Syntax and semantics of synchronous interworkings
I
Introduction

S. Mauw[1] M. van Wijk[2] T. Winter[3]

Interworkings are used in the analysis phase of the development process at PKI Nürnberg for describing the message interactions between functional blocks. The specification of one functional block contains the most common sequences of message interactions with all the neighbour blocks it communicates with. Interworkings are a synchronous variant of message sequence charts mentioned below. In close cooperation with PKI Nürnberg a project is performed in which an Interworking Tool Set (ITS) is specified and prototyped that can process, analyse, and combine interworkings. The first ideas about this tool set were raised in discussions with B. Lurz from PKI Nürnberg in 1991.

Message sequence charts have been used for a long time by CCITT Study Groups in their recommendations and within industry, according to different conventions and under various names. Initiated by Dr. E. Rudolph, the language for message sequence charts is being standardised by the CCITT. The reason to standardise message sequence charts is to make it possible to provide tool support for them, to exchange message sequence charts between different tools, to ease the mapping to and from SDL specifications, and to harmonise the use within the CCITT. Interworkings are also playing an important role in other areas than the telecommunication industry. Several object-oriented methods have absorbed Message Sequence Charts (ObjectOry [6], Rumbaugh [14]).

Compared to other trace languages interworkings have the advantage of a clear graphical layout and structuring. However, interworkings are only suitable for the description of relatively small parts of the system behaviour. Therefore, in order to describe the behaviour of a system more completely, composition operations on interworkings similar to the composition operations in for example LOTOS are required. In the literature already several solutions have been proposed in this area. In [13] the syntax of message sequence charts is enhanced with the concept of "conditions" providing the means to combine message sequence charts sequentially. The sequential composition operation is only described implicitly in this document. No other composition operations on message sequence charts are defined. In [3] a system is described as a hierarchy of services increasing in detail. The lowest level of services in the hierarchy is described in terms of message sequence charts. For the services several composition operations are defined (e.g. sequential-, parallel-composition, choice-operator). Although services can be combined the document does not discuss the related composition of the message sequence charts.

[1]Department of Mathematics and Computing Science, University of Technology Eindhoven, email:sjouke@win.tue.nl

[2]Department of Mathematics and Computing Science, University of Technology Eindhoven, Part of the work has been carried out to get a masters degree in computing science

[3]Philips Research Laboratories, Eindhoven, email:winter@prl.philips.nl

A collection of interworkings describes the behaviour of a system on a high level of abstraction. Each interworking is a projection of a part of the communication behaviour of a system onto a set of entities. In a telecommunication context where for example SDL is used as description language, an entity may be a process or a set of processes combined into one functional block. Interworkings are event oriented instead of entity oriented and only synchronous communication actions can be modelled. This means that messages can not be delayed as in message sequence charts where communication is asynchronous. Because in practice, this does not seem to be a restriction, we concentrate ourselves in this document on the simpler model of synchronous communication and keep the asynchronous variant for further study.

In order to reason about the meaning of interworkings and in particular about the composition operators on interworkings and consistency of interworkings, we needed a formal definition of the semantics. For this formal definition of interworkings and the composition operators we explore techniques from process algebra [1]. Process algebra is a commonly accepted technique for the description of communicating systems. Algebraic reasoning is very suitable for formal verification activities which concern in our case the consistency checking of interworkings. Additionally, via process algebra, it is easy to generate prototypes implementing the various operators, and a link to other algebraic description methods like for example LOTOS can easely be made. Although we restrict ourselves in this document to synchronous interworkings, we think that an extension to asynchronous communication actions (c.f. Message Sequence Charts) is possible. In [15], a formalisation of message sequence charts is given in terms of set theory. This formalisation has the advantage that it only uses basic constructs but has the disadvantage that the parallel composition operator can not be easily expressed in the formalism.

In chapter II of this document, we will define the concrete textual syntax `IW` for interworkings. In chapter III, we will give an informal description of the semantics of interworkings and a description of the composition operators. The informal semantics is based upon the formal description in chapter V. In order to be independent of the concrete syntax `IW` and in order to abstract from less relevant details we have introduced an intermediate language `T` for the description of the semantics of `IW`. In chapter IV this intermediate language `T` is described together with the translation rules from `IW` to `T`. In chapter V, a formal semantics for interworkings is defined with the use of the algebraic concurrency theory BPA.

For people who are only interested in the usage of interworkings the first two chapters should contain sufficient information to work with. The translation of `IW` to `T` and the formal semantics description is meant for those who want to get a thorough understanding of the semantics. Tool development for interworkings for example has to be based upon this formal description rather than on the informal description.

Acknowledgements

Thanks are due to many people for giving fruitful comments on the documents. Special thanks go to Jan Bergstra for his constructive advises concerning both the documents and the project, and to Jos Baeten for his comments on process algebra. We also want to mention Lex Augusteijn for his help with regard the Elegant compiler generator tool set which we used for the development of

prototype tools.

Syntax and semantics of synchronous interworkings

II

The concrete syntax IW

S. Mauw[1] M. van Wijk[2] T. Winter[3]

II.1 Introduction

In this document, a concrete syntax for interworkings, denoted by `IW`, is defined. The syntax is based upon the syntax as currently being used at PKI Nürnberg. In order to support the required functionality of the Interworking Tool Set we had to extend this syntax slightly. The extended syntax presented here only contains small modifications with respect to the interworking syntax as currently being used.

II.2 Syntax

II.2.1 Introduction

A description of the concrete syntax for interworkings is given below. The various sorts of comments are not included in the BNF notation in order to concentrate on the description of the behaviour of the processes. The various comment constructs will be discussed separately in section II.2.4.

We present the syntax by means of BNF rules of the following form:

expr1 expr2	*expr1* followed by *expr2*
expr1 | *expr2*	choice between *expr1* and *expr2*
[*expr*]	optional occurrence of *expr*
{ *expr* }	zero or more occurrences of *expr*
{ *expr* }+	one or more occurrences of *expr*
'string'	terminal symbol
expr1 // *expr2*	sequence of one or more *expr1's* separated by *expr2's*

II.2.2 BNF

```
iwdset                 ::= { iwd } .
```

[1]Department of Mathematics and Computing Science, University of Technology Eindhoven, email:sjouke@win.tue.nl

[2]Department of Mathematics and Computing Science, University of Technology Eindhoven, Part of the work has been carried out to get a masters degree in computing science

[3]Philips Research Laboratories, Eindhoven, email:winter@prl.philips.nl

```
iwd                   ::= 'INTERWORKING' iwname [ paramlist ]
                          'PROCESSES' processes 'ENDPROCESSES'
                           expressionlist
                          'ENDINTERWORKING' .

iwname                ::= Ident .

processes             ::= processname // ',' .

processname           ::= Ident .

expressionlist        ::= { expression } .

expression            ::= transmission
                      |   lost-transmission
                      |   macro
                      |   xmacro
                      |   process-action
                      |   timer-statement .

transmission          ::= processname 'SENDS' message
                          'TO' processname [ action ] .

lost-transmission     ::= processname 'SENDS' message
                          'TO' processname 'LOST' .

message               ::= msgname [ paramlist ] .

msgname               ::= Ident .

paramlist             ::= '(' [ parameters ] ')'
                      |   '{' [ parameters ] '}' .

parameters            ::= parameter // ',' .

parameter             ::= Ident [ paramlist ] .

action                ::= 'ACTION' actionname .

actionname            ::= Ident .

macro                 ::= 'MACRO' macroname [ paramlist ]
                          [ on-processes ] .

on-processes          ::= 'ON' processes .

xmacro                ::= 'XMACRO' macroname [ paramlist ] .

macroname             ::= Ident .

process-action         ::= processname action .

timer-statement       ::= timer-set
                      |   timeout .

timer-set             ::= processname 'SET' timername
```

```
                       ’(’ duration ’)’ .

timername             ::= Ident .

duration              ::= Ident
                      |   nat .

timeout               ::= processname ’TIMEOUT’ timername
                          [ action ] .

Ident                 ::= letter { letter | digit | symbol } .

letter                ::= ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’
                        | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ | ’m’ | ’n’
                        | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’
                        | ’v’ | ’w’ | ’x’ | ’y’ | ’z’ | ’A’ | ’B’
                        | ’C’ | ’D’ | ’E’ | ’F’ | ’G’ | ’H’ | ’I’
                        | ’J’ | ’K’ | ’L’ | ’M’ | ’N’ | ’O’ | ’P’
                        | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ | ’W’
                        | ’X’ | ’Y’ | ’Z’ .

symbol                ::= ’~’ | ’’’ | ’‘’ | ’@’ | ’#’ | ’$’ | ’^’
                        | ’&’ | ’_’ | ’+’ | ’=’ | ’|’ | ’\’ | ’[’
                        | ’]’ | ’<’ | ’>’ | ’?’ | ’/’ | ’.’ | ’:’
                        | ’;’ | ’"’ .


digit                 ::= ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’
                        | ’7’ | ’8’ | ’9 ’.


nat                   ::= digit { digit } .
```

II.2.3 Example

Hereafter, a small example of an interworking in the concrete syntax will be given. The graphical representation of this interworking can be found in figure **??** below.

```
INTERWORKING example

PROCESSES A, B, C, D
ENDPROCESSES

  D SENDS u TO C
  A SENDS v TO B
  B SENDS w TO C
  C SENDS x TO B
  B SENDS y TO A
  C SENDS z TO D

ENDINTERWORKING
```

II.2.4 Comments

Three different forms of comment may be included in interworkings. All three forms of comment run to the end of line. Below follows a short description of these three forms of comment. The description is extracted from the documentation on the current Interworking Processor.

