ITU - Telecommunication Standardization Sector **Temporary Document SAxx-E**

STUDY GROUP 10 **Original: English**

Sophia Antipolis, 98.10.20 – 98.10.23

Question: 9

SOURCE: RAPPORTEUR

TITLE: COMPOSITION OF MSC'S

Contact: Jan Docekal, Telelogic AB, jan.docekal@telelogic.com

Compositions of MSC's

Summary

In this document, we treat how decomposition should be handled. Some of the problems are described in TD026-E Geneva, 28 April –6 May. The document does not explicitly specify which changes should be done to the Z.120. The main purpose is to find out if the principles are sound. The document also brings up some open questions that should be answered for future work. The document is an upgraded version of B906 "Decomposition" presented in Berlin 98. This document takes a new approach to decomposition. It views decomposition as the result of composition. This is approach is taken, not because it is true in all situations, but because it hopefully provides new insights. New information is either directly taken from the minutes in Berlin [B903 Rev2-E] or are based on the discussions from that meeting.

Main principles

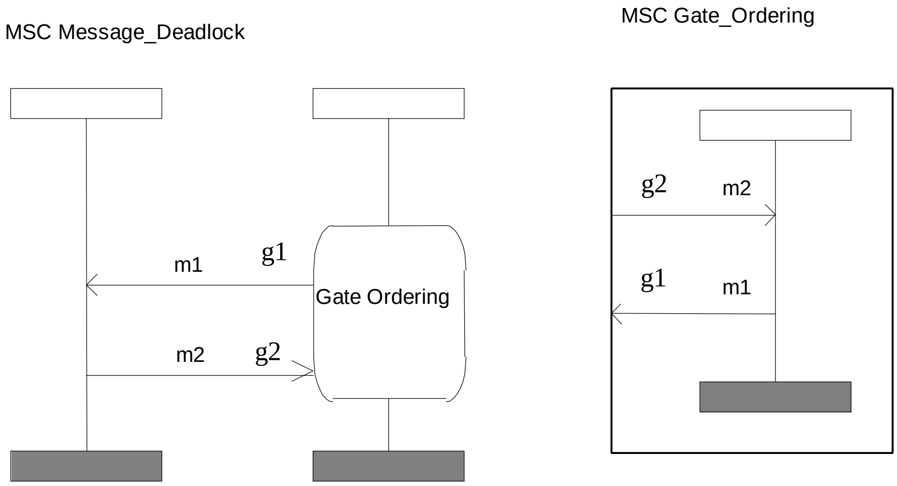
Decomposition should be specified in such a way that when a correct decomposed diagram is expanded the resulting diagram is also correct. In the case that we are forced to expand inline expressions, multiple diagrams will be the result. Diagrams must be constructed in such a way that regardless of in which order references and decomposition's are expanded they expand to the same diagram. This property is called commutative referencing. (In the case that we have inline expressions, only those cases that will render "connected" MSC will be treated as defined. At least one correct MSC must be the result when inline expressions are used. This means that when "expanding" all decomposition's and references we always obtain the same result regardless of the order in with the expansion is performed.

Transformations/Relations of decomposed symbols

This section will explain how a symbol or message placed on a decomposed instance is interpreted in the decomposed diagram and what restriction/rules applies. We will treat the symbols/messages in the order that they appear in Z.120. Inline expressions and MSC reference expressions will be treated in their own sections.

Messages/Environment and gates/General ordering

The main problem with gates is that they are unordered. This leads to the situation described in TD-026-E, 5.3 “Cyclic connectivity graph caused by MSC references with gates”, Geneva 28 Apr. – 6 May 1997.

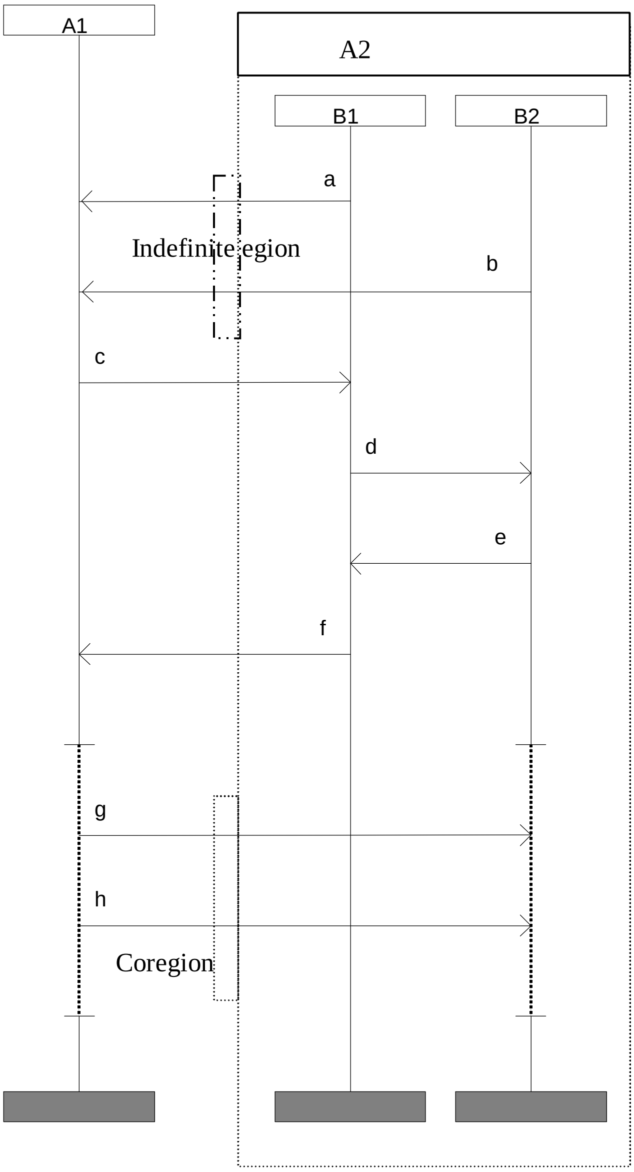


The above example is also applicable if the MSC reference is “replaced” by a decomposed instance, since the two concepts are similar.

The solution to this problem is to require that the gates be ordered in the same way in the gate interface. For PR, the order is specified by the position in the file. For GR, the order is specified by letting the environment be analogous to an environment instance axis (roll the MSC into a vertical cylinder where the environment edges touches each other – this is the environment instance).

