Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams

Christian Prehofer
DoCoMo Euro-Labs, Landsbergerstr. 312, 80687 Munich, Germany
prehofer@docomolab-euro.com

Abstract. This paper presents a new approach for modular design of highly-entangled software components by statechart diagrams. We structure the components into features, which represent reusable, self-contained services. These are modeled individually by statechart diagrams. For composition of components from features, we need to consider the interactions between the features. These feature interactions, which are well known in the telecommunications area, typically describe special cases or cooperations which only occur when features are combined. We describe these interactions graphically as well by partial statecharts. The main novelty is that full component descriptions are created automatically in a plug-and-play fashion by combining the statecharts for the required features and their interactions. Furthermore, we develop different classes of statecharts and show the interactions on a case-by-case basis. For composition, we use semantic refinement concepts for statecharts which preserve the original behavior.

Keywords: graphic description techniques, plug-and-play composition, feature interaction, statechart diagrams, semantic refinement, UML

1 Introduction

When talking about a piece of software, we often speak of a "feature" which this software has. For instance, when collecting requirements, features of a software system are common terminology. We propose here a graphic description method for feature-centric software development which supports composition of components from features in a modular way. The abstract behavior of features is modeled by statechart diagrams. However, features are often not independent and do interact in many ways. The main contribution of this paper is to describe features and their interactions graphically and to model their composition by stepwise refinement relations between statecharts.

A key problem is that features often have to cooperate or interact in unforeseen ways. In larger systems with many features, these interactions can lead to highly entangled code which is difficult to maintain. The problem of feature interactions is well established in the telecommunications area [3]. While most of the work in this area focuses on detecting interaction, we focus on software design methods which consider interactions. The problem is that handling interactions in most cases violates modularity. One often has to fix a special case in one feature which only occurs if another particular feature is present. Hence the code implementing these features becomes overloaded and obscures the core functionality of the feature as well as the dependencies.

For our development method, we employ graphic design of features by statechart diagrams (or abstract state machines). A novel point is that we model features and interactions as partial statecharts and transitions, which can be combined automatically when needed. This approach allows one to cross-cut large statechart diagrams into smaller features and their interactions. Based on semantic refinement concepts for statecharts, we present rules to create component models by combining the statecharts of the required features. These graphic refinement rules ensure that the feature behavior does not change during composition.

Our design method aims at graphic plug-and-play feature composition, which is essential in the following application scenarios:

- Many different versions of a software component are needed, each having different, often alternative features.
- Applications where monolithic software is unsuitable, since only the features needed by a customer are delivered. For instance, when downloading software on a limited mobile device, the download time and storage space on a mobile device should be kept minimal.
- Applications where entangled features are added or updated frequently during the live-cycle of a software component.

As an example consider an email server, where [4] has analyzed about 10 common features and discovered about 25 feature interactions. For instance, encryption and auto-responder interact as follows. The auto-responder answers emails automatically by quoting the subject field of the incoming email. If an encrypted email is decrypted first and then processed by the auto-responder, an email with the subject of the email will be returned. This however leaks the originally encrypted subject if the outgoing email is sent in plain. Hence for combining these seemingly independent features one must consider such special cases. We use in the sequel the above email example. The main features (see also [4]) are:

- Encryption and decryption for encrypted mails.
- Filtering particular emails, e.g. for virus protection.
- An auto-reply feature for automatic answers to all incoming emails.

In the following section we present the background on statecharts. Modular construction of statecharts modeling the behavior of features, interactions, and their combination is discussed in Section 3 through Section 5.

2 Statechart Diagrams and Semantic Refinement

Our graphic description method for developing software components is based on UML statechart diagrams [10]. When we model a software component with statecharts, we aim at specifying the behavior of its functions. We label transitions by the functions which trigger these transitions. An external function call triggers a transition labeled with this function depending on the current state of the statechart. Note that the finite number of states of a statechart usually represents an abstraction of the internal states a component can have.

We use the following UML notation for labeling transitions:

called_function() [condition] / action

A transition can be initiated by an external event, here called_function(). It may have a condition and it may have an action it initiates. This action describes the behavior triggered by the function. Note that all three labels may be empty. In case the trigger function is omitted, we have an internal transition without an external event, also called spontaneous transition.

An example is the statechart in Figure 1 describing the basic functionality of an email system in a feature called BasicEmail. We have two states, one called Waiting (for email) and the other called Emailarrived. The initial state is Waiting.

The transition labeled incoming() is triggered if an email has arrived and issues the operation get() to retrieve the email. The transition labeled deliver is the actual processing

of the new email which triggers the store() function to save the email. This simple model of an email system only handles one email at a time. Note that we indicate functions by parentheses as in f(), which does not mean that these are parameter-less functions in an implementation. Later we will introduce parallel statechart diagrams and hierarchical diagrams based on UML notation.

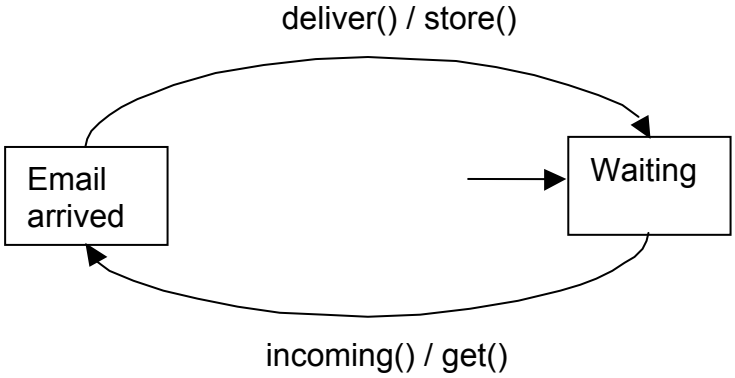

Figure 1: The BasicEmail Feature

2.1 Semantics

Our semantic model employs an external black-box view of the component. It is based on function calls from the outside which trigger transitions. Only the input and output are considered, not the internal states. A possible run can be specified by a trace of the externally called functions and the resulting actions of the statechart.