- (`! comment`) Lines starting with a `!` are passed directly to the output. Trailing comments starting with a `!` are appended to the right side of the output line.
- (`* comment`) Lines starting with a `*` produce an "empty" output line containing instance bars to which the comment is appended at the right. This type of comment may not occur as trailing comment.
- (`% comment`) Comment starting with a `%` does not produce any text in the output file.

II.2.5 Hyphenation

The hyphen character ("`-`") is used as line continuation symbol. A line feed after this continuation symbol will be discarded and the statement continues on the next line. The following example will be regarded as a syntactically correct statement.

```
P1 SENDS msg -
TO P3 ACTION act
```

The interworking syntax as it was used by PKI up till now was line oriented and thus a line continuation symbol was required in order to write a statement over multiple lines. The syntax as described above is not line oriented and the line continuation symbol is therefore not necessary any more. However, in order to comply with existing interworkings we have maintained the functionality of this symbol in the syntax.

II.2.6 Parameters of interworkings

Parameters of an interworking are textual placeholders for message names, process names, or message parameter names. A formal parameter of an interworking is implicitly typed by its occurrence in the expression list of the interworking. Only identifiers may be used as formal parameter, and a formal parameter may not occur both as process and as message or message parameter in that interworking, or as message and as message parameter.

An example of a parameterised interworking is given below. Note that the interworking is equal to the interworking in the example above, except that two parameters are added. Note that the first actual parameter in the corresponding macro call must be a process, and the second parameter must be a message name.

```
INTERWORKING macro_example (X, x)

PROCESSES X, B, C, D
ENDPROCESSES
```

```
  D SENDS u TO C
  X SENDS v TO B
  B SENDS w TO C
  C SENDS x TO B
  B SENDS y TO X
  C SENDS z TO D

ENDINTERWORKING
```

Syntax and semantics of synchronous interworkings
III
Informal Semantics

S. Mauw[1] M. van Wijk[2] T. Winter[3]

III.1 Introduction

In this document we will describe the concepts of interworkings and the composition operators informally. In section III.2 we will describe the semantics of interworking diagrams. After that, we will describe the two composition operators, interworking sequencing and interworking merge. Finally in section III.5, we will define the interworkings that can be constructed by means of the syntactical constructs and the two composition operators as well as some properties.

III.2 Interworking diagrams

III.2.1 Syntax

The basic graphical language of interworking diagrams is very simple and resembles the syntax for basic sequence charts [12]. Vertical lines are used to represent entities within a system and horizontal arrows are used to represent communications between two entities. These communication actions are interleaved with other constructs denoting behaviour of individual entities such as actions, timer statements, and lost messages. For these constructs the concrete graphical syntax is not yet defined and semantically they are of less interest than the communication actions. In this document we will therefore concentrate on the communication actions only. A detailed discussion on the syntax of interworkings lies beyond the scope of this document and for a definition of the concrete syntax we refer to [7].

III.2.1.1 Macros

As a means for structuring, it is possible to define macros for interworking diagrams that are used at various places within other interworkings. These macros are used as textual placeholders only and they have no additional semantical meaning. In order to reason about the semantics of interworkings it is therefore necessary that all macros are expanded. Nothing can be said about the communication behaviour of an interworking diagram containing unexpanded macros.

[1]Department of Mathematics and Computing Science, University of Technology Eindhoven, email:sjouke@win.tue.nl

[2]Department of Mathematics and Computing Science, University of Technology Eindhoven, Part of the work has been carried out to get a masters degree in computing science

[3]Philips Research Laboratories, Eindhoven, email:winter@prl.philips.nl

III.2.2 Semantics

A message interaction between two entities can be split into two different events; the message output and the message input. Interworkings are event oriented and only synchronous communication actions can be modelled. No global time axis is assumed for one interworking. Along each vertical entity-line the time is running from top to bottom, however, no proper time scale is assumed. Events of different entities are ordered only via messages. Within interworkings, no delay between a message output and the corresponding message input is explicitly modelled. Therefore, an interworking imposes only a partial ordering on the events being contained. In figure **??**, the messages `u` and `v` are not related and no ordering exists between them. This also holds for the messages `y` and `z`.

A collection of interworkings describes the behaviour of a system on a high level of abstraction. Each interworking describes a common sequence of communication actions performed by the contained entities. Note that the interworking does not specify communication actions with entities that are not contained. The behaviour of a single entity is thus only partially described by the events being contained in the interworking. The actual behaviour will be an interleaving with communication events concerning other entities. Between the output of `v` and the input of `y`, entity `A` in figure **??** for example can have interactions with an entity `E` which is not in the diagram. However, it can not have a communication action with either one of the entities `B`, `C`, or `D` between the output of `v` and the input of `y`. This high level of abstraction results from the fact that communication actions are considered to be synchronous.

III.3 Interworking sequencing

An interworking diagram can be constructed from atomic communication actions and applications of the interworking sequencing operator (denoted by $\circ_{iw}$). The interworking sequencing, or simply the sequencing of two interworking diagrams is the vertical concatenation of the two diagrams (See figure **??**).

In the above case where the interworking diagrams have all entities in common, the sequencing corresponds to a real sequentialisation in time which resembles the sequential composition or product of *ACP*. The operands need not necessarily have all entities in common. In the next example (Figure **??**) the two interworkings only have entity `B` in common.

When interworkings contain disjoint entities, the behaviour described by means of their sequencing will in most cases not be equal to simple sequential composition of the behaviours. Events that do not involve the same entity or are not related via messages will be unordered in the sequential composition. An example of the sequencing of two interworkings that contain unrelated messages is given in figure **??**. Note that the right hand side describes a single interworking in which the vertical position of the arrows does not indicate any ordering.

III.4 Interworking merge

The interworking merge, or simply called the merge (denoted by $\|_{iw}$), of two interworkings is their interleaved composition with the restriction that the interworkings are forced to synchronise on a set of communication actions. This set consists of the communication actions concerning every pair of entities which the interworkings have in common.

The merge synchronises on communication actions that the two interworking diagrams have in common. This means that in those cases where the interworking diagrams describe disjoint entities, the merge is equal to real parallel composition. An example of the merge of two interworking diagrams in which disjoint entities are described is given in figure ??. Note also that the merge of these two interworking diagrams is equal to the sequencing of these diagrams (See figure ??).

An interworking does not specify communication actions with entities that are not contained. The behaviour of a single entity is only partially described by the events contained in an interworking. The merge of two interworkings may return an interworking in which the sequence of events for a single entity is the result of interleaving the sequences of events in the original interworkings. As an example, figure ?? shows the merge of two interworking diagrams that have only one entity (`B`) in common.

For the interworking resulting from the merge in figure ??, nothing can be said about the order of the events for entity `B`. The input of `x` can equally well occur before or after the output of `y`. However, choices or decision points can not be expressed within the interworking diagrams themselves. But, because an interworking containing choices is semantically equivalent to a choice between various interworking diagrams, the result of the above merge can be described as the choice between two interworking diagrams, denoted by the + operator. Note that this + operator is not contained in the syntax for interworkings.

III.4.1 Consistency

When two interworkings are merged, the interworkings have to synchronise on all communication actions between entities which they have in common. In case this synchronisation succeeds, we will call the two interworkings *merge-consistent*. However, if two interworkings do have multiple entities in common but do not synchronise on all the communication actions between these entities then the interworkings are not merge-consistent and the merge of the interworkings will contain so called deadlock or inaction. The two interworking diagrams in figure ?? for example are not merge-consistent.

The textual and graphical syntax for interworking diagrams does not contain a construct to represent deadlock. In figure ??, the deadlock is depicted by means of a double horizontal line. It should be noted that deadlock is not related to any contained entity in particular. Deadlock is a property of the interworking as a whole.

III.5 Interworkings

If we call all objects that can be expressed in terms of the concrete syntax `IW` interworking diagrams, then interworkings are defined as all objects that can be constructed from these interworking diagrams and application of the sequencing and merge operator. With this definition, interworkings are closed under application of the sequencing and the merge operator, and repeated application of the operators is well defined in the underlying theory.

Since an interworking containing choices is semantically equivalent with a choice between interworking diagrams, it is always possible to represent an interworking in terms of the syntax `IW` as a collection of interworking diagrams. Note that interworking diagrams form a subset of interworkings. Both the interworking sequencing and the interworking merge apply to interworking diagrams as well as to interworkings in general. Hereafter, we will discuss some interesting properties that hold for interworkings. For a formal proof of these properties in terms of *ACP* we refer to chapter V.