If we take a compositional view on decomposition, it comes out quite natural that these relationships must hold. Consider the following MSC that has its decomposition written directly into the diagram:

MSC Composition



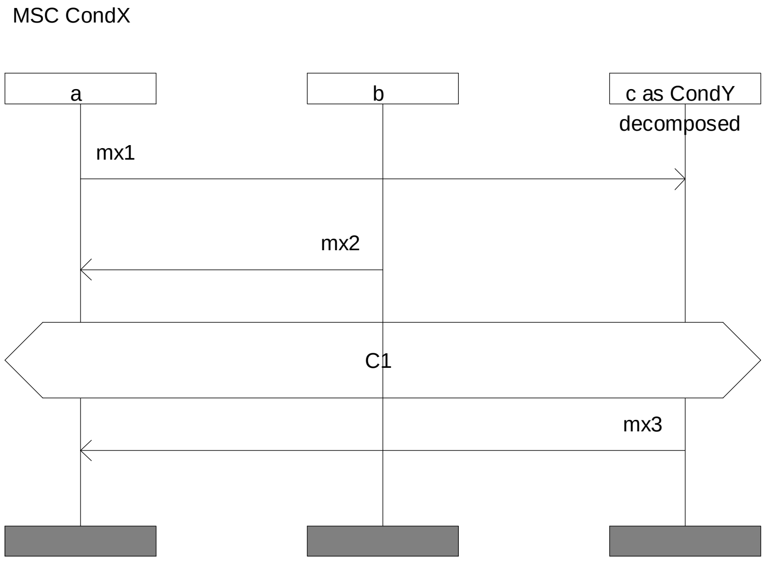
Once the compositional approach has been taken, it is not hard to se that the restrictions on gate ordering are quite reasonable. Since it should be possible to expand decomposition or a reference into a real diagram, which is a requirement for commutative referencing, a diagram that is composed gets implicitly an ordering of the gates on the environment. This ordering should be used to decide if a decomposed diagram fits with its decomposition. Another reason for requiring an ordering of the gates is that it should preserve a users mental health when finding out if diagrams connect. This statement is especially true if you mix decomposition with MSC references. Furthermore adding the requirement that the gates on the environment is ordered guarantees that the specified MSC is deadlock free if message arrows never goes upwards (this is stated in the Z.120 drawing rules). Note that this requirement does not diminish the expressiveness of the language. Instead, decomposition should be viewed as the result of composition, and then the ordering of the gates comes out naturally in the graphical syntax. This in turn puts semantic restrictions on the textual syntax of gate interfaces so that the gate names are ordered according to the new diagram.

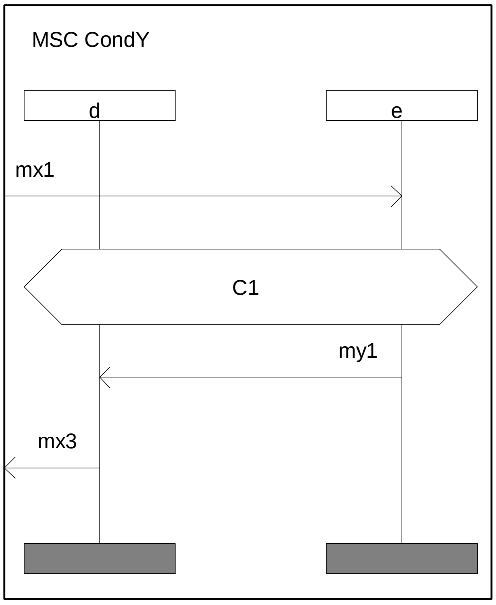
Conditions

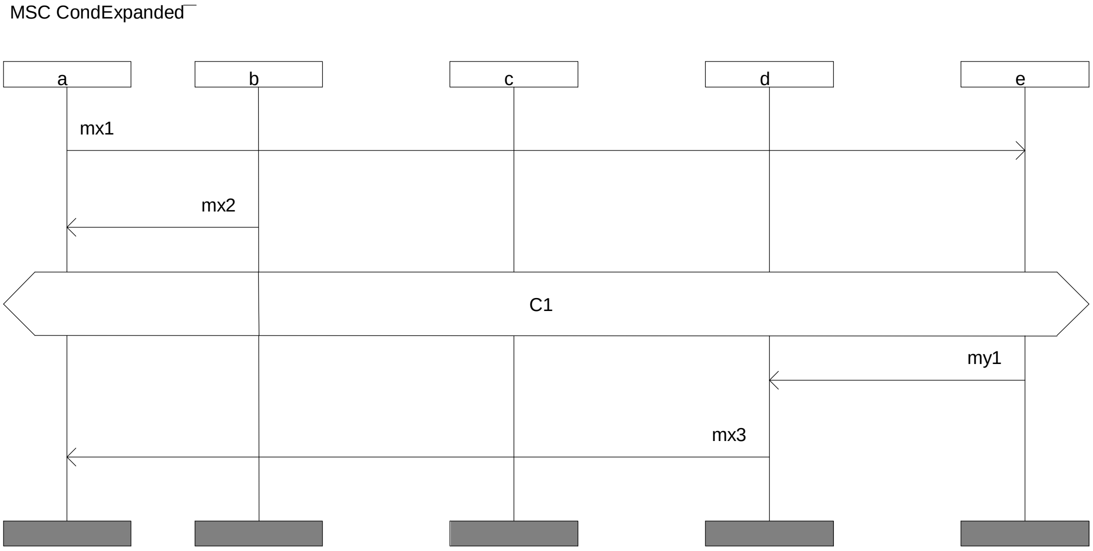
If a condition is present on a decomposed instance, it should also be present in the referenced diagram to comply with the principle of commutative referencing. A condition may also cross several instances of which some are decomposed. In this case, the "expanded" diagram is covered by the same condition.

Note that these requirements are not in line with what is stated in the standard (section 5.2, last two sentences).

Example:







Point of discussion:

- Is this definition too restrictive? Should it be enough that at least one of the instances has a corresponding condition in the relevant place?

Timers

If a timer is present on a decomposed instance, a timer with the same name (and parameters, if applicable) should be present on one of the instances in the referenced diagram at a corresponding location. "Compound" timers must naturally reside on one instance axis. Separate timers may reside on a different axis (as long as they do not constitute the same "physical" timer. It is not sure if this latest statement can be analysed, since we might have an MSC that is not a description since time T=0). (Timers should follow the same rules as messages in general and message to self specifically, se the semantics in section 5.2).

Actions

An Action on a decomposed instance has no formal correspondence in the referenced diagram. This is due to the fact that an action is an informal (and high level) description of what the referenced diagram should do at that point. This is in line with what in the Z.120 standard (section 5.2, last 2 sentences). Note that an action should be mapped to "something". It should be considered wrong to map it to "nothing". The Z.120 has to be updated in this respect.

Instance Creation (of decomposed Instances)

If an instance I is created and this instance is I is decomposed all instances in the referenced diagram must be created. (We are desperately in need of an example).

Open Questions:

- Does the standard describe how the decomposed instance is created at all? Ekkart Rudolph describes this problem in TD40-E, Geneva 24 March – 1 April 1998, and presents a possible solution.

- How are parameters passed to the referenced diagram in the case that two or more instances are created from the same instance creation of a decomposed instance?

There seems to be one logical restriction on decomposed instance creations: All instances in the diagram must be dynamically created, either directly by the (Z.120 non-existent) instance create message, or as normal instance creations. The problem here is not of a semantical nature, but instead to specify a mechanism that has as little restrictions as possible,

Instance Stop

If an instance stop occurs on a decomposed instance all instances in the referenced diagram must be stopped (as stated in Z.120).

Coregion

Coregion and decomposition are not treated in Z.120.

Normally, when sending and receiving messages from a decomposed instance, the actual order is specified by the referenced diagram, the only precondition is that signal sequence should be possible when the diagrams are expanded. Since a decomposed instance in some sense has the flavour of a coregion. Our goal is however to really preserve the coregion property. This leads to a number of cases.

