Theory of Handshake Circuits: Highlights

Igor Benko

April 22, 1994

Abstract

In this report we discuss the theory of handshake circuits developed in [vB92]. We focus on the decisions made in the development of the formalism, and on the consequences of these decisions for the design of a VLSI programming language Tangram, its semantics, and the compilation method from Tangram to handshake circuits.

1 Introduction

Asynchronous circuits have been recognized as a possible way for building circuits since the early days of computing. However, they have been neglected, mainly because it seemed they were substantially harder to design and build than clocked circuits. A clock was considered an essential component, since it introduced discreteness into time, and, furthermore, it made refreshing of dynamic memory elements easy. As clocked technology seems to be reaching its limits, asynchronous circuits have been becoming increasingly popular. The reason for this are a number of advantages for which asynchronous circuits have been advocated for; among them are the avoidance of many timing problems (in particular, the avoidance of clock skew), great potential for high speed and low power consumption, and a possibility of design reuse. One reason why asynchronous circuits have been considered difficult to design and build was the lack of an appropriate formalism that would be able to cope with parallelism, which inherently arises in an asynchronous computation.

In this report we investigate a formalism proposed in [vB92] and a VLSI programming language Tangram, based on this formalism. Handshake circuits introduced in the formalism capture a natural idea for an asynchronous communication: every communication has to be initiated by one partner and acknowledged by another partner. This approach, although very easy to grasp, results in a number of possibly redundant communications, since every request has to be acknowledged individually. Furthermore, in the underlying formalism we always have to worry about completing the handshakes properly, which results in rather complicated definitions and properties one has to deal with.

The remainder of the report is organized as follows. Section 2 introduces handshake structures, Section 3 introduces handshake processes, Section 4 defines handshake circuits, and Section 5 contains a discussion of the calculus of sequential handshake processes. VLSI programming language Tangram is introduced in Section 6. Section 7 briefly describes the compilation from Tangram to handshake processes. Finally, Section 8 collects comments on the theory and language developed in [vB92].

2 Handshake structures

In a handshake communication one of the partners plays the *active* role: it initiates communication and then waits for an acknowledgment. The other partner has the *passive* role: it waits until the communication is initiated by the active partner and then responds by an acknowledgment. Note that a handshake can be completed only when both parties are ready to participate in it: a handshake thus synchronizes the two communicating parties. The pair of links between two communicating parties is called a *channel*; the two terminals of a channel are called *ports*.

A port p can be defined as a pair of disjoint, finite, and non-empty sets of symbols $\langle \mathbf{0}p, \mathbf{1}p \rangle$, where $\mathbf{0}p$ denotes the set of request symbols and $\mathbf{1}p$ denotes the set of acknowledge symbols. A handshake then consists of an occurrence of an event (i.e., a symbol) from $\mathbf{0}p$ followed by an event from $\mathbf{1}p$. With $\mathbf{a}p$ we denote $\mathbf{0}p \cup \mathbf{1}p$, the *alphabet* of port p.

Note that for an active port p, $\mathbf{0}p$ is a set of output symbols and $\mathbf{1}p$ is a set of input symbols; the converse is true for a passive port. Input and output alphabets can be used to encode the data. We can, thus, distinguish different types of ports according to the direction where the data is sent:

- *input port* has several input symbols and one output symbol;
- *output port* has several output symbols and one input symbol;
- *nonput port* has one input and one output symbol;
- *biput port* has several input and several output symbols.

A port defines one connection between a component and its environment. Since a component may have several connections, we define a *port set* as a finite (possibly empty) set of ports. In order to distinguish communications that correspond to different ports of port set P, we require symbol sets of ports in a port set be disjoint; then P is called a *proper port set*:

$$(\forall a, b: \ a \in P \wedge b \in P: \ a = b \vee \mathbf{a}a \cap \mathbf{a}b = \emptyset)$$

The interface between a component and its environment consists of passive and active ports. Therefore, we define a *port structure* A as a pair $\langle A^\circ, A^\bullet \rangle$, where A° is the set of *passive* ports and $A^\bullet$ is the set of *active* ports. With $\mathbf{a}A$ we denote the *alphabet* of A, the set of all symbols in ports of the port structure A: $\mathbf{a}A = \mathbf{a}A^\circ \cup \mathbf{a}A^\bullet$ (note that the operator $\mathbf{a}$ extends naturally over port sets and port structures).

The sets A° and $A^\bullet$ need not be disjoint. Their intersection denotes the set of *internal ports* of the structure A; the ports that are not internal are called *external* ports. We say that port structures A and B are *compatible* if $A^\circ \cup A^\bullet \cup B^\circ \cup B^\bullet$ is a proper port set.

