orthogonal to the connectivity are the locking strategies; locking strategies and connectivity are working seamlessly together. We are using the word client to denote a developer who is working on a (possibly distributed) model. We are using the word model to denote a model or a part of a model.

2.2.1 Online-Connectivity

Online connectivity denotes working with other clients simultaneously on one model on a central server. When a client changes something on the model, every other client who is working on the same model has to be notified immediately about the change. Every change on a model is called a transaction, every transaction contains one change on a model, due to this granularity the server is able to apply the change to its stored central model and propagate the change to other clients without the need for a merging strategy.

Advantages

Immediate integration of the work in progress. Every change on a model by a client has to be immediately propagated to the server. The server stores the change and notifies each client who is working or viewing the same model. This procedure avoids merges to integrate changes performed on the different clients.

Disadvantages

Depending on a network connection. Immediate integration needs a continuous network connection. If the network connection is down, there has to be a strategy to overcome this loss. This situation leads us to typical offline situations.

Resolution of the Undo/Redo-Problem. The Undo/Redo-Problem results e.g. of the following procedure: A client *a* changes a part of a model and another client *b* performs changes based on the changes of the first clients' (*a*) work. If the first client *a* reverts his or her changes, what happens with the changes performed by client *b*?

A possible solution is to forbid the undo function. Another possible solution is a central online transaction history which allows the clients to see all changes from all clients ordered by the execution time of the transactions. This approach allows the users to gradually revert the changes.

2.2.2 Offline-Connectivity

In case of Offline-Connectivity, a client has no continuous network connection to a server where the whole model is stored. According to this, the client has a local copy of the model of interest on which he or she is working offline and locally.

Advantages

Independent work. The client who is working on a model is independent of time and location at which the client is working on the model.

Independent of a network connection. The client who is working on a model is independent of a continuous network connection.

Changes can be reverted locally. Any changes of the model can be undone locally. If a client changes something locally which leads to inconsistencies, it is easily possible to revert those changes locally.

Disadvantages

Integration of the local copy. In this scenario, the already mentioned merge problem is significant. If a client *a* carries out major changes (e.g. structural changes) and the client wants to store his changes on the server, those changes have to be merged with the model stored on the server. Concurrently, another client *b* could have changed the model as well and committed it before client *a* on the server. The server has to have a merge strategy to integrate the model of client *a* in such a way, that the model is still consistent and both changes on the model (of client *a* and client *b*) have to be taken into account.

2.3 Locking Strategies

When talking about distributed modelling, an important feature is the possibility to lock development artefacts to prevent other clients to change development artefacts (e.g. models) that are already in use. On the other side, there should also be a pos-

sibility to let clients work simultaneously on the same model. There are two important locking strategies: pessimistic and optimistic locking.

2.3.1 Pessimistic Locking

Pessimistic locking denotes the locking of a development artefact by a client to prevent the locked development artefact to be changed by another client. But read access is still possible for all clients.

Advantages

No merge conflicts. This is an important advantage, because a major problem of distributed modelling is circumvented. Because only one client is able to work on a model as long as it is locked, there is no need for a merge of models.

Disadvantages

No simultaneous work on the same model or model part is possible. As mentioned above, only one client is able to work on a model if it is locked. If this model is an important part of the system and many clients need to make changes to that model, this locking strategy could lead to unproductivity of the involved waiting clients and therefore to a major cost factor for the development company.

2.3.2 Optimistic Locking

Optimistic locking means that every client could work on the same model at the same time.

Advantages

Simultaneous work on the same model. Optimistic locking enables simultaneous work of different clients on the same model. When working with optimistic locking and two or more clients are working on the same model, a merge is needed. As already mentioned, merging models is not a simple task and should be further investigated [3].

Disadvantages

Merge conflicts are possible. As mentioned above, merging models is a complicated task. This is due to the fact that the developer who merges his or her changes with the changes of another developer has to understand the intention behind these changes. However, merging tools often show the changes between the two versions of the model only as primitive changes, making it difficult for the developer to understand the intention.