For instance, consider in this example traces for the statechart in Figure 1 of incoming() and deliver() transitions, triggered by external function calls.

Input sequence:
incoming(), deliver(), incoming(), deliver(), incoming(), deliver(),
Ouput sequence:
get(), store(),get(), store(), get(), store(),

We adopt the loose, "chaos" semantics [9] where the semantics of a component is given as the set of possible traces. The set of traces includes any possible trace by transitions specified in the statechart. In addition, any unspecified function call, e.g. a function call for which no transition is defined in the current state, leaves the statechart in chaos state and any behavior is permitted after that. For instance, the output for the input deliver() in the Waiting state is not defined in the statechart and hence anything is permitted after this. Our semantic model does not specify what happens in case the deliver() function is called in Waiting state. Hence the statechart does not fully specify a single implementation, but permits many possible implementations, which fulfill the specified input/output relation. More formally, we can describe the semantics as a set of deterministic, monotonic functions which are compatible the statechart. Each function represents a possible, deterministic implementation. Since many implementations are possible, we use a set of functions. This set of possible implementations can be reduced, which we view as refinement. Note that our statechart model permits a non-deterministic choice if several transitions are possible in one state, which is just a special case of loose semantics. For a more detailed treatment of the semantic background we refer to [1].

2.2 Refinement

Our notion of semantic refinement of statecharts aims at specifying a component in more detail and hence to reduce under-specification. We speak of semantic refinement of a statechart to another, if the refined one has fewer possible traces. As the loose semantics of a statechart is a set of functions, this can be formally expressed by reducing this set of functions. In this way, it is very suitable for abstract specifications and enables step-wise refinement by adding more specific behavior. For further details on semantic refinement relations for automata models, similar to statecharts, we refer to [9,5,7].

The main benefit of graphic refinement rules is their ease of use. The graphic refinement rules are based on syntactic input and output events. In some cases conditions of transitions have to be considered. Note that the rules do not cover any properties of possible input/output parameter or of internal state variables. To include these, the same semantic models can be used in this case [5,7], but deeper reasoning is required. In practice, this often means that the refinement is shallow and just implies compatibility with respect to the input and output messages.

The following graphic operations on statecharts are also refinements with respect to our semantics. These elementary statecharts refinement operations are proven to be semantically correct [5,7]:

- *Add new behavior* which was unspecified before, e.g. add a new state or add a new transition to a state, which did not exist at this state. Since this transition was not specified before, there is less chaotic behavior and the set of traces is decreased.
- *Eliminate alternative transitions*, if alternative transitions exit. This reduces non-determinism and specifies the behavior more precisely. Note that adding a condition to a transition can be handled in the same way, as it removes possible transitions at run time.
- *Add internal or compatible behavior*, which only adds new or internal behavior and does not change the original output. In this case of new behavior, we can abstract from the additional behavior and the old behavior remains unchanged. Under this abstraction, the original behavior is preserved.
- *Eliminating transitions for exceptional cases*. In this case, refinement only holds if some exceptional case does not occur. This is also called conditional refinement. Note that adding conditions to transitions is viewed as removing transitions.

In the following sections, we use the four refinement rules described above.

3 Modeling Features and Interactions by Statecharts

In this section, we describe graphic techniques to model the behavior of features. A feature, like a class in object-oriented design, offers an interface with functions and encapsulates internal state. We describe both features and interactions by partial statecharts to describe a high-level view of their behavior.

The novel point here is that we describe the features and their interactions modularly as fragments of statecharts. For a concrete feature combination, we show later how to combine these automatically to a statechart with the combined functionality. For the combination, we will make extensive use of hierarchical statecharts and parallel composition of statecharts.

When modeling features with statecharts, we can distinguish the following three kinds:

- Base features with a complete statechart, including an initial state and final states.

- Transition-based features which refine a transition locally, but do not add externally visible input transitions to a statechart. These features typically represent a service or an aspect which can be added to a feature combination with at least one base feature.
- State-extending features which add global states and externally-visible transitions. These features extend the states of some other features and also extend the externally visible interface.

We consider in the following these classes of features in the above order.

3.1 Base Features

Base features are self contained and can be used without any other features. An example of the first kind is the statechart in Figure 1 describing the basic functionality of an email system in a feature called BasicEmail.

The base features typically form the basis of a feature implementation. In contrast to others, they can also be used independently. For combination of base features, we will use parallel statecharts, as shown below.

3.2 Transition-based Features

Transition-based features provide services which can be modelled by internal transitions without persistent or global state. These features typically model auxiliary services and hence do not change the global control flow. They do not add externally visible input transitions to a statechart. The transition-based feature may however produce additional output or trigger transitions, e.g. in other, parallel statecharts as shown later.

We present transition-based features in the examples of the Forwarding and Reply features, each of which offers one function of the same name. In Figure 2, we show the internal states of the forward and reply functions. The feature Forwarding includes also a function call do_forward() which is not detailed here. The reply feature is similar. Note that we use two small circles to denote the start and end states of the transitions, which are determined later when composing features.

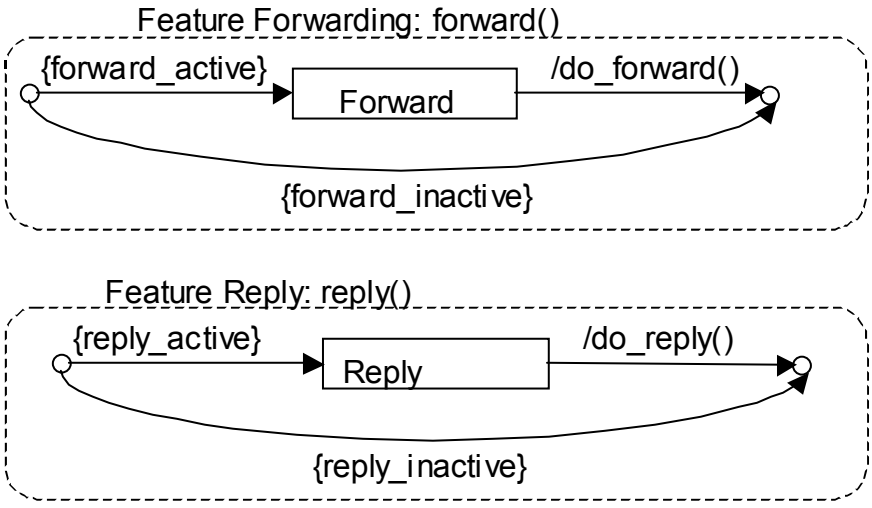

Figure 2: Internal View of Forwarding and Reply Features

Some transition-based features do not have internal states. Examples in the email application are the encryption and decryption features, which consist of a single transition each (Figure 3). An actual implementation of these features may have persistent state, but this is not modeled here.