So far we have defined an interface between a component and its environment, but we have not said anything about the interaction between the component and the environment. A common way of defining the behavior of a component is to specify its trace set: each trace records a possible sequence of communication events (denoted by symbols from port alphabets) in which the component is engaged. In our case we restrict behaviors to *handshake traces* only: a handshake trace is a trace where occurrences of $\mathbf{0}$-symbols (requests) and $\mathbf{1}$-symbols (acknowledgments) strictly alternate, beginning with a request. Our universe of behaviors is then defined as:

Definition 2.1 (Handshake traces) *The set of handshake traces with port structure A is denoted by A^H and defined as*

$$A^H = \{t : t \in (\mathbf{a}A)^* \wedge (\forall a, s: \ a \in \mathbf{a}A \wedge s \leq t : 0 \leq \mathrm{len}\cdot(s\lceil \mathbf{0}a) - \mathrm{len}\cdot(s\lceil \mathbf{1}a) \leq 1) : t\},$$

where with $s \leq t$ we denote s is a prefix of t, and $\mathrm{len}\cdot t$ *denotes the length of trace t.*

An early decision to consider handshake behaviors only draws many consequences. One can immediately question whether such a limitation should be built so deeply into a formalism, or should handshake traces rather be introduced later, for example at a time when a compilation method is defined, leaving more space for different compilation methods, and more space for eventual optimizations.

An interface and a behavior definition can now be combined into a *handshake structure*: handshake structure S is a pair $\langle \mathbf{p}S, \mathbf{t}S \rangle$, where $\mathbf{p}S$ is a port structure and $\mathbf{t}S \subseteq (\mathbf{p}S)^H$ is a set of handshake traces. Intuitively, $\mathbf{t}S$ defines a behavior of S. We remark it is not specified how the behavior is defined: $\mathbf{t}S$ could contain all possible behaviors of S, or, it could contain only a set of distinguished traces from which the complete behavior of S can be derived.

3 Handshake processes

Handshake structures were introduced to model physical devices, in particular, asynchronous circuits. In order to bring the model as close as possible to the physical reality, and in order to make a mathematical model feasible, we introduce a number of properties to characterize handshake structures used for circuit description. Such handshake structures are called *handshake processes*.

A handshake process is used to represent an external behavior of a component. Hence, a handshake process has no internal ports. We also require the trace set be non-empty: even if a component is always idle, its behavior can be specified as an empty string. The remaining three properties deserve a more detailed discussion.

3.1 Quiescence

Let trace t represent a behavior of handshake process P. After P has engaged in t, the environment may not be able to obtain an additional output from P. There are two reasons why this can happen: either the (required) behavior of P does not permit an extension of t with an output symbol, or, even if t could be extended with an output symbol, P may nondeterministically choose to remain idle. Such a trace is called a *quiescent* trace.

The traces contained in a trace set of a handshake process are the quiescent traces of that process. Therefore, behavior of a process is specified by listing all points when the process may (or must) wait for an input from the environment. The set of all observable traces for handshake process P is then the prefix closure of $\mathbf{t}P$, denoted by $(\mathbf{t}P)^{\leq}$.

Why does [vB92] specify behavior of a component with a set of quiescent traces and not with the set of all possible traces ? A reason for this may be that the formalism as defined in [vB92] does not provide any mechanism for imposing restrictions on a behavior of a component, or on behavior of the component's environment. Thus, if the set of all possible traces was used to specify the behavior, we would not have any means to specify when a process can decide to wait for an input.

A special subset of quiescent traces are *passive traces*. In order to introduce them, we first define a *successor set* of a trace t with respect to handshake structure S, denoted by $suc(t, S)$, as a set of symbols that can be used to extend t:

$$suc{\cdot}(t, S) = \{a :\ a \in \mathbf{a}A \wedge ta \in \mathbf{t}S^{\leq}\}$$

If the successor set of trace t does not contain any output symbols, t is called a *passive trace* in handshake structure S; this property of t is denoted by $pas{\cdot}(t, S)$. A handshake structure is passive, if its initial behavior, i.e., an empty trace, is passive.

Finally, we define a *passive restriction* of handshake structure S, denoted by $Pas\cdot S$, as a handshake structure with the same port structure as S, and with the trace set that contains all passive prefixes of all traces in $\mathbf{t}S^{\leq}$:

$$Pas\cdot S = \langle \mathbf{p}S, \{t:\ t \in \mathbf{t}S^{\leq} \wedge pas\cdot(t, S):\ t\}\rangle$$

For a handshake process S, we require that trace set of $Pas\cdot S$ is a subset of $\mathbf{t}S$. This is a necessary requirement, because passive traces are quiescent a priori. One can, however, specify additional quiescent traces and with this impose additional restrictions on behavior of a component. We comment that this kind of specification seems quite complicated, especially because [vB92] does not offer a quiescent-trace calculus that would support an easy specification of a trace set. Trace sets are thus specified either with a graphic notation which is not appropriate for complex structures, or they are specified using various closure operators that are not intuitive.

3.2 Reordering

An important property of asynchronous circuits is *delay insensitivity*. Intuitively, a circuit is delay-insensitive if the correctness of its operation does not depend on delays in components and wires. Reordering deals with delays in connections between two components. For example, assume that, after component P has exhibited behavior t, the environment sends signals along links a and b in that order, and that P is ready to receive them, i.e., $tab \in \mathbf{t}P^{\leq}$. If no assumptions about communication delays are made, a and b may arrive in the opposite order, therefore we require that tba belongs to $\mathbf{t}P^{\leq}$. We say that trace tba *reorders* trace tab. For a similar reason we allow reordering of two output symbols: for outputs c and d, trace tdc reorders trace tcd.