Rules:

- If there is a coregion in the referenced diagram and two or more messages from this coregion goes to the environment, these messages should exit the decomposed instance within an coregion.
- If two messages enter through the same coregion as a decomposed instance, there are two cases:
  1. They got to the same instance, in that case they must enter that instance within the same coregion.
  2. They go to different instances: In that case, these instances must be connected with messages in a way that makes the order from the incoming messages significant. There should be a coregion in a suitable place so that it overrides this dependency. If on the other hand the messages go to different instances that are not sequentially dependent on each other, they should not enter through a coregion. Instead they should enter through an “indefinite” region.
- A combination of the two above if there are both incoming and outgoing messages.

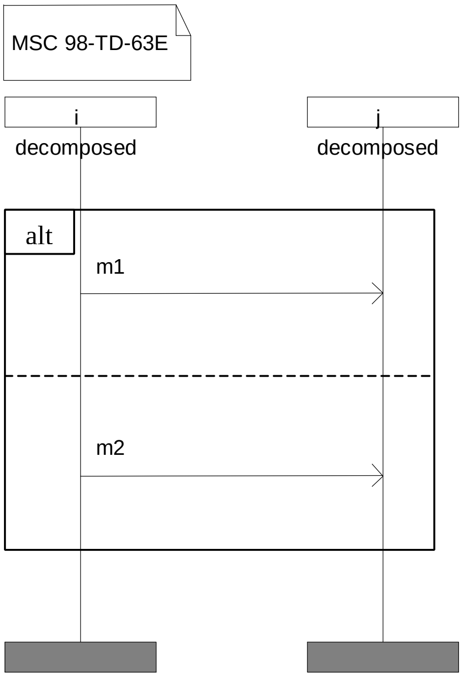
It might also be the case that since we now have **par** as an inline expression, we perhaps no longer need coregions. I propose an investigation to find out if coregion in combination with general ordering is a redundant construct.

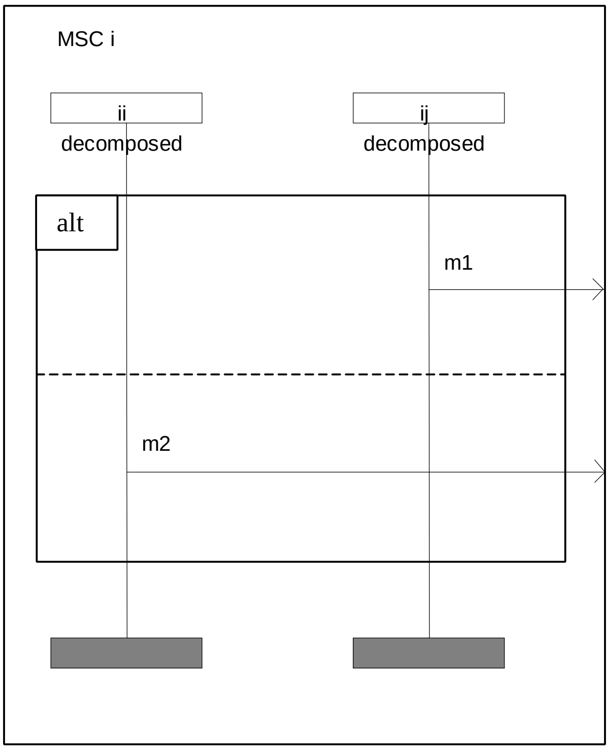
Inline Expressions and MSC Reference expressions

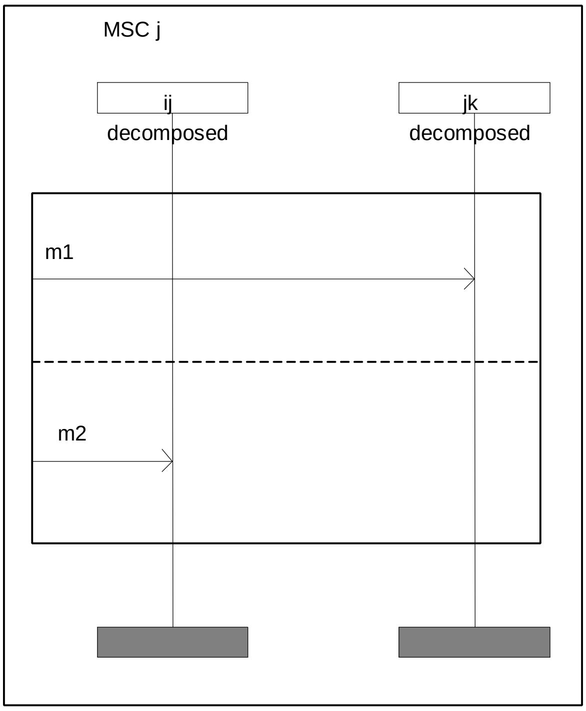
The status in the grammar is that Inline expressions can not be attached to decomposed instances. We naturally want to change this. Please note that the discussion below assumes that the gates are ordered. It is true that this is a restriction compared with the original Z.120, but this restriction does not diminish the expressive power of the language, instead it is used to extend it.

If there is an inline expression enclosing a decomposed instance it is required that the constructs inside the inline expression could be expanded consistently. This means that messages exchanges with the environment in the referenced diagram in principle must follow the same inline expressions structure as the one present on the decomposed instance. Messages (and instance) that are not subject to these restrictions need of course not be enclosed by inline expressions.

The connectivity principle in its simplest case may be described like this:







Unfortunately, not all inline expressions are alternatives. To be able to determine how inline expressions can match each other we will specify a gate interface type algebra to specify how inline expressions can be connected. This algebra will also specify how the inline expressions are expanded. The type can also be viewed as a compatibility profile since there are reduction rules for gate types that specify which gates are compatible with each other.

Algebra

The algebra consists of binary and unary operators. The operators have the same names as the inline expression operators. The parameters for an operator is listed behind the operator (e.g. unary operator X; binary operator X, Y, …). We have the following operators:

- Binary: (**alt**, **par**)
- Unary: (**exc**, **loop**, **opt**)

Gate interfaces

We will now define how gate interfaces are expressed. A gate interface is quite similar to a function/procedure prototype, where data types are listed in some order. The difference with gate interfaces is that equal types can not be connected. Instead, types that are the complement of each other fit together. The interface A has a complement interface A’. Suppose we have the gate interface A which have gate interface (*out* g1, *in* g2) then this interface can be connected to A’ (*in* g1, *out* g2). Please note that when connecting interfaces trough decomposition not only messages are part of the interface, but other constructs (e.g. timers, coregions, conditions, etc.). These constructs are however not treated below.

When a gate interface is connected to another it will be written as (A : A’).

When a transformation rule will be defined for a gate interface it will be written as *expr* => *expr*. Please note that the transformation rules below are only applicable to gate interface compatibility.

When there are several interfaces of the same kind, they are denoted by an index (e.g. A_1, A_2). The index 0 (e.g. A_0) means the empty set.

Rewrite rules and Possible traces

The rewrite rules reduce the complexity of an expression. If you have two expressions and want to see if they may connect to the same expressions, apply the reduction rules to them until no more rules are possible to apply. If they are equal then they may be connected to the same interface.