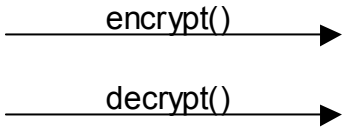

Figure 3: Encryption and Decryption Features

In general, a transition-based feature is modeled by a number of transition functions which we specify without detailing the start and end states. Furthermore, we may use an internal statechart to model the internal behavior of this feature. For refinement, we also make the assumption that the local statechart only consists of internal or spontaneous transitions. A schematic example is shown in Figure 4 below.

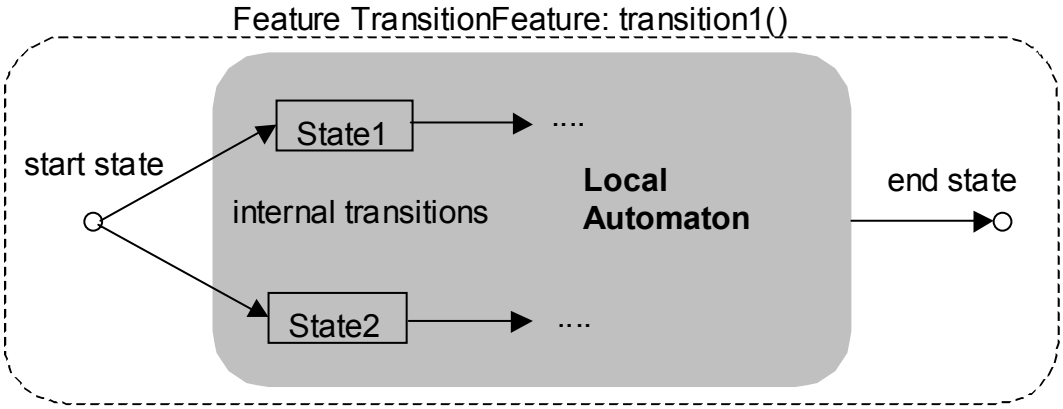

Figure 4: Partial Statechart for Transition-oriented Features

3.3 Global State-extending Features

Another main category of features is that of state-extending features which extend the set of states and add global transitions. In this way, they extend the external interface. For instance, consider a feature with a new state called MaintenanceMode (Figure 5). This partial statechart does not have initial or final states; its states will be reached by transitions from the states of other features. This will be specified separately in the feature interactions. Note that this statechart extends the external interface by new functions.

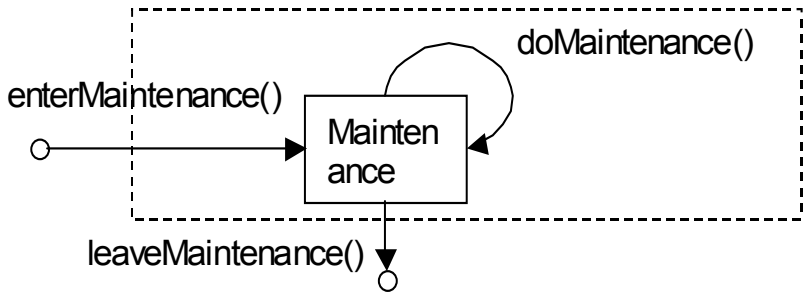

Figure 5: The MaintenanceMode Feature

The definition of a partial statecharts is a statechart with two special transitions for entering and leaving the statechart. These do not have start or end states, respectively. This indicates that the statechart shall not be used in isolation and shall be combined with others. We do not introduce named start/end states, since feature combination in this case would require to rename states which can cause complications, e.g. if other features or interactions refer to this state.

For combination, these start/end transitions will be mapped to specific states. We will also permit that the start function may be triggered from several states, unlike the exit function. Compared to the above transition-oriented features, the new state Maintenance is global, as well as the functions of the transition are visible externally. As we will show in the next section, composition of such features consists of two steps. First we merge this partial statechart with another statechart. In addition, other adaptations may be needed.

4 Resolving Feature Interactions

Interaction handling adapts a feature into the context of another one in order to resolve feature interactions. An adaptor A → B defines a refinement of a composed statechart which includes feature A (and possibly others) with feature B. With statecharts, we will use two techniques to refine a feature A to the context of a new feature B:

- The transitions of feature A may be refined. We model this by restricting a transition or by inserting a local statechart, leading to hierarchical statecharts. We will also use this kind of refinement to specify the start or end states of a transition, as shown below.
- New transitions from the states of A to states of B triggered by function calls of B can be added if B is state-oriented. We also refer to [5] for this kind of refinement.

The goal here is to denote this refinement just by describing the necessary refinement steps in isolation, without the context of other features. In addition, it is important to describe this adaptor generically such that it can be used for any combination of features including feature A. This is the essential abstraction which permits to compose features in a flexible way.

4.1 Interaction of Transition-oriented Features

In this section, we describe features which refine transitions by local statecharts. More precisely, this can be seen as an hierarchical statechart. We first focus on adapting base features to transition-oriented features. For instance, Figure 6 shows how the deliver function is adapted to the decryption feature. Note that we do not specify the refinement for individual transitions but for functions labeling the transitions. Hence two transitions with the same label are refined in the same way.