Reordering may also be required for an output and an input communication. Let a be an input, and c an output symbol. If $tca \in \mathbf{t}P^{\leq}$, input a was not required by P in order to produce output c. Thus, if a has smaller delay, it could arrive before P produces c, so tac reorders tca, *provided, that both are handshake traces*. We comment that the reordering allows input a that is supposed to be the environment's response to output c to arrive even *before* c is produced.

Reordering can be formalized as a binary relation: for a port structure B, and handshake traces t and s, $t\mathbf{r}_B s$ means that t reorders s. If B is obvious, the subscript may be omitted. It can be seen that $\mathbf{r}$ is a preorder on B^H, so we can introduce a *reorder closure* of a handshake trace set T, denoted by $T^{\mathbf{r}}$, and *reorder closedness* of T, denoted by $(\mathbf{r})\cdot T$. Both operations are extended to handshake structures in an obvious way.

We require that the trace set of a handshake process is reorder closed. A consequence of this requirement is delay insensitivity of handshake processes (proved to satisfy Udding's definition [Udd86]). Although delay insensitivity is a desired property of an asynchronous circuit, it seems that restriction to delay-insensitive circuits *only* does not contribute to flexibility of a design methodology based on a theory of handshake processes; for example, speed independent circuit design is not supported by the proposed theory.

3.3 Receptiveness

The theory of handshake processes as developed in [vB92] does not contain a notion of computation interference. Furthermore, the reordering property of handshake processes makes it impossible to control when an input arrives to a receiving side of a channel. Therefore, there is a need for a property of handshake processes that guarantees every communication can take place without the danger that the receiving side is not ready for it. This property is called *receptiveness*.

Let B be a port structure, $t \in B^H$, and $u \in (\mathbf{i}B \setminus \mathbf{o}B)^*$, i.e., t is a handshake trace and u is a string containing input symbols only. Then, handshake trace tu is an *input extension* of t in B, denoted by $tu\,\mathbf{x}_B\,t$. If B is obvious from the context, we may omit the subscript. Relation $\mathbf{x}$ is a preorder. Therefore, for a handshake-trace set T, we use $T^{\mathbf{x}}$ to denote the input-extension closure of T, and $(\mathbf{x})\cdot T$ to denote the input-extension closedness of T.

If the prefix closure of the trace set of a handshake structure is input-extension closed, the handshake structure is called *receptive*. Intuitively, a receptive handshake structure is always ready to accept an input from the environment. This means that any passive port is always ready to accept an initiation of a handshake from the environment (if a handshake is not in "progress", of course), and that any active port is, after it has initiated a handshake, always ready to accept the acknowledgment.

A handshake process is required to be receptive. Because a receptive handshake structure is always ready for an input from the environment, one can see that a parallel composition of receptive structures is free of computation interference. We remark that in contrast to [Ebe91], [vB92] does not distinguish between parallel composition and decomposition. In [Ebe91], parallel composition is intended to specify the behavior of one process, making it possible to express a parallel nature of the behavior. Decomposition, however, is intended to describe the network of properly connected components that satisfy certain behavioral conditions. In a sense, van Berkel's parallel composition corresponds to Ebergen's decomposition, with an additional simplification that all processes with connectable port structures can be connected in parallel.

We also mention that the definition of receptiveness as defined is [vB92] differs from the original definition of receptiveness [Jos92]. Josephs's receptive process never blocks an input from the environment, but, if the process is not ready to accept a particular input, the subsequent behavior of the process is undefined. Van Berkel's receptive process never blocks an input from the environment and is *always ready* to accept any possible input. In Section 5.1.9 we show that van Berkel's notion of receptiveness causes problems in defining a *choice* operator for (sequential) handshake processes.

The requirement that a handshake process must be receptive was introduced mainly to simplify the definition of parallel composition. A consequence of this requirement is, however, costly circuit realizations. In particular, readiness for inputs through passive ports requires latches that "remember" input transitions for further processing. Furthermore, certain Tangram constructs (choice, for example) require arbiters to arbitrate between transitions through the guard inputs. Although [vB92] recognizes this deficiency and briefly discusses a possibility of replacement of a receptive structure with a non-receptive one, no solid proposal is given. We believe one of the main problems is that parallel composition and decomposition are not distinguished. The consequence is that a definition of a parallel composition in an environment, where computation interference is possible, seems quite complicated.

3.4 Handshake process

Now we are ready to define a handshake process:

Definition 3.1 (Handshake process) *A handshake process is a handshake structure $\langle A, T\rangle$ that satisfies the following conditions:*

1. $A^{\circ} \cap A^{\bullet} = \emptyset$ — *no internal ports*
2. $T \neq \emptyset$ — *non-empty trace set*
3. $\mathbf{t}(\mathrm{Pas}\cdot\langle A, T\rangle) \subseteq T$ — *passive traces are quiescent*
4. $(\mathbf{r})\cdot T$ — *reorder closedness*
5. $(\mathbf{x})\cdot T^{\leq}$ — *receptiveness*

The set of all handshake processes with alphabet A is denoted by $\Pi\cdot A$.