The possible traces that are listed after each rewrite rule assumes that the expression that describes a gate interface that is on the left side has been fitted to an gate interface that is the complement of the expression on the right side of the rewrite rule.

Please note that rules marked transmutation do not reduce the complexity of the expression, instead it is transformed into another form.

seq

This is not an inline expression (it is an MSC reference expression) but will be treated here anyway. A specific gate interface will be denoted by capital letters so A might consist of gates *in* g1 and *in* g2 (in that order!) and B might consist of gate *in* g1 and *out* g3. This could be expressed with the **seq** operator

A = *in* g1 **seq** *in* g2; B = *in* g1 **seq** *out* g3, A’ = *out* g1 **seq** *out* g2

In our type system, we will however simply write:

A, B, A’

Note: The trivial case where two equal expressions are connected to each other is described first in all rules. They are added to make the notation simpler to understand, but are *not* part of the rewrite rules. The normal connection also uses the notation for possible traces.

alt

alt A_1, A_2, …, An : **alt** A'_1, A'_2, …, A'_n	(A_1 : A'_1), (A_2, A'_1), …, (A_n : A'_1), (A_2 : A'_1), (A_2 : A'_2), …, (A_2 : A'_n), …, (A_n : A_1), …, (A_n : A'_n)
Rewrite Rule	*All possible traces*
alt A_1, A_2 => A	(A_1 : A’), (A_2, A’)
alt A_1, A_2, …, A_n => **alt** A_1, A_2, ..,A_{n-1}	(A_1 : A'_1), (A_2, A'_1), …, (A_n : A'_1), … , (A_n : A'_{n-1})
alt A_1, A_2, …, A_n , B => **alt** A_1, …A_{n-1} , B	(A_1 : A'_1), (A_2, A'_1), …, (A_n : A'_1), … , (A_n : A'_{n-1}), (A_1 : B’), …, (A_n : B’)

exc

(A **exc** B) C : (A’ **exc** B’) C’	(A B : A’ B’), (A C : A’ C’)
Rewrite Rule	*Possible traces*
(A **exc** B), C, D, … => (A **alt** B), C, D, …	(A B : A’ B’), (A, C, D, … : A’ C’ D’, …)
A **exc** B => A opt B	(A : A’), (A B : A’ B’)

Since the **exc** operator always covers all instances in an MSC, these mappings are only applicable when the mapping is performed on an decomposed instance and only when raising the abstraction level. It could also be the case that new rules for continuations will change how the **exc** operator works, and that it will only be possible to map **exc** to **exc.** Currently, the scope for and **exc** is an MSC. With continuations, this may change and then the rules above become invalid. It is perhaps wise to have this restriction until continuations have been settled.

loop

loop <0, n> A : **loop** <0, n> A’	$(A_0, \ldots A_n: A'_0, \ldots A'_n)$
Rewrite Rule	*Possible traces*
loop <1, 1> A => A	(A : A’)
loop <1, n> A => **loop** <1, n-1> A where 1 < n <= **inf**	$(A_1, \ldots A_{n-1} : A'_1, \ldots A'_{n-1})$
loop <n, **inf**> A => **loop** <n+1, **inf**> A where 0 < n < **inf**	$(A_{n+1}, \ldots A_{\mathbf{inf}} : A'_{n+1}, \ldots A'_{\mathbf{inf}})$
loop <0, 1> A => A	(A : A’)
loop <0, 1> A => **opt** A	(:), (A : A’)
loop <n, m> A => $(A_n, \ldots A_m)$ where -1 < n =< m < **inf**	$(A_0, \ldots A_n: A'_0, \ldots A'_n)$ Transmutation
loop <0, n> A => **opt** $(A_1,\ldots,A_n)$ where 0 < n < **inf**	(:), $(A_1, \ldots, A_n : A'_1, \ldots, A'_n)$ Transmutation

opt

opt A : **opt** A’	(:), (A : A’)
Rewrite rule	*Possible traces*
opt A => A	(A : A’)

par

Since a **par** expression gate interface is similar to a coregion we get the following rules:

par A, B, …, X, Y : **par** A’, B’, …, X’, Y’	All permutations…
Rewrite rule	*Possible traces*
par A, B, …, X, Y => (par A, B, …., X) **seq** Y	(see example below)
par A, B => A’, B’	(A, B : A’, B’), (B, A : A’, B’)

Note that since no notation for parallel traces have been given only a simple example is given for possible traces, since the **par** operator quickly expands into many different traces.

Example of applying the rules

Suppose we have the following MSC Reference expressions A1 **alt** B1 **alt** C1 **opt** D1 and A1 **alt** B1 **loop** D1. The MSC's have the following Interfaces:

MSC	Gate Interface	MSC	Gate Interface
A1, B1	X	A2	X'
C1	Y	B2	Y'
D1	Z	C2	Z'

We transform the first interface to gate expressions and get:

(**alt** X, X, Y) **opt** Z

(**alt** X, Y) **opt** Z

(**alt** X, Y) Z

Now this expression can not be reduced any more, so we make the same procedure for the other expression:

(**alt** X' Y') **loop** <0, **inf**> Z'

(**alt** X' Y') **loop** <1, 1> Z'

(**alt** X' Y') Z'

Now no more reductions can be performed. If we compare both expressions we see that their gate interfaces match.

Nested inline expressions

It is not as complex as it seems, since it is the outermost inline expression that specifies the gate interface wee don not need to treat nesting of inline expressions.

Inline expressions and decomposition

Inline expressions interface through decomposition in the same way as inline expressions interface with each other.

Example:

In the decomposed diagram you have the gate interface alt A, A, A, A. The decomposed instance might then expose either gate interface A' or gate interface alt A', A' etc. One restriction could however be considered: Reduction rules may only be applied when raising the abstraction level. (e.g. you have an loop construct in the referenced diagram in an decomposition but on the decomposed instance you have three messages). It might also be the case that these restrictions only should be true for some reduction rules.

MSC References

The status in the grammar is that Inline expressions can not be attached to decomposed instances. We naturally want to change this.

To resolve the problem the principle of commutative referencing has been established se the example in "Main principles" above. Most probably, these principles must be formalised.

Note that this principle only describes the case when we have “MSC references” not when we have “MSC reference expressions”. In the later case, we have an analogous situation as with inline expressions.

Appendix A has an example on how decomposition and MSC’s interact when decomposition and references are combined.

Gates and MSC references

Z.120 says:

Gates, which are not connected on an MSC reference, are implicitly defined to propagate to the next enclosing frame (either MSC frame or inline expressions frame). A propagated gate will have the same gate name as the gate it propagated from. This is a practical shorthand which saves a connection line which otherwise would clutter the diagram.

I would like to point out that this approach does not distinguish information hiding from information “obscurement”. Unfortunately, drawing messages in HMCS becomes unpractical if the user is forced to draw messages to the frame form the MSC references. In addition, MSC’s can refer to HMSC’s, which have no “observable” gate interface. I propose an investigation that should find out if references can and should expose their gate interface inside MSC’s.

MSC Reference expressions

MSC reference expressions has additional complexity compared to MSC references since they are a combination of MSC references and inline expressions. Each referenced diagram has a gate interface type. These gate interface types are then synthesised into the new type by the operators in the MSC reference expression and the rules above.