In the example, the function deliver() is expanded by a statechart with two transitions and a newly added state, where decrypt() is a function of the added feature. In this way, we refine the deliver() function to first execute the decrypt() operation and then the original deliver() operation. Note that *BasicEmail.deliver()* is now a special internal operation which is not externally triggered. We denote this by adding the feature name and by italics. It denotes the internal operation to be triggered by deliver() and is implicitly triggered. Externally, it is viewed as a spontaneous transition. Also it is important for further refinement steps, which can refine this transition. This simplifies the technical treatment, because we do not refine local statecharts but only transitions.

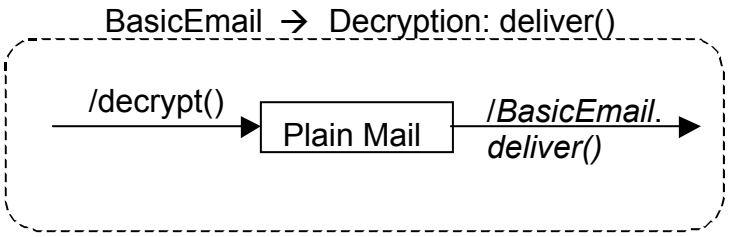

Figure 6: Adapting BasicEmail to Decryption

In a similar way we can adapt BasicEmail to Forwarding, as shown in Figure 7. In this figure, the function forward(), as defined above, is not shown in detail.

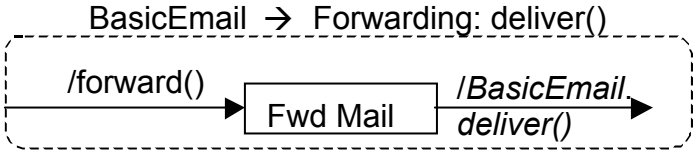

Figure 7: Adapting BasicEmail to Forwarding

Next we consider the case of an adaptor between two transition-oriented features. Interaction between automatic reply and decryption is a typical special case, which can often be modeled by adding or restricting transitions. The interaction here can be handled similar to the above and is shown in Figure 8. Furthermore, we restrict both transitions by adding conditions. As we will see later, do_reply() will occur as an action of a transition which we refine by the statechart in Figure 8.

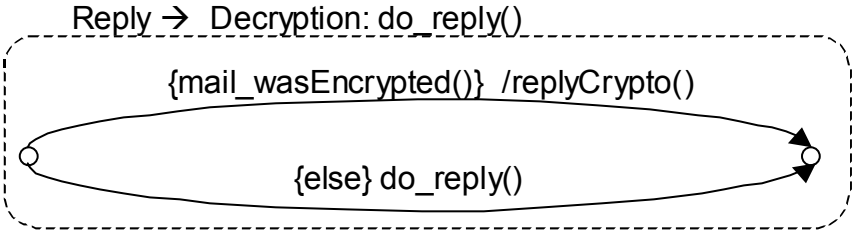

Figure 8: Adapting the Method do_reply of the Feature Reply to the Decryption Feature

In the general case, an adaptor can refine a transition of the adapted feature by extending the transition to a local statechart with internal states and transitions. We do not permit externally triggered functions in this refinement, since this may affect the external semantic behavior. Externally visible transitions in local statecharts would syntactically be possible, but this does not follow our notion of semantic refinement. It would require a different or more complex notion of refinement. (See also [5,1] for a detailed treatment of refinement.) The function which is refined can however be used in the operation associated with an internal transition.

We can express the general case with following schematic adaptor shown in Figure 9. We adapt all transitions labeled with the same function in the same way by one adaptor. In case two transitions (from different states) are triggered with the same function, these may not have different refinements.

The inner "black-box" statechart can use the functions of the feature A, e.g. A.a(), as well as the functions of features B and C. No others are allowed, since the feature adaptors have

to be generic to be added to any feature combination which includes the adapted feature. Furthermore, only internal or spontaneous transitions are permitted, as the external input behavior shall not change. Note that we use the notation F.f () to denote a function f of the feature F.

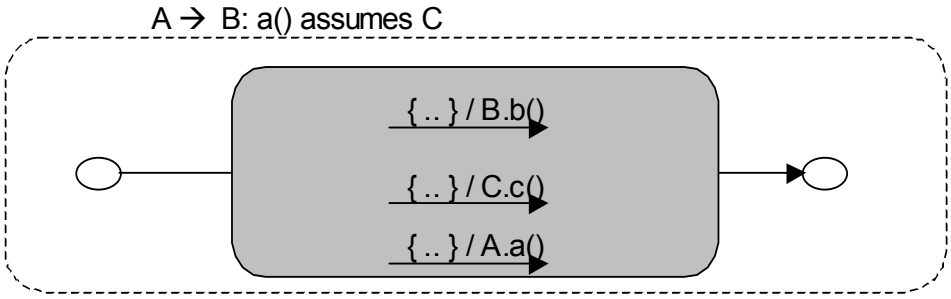

Figure 9: Schematic adaptor for a transition, triggered by a(), of feature A to B

4.2 Interaction of State-extending Features

Interaction in this case defines the merging of two statecharts by adding transitions between the states of the features. For this, we need to map the anonymous start/end states of the state-oriented feature to states of another feature. Furthermore, transitions may be refined by conditions or actions. In the following, we consider the possible interactions based on our classification of features.

A typical interaction with a base feature is illustrated by the example in Figure 10 for adding the BasicEmail feature to the MaintenanceMode feature. We only show the relevant states for both features and indicate the MaintenanceMode feature by the gray area. The interaction defines that the latter feature can be reached from the Waiting state of the BasicEmail feature. We view this interaction as a refinement of the MaintenaneMode feature, since the transitions of this feature are refined by specific start/end states.