III.5.1 Properties

Proposition 1 *Merging is commutative; if x and y are two interworkings then the merge of x and y is equal to the merge of y and x;*

$$x \parallel_{iw} y = y \parallel_{iw} x$$

Interworking sequencing in general is not commutative. Only in case two interworkings have no entities in common then the sequencing of these interworkings is commutative. Note that the entities of an interworking are defined as the entities that are involved in the communication actions contained in that interworking.

Proposition 2 *If interworking x and interworking y have no entities in common, then the sequencing of x and y is equal to the sequencing of y and x;*

$$x \circ_{iw} y = y \circ_{iw} x$$

An example of this commutativity can be seen in figure **??**. Note that in this case that two interworkings have no entities in common the sequencing of the interworkings is equal to the merge of the interworkings: $(E(x) \cap E(y) = \emptyset \Rightarrow x \circ_{iw} y = x \parallel_{iw} y)$. A special case of the above proposition occurs when x is equal to deadlock; $x = \delta$. Because deadlock is not related to any entity it holds that $\delta \circ_{iw} y = y \circ_{iw} \delta$ (i.e. deadlock propagates until the "end" of the interworking).

Two other important properties of interworkings and the sequencing and merge operators concern the associativity.

Proposition 3 *Sequencing is associative; For interworkings x, y, and z the sequencing of x and y sequenced with z is equal to the x sequenced with the sequencing of y and z;*

$$(x \circ_{iw} y) \circ_{iw} z = x \circ_{iw} (y \circ_{iw} z)$$

The associativity of the merge operator depends on the operands. Only if the interworkings are mutually merge-consistent it holds that the merge is associative.

Proposition 4 *Let x, y and z be interworkings, let x be of the form x', $x' \circ_{iw} \delta$ or δ, y be of the form y', $y' \circ_{iw} \delta$ or δ and z be of the form z', $z' \circ_{iw} \delta$ or δ, such that x', y' and z' are pairwise consistent interworkings, then the merge of x and y merged with z is equal to x merged with the merge of y and z;*

$$(x \parallel_{iw} y) \parallel_{iw} z = x \parallel_{iw} (y \parallel_{iw} z)$$

The fact whether the merge of a collection of interworkings contains deadlock or not does not depend on the order of application of the merge operator. As a result of the above properties we can conclude that the final result of merging a collection of interworkings is not influenced by the order in which the merge operator is applied to the various elements of the collection. This means that we may generalise the definition of merge-consistency to collections of interworkings.

Syntax and semantics of synchronous interworkings

IV

The syntax of T and denotational semantics of IW

S. Mauw[1] M. van Wijk[2] T. Winter[3]

IV.1 Introduction

In this document we present the syntax of T, a new language to describe interworkings, and the denotational semantics of IW. Those two subjects are presented in one document because they are strongly related. The semantics of IW will be given in terms of T. The language T is created in parallel with the theoretical part [10]. T maps directly onto the theory described in that part.

IV.2 The language T

IV.2.1 Another language to describe interworkings

IW is a language which is used for describing interworking diagrams. All aspects of interworking diagrams can be expressed within IW and IW expressions can be interpreted as interworking diagrams. In order to make tools for interworking processing we studied the syntax of IW and the tools requirements. In doing so so we encountered a number of reasons to create another language to describe interworkings.

One of the first things we encountered was that in order to describe the result of the merge of two interworkings a choice operator was needed as shown in [8]. It is possible to enhance IW with this operator but this should not be done because IW is an intensively used language. There are that much systems described in IW that changes are not accepted by users. Also the users are not interested in interworkings containing that kind of choices. To be able to describe the merge of two interworkings a language with a choice operator is needed.

The language IW has elements which can be replaced by syntactically different but semantically identical elements. These elements are the compound statements timeout-action and send-action and the (x)macros. They are useful during the creation and description of interworking sets as is done at PKI but they impose problems on the application of the interworking operators on IW. Macros and compound statements are not defined in BPA_{iw}.

[1]Department of Mathematics and Computing Science, University of Technology Eindhoven, email:sjouke@win.tue.nl

[2]Department of Mathematics and Computing Science, University of Technology Eindhoven, Part of the work has been carried out to get a masters degree in computing science

[3]Philips Research Laboratories, Eindhoven, email:winter@prl.philips.nl

Compound statements as used in IW have to be split before merging. They can be replaced by other statements as shown in the following example.

```
p TIMEOUT timer ACTION action
```

the statement above means the same as

```
p TIMEOUT timer
p ACTION action
```

Macros should be expanded before interworkings containing macros are merged. Another problem regarding macros is that they do not have to be defined. The formal semantics of an undefined macro is not determined although the meaning of such a macro can be clear to a user of an interworking containing that macro. Tools used to process interworkings can hardly recognize and interpret such macros in the way users can do.

The parameters of message identifiers are a way of enlarging the meaning of messages to the user. The meaning of a message and parameters combination is just that of one identifier. A message name parameters combination in IW will be translated into one message identifier of T. This is expressed in the denotational semantics of IW.

It is possible to create, use other or change interworking describing languages, which have semantics definable in T. It is then possible to use the existing syntax, tools and semantics of T, perhaps with some changes and enhancements, to process and analyse such languages. Also by using compilers to and from T it is possible to translate those languages to each other. The use of an extra language makes the tools syntax independent.

IW is not able to express all needed expressions over interworkings. IW also contains semantically not relevant and difficult to merge items. In order to describe all possible interactions of interworkings in a clear way the language T and its theoretical foundations were created.

IV.2.2 Design of T

A design goal for T was to create a language that was as close to the theoretical foundations in chapter [10] as possible. We choose the elements of the language T to be expressions over the basic process algebra operators $+$ and $\cdot$, and the two interworking operators, sequencing and merge.

Macros need to be unfolded before interworkings can be merged. Since interworkings are finite processes, recursive definitions as by using macros are not needed to describe interworkings. A language to describe the semantics and combination of interworkings in IW should not use macros. The language T should also not have compound statements as is explained in the previous section.

The interworking sequencing and merge are elements of the language T. So any process that can be expressed using these operators can be expressed in T. In particular the merge of inconsistent interworkings which contains deadlock. Therefore deadlock needs to be an element of T. The availability of deadlock allows the study of failure behaviour of merged, inconsistent interworkings.

T should consist of the following items: the operators of BPA, $+$ and $\cdot$, the interworking operators, $\circ_{iw}$ and $\|_{iw}$, the atom δ and the other atoms for BPA_{iw} and a way of linking interworking names and the semantics of their expression lists. All these items are represented in the language T as given in the next section.

This grammar is chosen to have such a form that the shape of the derivation tree of expressions in T reflects which operators operate on which operands. The grammar reflects also the priorities of the operators which is $+ < \circ_{iw} < \|_{iw} < \cdot$.

IV.2.3 BNF

In this section, the concrete syntax of T is given by means of the following BNF. All undefined non terminals, which are Eid, Tid, Did, Aid, Mid, are identifiers as defined in [7]. Mid's are a bit different because they may also contain '(', ')' and ',' symbols as used for messages and their parameters in IW. The rules which describe the grammar of T have the same meaning as those which describe the grammar of IW.

```
T                  ::= { process }

process            ::= processname '=' expression

expression         ::= iwmerge { '+' iwmerge }

iwmerge            ::= iwseq { '||' iwseq }

iwseq              ::= seq { 'o' seq }

seq                ::= atom_or_expression { '.' atom_or_expression }

atom_or_expression ::= atom
                     | '(' expression ')'

atom               ::= 'C(' Eid ',' Eid ',' Mid ')'
                     | 'Lost(' Eid ',' Eid ',' Mid ')'
                     | 'Timerset(' Eid ',' Tid ',' Did')'
                     | 'Timeout(' Eid ',' Tid ')'
                     | 'Action(' Eid ',' Aid ')'
                     | 'Delta'
```

IV.2.4 An example in T

Here follows an example of interworkings defined in T. Note that p is not a single interworking but a more complex combination of interworkings containing just one expression.

```
p = ( C(x,y,m) || C(x,z,m) ) o Action(x,b)
q = Timerset(x,timer,5) o Timeout(x,timer) o Action(y,h)
```

Process p is the merge of two communications sequenced with an action, q is the sequencing of a timerset, a timeout and an action.

IV.2.5 The semantics of T

The semantics of T is given by a direct mapping of the expressions of T to an extension of BPA. This extension of BPA is defined in [10]. The mapping is given by a simple replacement. This is the replacement of `||` by $\|_{iw}$ and `o` by $\circ_{iw}$.

We do not look at the semantics of a set of processes, because the semantics of a set of processes is determined by the relations between those processes. These are relations like being a correct refinement of another process or being the sequencing or merge of different processes. These relations are only visible to users of these processes and are not described or visible within the language. So the semantics of T is sufficiently defined by a mapping of the expressions of T towards BPA_{iw}.

IV.3 Abstract syntax of IW

IV.3.1 Introduction

The abstract syntax of IW is derived from the concrete syntax of IW by removing all semantically non-relevant items. Occurring optional actions can be replaced by a normal action expression as shown in the example on page 14. There is no difference in semantics between an expandable macro and an xmacro, both their semantics are the semantics of the corresponding expression list after parameter substitution, so both are named macro.