We can introduce a natural ordering relation between handshake structures with port structure A: if $\mathbf{t}S \supseteq \mathbf{t}T$, we say S *refines* to T, and denote it by $S \sqsubseteq T$. According to this ordering, the greatest lower bound of handshake structures S and T can be obtained as the *union* of the structures, defined as $\langle A, \mathbf{t}S \cup \mathbf{t}T\rangle$, and denoted by $S \sqcap T$. [vB92] proves that the union of two handshake processes is also a handshake process, which behaves either as

S or as T. Because the choice between S and T is nondeterministic, $S \sqcap T$ is also called a *nondeterministic composition* of S and T.

The intersection of trace structures S and T is defined as $\langle A, \mathbf{t}S \cap \mathbf{t}T \rangle$, and denoted by $S \sqcup T$. If the least upper bound of handshake processes S and T exists in $\Pi \cdot A$, it can be obtained as an intersection of the two processes.

One can prove that $(\Pi \cdot A, \sqsubseteq)$ is a complete partial order with $\langle A, A^H \rangle$ as least element, and $(\sqcup i : 0 \leq i : P_i)$ as limit of chain $P_0 \sqsubseteq P_1 \sqsubseteq P_2 \ldots$. This property of $\Pi \cdot A$ comes useful when we define the infinite repetition of a process as the least fixpoint of an equation of the form $P = F \cdot P$.

4 Handshake circuits

A handshake circuit is defined as a set of handshake processes, that operate in parallel. The question we have to address is whether any two handshake processes can be connected in parallel. The problem that arises here is called *output interference*: outputs must not be connected to outputs. Therefore, parallel composition is defined for *connectable* handshake processes only. Handshake processes R and S are connectable, denoted by $R \bowtie S$, if

1. their port structures, $\mathbf{p}S$ and $\mathbf{p}R$, are compatible,
2. $\mathbf{p}S^\circ \cap \mathbf{p}R^\circ = \emptyset$, and $\mathbf{p}S^\bullet \cap \mathbf{p}R^\bullet = \emptyset$.

The first condition requires that ports of R and S are either identical, or they have disjoint symbol sets; the second conditions prevents connections of outputs to outputs.

Because handshake processes that form a handshake circuit operate in parallel, we require they be pairwise connectable:

Definition 4.1 (Handshake Circuit) *A handshake circuit is a finite set of pairwise connectable handshake processes.*

How do we define the behavior of the parallel composition of two connectable handshake structures ? The first attempt could be defining the behavior as the *weave* of two structures: the weave of connectable handshake structures R and S, denoted by $R \,\mathbf{w}\, S$, is defined as

$$R \,\mathbf{w}\, S = \langle A, \{t : t \in A^H \wedge t\lceil \mathbf{p}R \in \mathbf{t}R \wedge t\lceil \mathbf{p}S \in \mathbf{t}S\} \rangle$$

where $A = \mathbf{p}R \cup \mathbf{p}S$. This is a standard definition of the weave — behavior of the weave must be in accordance with behaviors of the weavands [vdS85].

The weave of two handshake processes is, however, not a handshake process in general, because of its internal ports. An attempt to remedy this shortcoming is to project the quiescent traces of the weave on the alphabets of external ports and conceal the internal ports. Handshake processes are, unfortunately, not closed under such an operation (known as *blending* [vdS85]). The reason is that we do not consider *divergences*, i.e. traces that lead to unbounded amount of internal communication without any external communication. Once a parallel composition of handshake processes engages in a so-called infinite chatter, the environment is not able to obtain an output from the parallel composition, therefore a divergence is a quiescent trace. From the point of view of a handshake process participating in the parallel composition, a divergence may not be a quiescent trace, because it may involve outputs to another process participating in the infinite chatter. Therefore, we have to add divergences to the set of quiescent traces of a parallel composition.

Formally, for a handshake structure $\langle A, T\rangle$ we define the *divergences of* $\langle A, T\rangle$ as the trace set

$$div\cdot\langle A, T\rangle = \{t : (\forall n : 0 \leq n : (\exists u : u \in (A^\circ \cap A^\bullet)^* \wedge tu \in T : n < len\cdot u)) : t\}$$

Recall, that $A^\circ \cap A^\bullet$ is the set of internal symbols of A.

The parallel composition of connectable handshake processes P and Q, can now be defined as

$$P \parallel Q = \langle A, (\mathbf{t}P \;\mathbf{w}\; \mathbf{t}Q) \cup div\cdot(P^{\leq} \;\mathbf{w}\; Q^{\leq}))\lceil A\rangle,$$

where $A = \mathbf{e}(\mathbf{p}P \cup \mathbf{p}Q)$, the set of external ports of the composition of P and Q.

We remark that [vB92] does not investigate characterization of divergences any further.

5 Calculus of sequential handshake processes

In order to simplify specification of handshake circuits, it is desirable to provide a calculus for handshake processes. One of the desirable operations in such a calculus would be a sequential composition: a sequential composition of processes P and Q first behaves as P and then, after a "successful termination" of P, it behaves like Q. A possible way to introduce the notion of successful termination is to specify a set of *terminal traces* for each handshake process. We expect that upon successful termination all initiated handshakes have been completed, i.e. all terminal traces must be *closed*. Furthermore, reordering should have no effect on termination, therefore the set of terminal traces should be reorder closed.