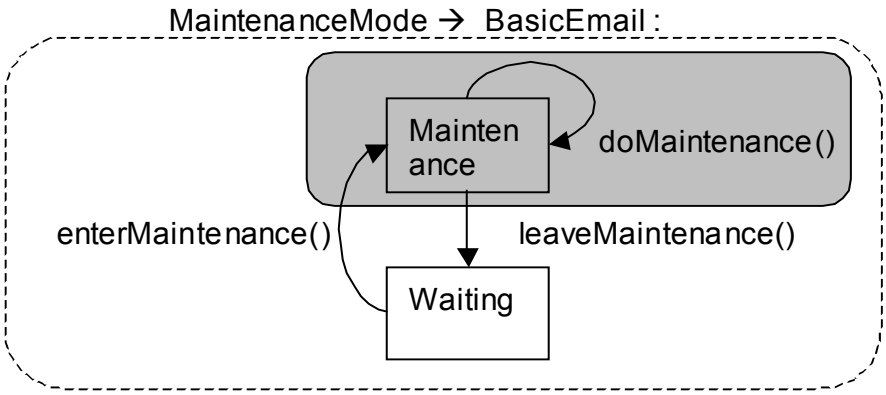

Figure 10: The MaintenanceMode Interaction with BasicEmail Feature

Another example is shown for a feature called ErrorCase with the state EmailError and the two functions error() and resume().

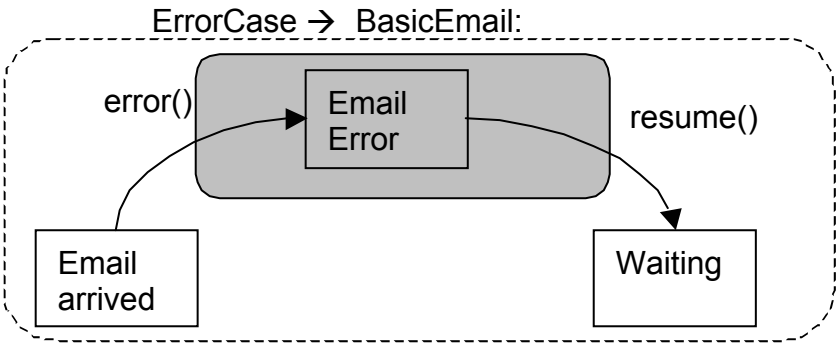

Figure 11: Interaction of Error and BasicEmail features

In the general case, an adaptation A → B, with one base and one state-extending feature means to add new transitions from states of A to states of B, which are labeled by functions of A. For instance, in the above example the other direction for adaptation is also possible. In this alternative, adapting BasicEmail to ErrorCase would allow one to add transitions labeled with deliver() or incoming() of BasicEmail to the EmailError state. Regarding semantic refinement in this variation, we have to make sure that this transition has not been defined before. For instance, it would be legal in this variation to add a transition labeled deliver() from Waiting to EmailError, since this is not defined yet. On the other hand, this is not possible for the Emailarrived state, since this would overlap with an existing transition. Although this might be viewed as a non-deterministic statechart, this leads to semantic refinement problems as discussed in [5].

The case of an adaptation between two state-extending features is similar to the above. The difference is that in combination of several features, only one interaction may define the exit transition of the new statechart. On the other hand, we show that there can be several transitions labeled with the function entering the new feature. An example for this case is the adaptation of MaintenanceMode to ErrorCase, as shown in Figure 12. For this, we add a new transition labeled enterMaintenace() from the EmailError state. In this way, the MaintenanceState is also reachable in an error case, which resolves an important interaction. Note that we do not fix a state for the exit transition from Maintenance, since we assume that this will be done by the base feature, here the BasicEmail feature. As a general rule, only one feature in a combination can define the exit state. In contrast, there can be several transitions for entering the Maintenance state. For instance, there can be transitions from both the Error and BasicEmail features.

In addition to this mapping, we may have to refine the transitions, as for transition-based features. In this example, the transition leaveMaintenance() has been restricted. We refine the leaveMaintenance message to leave the Maintenance state only if no errors exist. In other words, it is possible to enter maintenance from error state, but then the error must be fixed in the maintenance state.

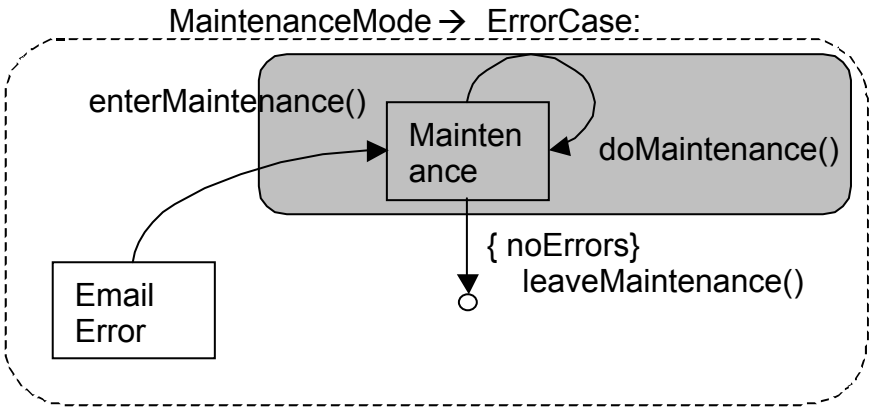

Figure 12: The MaintenanceMode Interaction with BasicEmail Feature

The adaptation of a transition-oriented to a state-extending feature is more restricted, since we may not add transitions from local to global states. Hence only transition refinement by conditions or actions is possible.

We have discussed the combination of state-extending and base features. The remaining case of adaptation of a state-extending to a transition-oriented feature is analogous to the case for base features shown above.

4.3 Interaction of Base Features

For base features, we distinguish several cases. The cases where a base feature interacts with a transition-based or state-extending feature have already been treated above. In case of interactions between two base features, we use parallel statecharts for combination, as shown below. Since these statecharts operate separately, we can only restrict the transitions of the other base feature, as shown in the examples above. Adding transitions between parallel statecharts is not permitted.

5 Combining Features and Interactions as Refinement

We show in the following how to combine features and their interaction handlers in an automatic way. If no adaptor between two features exists, we assume that the features can be combined in any arbitrary order. If feature A must be adapted in the presence of B, A must clearly be added before B. In this way, the interaction handling defines the order of the feature combination. Given a selection of features, a possible sequence for composition can be inferred. Alternatively, a particular order can be given explicitly. Then the statechart for a concrete feature combination can be determined, as we show in this section. We use several forms of statecharts refinement, including transition refinement and parallel statecharts. For a more detailed semantic treatment of refinement we refer to [5,9].