Gates and MSC reference expressions

MSC reference expressions ads additional complexity to MSC references which may make it even harder for the user to know what gate interface he is dealing with.

Action Points and/or Questions

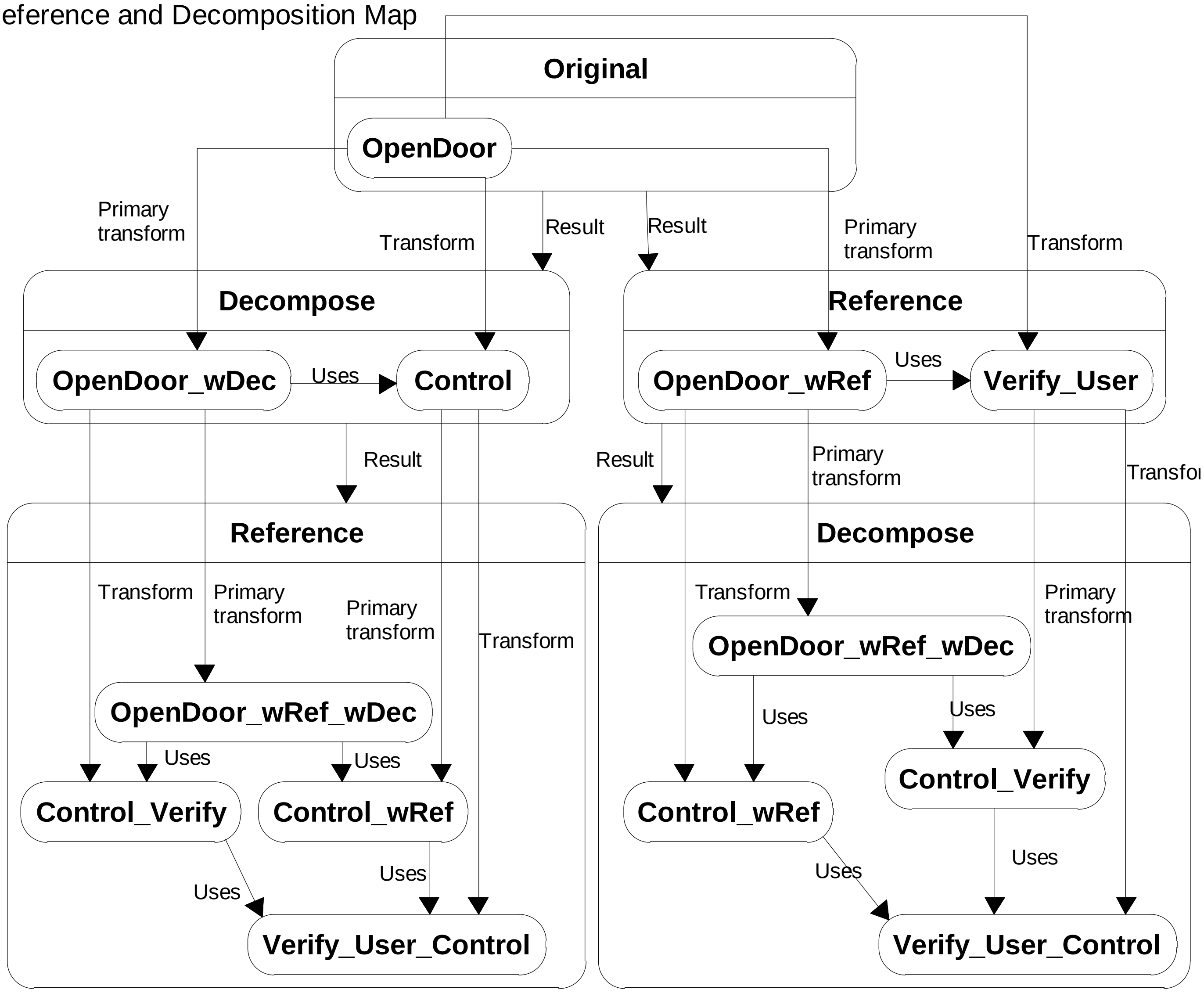
In this section, I summarise which issues I would appreciate that the meeting should take a position to.

1. Are the expansion rules for inline expressions reasonable. Note that these rules are also applied for the definition of inline expressions on decomposed instances.
2. Find out if MSC references in MSC’s always should expose their gate interface.
3. How should instance creation parameters be passed to the instances created in an decomposition.
4. Is coregion with general ordering an redundant construct.
5. Should a condition on a decomposed instance corresponds to a global condition in the referred diagram, or is it enough that at least one instance in the referred diagram has a corresponding condition.

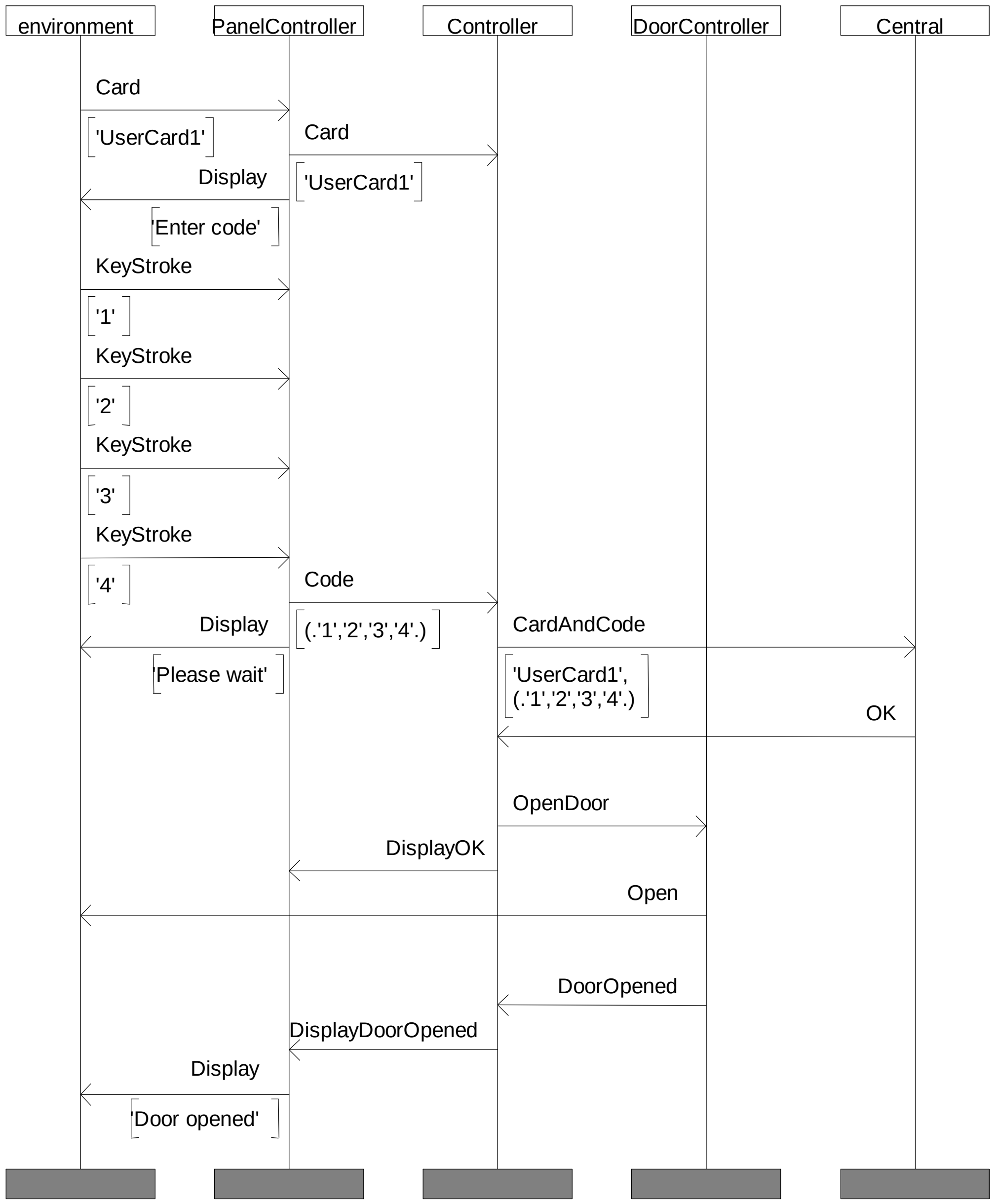
Appendix A: MSC references and Decomposition.