A combination of a handshake process and a set of its terminal traces is called *sequential handshake process* and is defined as:

Definition 5.1 (Sequential handshake process) *Let P be a triple $\langle A, T, U\rangle$, where A is a port structure and T, the set of quiescent traces, and U, the set of terminal traces, are subsets of A^H. Furthermore, let $V = T \cup U^{\mathbf{xr}}$. P is a sequential handshake process if:*

1. $A^\circ \cap A^\bullet = \emptyset$ — *no internal ports;*
2. $V \neq \emptyset$ — *non-empty trace set;*
3. $\mathbf{t}(\mathrm{Pas}\cdot\langle A, V\rangle) \subseteq V$ — *all passive traces are quiescent or terminal;*
4. $(\mathbf{r})\cdot T \wedge (\mathbf{r})\cdot U$ — *reorder closedness;*
5. $(\mathbf{x})\cdot V^{\leq}$ — *receptiveness;*
6. $(\forall u :\ u \in U : \mathrm{closed}\cdot u)$ — *terminal traces are closed,*

Note that a trace is allowed to be both, quiescent and terminal. The set of terminal traces of sequential handshake process P can also be denoted as $\mathbf{u}P$.

The set of all sequential handshake processes with port structure A is denoted by $\Sigma\cdot A$. There are a number of generic sequential handshake processes in $\Sigma\cdot A$ we use later in the calculus:

> $CHAOS\cdot A$: the least predictable sequential handshake process. It can engage in a handshake through any of its ports at any time, it can become quiescent at any time, and it can terminate after any closed trace:
>
> $$CHAOS\cdot A = \langle A, A^H, \{t :\ t \in A^H \wedge closed\cdot t\}\rangle$$
>
> $RUN\cdot A$: Always willing to engage in a handshake in any of its ports, but it never terminates:
>
> $$RUN\cdot A = \langle A, \mathbf{t}Pas\cdot\langle A, A^H\rangle, \emptyset\rangle$$
>
> $STOP\cdot A$: never completes a handshake through any of its ports, it also never terminates:
>
> $$STOP\cdot A = \langle A, \emptyset^{\mathbf{x}}, \emptyset\rangle$$
>
> $SKIP\cdot A$: never engages in a handshake through any of its ports. All it does is terminate successfully:
>
> $$SKIP\cdot A = \langle A, \emptyset, \{\varepsilon\}\rangle$$

A sequential handshake process with the empty set of terminal traces never successfully terminates. Such a process is called a *permanent sequential process*.

We can define the notion of refinement in a similar way we did with the handshake processes: sequential handshake process P refines to sequential handshake process Q, denoted by $P \sqsubseteq Q$, if $\mathbf{t}Q \subseteq \mathbf{t}P$ and $\mathbf{u}Q \subseteq \mathbf{u}P$. A possible interpretation of $P \sqsubseteq Q$ is that Q *implements* P. The reasoning behind such an interpretation is that in the process of specification we do not wish to introduce too many restrictions for the implementation; in particular, we wish to take advantage of nondeterminism in the specification. For the implementation purposes it may be, however, sufficient to realize only a subset of the specified behaviors and thus obtain cheaper circuits. Although this decision may seem correct from the practical point of view, we comment that it allows implementations using "accept-all-produce-nothing" components only. Another possible decision is that an implementation should be able to produce all specified behaviors [Ebe91]. In [Pee90] it was shown that such requirement may be too weak (as it allows implementations that may exhibit deadlock, for example) and too strong (as it does not allow certain simple implementations) at the same time.

The definitions of the union and the intersection of sequential handshake processes with port structure A are also similar to the definitions concerning handshake processes:

union: $P \sqcap Q = \langle A, \mathbf{t}P \cup \mathbf{t}Q, \mathbf{u}P \cup \mathbf{u}Q \rangle$; $P \sqcap Q$ is the greatest lower bound of P and Q in the partial order $\left(\Sigma \cdot A, \sqsubseteq\right)$;

intersection: $P \sqcup Q = \langle A, \mathbf{t}P \cap \mathbf{t}Q, \mathbf{u}P \cap \mathbf{u}Q \rangle$; $P \sqcup Q$ is not a sequential handshake process in general.

One can show that $\left(\Sigma \cdot A, \sqsubseteq\right)$ is a complete partial order with $CHAOS \cdot A$ as least element, and $(\sqcup i \,:\, 0 \leq i \,:\, P_i)$ as limit of chain $P_0 \sqsubseteq P_1 \sqsubseteq P_2 \ldots$

5.1 Process calculus

The calculus we describe in this section is limited to sequential processes with undirected ports (only one output and one input symbol), therefore data transfer is excluded. Although [vB92] does not extend the calculus to the level it could cope with data transfer, it is suggests this could be done within the same mathematical framework. Since the calculus is used to define a semantics of Tangram, where data types are present, we would prefer to have a precise treatment of the extended calculus.