In order to obtain practical composition schemes, we follow several design principles as discussed in [8,6]:

- We limit our model to feature interactions between two features. As we focus on a general purpose software development method, we do not focus on modeling such three-way interactions [12]. We refer to [8] for a more detailed discussion.
- Asymmetric interaction handling: in case of an interaction only one feature is adapted.
- Composition of features in a sequential order is used as it is the simplest and most natural composition technique.

5.1 Transition Refinement

We cover in the following the case of adding transition-based features. In this case, the combined statechart can be obtained by unfolding the interaction handlers in the statechart one after the other. Adding features with global control state is considered in the following section.

We consider in the example a typical case with a base feature, which is externally visible, and add auxiliary features, which do not change the external interface. We first consider adding features to the base feature, and then we look at other kinds of interactions. This combination proceeds sequentially and produces a typical pipeline-architecture, where the input is passed from one feature to another. For instance, the message is first forwarded, and then decrypted, and finally it is delivered to the client. Hence the interaction consists of extending the message delivery function of the basic email feature. For the reverse direction, not shown here, one has to extend the message incoming function in a similar way.

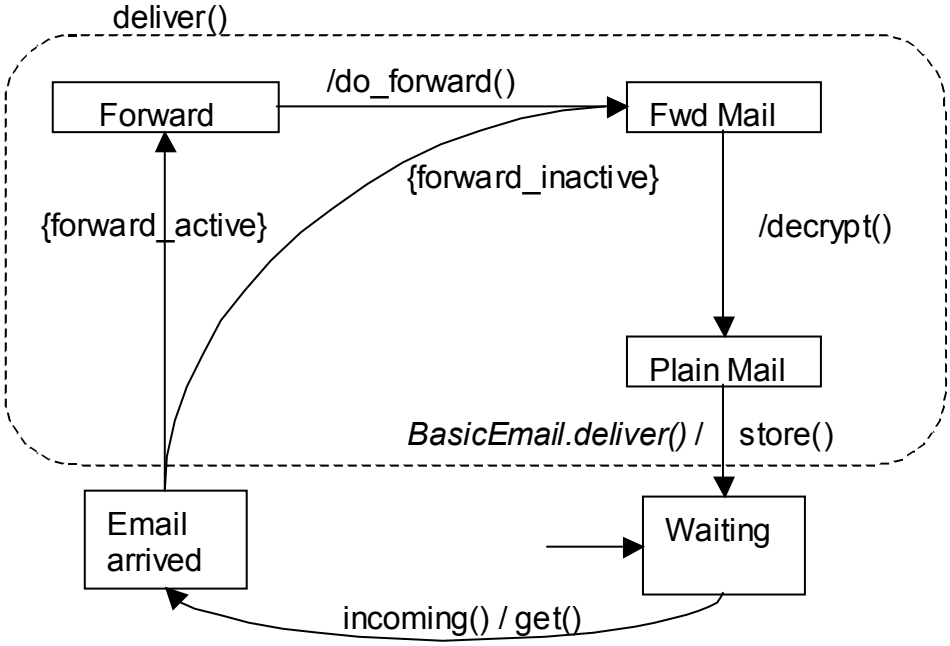

Figure 13: Combination of Basic Email, Decryption and Forwarding

The hierarchical statechart in Figure 13 shows the combination of three features, extending the basic email feature. Note that the delivery() function has been refined twice by two feature interactions. As only new behavior is added, refinement by abstraction is easy to show.

In similar fashion, we can combine three other features, including the reply and decrypt features. We first define the interaction between the Reply and the BasicEmail feature, as shown in Figure 14. In this figure, the function forward(), as defined above, is not shown in detail.

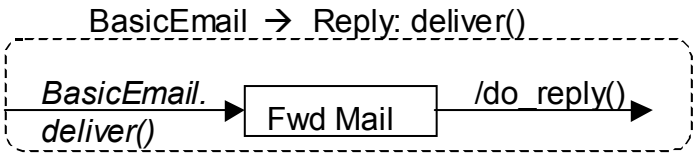

Figure 14: Adapting BasicEmail to Reply

The combination of the three features in Figure 15 also illustrates the interaction between the reply and decrypt features, which is not shown explicitly. For this interaction, we refine the do_reply() function to account for the interaction and use a function replyCrypto() in case the email was encrypted. This function has to handle the interaction of leaking the previously encrypted header of the email, as detailed above.