2.4 Access to Models and Locking Strategies

Each kind of connectivity (offline or online) can be assigned to every locking strategy (pessimistic or optimistic). There are four possible combinations for parallel development. In case of distributed development, online connectivity is not possible. The four states are:

- *Offline connectivity with pessimistic locking*: This constellation can be found when a client *a* is working with a local copy of model *m*. *a* locks the model *m* and works hereafter offline. For any other client, a change on model *m* is impossible, as long as client *a* has not released the lock on model *m*. After client *a* releases the lock, every other client is able to lock the model *m*.
- *Online connectivity with pessimistic locking*: This constellation is almost the same as the scenario above. There is just one difference: Client *a* stays online when working on model *m*. Every change on model *m* is propagated to all other clients, but there is no possibility to change the model for any client except client *a*.
- *Offline connectivity with optimistic locking*: In this constellation, a client *a* is working on a local copy of a model *m*. Once the client has finished its work on the local copy *m'*, he tries to merge the changed local model *m'* with the model saved on the server system *m*. There is still a possibility that another client *b* changed the model *m* as well locally and produced a model *m''* which has to be merged as well. This leads to a full set of merge problems.
- *Online connectivity with optimistic locking*: In this constellation a client *a* is working on the model *m* directly on the server system. Every other client, e.g. client *b* is able change the model *m* as well at the same time. If client *a* is changing something on model *m*, client *b* or every other client who is working on the model *m* has to be notified immediately. There is no merge situation in this scenario, because every change is propagated to each client who is working on the same model.

2.5 Conflict Granularity

The conflict granularity determines the size of model parts that are atomic when it comes to avoiding or determining conflicts. The conflict granularity serves two purposes, depending on the kind of locking strategy that is used. For pessimistic locking, it can be used to determine the size of the model part that can be locked separately from other model parts. For optimistic locking, it can be used to determine conflicts in a merge situation – if two clients change the same model part, then their changes conflict with each other. The conflict granularity can be either fine-grained or coarse-grained.

2.5.1 Fine-Grained Conflict Granularity

The conflict granularity can be very fine-grained, i.e. on the level of single model elements. A conflict thus only occurs if two clients simultaneously change the very same model element.

Advantages

Low probability for conflicts. Since only elements are locked and a model usually consists of a lot of elements, the probability for conflicts is rather low.

Independent of the modelling language. Locking models on the level of elements is independent of the modelling language and can thus be applied to any modelling language.

Disadvantages

Locks for each element. A lock has to be maintained for each model element or a conflict has to be resolved for each model element. This results in a lot of effort for obtaining locks or resolving conflicts.

Risk of inconsistency. If there are non-local consistency rules that apply to a group of several elements, handling conflicts on the element-level may result in inconsistent models. If two elements are only consistent with each other in certain cases, then two clients each of which changes one of the elements are not in conflict, but may produce an inconsistent model.

2.5.2 Coarse-Grained Conflict Granularity

The conflict granularity can be very coarse-grained, i.e. on the level of a module. A conflict thus already occurs if two clients simultaneously perform changes on the very same module.

Advantages

High-level locking. Fewer locks are required to lock larger parts of the model as well as fewer merge decisions are needed to resolve conflicts. This reduces the effort for obtaining locks or resolving conflicts.

Preservation of consistency. Inconsistencies resulting from changing elements that are in a consistency relation with each other can be avoided by defining the modules appropriately so that these elements are in the same module.

Disadvantages

High probability for conflicts. The probability for conflicts is rather high, since there are usually much fewer modules than elements and changes to the same module result in a conflict.

Dependent on the modelling language. The modules that serve as conflict granularity have to be defined for each modelling language, thus making the coarse-grained conflict granularity dependent of the modelling language.

2.6 Change Tracking

To determine conflicts or to merge models, the changes applied to a model need to be tracked by the Version Control (VC) system. There are two major means to track changes: state-based and operation-based change tracking [3].

2.6.1 State-based Change Tracking

State-based approaches only store states of a model, and thus need to derive differences by comparing two states, e.g. a version and its successor, after the changes occurred. This activity is often referred to as differencing.

Advantages

Tool independence. Since the VC system is not required to be able to observe the changes while they occur, a total separation of the modeling tools and the VC system is possible.

Disadvantages

Computationally expensive. Due to the graph isomorphism problem, calculating the difference is a computationally complex endeavour – especially if changes between many states need to be retrieved, or the model is of a large size.