We begin the description of the calculus with the definition of four basic sequential processes:

1. $\mathbf{stop} = STOP{\cdot}\langle\emptyset,\emptyset\rangle = \langle\langle\emptyset,\emptyset\rangle,\{\varepsilon\},\emptyset\rangle$ — **stop** can be seen as an unconditional deadlock;
2. $\mathbf{skip} = SKIP\cdot\langle\emptyset,\emptyset\rangle = \langle\langle\emptyset,\emptyset\rangle,\emptyset,\{\varepsilon\}\rangle$ — **skip** can be seen as an "empty action";
3. $a^\circ = \langle a^\circ,\{\varepsilon\},\{a_0a_1\}\rangle$ — a° denotes a process that can be involved in one handshake, initiated by the environment;
4. $a^\bullet = \langle a^\bullet,\{a_0\},\{a_0a_1\}\rangle$ — $a^\bullet$ denotes a process that can be involved in one handshake, initiated by the process itself.

We comment that a° (and $a^\bullet$) may denote both, a port structure and a sequential handshake process. This is only one instance of large number of overloadings used in [vB92].

In the remainder of this section we define a number of operations on sequential handshake processes; [vB92] proves that sequential processes are closed under all operations defined.

5.1.1 Parallel composition

Parallel composition is one of the most important operations in the calculus. In [vB92] the parallel composition of connectable sequential processes P and Q is denoted by $P \parallel Q$ and defined as follows:

Let $A = \mathbf{e}(\mathbf{p}P \cup \mathbf{p}Q)$, the set of external ports of P and Q, and let the weave of trace sets V and W (defined as usual) be denoted as $V\,\mathbf{w}\,W$. Then

1. $\mathbf{p}(P \parallel Q) = A$ — parallel composition conceals the common ports. This decision contributes to simplicity of the calculus, but it also makes the parallel composition of sequential handshake processes inappropriate for semantics of the parallel composition in Tangram. The result is that [vB92] has to redefine the parallel composition in order to accommodate the desired properties of Tangram, and the properties of the compilation method from Tangram to handshake circuits.
2. $\mathbf{t}(P \parallel Q) = \Big(\underbrace{\mathbf{t}P\,\mathbf{w}\,\mathbf{t}Q}_{1} \cup \underbrace{(\mathbf{u}P)^{\mathbf{x}}\,\mathbf{w}\,\mathbf{t}Q}_{2} \cup \underbrace{(\mathbf{u}Q)^{\mathbf{x}}\,\mathbf{w}\,\mathbf{t}P}_{3} \cup \underbrace{div{\cdot}(\mathbf{t}P^{\leq}\,\mathbf{w}\,\mathbf{t}Q^{\leq})}_{4}\Big)\lceil A$ — [vB92] does not devote a single word of explanation to this definition. We believe, the intuition behind the definition is that the parallel composition is quiescent if (1), both processes are quiescent, (2) and (3), one process has terminated and another process is quiescent, and (4), infinite chatter (livelock) occurs.
3. $\mathbf{u}(P \parallel Q) = (\mathbf{u}P\,\mathbf{w}\,\mathbf{u}Q)\lceil A$ — the parallel composition terminates successfully if both processes terminate successfully.

The next two operations deal with modifying the port structure of a sequential process. Sometimes, operations on processes are defined if they have equal port structure, therefore there is a need for an operation that extends a port structure of a sequential process, the *extension*. Sometimes, however, we wish a handshake through certain ports be hidden from the environment; the corresponding operation is called *concealment*.

Before we define the operations introduced above, we introduce the notion of *conformance* of two port structures: Port structures A and B are *conformant*, denoted by $A \bigtriangleup\!\!\bigtriangleup B$, if

1. A and B are compatible (their ports are either equal or have disjoint port alphabets);
2. $A^\circ \cap B^\bullet = \emptyset$ and $A^\bullet \cap B^\circ = \emptyset$ — a port that is common to A and B is either active in both A and B, or passive in both A and B.

The second requirement can be interpreted as a request that processes with port structures A and B share the same end of a channel. The need for this requirement will become apparent when we define the nondeterministic composition of sequential processes.

5.1.2 Extension

Now we are ready to define the *extension*: Let A be a port structure, P a sequential process, and let $A \bigtriangleup\!\!\bigtriangleup P$. Then, the extension of P with A is denoted by $(A)\cdot P$ and is defined as

$$(A)\cdot P = P \parallel \mathit{SKIP}\cdot(A \setminus \mathbf{p}P)$$

Note that the second operand in the parallel composition in fact only "takes care" of extending the port structure.