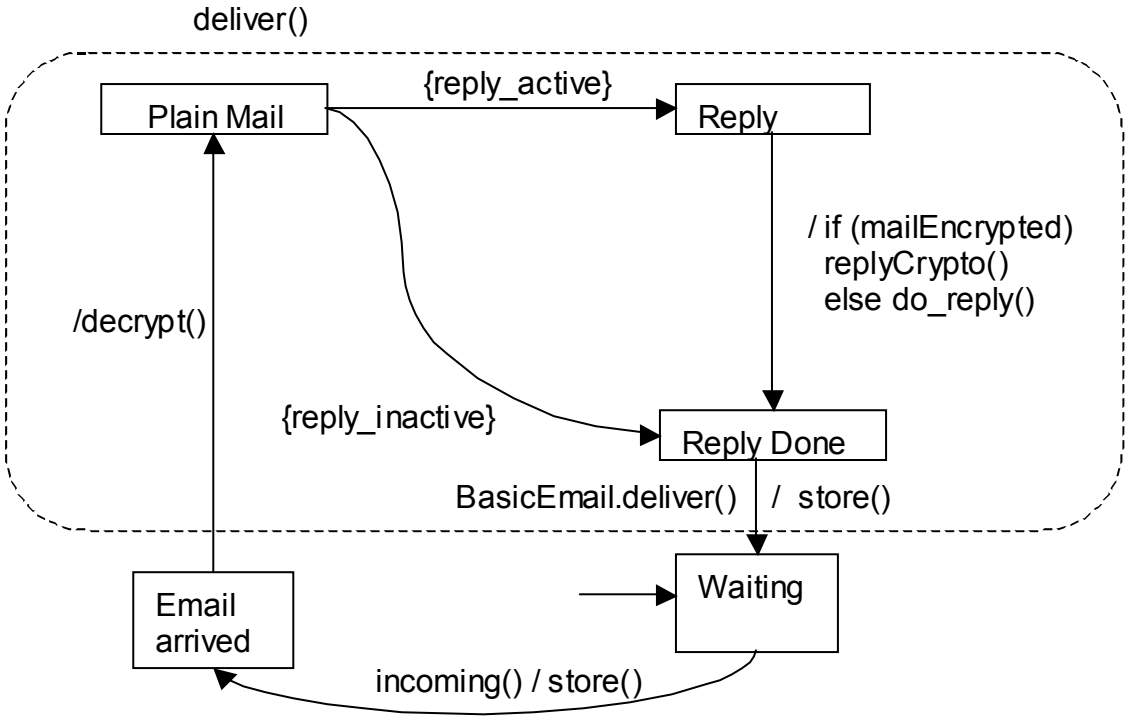

Figure 15: Combination of Basic Email, Auto Reply and Decryption

With respect to the original BasicEmail, the refinement relation is clear; all the new output operations have to be abstracted. For the refinement from the combination of BasicEmail and Auto Reply to this combination of three features, exceptions have to be considered. Since the normal do_reply() function is not used in case of encrypted emails, we have to consider this as an exception. Unless the exception occurs, the refinement holds with the appropriate abstraction of the new output.

5.2 Combination of Base Features

So far, we have expanded transitions for combination. For base features this method is however insufficient. In the case of two base features, we combine features with individual statecharts. We obtain two parallel statecharts with disjoint function labels. This is typically needed if features add external methods and may affect the control state.

For instance, consider a generic locking feature, which disables all function calls to an object which change the state. In the case of the mailer, this may be used to stop receiving/sending, e.g. to configure or inspect the mailer.

The lock feature has two externally visible functions, lock and unlock, and two states. It is shown in Figure 16.

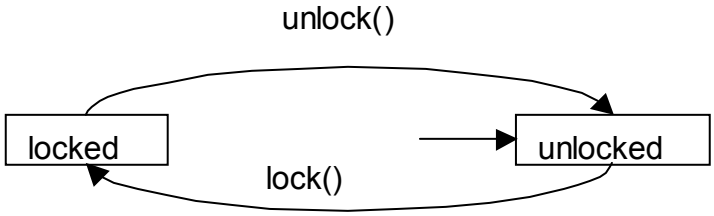

Figure 16: The Lock Feature

When combining the BasicEmail and the lock features, the interaction handling shall block transitions if the statechart is locked. We show the combined statechart in Figure 17; it uses several interaction adaptors which are not shown separately. The first interaction is the blocking of transitions by adding conditions to the transitions, which is a particular form of transition refinement. Adaptors can extend the functions in the parallel statecharts and can add function calls to other statecharts. For instance, an adaptor to lock may invoke the lock() operation of the Lock feature. For illustrating this, we show here a new function of the basic email feature, called reset. When adapting this function to Lock, this transition has to unlock the Lock.

When adding another feature we may have to adapt all the previously adapted functions. With parallel statecharts, only the transitions of one statechart have to be adapted. Furthermore, we need to restrict the transitions of the BasicEmail feature by some conditions which check if the statechart is not locked. These conditions are not detailed here and are specified in an adaptor similar to the above cases.

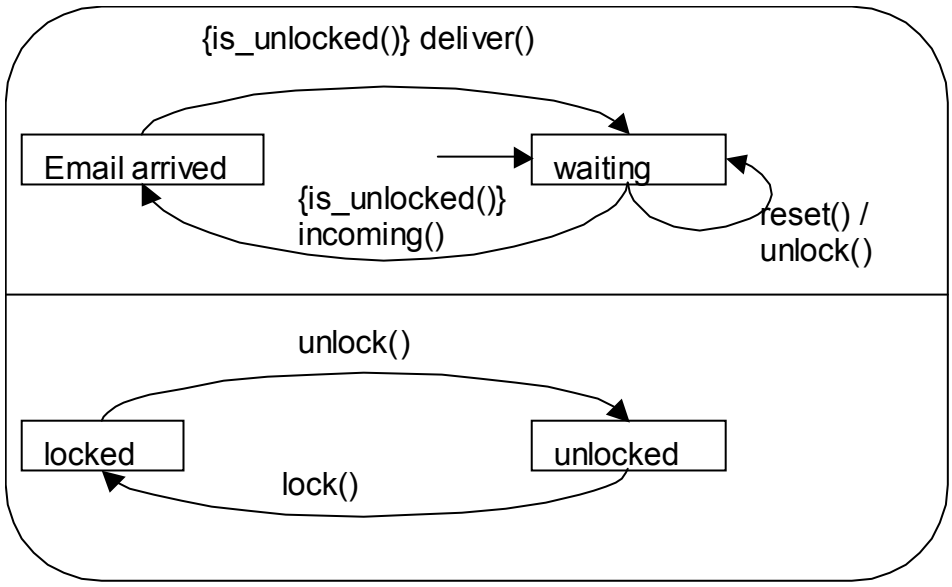

Figure 17: Parallel Statechart with the adapted BasicEmail and Lock Features

An interesting case occurs when a state-oriented feature is added to email which interacts with lock. In this case, only adaptation of transitions is permitted. It is not permitted that interaction handling creates transitions between states of parallel-composed statecharts. This would lead to semantic problems as it contradicts the concept of parallel composition. Since the new statechart only adds new behavior, refinement is easy to show.

5.3 Features which add Global States

In the following, we discuss the combination of features which add global states. For this, we have two steps:

- Merge the statecharts of the new feature with an existing statechart with some features.
- Adapt all the previously added features. For this, we add the transitions and transition refinements for each adaptor of all the features added earlier.