The ON PROCESSES part is not needed for the definition of the semantics of a macro call nor is the PROCESSES part of an interworking needed for the definition of the semantics of that interworking.

Not defined macros are not allowed, since they have an undefined semantics. The difference between message- and interworking/macro-parameters is made explicit in the abstract syntax because their semantics are different. The semantics of interworking parameters is a list of textual place holders for substitution purposes, so each interworking parameter is a single identifier. The semantics of a message parameter is also an identifier but it is composed of a combination of identifiers.

IV.3.2 Abstract syntax

The abstract syntax of IW is given by the same king of grammar rules as used for the description of the other grammars. An object between <, > denotes the abstract representation of that object. Names of objects (e.g. interworkings, processes, messages) do not have any semantical meaning.

$$\begin{array}{lll}
<\mathit{iwdset}> & ::= & \mathit{empty}() \\
 & \mid & \mathit{nonempty}(<\mathit{iwd}>, <\mathit{iwdset}>) \\
 & & \\
<\mathit{iwd}> & ::= & \mathit{iwd}(<\mathit{iwname}>, <\mathit{iwparameters}>, <\mathit{expressionlist}>) \\
 & & \\
<\mathit{iwparameters}> & ::= & \mathit{empty}() \\
 & \mid & \mathit{nonempty}(<\mathit{iwparameter}>, <\mathit{iwparameters}>) \\
 & & \\
<\mathit{expressionlist}> & ::= & \mathit{empty}() \\
 & \mid & \mathit{nonempty}(<\mathit{expression}>, <\mathit{expressionlist}>) \\
 & & \\
<\mathit{expression}> & ::= & <\mathit{transmission}> \\
 & \mid & <\mathit{losttransmission}> \\
 & \mid & <\mathit{macro}> \\
 & \mid & <\mathit{processaction}> \\
 & \mid & <\mathit{timerset}> \\
 & \mid & <\mathit{timeout}> \\
 & & \\
<\mathit{transmission}> & ::= & \mathit{transmission}(<\mathit{processname}>, <\mathit{message}>, <\mathit{processname}>)
\end{array}$$

$$
\begin{array}{lll}
< \mathit{losttransmission} > & ::= & \mathit{losttransmission}(< \mathit{processname} >, < \mathit{message} >, < \mathit{processname} >) \\
< \mathit{message} > & ::= & \mathit{message}(< \mathit{msgname} >, < \mathit{parameters} >) \\
< \mathit{parameters} > & ::= & \mathit{empty}() \\
 & | & \mathit{nonempty}(< \mathit{parameter} >, < \mathit{parameters} >) \\
< \mathit{parameter} > & ::= & \mathit{parameter}(< \mathit{paramname} >, < \mathit{parameters} >) \\
< \mathit{macro} > & ::= & \mathit{macro}(< \mathit{macroname} >, < \mathit{iwparameters} >) \\
< \mathit{processaction} > & ::= & \mathit{processaction}(< \mathit{processname} >, < \mathit{actionname} >) \\
< \mathit{timerset} > & ::= & \mathit{timerset}(< \mathit{processname} >, < \mathit{timername} >, < \mathit{duration} >) \\
< \mathit{timeout} > & ::= & \mathit{timeout}(< \mathit{processname} >, < \mathit{timername} >)
\end{array}
$$

IV.4 Semantics of IW

IV.4.1 Introduction

The semantics of IW will be given for syntactically and semantically correct interworking sets. We give the semantics of IW in terms of the abstract syntax of IW and the concrete syntax of T. To determine the semantics of a macrocall the semantics of the defining interworking after parameter substitution should be available at the moment the semantics of that call is determined. To accomplish that availability the definition of the semantics is split into two phases. First an environment which allows parameter substitution on expression lists is created. Thereafter that environment is used to determine the final semantics.

IV.4.2 Sets and constants

In this section the sets, constants and basic operators required to define the semantics of IW in terms of T are presented. Emphasized The language of a non-terminal from the grammars of T or IW is denoted by the italicized version of that non-terminal. For example *iwdset* denotes the set of all derivable interworkings, using the grammar of IW, starting at $< \mathit{iwdset} >$ in the abstract grammar of IW, *expressions* denotes the set of all objects derivable from expression in the grammar of T.

$\perp$: denotes anything undefined.
ε : empty string or identifier.
$Ident$: the set of all identifiers.

macros
$macros = \lambda\,\overline{x} \in Ident^* \cdot expressionlist$
This is a set of functions which denote the parameter substitution on an expression list. $\overline{x}$ is a tupel of idents. An example of such a function is:

$$\lambda(x,y,z) \in Ident^* \cdot \begin{array}{l} x \textsf{ SET } y\ (z) \\ x \textsf{ SENDS mesg TO d} \end{array}$$

Its application is as follows:

$$\left(\lambda(x,y,z) \in Ident^* \cdot \begin{array}{l} x \textsf{ SET } y\ (z) \\ x \textsf{ SENDS mesg TO d} \end{array}\right) \textsf{(a,b,c)} = \begin{array}{l} \textsf{a SET b (c)} \\ \textsf{a SENDS mesg TO d} \end{array}$$
.

environment
$Env = Ident \rightarrow macros \cup \{\perp\}$
This environment is used to establish links between interworking names and parametrized expression lists. The bottom element is added to make sure the environment is defined for all identifiers. The parameters used for the macros will be the interworking parameters. An example of such an environment is the environment function r. r is a defined environment function for the following interworking.

```
INTERWORKING iw (a,b,c,d)
PROCESSES a,b
ENDPROCESSES
   a SENDS c TO b
   b ACTION d
ENDINTERWORKING
```

Now r(iw)(p,q,s,t) is $\begin{array}{l} \textsf{p SENDS s TO q} \\ \textsf{q ACTION t} \end{array}$.

The creation of such an environment is defined in section IV.4.4.

environment creation function
$/ : Ident \times macros \rightarrow Env$
$b/a = \lambda y \in Ident \cdot y = b \rightarrow a, \perp$ environment creation function.
This is a function over Ident, with $b/a(b) = a$, and for $c \neq b$, $b/a(c) = \perp$.
This function will be used to establish links between interworking names and parametrized expression lists.

environment composition
$[\,] : Env \times Env \rightarrow Env$
$f[g] = \lambda y \in Ident \cdot g(y) = \perp \rightarrow f(y), g(y)$ environment composition,
$f[g]$ is a function which returns the application of f if the application of g on the actual Ident yields $\perp$.
This function will be used to combine environment functions defined over interworkings to environment functions defined over interworking sets.

identifier concatenation

$+\!\!+ : Ident \times Ident \to Ident$ identifier concatenation,
"abc" $+\!\!+$ "xyz"= "abcxyz"

expression sequencing

$+\!\!+ : expression \times expression \to expression$ expression sequencing,
$\circ$ is the interworking sequence operator in T. The function $+\!\!+$ is needed because the interworking sequencing is not defined for empty processes. Interworkings can have an empty expression list, so the semantics of a macro call can be ε, which can only be sequenced by using the forementioned auxiliary operator.

$$a +\!\!+ b = \begin{cases} a \neq \varepsilon \wedge b \neq \varepsilon & \to a \circ b \\ a \neq \varepsilon \wedge b = \varepsilon & \to a \\ a = \varepsilon \wedge b \neq \varepsilon & \to b \\ a = \varepsilon \wedge b = \varepsilon & \to \varepsilon \end{cases}$$

IV.4.3 Semantics

The semantics function has an interworking set as input and returns its semantics in terms of T. The semantics of IW is given by the function

$$[\![\ ]\!] : iwdset \to T.$$

This function is defined in two stages. First an environment is defined which allows parameter substitution on named expression lists. Then that environment is used to determine the semantics of the interworking set in terms of T.

$[\![< iwdset >]\!]$ is defined as $[\![< iwdset >]\!]_r$, where $r = env(< iwdset >)$. The function *env* is defined in section IV.4.4. $[\![< iwdset >]\!]_r$ is the application of the function $[\![\ ]\!] : iwdset \times Env \to T$ which is defined in section IV.4.5. It is possible to write this function down in one term as $[\![< iwdset >]\!]_{env(< iwdset >)}$. In this case it should be guaranteed that each parametrized expression list of $env(< iwdset >)$ is evaluated before it is needed in the determination of the semantics of a macro call.

IV.4.4 Environment creation

The environment created by the function *env* denotes for each interworking of the interworking set the parameter substitution on its expression list. The application of this function on an interworking name gives a substitution function on that interworkings expression list.

iwdset

The environment of an interworking set is the composition of the environments of its interworkings.