Loss of information. State-based approaches can neither completely and correctly derive the exact temporal order of the changes nor are they able to derive composite changes.

2.6.2 Operation-based Change Tracking

Operation-based approaches record the changes, while they occur, and store them in a repository. There is no need for differencing, since the changes are recorded and stored, and thus do not need to be derived later on.

Advantages

No computation effort. Since the changes are recorded, no computation effort is necessary to derive the changes, when they are required for the different use cases.

More accurate information. Since they record the changes, operation-based approaches retain the exact temporal order of the changes as well as composite changes. Especially information about composite changes may help to produce better merge results.

Disadvantages

Tool dependence. Operation-based approaches need to be integrated into the tool used for changing the models. As a consequence, they cannot be used for existing tools which do not provide such functionality.

3 Distributed Modelling in Context of the Functional and Logical View

Functions of the Functional View are refined into one or more components of the Logical View. The Functional View should be as complete as possible before the development team starts to work on the Logical View. Distributed modelling can then be seen as the simultaneous work on the Functional and Logical View. We assume that the Functional View has to be as stable as possible, before developers should work simultaneously on the Logical View.

Working in parallel on the Logical View leads to the following problems:

a) *How to distribute the models*: There could be an initial component model, which is the starting point for different developers. We assume that after the definition of the interfaces and the interface behaviours developers can work independently on the components.

b) *How to handle changes in the Functional View*: Changing something in the Functional View leads to inconsistencies in the Logical View. Changing top-level functions (high level of granularity) is problematic, since big parts of the Logical View could be affected. A change process must be established for modifying interfaces.

3.1 Problems when working on two different layers

Working on the Abstraction Layers in the context of distributed modelling means the simultaneous work on the Functional and Logical View. The Functional View is connected to the Logical View, because it is the refinement of user functions into logical components. These layers are therefore connected.

3.1.1 The Functional and the Logical View

An initial component model could serve as starting point for distributed modelling. The initial component model should contain interfaces and interface behaviours. Each component is, as already mentioned, a refinement of one or more functions, therefore the developers have to have access to the functions they should implement. A component therefore is an implementation of one or more subtrees of functions. The functions have to be accessible for every developer in the project. If a function is changed, the affected components have to be changed as well. Therefore it is important that a technique exists to easily find the affected components. If a function is changed, it may also influence the interfaces and the interface behaviours of components. Hence, the notification of changes in the functional view is very important for distributed modelling based on the abstraction layers.

3.1.2 How to find an Initial Component Model

When creating an initial component model, the following influence factors have to be taken into account:

a) Conflict Granularity Level

b) Access Form
c) Locking Strategy
d) Or all together

Why should those factors influence the initial component model? Depending on how many developers are working on a model, there has to be a strategy, how to distribute the work on a model in such a way that the developers could work simultaneously on that model.

The initial component model must at least consist of a component and its interface. If the initial component model contains just one component, conflicts are very likely to occur, since all developers are working on the same components. To reduce the risk of conflicts, it is necessary to partition the component at least into as many independent parts as there are developers. The problem, which arises here, is, it is not easy to find those independent parts. This leads to the assumption, that the only feasible way is to define an initial component model whose interfaces and whose input and output behaviours are well defined. In that case, the result of the composition of the distributed components is clear from the very beginning of the work.

Depending on the locking strategy, only one person might to be able to change the model. If one person locks the initial components, it won't be possible to change the model for other developers. The locking strategy is as well dependent on the form of representation of the model in the repository as it is on the possible granularity level. It can't be seen independent of the underlying technology, which has to support locking strategies as well as possible granularity levels. In the case that these requirements are in place, it is possible to distribute the initial component model to the developers without disturbing the work of each other.

3.2 Distributed Development Process

Depending on the access form, it could be very difficult to integrate the initial component, if every developer has the same right to change parts of the initial component. Therefore it is advisable to assign certain roles to the involved persons. The functional model and initial components can be developed by some key developers or the system/software architect himself. The developers work on the components assigned to them. For integration (i.e. merge) of the developers' changes into the server model a two-level procedure can be defined. Developers work on a copy of the model. When they have finished their task, the integrator merges the second-level model into the main model. This is more safe because integrators know the whole system and can assess the side effects of changes. The results of other developer groups have to be propagated to the second level model. This is done by the integrators by replacing the second level model by the main model at defined points of time. Thus a process as shown in Figure 3 is determined. Critical changes of the functional model or initial components can be included in this process in a defined way.