For instance, we can combine the Maintenance, Error, and BasicEmail features as shown in the Figure 18 below. Note that we first add the Maintenance feature, which is then first adapted to the Error feature and then to the BasicEmail feature. Each of these adaptations adds one transitions labeled enterMaintenance(). As these transitions and the states are new, it is easy to see that the semantic refinement relation holds.

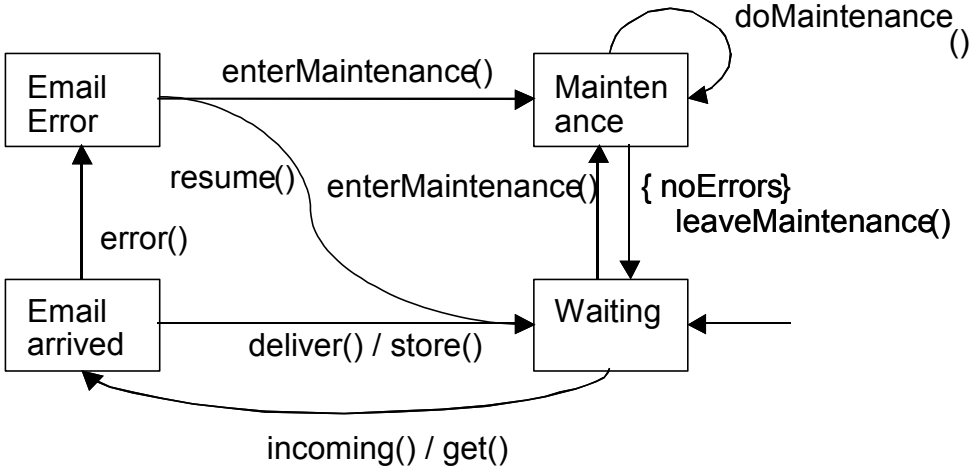

Figure 18: Combination of Maintenance, Error, and BasicEmail features

6 Conclusions

We have presented a set of graphic specification techniques which allow one to structure highly entangled software systems. With our graphic design techniques, we have shown plug-and-play concepts for the construction of complex statecharts, which describe a component with several features. The main contribution is the graphic combination rules of features, which are based on semantic refinement. In this way we can select any combination from a set of features and the complete statechart can be created automatically. As we use each feature only one time and the selected set is ordered implicitly by the feature interaction handlers, we have an exponential number of possible feature combinations. This is based on a quadratic number of interactions.

There exist several interesting extensions in the area of parallel statecharts for further work. For instance, we have focused on sequential composition of statecharts. With parallel statecharts, one may specify features which only affect one of the parallel statecharts or even add the same feature twice in each of the parallel statecharts. Furthermore, we have not considered hiding operations for the external interface. For parallel statecharts, it is possible that one of them is controlled by the other and is not externally visible. In this case, an extension for syntactical interface hiding may be useful.

Our approach was guided by semantic refinement concepts based on a black box component view. We have shown that our combination methods preserve the external interface and only reduce the set of possible execution traces. By nature of our approach, we have only considered syntactic criteria which are compatible with semantic refinement.

For a deeper semantic analysis of the behavior of components, formal specification technologies can be used as e.g. considered in [7,1]. The work in[11] is similar as it aims at incremental refinement on different levels of abstraction, but without graphic description.

There is very little work on modularization of statecharts based description. The work in [5] covers incremental development of statecharts, but does not consider features as independent entities and also does not consider interactions. Most of the work on feature interaction aims at detecting interaction, but does not consider systematic development of features. There is other work on modeling telecommunication features using statecharts [12] which however aims at formal verification using model checking.

7 References

[1] M. Broy and K. Stolen, Specification and Development of Interactive Systems. Springer-Verlag, 2001

[2] Tzilla Elrad, R. E. Filman, A. Bader, Aspect-oriented programming: Introduction, Communications of the ACM, Volume 44 , Issue 10 (October 2001)

[3] Workshops in Feature Interaction 1992-2000, Proceedings published at IOS Press, Netherlands, www.iospress.nl/site/html/tel.html

[4] R. J. Hall, Feature Interactions in Electronic Mail, IEEE Workshop on Feature Interaction, IOS-Press, 2000

[5] C. Klein, C. Prehofer und B. Rumpe, Feature Specification and Refinement with State Transition Diagrams, In.: P. Dini (Ed..), Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems, IOS-Press, 1997.

[6] C. Prehofer, Feature-Oriented Programming: A Fresh Look at Objects, ECOOP '97 -- Object-Oriented Programming, Springer LNCS 1241, 1997.

[7] C. Prehofer, Flexible Construction of Software Components: A Feature-Oriented Approach, Habilitation Thesis, Technical University of Munich, 1999

[8] C. Prehofer, Feature-Oriented Programming: A New Way of Object Composition, Concurrency and Computation. 2001

[9] B. Rumpe and C. Klein, Statecharts Describing Object Behavior, In: Specification of Behavioral Semantics in Object-Oriented Information Modeling, Kilov, H. and Harvey, W.(Eds.), Kluwer, 1996

[10] The OMG's Unified Modelling Language, version 1.4, http://www.omg.org/UML/, 2002

[11] Bengt Jonsson, Tiziana Margaria, Gustaf Naeser, Jan Nyström, and Bernhard Steffen. Incremental Requirement Specification for Evolving Systems. *Nordic Journal of Computing*, 8(1):65-87, Spring 2001.

[12] Gregory W. Bond, Franjo Ivancic, Nils Klarlund, Richard Trefler, ECLIPSE Feature Logic Analysis, Internet Telephony Workshop 2001, April 2001, Columbia University, New York City, USA

[13] M. Kolberg, E.H. Magill, D. Marples and S. Reiff. Second Feature Interaction Contest Results. In M. Calder and E. Magill, editors, Feature Interaction in Telecommunications and Software Systems VI, IOS Press, Glasgow, May 2000.