$env : iwdset \to Env$

$env(empty()) = \lambda y \in Ident \cdot \perp$

$env(nonempty(< iwd >, < iwdset >)) =$

$env(< iwd >)\ [\,env(< iwdset >)\,]$

iwd

By taking the iwparameters as the parameters for the macro, the substitution function of $< iwname >$ equals to the macrocall MACRO iwname iwparameters

$env : iwd \to Env$

$env(iwd(< iwname >, < iwparameters >, < expressionlist >)) =$

$< iwname > / \lambda(p_1, p_2, \cdots, p_n) \cdot < expressionlist >$

where $(p_1, p_2, \cdots, p_n) = [\![< iwparameters >]\!]$

iwparameters

$[\![\ ]\!] : iwparameters \to Ident^*$

$[\![empty()]\!] = ()$

$[\![nonempty(< iwparameter >, < iwparameters >)]\!] =$

$(< iwparameter >, p_1, p_2, \cdots, p_n)$ where $(p_1, p_2, \cdots, p_n) = [\![< iwparameters >]\!]$

IV.4.5 Translation and macro expansion

In this section the semantics of an interworking set in terms of T is given. The application of the environment function *env* on the current interworking set is needed to determine the semantics of macro calls, we call that application r, $r = env(< iwdset >)$.

iwdset

The concatenation in the last line of this paragraph denotes some placing together of processes within a set of processes described in T.

$[\![\ ]\!] : iwdset \times Env \to T$

$[\![empty()]\!]_r = \varepsilon$

$[\![nonempty(< iwd >, < iwdset >)]\!]_r =$

$[\![< iwd >]\!]_r [\![< iwdset >]\!]_r$

iwd

All interworkings, also macros, are translated. The parameters of interworkings are not needed to determine the semantics of an interworking.

$[\![\ ]\!] : iwd \times Env \to process$

$[\![iwd(< iwname >, < iwparameters >, < expressionlist >)]\!]_r =$

$< iwname > = [\![< expressionlist >]\!]_r$

expression list

The semantics of an expression list is the interworking sequencing of the semantics of all expressions in the expression list. The interworking sequencing is not defined for empty expressions. Since interworkings and by that macros can be empty the auxiliary operator $+\!\!+$ on expressions is used to remove empty expressions before using the interworking sequencing operator. This operator also avoids the problem of checking whether the remaining expression list is empty. The semantics of an expression is a direct translation to T except for macros, which have to be expanded first.

$[\![\ ]\!] : expressionlist \times Env \to expression$

$[\![empty()]\!]_r = \varepsilon$

$[\![nonempty(< expression >, < expressionlist >)]\!]_r =$

$[\![< expression >]\!]_r ++ [\![< expressionlist >]\!]_r$

transmission

$[\![\ ]\!] : transmission \times Env \to expression$

$[\![transmission(< processname1 >, < message >, < processname2 >)]\!]_r =$

$\mathsf{C}(< processname1 >, < processname2 >, [\![< message >]\!])$

lost transmission

$[\![\ ]\!] : losttransmission \times Env \to expression$

$[\![losttransmission(< processname1 >, < message >, < processname2 >)]\!]_r =$

$\mathsf{Lost}(< processname1 >, < processname2 >, [\![< message >]\!])$

message

$[\![\ ]\!] : message \to Ident$

$[\![message(< msgname >, < parameters >)]\!] =$

$$< msgname > ++ \begin{cases} < parameters > \neq empty() \to \text{"("} ++ [\![< parameters >]\!] ++ \text{")"} \\ < parameters > = empty() \to \varepsilon \end{cases}$$

parameter

$[\![\ ]\!] : parameter \to Ident$

$[\![parameter(< paramname >, < parameters >)]\!] =$

$$< paramname > ++ \begin{cases} < parameters > \neq empty() \to \text{"("} ++ [\![< parameters >]\!] ++ \text{")"} \\ < parameters > = empty() \to \varepsilon \end{cases}$$

parameters

$[\![\ ]\!] : parameters \to Ident$

$[\![empty()]\!] = \varepsilon$

$[\![nonempty(< parameter >, < parameters >)]\!] =$

$$[\![< parameter >]\!] ++ \begin{cases} < parameters > \neq empty() \to \text{","} ++ [\![< parameters >]\!] \\ < parameters > = empty() \to \varepsilon \end{cases}$$

macro

The semantics of a macro is the semantics of its defining interworkings expression list after parameter substitution. The substitution function on the mentioned expression list is $r(< macroname >)$, this function will be applied on the actual macro parameters.

$[\![\ ]\!] : macro \times Env \to expression$

$[\![macro(< macroname >, < iwparameters >)]\!]_r =$

$[\![r(< macroname >)[\![< iwparameters >]\!]]\!]_r$

processaction

$[\![\ ]\!] : processaction \times Env \to expression$

$[\![processaction(< processname >, < actionname >)]\!]_r =$

$\mathsf{Action}(< processname >, < actionname >)$

timerset

$[\![\ ]\!] : \mathit{timerset} \times \mathit{Env} \to \mathit{expression}$

$[\![\mathit{timerset}(<\mathit{processname}>,<\mathit{timername}>,<\mathit{duration}>)]\!]_r =$

$\mathsf{Timerset}(<\mathit{processname}>,<\mathit{timername}>,<\mathit{duration}>)$

timeout

$[\![\ ]\!] : \mathit{timeout} \times \mathit{Env} \to \mathit{expression}$

$[\![\mathit{timeout}(<\mathit{processname}>,<\mathit{timername}>)]\!]_r =$

$\mathsf{Timeout}(<\mathit{processname}>,<\mathit{timername}>)$

IV.4.6 An example on application of the semantics function

The interworkings in this example are given in the concrete syntax of IW. They are syntactically and semantically correct so it is possible to determine the semantics of this interworking set.

Take the following interworking set, in IW lay out.

```
INTERWORKING iw1
PROCESSES p,q,r
ENDPROCESSES
   MACRO macro(p,a,123)
   p ACTION x
   p SENDS m TO r
ENDINTERWORKING

INTERWORKING macro(x,y,z)
PROCESSES x,q ENDPROCESSES
   q SENDS m TO x
   x SET y (z)
   x TIMEOUT y
ENDINTERWORKING
```

To determine the semantics of this interworking set we first determine the environment function r, where after we apply the translation part using this r.

$r =$

$$\mathit{env}\left(\begin{array}{l}
\mathsf{INTERWORKING\ iw1}\\
\mathsf{PROCESSES\ p,q,r}\\
\mathsf{ENDPROCESSES}\\
\quad\mathsf{MACRO\ macro(p,a,123)}\\
\quad\mathsf{p\ ACTION\ x}\\
\quad\mathsf{p\ SENDS\ m\ TO\ r}\\
\mathsf{ENDINTERWORKING}\\
\\
\mathsf{INTERWORKING\ macro(x,y,z)}\\
\mathsf{PROCESSES\ x,q}\\
\mathsf{ENDPROCESSES}\\
\quad\mathsf{q\ SENDS\ m\ TO\ x}\\
\quad\mathsf{x\ SET\ y\ (z)}\\
\quad\mathsf{x\ TIMEOUT\ y}\\
\mathsf{ENDINTERWORKING}
\end{array}\right)$$

$$=$$

$$env\left(\begin{array}{l}\textsf{INTERWORKING iw1}\\ \textsf{PROCESSES p,q,r}\\ \textsf{ENDPROCESSES}\\ \quad\textsf{MACRO macro(p,a,123)}\\ \quad\textsf{p ACTION x}\\ \quad\textsf{p SENDS m TO r}\\ \textsf{ENDINTERWORKING}\end{array}\right)\left[env\left(\begin{array}{l}\textsf{INTERWORKING macro(x,y,z)}\\ \textsf{PROCESSES x,q}\\ \textsf{ENDPROCESSES}\\ \quad\textsf{q SENDS m TO x}\\ \quad\textsf{x SET y (z)}\\ \quad\textsf{x TIMEOUT y}\\ \textsf{ENDINTERWORKING}\end{array}\right)\right]$$

$$=$$

$$\textsf{iw1}/\lambda()\cdot\ \begin{array}{l}\textsf{MACRO macro(p,a,123)}\\ \textsf{p ACTION x}\\ \textsf{p SENDS m TO r}\end{array}\quad\left[\textsf{macro}/\lambda(x,y,z)\cdot\ \begin{array}{l}\textsf{q SENDS m TO } x\\ x\ \textsf{SET}\ y\ (z)\\ x\ \textsf{TIMEOUT}\ y\end{array}\right]$$

Now we can determine the semantics of that interworking set using the function r.