The detailed process how to develop/integrate changes of the model depends on the working style which is more or less prescribed by the modelling tool. Some tools allow to separate the abstraction layers in different sub-models which are more or less independent from one another. This supports avoidance of conflicts. Other tools al-

low only for one single model per “project” or repository. These tools are less suited for large-scale development of model-based software.

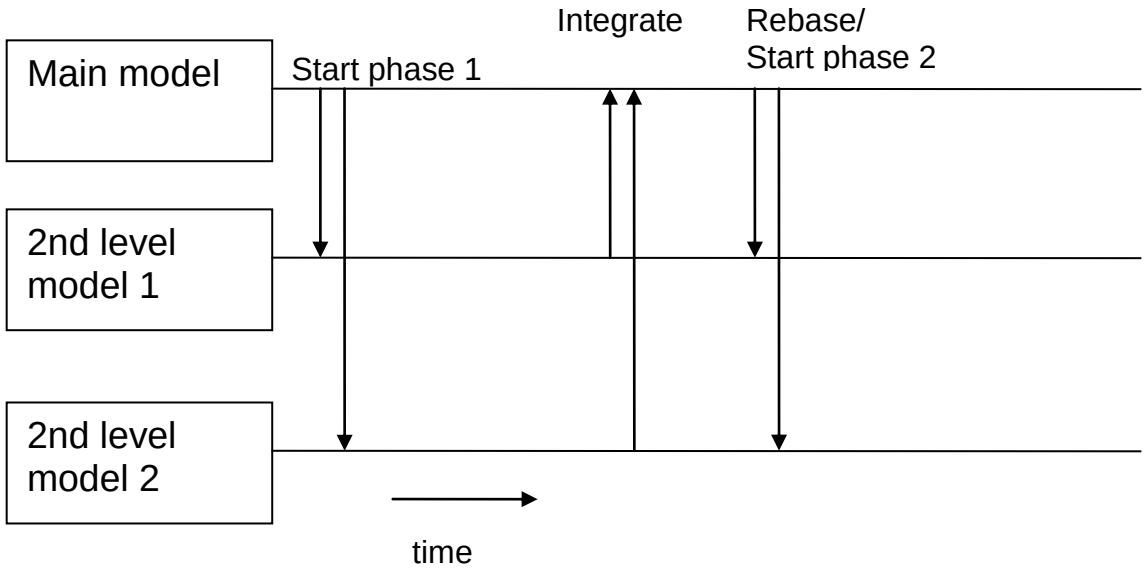


Figure 3 Distributed development process

4 Summary

To manage their complexity, systems are implemented using different layers of abstraction. First, the user functions are defined. Then, the user functions are implemented by software components. Finally, these software components are mapped onto hardware components. User functions are often implemented simultaneously by different developers. Since several functions may be mapped onto the same software component, conflicts may arise between the work of the different developers. To avoid such conflicts, we presented a number of techniques for parallel and distributed modeling. We have analyzed these techniques in the context of the transition from user functions to software components.

5 References

[1] Daniel Ratiu, Wolfgang Schwitzer, Judith Thyssen. A System of Abstraction Layers for the Seamless Development of Embedded Software Systems. TUM-I0928. 2009

[2] Alexander Harhurin, Florian Hölzl, Thomas Kofler. SPES Metamodel. SPES Deliverable D1.2.B-6. 2010

[3] Maximilian Koegel, Helmut Naughton, Jonas Helming, Markus Herrmannsdoerfer. Collaborative Model Merging. ONWARD! '10: ACM Conference on New Ideas in Programming and Reflections on Software. 2010

[4] Maximilian Koegel, Markus Herrmannsdoerfer, Yang Li, Jonas Helming, Joern David. Comparing State- and Operation-based Change Tracking on Models. EDOC '10: 14th IEEE International EDOC Conference. 2010