5.1.3 Concealment

The *concealment* is the operation that hides handshakes through certain ports from the environment. Note, that an effect of concealment can be hiding of unbounded number of handshakes through the concealed port, therefore the definition must take into account the possible divergences: Let A be a port structure, $\bar{A}$ be the reflection of port structure A (the roles of active and passive ports are swapped), P a sequential process, $A \bigtriangleup\!\!\bigtriangleup P$, and $B = \mathbf{p}P \setminus A$. Handshake process P with A concealed, denoted by $|[A \mid P]|$, is defined as

$$|[A \mid P]| = \langle B, (\mathbf{t}P \cup \mathit{div}\cdot\langle \mathbf{p}P \cup \bar{A}, \mathbf{t}P^{\leq}\rangle)\lceil B, \mathbf{u}P\lceil B\rangle$$

5.1.4 Nondeterministic composition

The *nondeterministic composition* of sequential handshake processes P and Q can be interpreted similarly as the nondeterministic composition of handshake processes: it behaves either like P or like Q, where the selection between P and Q is nondeterministic. Formally, the nondeterministic composition of conformant sequential processes P and Q, denoted by $P \sqcap Q$, is defined as

$$P \sqcap Q = (\mathbf{p}Q)\cdot P \sqcap (\mathbf{p}P)\cdot Q.$$

Note that operator $\sqcap$ stands for the *nondeterministic composition* on the left-hand side of the equation, and for the *union* on the right-hand side... We require that P and Q are conformant because the decision which behavior the nondeterministic composition of the two follows must not affect roles (active/passive) of the ports with respect to the environment. Furthermore, we have to "equalize" the alphabets of both operands of the union, because the port structure of the composition must be invariant to the choice of the behavior.

5.1.5 Sequential composition

Sequential composition was the main reason for introducing sequential processes: the composition of processes P and Q, first behaves like P, and, upon successful termination of P, continues to behave like Q.

If P and Q have the same port structure A, their sequential composition is defined as

$$P ;_A Q = \langle A, \mathbf{t}P \cup (\mathbf{u}P\,\mathbf{t}Q)^{\mathbf{r}}, (\mathbf{u}P\,\mathbf{u}Q)^{\mathbf{r}} \rangle,$$

where the concatenation of two sets is denoted by juxtaposition. Note that a quiescent trace of the sequential composition is either a quiescent trace of P, or a terminal trace of P, concatenated with a quiescent trace of Q (modulo reordering). The terminal traces of $P;_A Q$ are composed of the terminal traces of P, concatenated with terminal traces of Q.

If P and Q do not have the same port structure, we can first use the extension operator to "equalize" their port structures, and then use the restricted sequential composition given above. In this case we also require that the port structures of P and Q are conformant. The definition of the general sequential composition is then

$$P; Q = (\mathbf{p}Q)\cdot P \;;_B\; (\mathbf{p}P)\cdot Q,$$

where $B = \mathbf{p}P \cup \mathbf{p}Q$.

5.1.6 N-fold repetition

The N-fold repetition corresponds to a loop in programming languages. It is denoted by $\#N[P]$ and is defined recursively: The 0-fold repetition of P behaves like **skip** (successful termination only). For $N > 0$, the N-fold repetition behaves like P, and after successful termination of P it behaves like the $(N-1)$-fold repetition of P.

5.1.7 Infinite repetition

The infinite repetition of P behaves like an infinite sequential composition of P. It is denoted by $\#[P]$, and defined as the least fixpoint of F, where F is defined as $F{\cdot}X = P;X$. Because $(\Sigma{\cdot}A, \sqsubseteq)$ is a complete partial order and because the sequential composition is continuous, the fixpoint above can be calculated as the limit of the chain $(i\,:\,0 \le i\,:\,\#i[P];\,\mathit{CHAOS}{\cdot}\mathbf{p}P)$.

5.1.8 Enclosure

Assume P is a sequential process and $a^\circ \notin \mathbf{p}P$. The enclosure of P by a passive handshake a° can be interpreted as follows: The first phase of the handshake on port a° (initiated by the environment) enables process P. When P successfully terminates, the handshake on a° is completed. The formal definition of enclosure of P by a handshake on a°, denoted by $a^\circ\,:\,P$, is

$$a^\circ\,:\,P = \langle a^\circ \cup \mathbf{p}P, \mathbf{t}(\mathit{STOP}{\cdot}\mathbf{p}P) \cup (\{a_0\}\,\mathbf{t}P)^{\mathbf{r}}, (\{a_0\}\,\mathbf{u}P\,\{a_1\})^{\mathbf{r}}\rangle$$

Because of the reordering and receptiveness, the enclosure "enjoys" a rather unusual property

$$a^\circ\,:\,b^\circ\,:\,P = b^\circ\,:\,a^\circ\,:\,P$$

5.1.9 Choice

A sequential process of a form $a^\circ : P$ or $a^\circ; P$ is called a *guarded* process — the first phase of a handshake on a° can be seen as a guard for process P. Sometimes, we are interested in a process that behaves either like $a^\circ; P$ or $b^\circ; Q$, such that the environment may choose between P and Q by either offering a_0 or b_0.

The choice can be formally defined as follows: Let P and Q be two conformant guarded processes with disjoint guards a and b. If P is the enclosure $a^\circ\,:\,P'$, we require $a^\circ \notin \mathbf{p}P'$, and similarly for b and Q'. The choice between P and Q is denoted by $[P \mid Q]$ and is defined as

$$[P \mid Q] = \langle A, \mathbf{t}((A)\cdot P) \setminus \{b_0\} \cup \mathbf{t}((A)\cdot Q) \setminus \{a_0\}, \mathbf{u}P \cup \mathbf{u}Q \rangle,$$

where $A = \mathbf{p}P \cup \mathbf{p}Q$.