The first diagram gives a road map of what happens when decomposition and references are added to an MSC Diagram. The following diagrams are the MSC's refernced in the road map. Note that in a normal case, the diagram OpenDoor would not changed name. To be able to follow what happens the we have changed the names in this example. To track which diagram is the "original", please follow the arrows that have "Primary transform" attached to its arrows. Note also that the final names of the diagrams in a real world case would differ depending on the order in which the operations where performed, due to different abstractions.

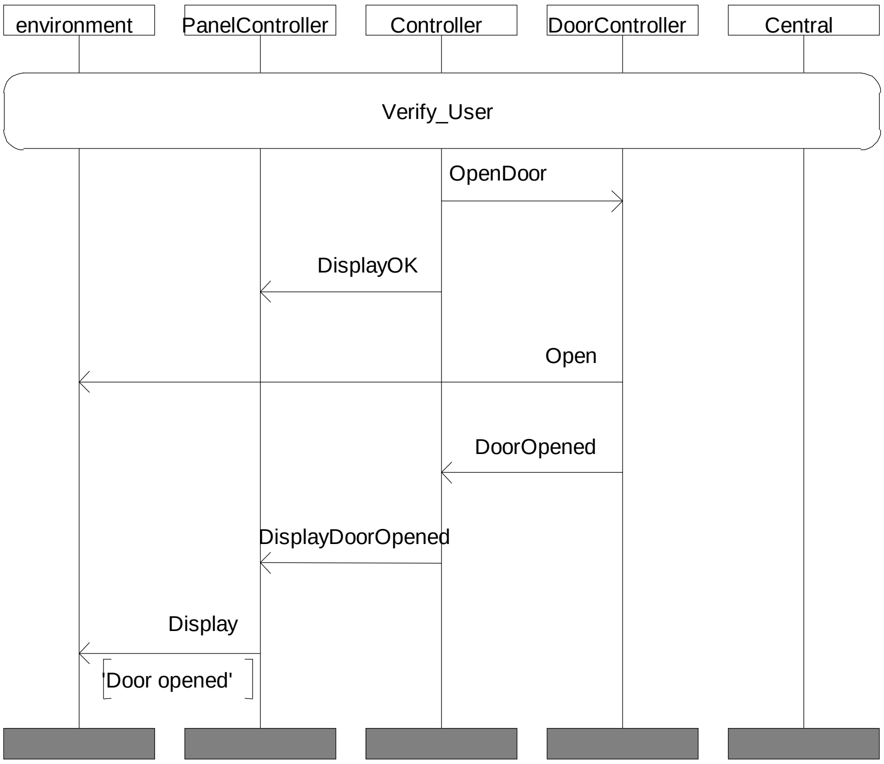
Reference and Decomposition Map



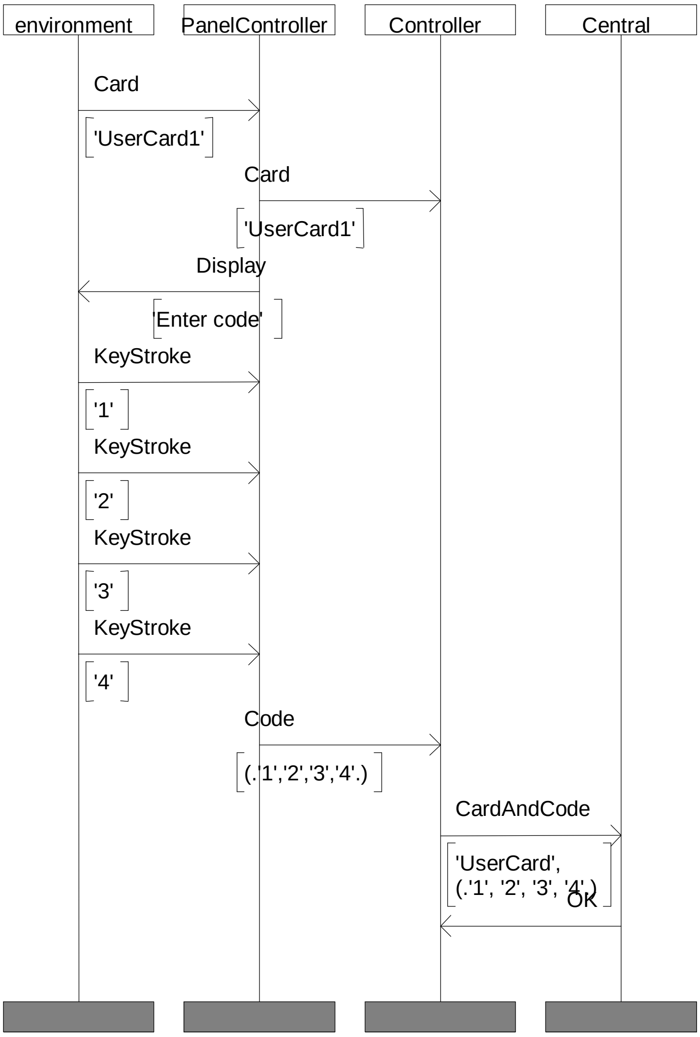
MSC OpenDoor



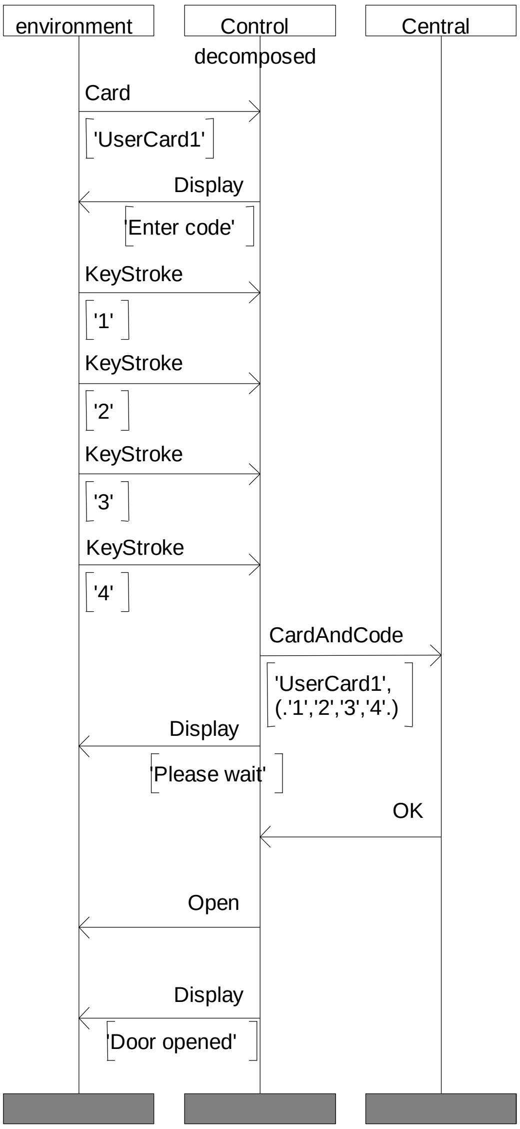
MSC OpenDoor_wRef

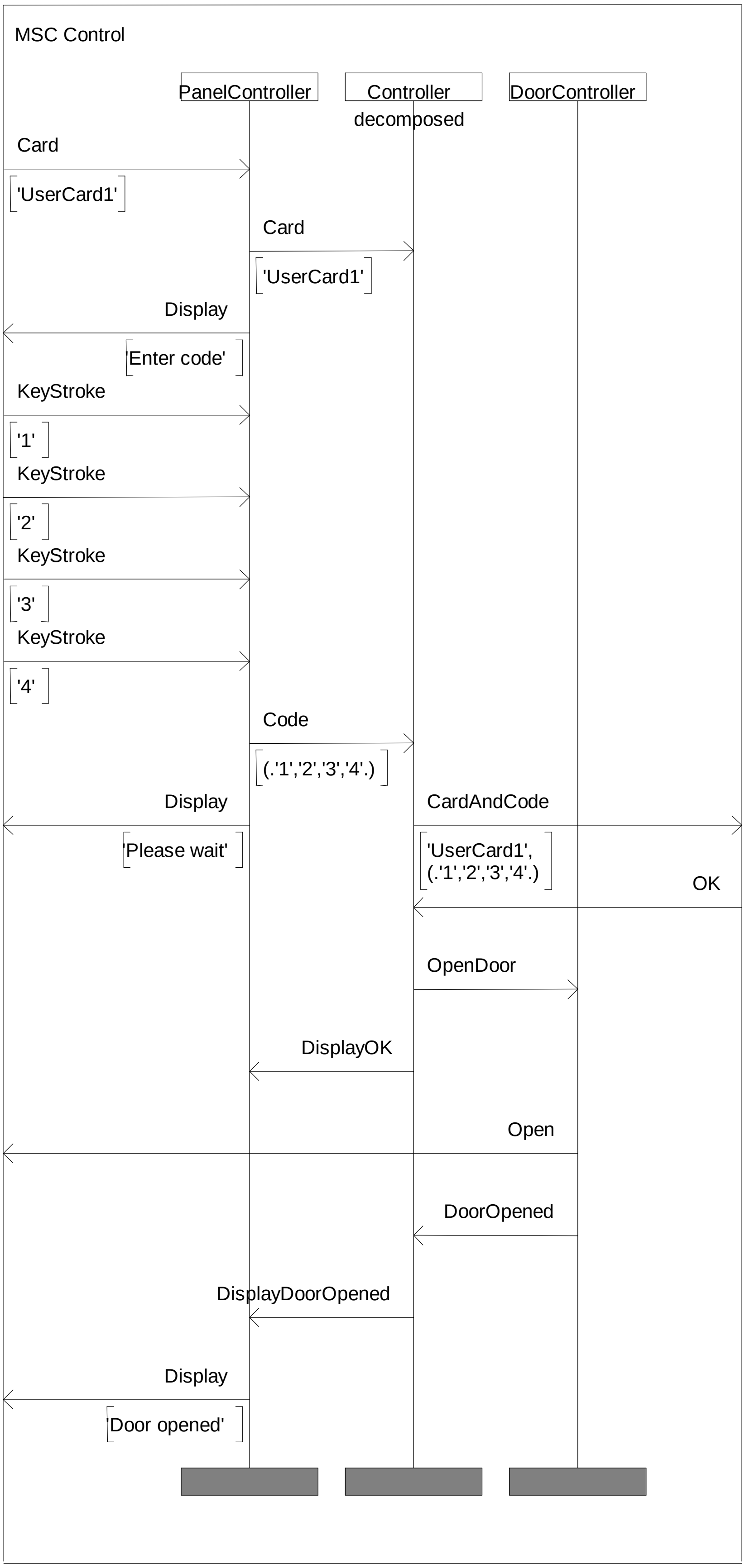


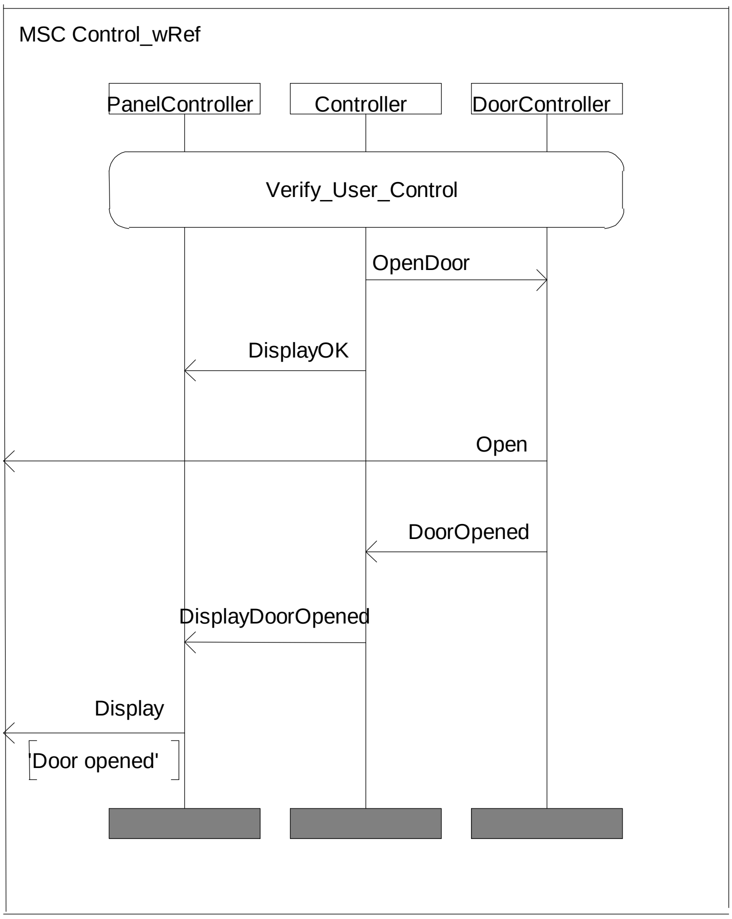
MSC Verify_User



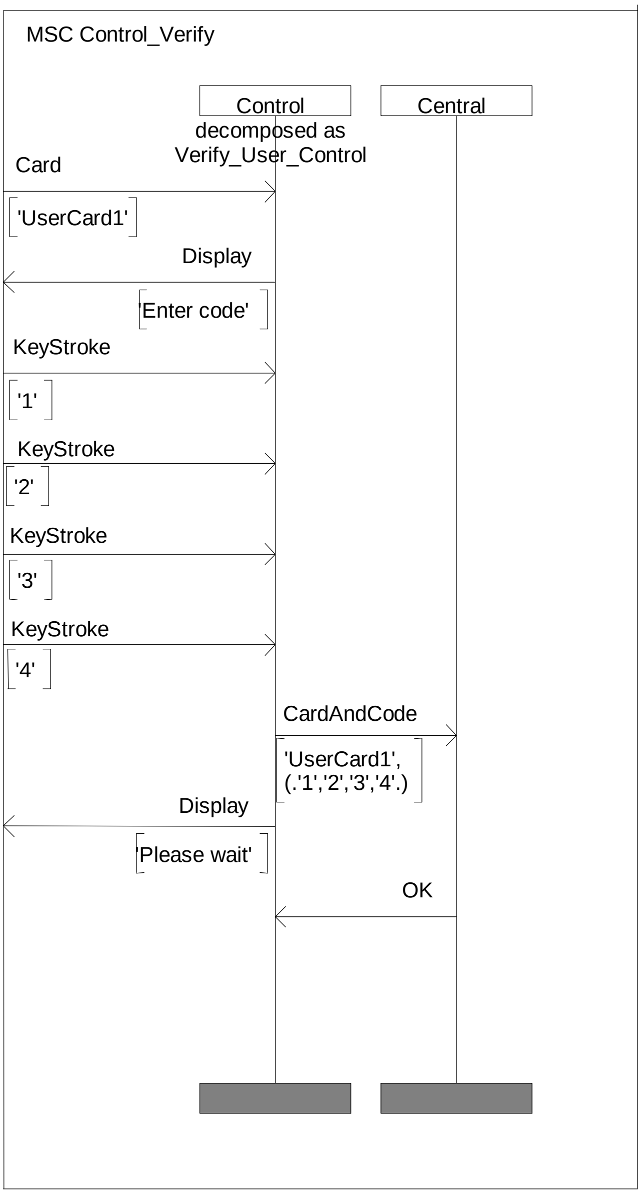
MSCOpenDoor_wDeco



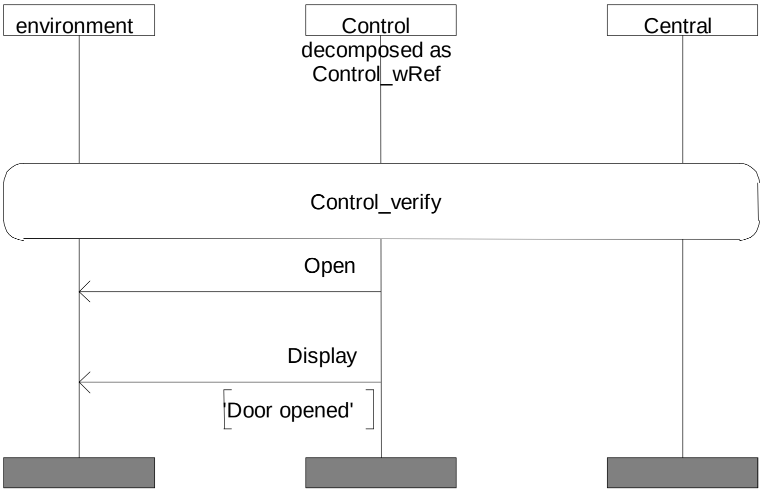

MSC Control
PanelController
Controller
decomposed
DoorController
Card
'UserCard1'
Card
'UserCard1'
Display
'Enter code'
KeyStroke
'1'
KeyStroke
'2'
KeyStroke
'3'
KeyStroke
'4'
Code
(.'1','2','3','4'.)
Display