$$\left[\!\!\left[\begin{array}{l}\textsf{INTERWORKING iw1}\\ \textsf{PROCESSES p,q,r}\\ \textsf{ENDPROCESSES}\\ \quad\textsf{MACRO macro(p,a,123)}\\ \quad\textsf{p ACTION x}\\ \quad\textsf{p SENDS m TO r}\\ \textsf{ENDINTERWORKING}\\ \\ \textsf{INTERWORKING macro(x,y,z)}\\ \textsf{PROCESSES x,q}\\ \textsf{ENDPROCESSES}\\ \quad\textsf{q SENDS m TO x}\\ \quad\textsf{x SET y (z)}\\ \quad\textsf{x TIMEOUT y}\\ \textsf{ENDINTERWORKING}\end{array}\right]\!\!\right]_r$$

$$=$$

$$\left[\!\!\left[\begin{array}{l}\textsf{INTERWORKING iw1}\\ \textsf{PROCESSES p,q,r}\\ \textsf{ENDPROCESSES}\\ \quad\textsf{MACRO macro(p,a,123)}\\ \quad\textsf{p ACTION x}\\ \quad\textsf{p SENDS m TO r}\\ \textsf{ENDINTERWORKING}\end{array}\right]\!\!\right]_r\quad\left[\!\!\left[\begin{array}{l}\textsf{INTERWORKING macro(x,y,z)}\\ \textsf{PROCESSES x,q}\\ \textsf{ENDPROCESSES}\\ \quad\textsf{q SENDS m TO x}\\ \quad\textsf{x SET y (z)}\\ \quad\textsf{x TIMEOUT y}\\ \textsf{ENDINTERWORKING}\end{array}\right]\!\!\right]_r$$

$$=$$

$$\textsf{iw1} = \left[\!\!\left[\begin{array}{l}\textsf{MACRO macro(p,a,123)}\\ \textsf{p ACTION x}\\ \textsf{p SENDS m TO r}\end{array}\right]\!\!\right]_r\quad\textsf{macro} = \left[\!\!\left[\begin{array}{l}\textsf{q SENDS m TO x}\\ \textsf{x SET y (z)}\\ \textsf{x TIMEOUT y}\end{array}\right]\!\!\right]_r$$

$$=$$

$$\textsf{iw1} = \begin{array}{ll} & [\![\textsf{MACRO macro(p,a,123)}]\!]_r\\ +\!\!+ & [\![\textsf{p ACTION x}]\!]_r\\ +\!\!+ & [\![\textsf{p SENDS m TO r}]\!]_r\end{array}\quad\textsf{macro} = \begin{array}{ll} & [\![\textsf{q SENDS m TO x}]\!]_r\\ +\!\!+ & [\![\textsf{x SET y (z)}]\!]_r\\ +\!\!+ & [\![\textsf{x TIMEOUT y}]\!]_r\end{array}$$

$$=$$

$$\mathsf{iw1} = \begin{array}{cl} & [\![\mathsf{r(macro)(p,a,123)}\ ]\!]_r \\ +\!\!+ & \mathsf{Action(p,x)} \\ \mathsf{o} & \mathsf{C(p,r,m)} \end{array} \qquad \mathsf{macro} = \begin{array}{cl} & \mathsf{C(q,x,m)} \\ \mathsf{o} & \mathsf{Timerset(x,y,z)} \\ \mathsf{o} & \mathsf{Timeout(x,y)} \end{array}$$

$=$

$$\mathsf{iw1} = \begin{array}{cl} & \left[\!\!\left[\begin{array}{l} \mathsf{q\ SENDS\ m\ TO\ p} \\ \mathsf{p\ SET\ a\ (123)} \\ \mathsf{p\ TIMEOUT\ a} \end{array} \right]\!\!\right]_r \\ +\!\!+ & \mathsf{Action(p,x)} \\ \mathsf{o} & \mathsf{C(p,r,m)} \end{array} \qquad \mathsf{macro} = \begin{array}{cl} & \mathsf{C(q,x,m)} \\ \mathsf{o} & \mathsf{Timerset(x,y,z)} \\ \mathsf{o} & \mathsf{Timeout(x,y)} \end{array}$$

$=$

$$\mathsf{iw1} = \begin{array}{cl} & \mathsf{C(q,p,m)} \\ \mathsf{o} & \mathsf{Timerset(p,a,123)} \\ \mathsf{o} & \mathsf{Timeout(p,a)} \\ \mathsf{o} & \mathsf{Action(p,x)} \\ \mathsf{o} & \mathsf{C(p,r,m)} \end{array} \qquad \mathsf{macro} = \begin{array}{cl} & \mathsf{C(q,x,m)} \\ \mathsf{o} & \mathsf{Timerset(x,y,z)} \\ \mathsf{o} & \mathsf{Timeout(x,y)} \end{array}$$

So the semantics of this interworking set is as follows:

iw1 = C(q,p,m) o Timerset(p,a,123) o Timeout(p,a) o Action(p,x) o C(p,r,m)
macro = C(q,x,m) o Timerset(x,y,z) o Timeout(x,y)

Syntax and semantics of synchronous interworkings

V

A formal semantics fo interworkings

S. Mauw[1] M. van Wijk[2] T. Winter[3]

V.1 Introduction

In [7] we defined the specification language IW, which can be used to describe synchronous interworking diagrams. The semantics of this language is given by a translation to the language T (see [9]). The expressions of this intermediate language directly map onto the process theory BPA_{iw}, which will be introduced in this paper. BPA_{iw} is an extension of BPA, which stands for *Basic Process Algebra* [1]. In this paper we will first give an introduction to BPA, then we define the two interworking operators and the class of interworkings. Finally we give some useful properties of interworkings. The main result for the interworking toolset is that every interworking can be represented by a collection of locally deterministic interworkings.

V.2 Basic Process Algebra

BPA (Basic Process Algebra) is an algebraic theory for the description of process behavior. The theory BPA has been extended with several operators in order to express special features more easily. One of the extensions is the theory ACP (Algebra of Communicating Processes) [1, 2], which has facilities for parallelism. ACP can be compared to CCS [11] and CSP [5].

V.2.1 Actions and Processes

We consider two basic notions: *atomic actions* and *processes*. An atomic action is an indivisible unit of behavior, such as the insertion of a coin in a coffee dispenser or the communication of some piece of information between two agents. A process is the description of the (possible) behavior of a system. Atomic actions will be denoted by $a, b, \ldots$ and processes by $x, y, \ldots$.

Every atomic action will be considered as a process. Processes can be constructed from simpler processes with the use of two operators. The *sequential composition* of processes x and y (notation $x \cdot y$ or xy for short) is the process

[1]Department of Mathematics and Computing Science, University of Technology Eindhoven, email:sjouke@win.tue.nl

[2]Department of Mathematics and Computing Science, University of Technology Eindhoven, Part of the work has been carried out to get a masters degree in computing science

[3]Philips Research Laboratories, Eindhoven, email:winter@prl.philips.nl

that first executes x, and upon completion of x starts y. The *alternative composition* of x and y (notation $x + y$) is the process that either executes x, or executes y (but not both). We will not specify how this choice is made.

These operators are defined by the equations from table V.1.

x + y = y + x
(x + y) + z = x + (y + z)
x + x = x
(x + y)z = xz + yz
(xy)z = x(yz)

Table V.1: Basic Process Algebra

The equations state that the alternative composition is commutative, associative and idempotent, and that the sequential composition is associative only. Note that the $\cdot$ only distributes over a $+$-operator at the left-hand side.

V.2.2 Deadlock

In order to describe processes which may not terminate successfully, we extend BPA with features for unsuccessful termination. We use the term *deadlock* for unsuccessful termination. The special atomic action δ denotes a deadlocked process. Furthermore we introduce the encapsulation function, which renames atomic actions from a given set into δ. It is denoted by ∂_H, where H is a set of atomic actions. The equations in table V.2 define deadlock and encapsulation. The first equation says that no deadlock will ever occur as long as there is an alternative that can proceed. The second equation states that after a deadlock has occurred, no other actions can possibly follow. The predicate *isdelta* determines whether a process equals δ, and the predicate *df* determines whether a process is deadlock-free or not.

The equations from tables V.1 and V.2 define the theory $BPA_{\delta,\partial_H,df}$. We consider the term model as the semantics for the collection of process expressions.

V.3 Basics

The collection of atomic actions can be considered as a parameter of BPA. In the setting of interworkings, we will only consider communication actions of a fixed format and some primitive actions, so we specialize the set of atomic actions.

Let EID and MID be finite sets, containing entity identifiers and message identifiers. Define the following collection of communication actions.

$$A_c = \{c(p, q, m) | p, q \in EID, m \in MID\}$$

Action $c(p, q, m)$ means that entity p sends message m to entity q. Furthermore we define a set of primitive actions. let AN, TID and DID be finite sets of primitive action names, timer identifiers and duration identifiers, respectively. Then we define the set of primitive actions as follows.

$x + \delta = x$	
$\delta x = \delta$	
$\partial_H(a) = a$	if $a \notin H$
$\partial_H(a) = \delta$	if $a \in H$
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	
$\partial_H(xy) = \partial_H(x) \cdot \partial_H(y)$	
$\neg isdelta(a)$	if $a \not\equiv \delta$
$isdelta(\delta)$	
$isdelta(ax) = isdelta(a)$	
$isdelta(x + y) = isdelta(x) \wedge isdelta(y)$	
$\neg df(\delta)$	
$df(a)$	if $a \not\equiv \delta$
$df(ax) = df(x)$	if $a \not\equiv \delta$
$df(x + y) = df(x) \wedge df(y)$	if $\neg isdelta(x)$ and $\neg isdelta(y)$

Table V.2: Deadlock

$$\begin{aligned} A_p = \quad & \{act(p, an) | p \in EID, an \in AN\} \cup \\ & \{lost(p, q, m) | p, q \in EID, m \in MID\} \cup \\ & \{timerset(p, t, d) | p \in EID, t \in TID, d \in DID\} \cup \\ & \{timeout(p, t) | p \in EID, t \in TID\} \end{aligned}$$

The primitive action $act(p, an)$ is interpreted as the fact that entity p executes the non-communication action an. The action $lost(p, q, m)$ means that a communication from entity p to q has failed. If entity p sets timer t with duration d it is denoted by $timerset(p, t, d)$ And action $timeout(p, t)$ means that timer t of entity p signals a timeout.

If we extend this collection of atomic actions with δ we get A_δ.

$$A = A_c \cup A_p$$
$$A_\delta = A \cup \{\delta\}$$

Unless stated differently, all variables $a, b, \ldots$ range over A_δ.

Furthermore, we need an auxiliary function E, which determines the entities involved in an action. This function is defined in table V.3. Note that there is only one entity involved in a failed communication.

V.4 Interworking Operations

Although all process expressions define some behavior, we will consider only certain processes as interworkings. For the construction of interworkings we define two operators, which are defined on all processes: the *interworking sequencing* and the *interworking merge*.

$E(\delta) = \emptyset$	
$E(c(p,q,m)) = \{p,q\}$	
$E(act(p,an)) = \{p\}$	
$E(lost(p,q,m)) = \{p\}$	
$E(timerset(p,t,d)) = \{p\}$	
$E(timeout(p,t)) = \{p\}$	
$E(ax) = E(a) \cup E(x)$	if $a \not\equiv \delta$
$E(x+y) = E(x) \cup E(y)$	

Table V.3: The Entity function

V.4.1 Interworking Sequencing

The interworking sequencing of two processes (notation $\circ_{iw}$) is their interleaved composition with the restriction that actions involving common entities are performed first by the left-hand process and then by the right-hand process. First we define the set of actions generated by the entities of a given process.

$$\alpha_E(x) = \{a \in A | E(a) \cap E(x) \neq \emptyset\}$$

The definition of the interworking sequencing resembles the definition of the communication free merge from ACP. We use the auxiliary operators left sequencing ($\mathbf{L}\!\circ_{iw}$) and right sequencing ($\mathbf{R}\!\circ_{iw}$) which have the following intuitive meaning. The left sequencing of two processes means that the left operand is forced to do the first step. The right sequencing of two processes means that the right operand has to do the first step, but it may only execute this step if it is not blocked by some action from the left operand which involves the same entity. The encapsulation operator (∂_H) is used for blocking unwanted actions.

$x \circ_{iw} y = x \,\mathbf{L}\!\circ_{iw}\, y \;+\; x \,\mathbf{R}\!\circ_{iw}\, y$
$a \,\mathbf{L}\!\circ_{iw}\, x = ax$
$ax \,\mathbf{L}\!\circ_{iw}\, y = a(x \circ_{iw} y)$
$(x+y) \,\mathbf{L}\!\circ_{iw}\, z = x \,\mathbf{L}\!\circ_{iw}\, z \;+\; y \,\mathbf{L}\!\circ_{iw}\, z$
$x \,\mathbf{R}\!\circ_{iw}\, a = \partial_{\alpha_{E(x)}}(a) \cdot x$
$x \,\mathbf{R}\!\circ_{iw}\, ay = \partial_{\alpha_{E(x)}}(a) \cdot (x \circ_{iw} y)$
$x \,\mathbf{R}\!\circ_{iw}\, (y+z) = x \,\mathbf{R}\!\circ_{iw}\, y \;+\; x \,\mathbf{R}\!\circ_{iw}\, z$

Table V.4: Interworking Sequencing

Example 1 *Assuming that p, q, r, s and t are all different entities, we have the following derivation.*

$$\begin{aligned}
&c(p,q,m) \circ_{iw} (c(p,r,n) \circ_{iw} c(s,t,l)) = \\
&c(p,q,m) \circ_{iw} (c(p,r,n) \cdot c(s,t,l) + c(s,t,l) \cdot c(p,r,n)) = \\
&c(p,q,m) \cdot (c(p,r,n) \cdot c(s,t,l) + c(s,t,l) \cdot c(p,r,n)) \;+ \\
&c(s,t,l) \cdot c(p,q,m) \cdot c(p,r,n)
\end{aligned}$$

V.4.2 Interworking Merge

The interworking merge of two processes (notation $\|_{iw}$) is their interleaved composition, except that the processes are forced to synchronize on a set of atomic actions. This set consists of the actions concerning every pair of entities which the processes have in common. First we define the S-interworking merge (notation $\|_{iw}^{S}$) for a subset S of A. As in the definition of the parallel composition in ACP we need two auxiliary operators: the left interworking merge ($\llfloor_{iw}^{S}$) and the synchronization interworking merge ($|_{iw}^{S}$). The left interworking merge forces that the first step is taken from the left argument, provided that this action is not an element of the set S. The synchronization interworking merge can only execute an action if both operands can perform this same action and this action is an element of the set S. The function γ_S determines whether two actions have to synchronize.

$$
\boxed{\begin{array}{l}
x \|_{iw}^{S} y = x \llfloor_{iw}^{S} y + y \llfloor_{iw}^{S} x + x |_{iw}^{S} y \\
a \llfloor_{iw}^{S} x = \partial_S(ax) \\
ax \llfloor_{iw}^{S} y = \partial_S(a) \cdot (x \|_{iw}^{S} y) \\
(x+y) \llfloor_{iw}^{S} z = x \llfloor_{iw}^{S} z + y \llfloor_{iw}^{S} z \\
a |_{iw}^{S} b = \gamma_S(a,b) \\
ax |_{iw}^{S} b = \gamma_S(a,b) \cdot \partial_S(x) \\
a |_{iw}^{S} bx = \gamma_S(a,b) \cdot \partial_S(x) \\
ax |_{iw}^{S} by = \gamma_S(a,b) \cdot (x \|_{iw}^{S} y) \\
(x+y) |_{iw}^{S} z = x |_{iw}^{S} z + y |_{iw}^{S} z \\
x |_{iw}^{S} (y+z) = x |_{iw}^{S} y + x |_{iw}^{S} z \\
\gamma_S(a,b) = \begin{cases} a & \text{if } a = b \wedge a \in S \\ \delta & \text{otherwise} \end{cases}
\end{array}}
$$

Table V.5: Interworking Merge

Define the function α_{CE}, which determines the actions performed by common entities as follows.

$$\alpha_{CE}(x,y) = \{c(p,q,m) \in A | p,q \in E(x) \cap E(y)\}$$

Then we define the interworking merge of two processes as in the second part of table V.5.

$$\boxed{x \|_{iw} y = x \|_{iw}^{\alpha_{CE}(x,y)} y}$$

Table V.5: Interworking Merge (continued)

Note that we do not force synchronization of primitive actions. Only communication actions have to synchronize. However, we require that the ordering of actions with respect to every single entity is preserved.

Example 2

$$c(p,q,k) \cdot act(q,a) \cdot c(q,r,l) \parallel_{iw} c(q,r,l) \cdot c(r,s,m) =$$
$$c(p,q,k) \cdot act(q,a) \cdot c(q,r,l) \cdot c(r,s,m)$$

V.5 Interworkings

The theory $BPA_{\delta,\partial_H,df}$ with the two operators for interworking merge and interworking sequencing will be called BPA_{iw}. An *interworking* is a process which can be constructed only from atomic actions and applications of the interworking sequencing and the interworking merge operator. The set of interworkings is denoted by IW. The set of deadlock free interworkings is denoted by IW_{df}. A process which is only constructed of atomic actions and the interworking sequencing operator is called a locally deterministic interworking. The set of local deterministic interworkings is called IW^{ld}. Likewise, IW^{ld}_{df} is used for deadlock free locally deterministic interworkings.

It is not always the case that the merge of two deadlock free interworkings is again deadlock free. We say that two interworkings are *consistent* if their interworking merge is deadlock free.

Example 3 *The following example shows two interworkings which don't agree on the communications with respect to entities q and r.*

$$c(p,q,k) \cdot c(q,r,l) \parallel_{iw} c(s,t,m) \cdot c(r,q,n) =$$
$$c(p,q,k) \cdot c(s,t,m) \cdot \delta + c(s,t,m) \cdot c(p,q,k) \cdot \delta$$

V.6 Properties

In this section we present some useful properties about BPA_{iw}. In general x, y and z are finite process expressions in BPA_{iw} and S is a set of actions,

Proposition 1 (Elimination) *(1) Every process in BPA_{iw} is equal to a process which has one of the following forms:*

$$\delta,\ a,\ a \cdot x,\ x + y$$

(2) BPA_{iw} is a conservative extension of BPA.

Proof Term rewrite analysis as in [1].

The operators $\parallel^S_{iw}$ and $\parallel_{iw}$ are commutative, while the arguments of the $\circ_{iw}$ operator may only be interchanged if they have no entities in common.

Proposition 2

$$E(x) \cap E(y) = \emptyset \Rightarrow x \circ_{iw} y = y \circ_{iw} x \quad \text{(V.1)}$$
$$x \parallel^S_{iw} y = y \parallel^S_{iw} x \quad \text{(V.2)}$$
$$x \parallel_{iw} y = y \parallel_{iw} x \quad \text{(V.3)}$$

Proof For (1) use

$$E(x) \cap E(y) = \emptyset \Rightarrow x \, \mathbf{L}\!\circ_{iw} y = y \, \mathbf{R}\!\circ_{iw} x$$

(2) and (3) are clear by symmetry of the definitions.