'Please wait'
CardAndCode
'UserCard1',
(.'1','2','3','4'.)
OK
OpenDoor
DisplayOK
Open
DoorOpened
DisplayDoorOpened
Display
'Door opened'

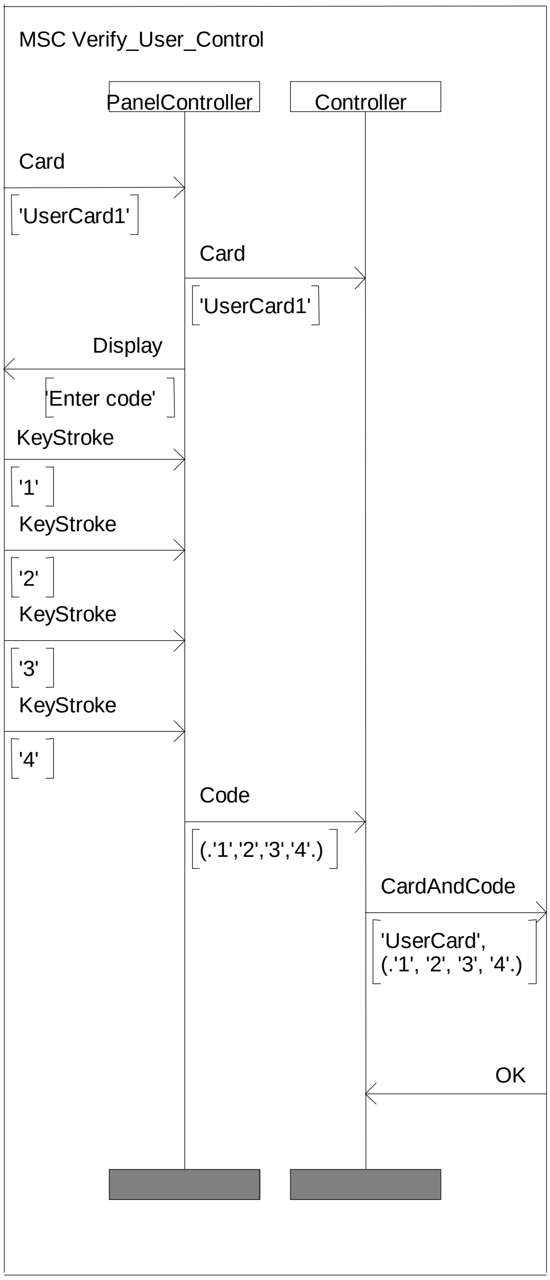

MSC Control_wRef
PanelController
Controller
DoorController
Verify_User_Control
OpenDoor
DisplayOK
Open
DoorOpened
DisplayDoorOpened
Display
'Door opened'

MSC Control_Verify



MSC OpenDoor_wRef_wDec



MSC Verify_User_Control
PanelController
Controller
Card
'UserCard1'
Card
'UserCard1'
Display
'Enter code'
KeyStroke
'1'
KeyStroke
'2'
KeyStroke
'3'
KeyStroke
'4'
Code
(.'1','2','3','4'.)
CardAndCode
'UserCard',
(.'1', '2', '3', '4'.)
OK