The operators $\parallel^S_{iw}$ and $\circ_{iw}$ are associative, while associativity for the $\parallel_{iw}$ operator only holds for pairwise consistent interworkings.

Proposition 3

$$(x \circ_{iw} y) \circ_{iw} z = x \circ_{iw} (y \circ_{iw} z)$$

Proof Simultaneous induction on the total number of symbols in x, y and z of the following four propositions.

$$\begin{array}{l}
(x \,\mathbf{L}\!\circ_{iw} y) \,\mathbf{L}\!\circ_{iw} z = x \,\mathbf{L}\!\circ_{iw} (y \circ_{iw} z) \\
(x \,\mathbf{R}\!\circ_{iw} y) \,\mathbf{L}\!\circ_{iw} z = x \,\mathbf{R}\!\circ_{iw} (y \,\mathbf{L}\!\circ_{iw} z) \\
(x \circ_{iw} y) \,\mathbf{R}\!\circ_{iw} z = x \,\mathbf{R}\!\circ_{iw} (y \,\mathbf{R}\!\circ_{iw} z) \\
(x \circ_{iw} y) \circ_{iw} z = x \circ_{iw} (y \circ_{iw} z)
\end{array}$$

Proposition 4

$$(x \parallel^{S}_{iw} y) \parallel^{S}_{iw} z = x \parallel^{S}_{iw} (y \parallel^{S}_{iw} z)$$

Proof Simultaneous induction on the total number of symbols in x, y and z of the following three propositions.

$$\begin{array}{l}
(x \lfloor\!\lfloor^{S}_{iw} y) \lfloor\!\lfloor^{S}_{iw} z = x \lfloor\!\lfloor^{S}_{iw} (y \parallel^{S}_{iw} z) \\
(x |^{S}_{iw} y) |^{S}_{iw} z = x |^{S}_{iw} (y |^{S}_{iw} z) \\
(x |^{S}_{iw} y) \lfloor\!\lfloor^{S}_{iw} z = x |^{S}_{iw} (y \lfloor\!\lfloor^{S}_{iw} z) \\
(x \parallel^{S}_{iw} y) \parallel^{S}_{iw} z = x \parallel^{S}_{iw} (y \parallel^{S}_{iw} z)
\end{array}$$

Proposition 5 *Let x, y and z be interworkings, let x be of the form x', $x' \cdot \delta$ or δ, y be of the form y', $y' \cdot \delta$ or δ and z be of the form z', $z' \cdot \delta$ or δ, such that x', y' and z' are pairwise consistent interworkings, then we have*

$$(x \parallel_{iw} y) \parallel_{iw} z = x \parallel_{iw} (y \parallel_{iw} z)$$

Proof (Omitted)

Example 4 *The fact that this proposition requires that the interworkings have to be consistent is shown by the following calculations.*

$$\begin{array}{l}
c(p,q,m) \parallel_{iw} (c(p,q,n) \parallel_{iw} c(p,q,n)) = c(p,q,m) \parallel_{iw} c(p,q,n) = \delta \\
(c(p,q,m) \parallel_{iw} c(p,q,n)) \parallel_{iw} c(p,q,n) = \delta \parallel_{iw} c(p,q,n) = c(p,q,n) \cdot \delta
\end{array}$$

Proposition 6

$$x \circ_{iw} \delta = x \cdot \delta \tag{V.1}$$

$$\delta \circ_{iw} x = x \cdot \delta \tag{V.2}$$

$$x \parallel_{iw} \delta = x \cdot \delta \tag{V.3}$$

Proof (1) Simultaneous induction with

$$x \,\mathbf{L}\!\circ_{iw} \delta = x \cdot \delta$$

(2) follows from (1) and proposition 2.

(3) Simultaneous induction of

$$\begin{array}{l}
x \lfloor\!\lfloor^{\emptyset}_{iw} \delta = x \cdot \delta \\
x \parallel^{\emptyset}_{iw} \delta = x \cdot \delta
\end{array}$$

using

$$\begin{array}{l} \delta \mathbin{\lfloor\!\lfloor}^{S}_{iw} x = \delta \\ \delta |^{S}_{iw} x = \delta \end{array}$$

The following proposition implies that a deadlock in an interworking is *global* and will be reached only if all admissible actions are executed.

Proposition 7 *Every interworking is deadlock free, equal to δ or of the form $p \cdot \delta$ where p is a deadlock free interworking.*

Proof (Omitted)

For the following properties we need the notion of a *trace* of a process. A trace is an element of $A^+ \cup A^* \ \delta \cup \{\delta\}$. So a trace is a non-empty sequence of atomic actions, with the restriction that only the last action may be a δ. Concatenation of traces is denoted by placing the traces next to each other. We define the set of traces of a process as follows.

$$\begin{array}{ll} tr(a) = \{a\} & \\ tr(ax) = \{at | t \in tr(x)\} & \text{if } a \neq \delta \\ tr(x+y) = tr(x) \cup tr(y) & \text{if } x \neq \delta \wedge y \neq \delta \end{array}$$

A process x is *deterministic* if it is of the following form.

$$\sum_{i \in I} a_i \cdot x_i + \sum_{j \in J} a_j$$

where we require that all a_k are different for $k \in I \cup J$ and all x_i are deterministic.

Proposition 8 *Every interworking is deterministic.*

Proof This follows directly from the definitions of the interworking operators.

Every interworking is completely determined by its trace set.

Proposition 9 *For interworkings x and y*

$$tr(x) = tr(y) \Rightarrow x = y$$

Proof See [4] for a proof that this proposition holds for all deterministic processes.

By $\tilde{x}$ we denote the process x where all sequential operators are replaced by interworking sequencing operators.

$$\begin{array}{l} \tilde{a} = a \\ \widetilde{a \cdot x} = a \circ_{iw} \tilde{x} \\ \widetilde{x + y} = \tilde{x} + \tilde{y} \end{array}$$

The class IW is closed under this operation. Moreover we have the following proposition.

Conjecture 1 *For every interworking x*

$$x = \tilde{x}$$

This operation can also be defined on traces.

$$\begin{aligned}&\tilde{a} = a\\ &\widetilde{ax} = a \circ_{iw} \tilde{x} \quad \text{if } a \neq \delta\end{aligned}$$

We conclude with the fact that every interworking is completely determined by a set of locally deterministic interworkings.

Conjecture 2 *For every interworking x*

$$tr(x) = \bigcup_{t \in tr(x)} tr(\tilde{t})$$

Bibliography

[1] J.C.M. Baeten, W.P. Weijland, Process algebra, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, ISBN 0 521 40043 0, 1990.

[2] J.A. Bergstra & J.W. Klop, Process algebra for synchronous communication, Inf. & Control 60, pp. 109-137, 1984.

[3] V. Encontre, e.a., Combining Services, Message Sequence Charts and SDL: Formalism, Method and Tools, SDL '91, Evolving Methods, Elsevier Science Publishers, ISBN 0 444 88976 0, 1991.

[4] J. Engelfriet, Determinacy $\rightarrow$ (observation equivalence $=$ trace equivalence), TCS 36(1), pp.21-25, 1985.

[5] C.A.R. Hoare, Communicating sequential processes, Prentice-Hall, 1985.

[6] I. Jacobson, a.o., Object-Oriented Software Engineering, A Use Case Driven Approach, Addison Wesley, ISBN 0 201 54435 0, 1992.

[7] S. Mauw, M. van Wijk, T. Winter, Syntax and semantics of synchronous interworkings, The concrete syntax IW, 1992.

[8] S. Mauw, M. van Wijk, T. Winter, Syntax and semantics of synchronous interworkings, Informal semantics, 1992.

[9] S. Mauw, M. van Wijk, T. Winter, Syntax and semantics of synchronous interworkings, The syntax of T and denotational semantics of IW, 1992.

[10] S. Mauw, M. van Wijk, T. Winter, Syntax and semantics of synchronous interworkings, A formal semantics, 1992.

[11] R. Milner, A calculus of communicating systems, Springer LNCS 92, 1980.

[12] E. Rudolph, Syntax and Semantics of Basic Sequence Charts, Contribution to the Study Group X, WP X/3, Q8, CCITT meeting, Geneva 1992.

[13] E. Rudolph, P. Graubmann, J. Grabowski, Towards an SDL-Design-Methodology Using Sequence Charts Segments, SDL '91, Evolving Methods, page 237-252, Elsevier Science Publishers, ISBN 0 444 88976 0, 1991.

[14] J. Rumbaugh, a.o., Object-Oriented Modeling and Design, Prentice Hall International, ISBN 13 630054 5, 1991.

[15] P.A.J. Tilanus, A formalisation of Message Sequence Charts, SDL '91, Evolving Methods, page 273-288, Elsevier Science Publishers, ISBN 0 444 88976 0, 1991.

Contents