[vB92] states that process $[a^\circ\,;\,P \mid b^\circ\,;\,Q]$ is different from $a^\circ; P \sqcap b^\circ; Q$, in which the environment *may* offer *both* a_0 and b_0 and the choice between P and Q is made nondeterministically. We comment that because of the way [vB92] defines receptiveness, there is no means to prevent the environment of the choice process, $[a^\circ\,;\,P \mid b^\circ\,;\,Q]$, to offer both a_0 and b_0, and thus reduce the choice to the nondeterministic composition. This means that the definition of receptiveness in a sense limits the expressivenss of the theory of handshake processes. If the definition of receptiveness followed [Jos92], one would have means to express the requirement that only a subset of inputs is allowed to arrive at a certain time.

The choice operator is probably introduced in order to provide semantics for the `if` construct in Tangram. Unfortunately, [vB92] provides the semantics for a typeless subset of Tangram, where the `if` construct is not present.

6 Tangram

Tangram is the VLSI programming language based on CSP. A Tangram program consist of *commands* that can be divided into primitive commands, composite commands, and guarded commands. An execution of a command may result in a number of interactions with the environment through external ports, or by reading from or writing into command's external variables. A Tangram program is a command without external variables, prefixed by an explicit definition of its external ports.

Not all compositions of commands are valid. In a sequential composition, for example, the two constituent commands must agree in the input/output direction of their common ports. This means that in order to precisely define Tangram's constructs, we have to precisely define the conditions for *alphabet structures* of construct's constituents.

6.1 Alphabet structures

All ports and variables in Tangram are typed, with *Val* denoting the set of all possible values in Tangram: integers, Boolean values `true` and `false`, and the special *null* value $\tilde{}$. A *type* is a finite subset of *Val*; the set of all types, i.e., the power set of *Val* is denoted by $\mathcal{P}\cdot \mathit{Val}$.

The type information of ports and variables of a Tangram command is recorded in *alphabet structure*, defined as:

Definition 6.1 (Alphabet structure) *An alphabet structure A is a 5-tuple*

$$\langle \mathbf{p}?A, \mathbf{p}!A, \mathbf{v}?A, \mathbf{v}!A, \tau_A \rangle,$$

where

1. $\mathbf{p}?A$ *is the set of* input ports, *and* $\mathbf{p}!A$ *is the set of* output ports. $\mathbf{p}?A$ *and* $\mathbf{p}!A$ *must be disjoint; their union is denoted by* $\mathbf{p}A$.
2. $\mathbf{v}?A$ *is the set of* read ports, *and* $\mathbf{v}!A$ *is the set of* write ports. $\mathbf{v}?A$ *and* $\mathbf{v}!A$ *need not be disjoint, because a process can read from and write into the same variable. The union of the two sets is denoted by* $\mathbf{v}A$.
3. $\mathbf{p}A$ *and* $\mathbf{v}A$ *are disjoint. Their union, i.e., the set of all ports is denoted by* $\mathbf{c}A$.
4. τ_A *is the type function of alphabet structure A. It assigns a type to each port in* $\mathbf{c}A$*:*

$$\tau_A : \mathbf{c}A \to \mathcal{P}\cdot\mathrm{Val}$$

The empty alphabet structure $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ can be abbreviated to $\emptyset$.

Note that communication with variables is distinguished from communication among "ordinary" processes. The reason for this are the special properties of variables. We wonder whether this distinction has to be built so deeply into the formalism. In an alternative approach, variables could be seen as sequential processes, so the specification of their behavior would contain all necessary restrictions that should apply to a variable.

Since a port can be typically used either for communication (send/receive/synchronize) with other processes, or for communication with a variable (read/write), we can define a number of canonical alphabet structures, called *port definitions*:

1. $a\tilde{\ } = \langle \{a\}, \emptyset, \emptyset, \emptyset, \{(a, \tilde{\ })\} \rangle$ — *synchronization* port; no data is transferred, used for synchronization only;
2. $a?T = \langle \{a\}, \emptyset, \emptyset, \emptyset, \{(a, T)\} \rangle$ — an input port of type T; a value of type T can be received through such a port;
3. $a!T = \langle \emptyset, \{a\}, \emptyset, \emptyset, \{(a, T)\} \rangle$ — an output port of type T; a value of type T can be sent through such a port;

4. $x : T = \langle \emptyset, \emptyset, \{x\}, \{x\}, \{(x, T)\} \rangle$ — a variable of type T; a value of type T can be read from or written into such a variable.

Note that it is easy to find equivalent port structures for all port definitions. The only decision we have to make is whether to make a port passive or active. This decision can significantly influence the cost of the resulting circuits, therefore we discuss it later, together with other translation issues.

There are a number of relations and operations on alphabet structures we use in the remainder of this section. Most of these notions have their equivalents among the notions concerning port structures, they even enjoy the same names. In the following definitions, A and B are alphabet structures:

1. A and B are *type compatible* if the names they have in common are either ports or variables in *both* alphabet structures, and if these common names are of the same type.

2. A and B are *conformant*, denoted by $A \triangle\!\triangle B$, if they are type compatible and their common input and output ports agree in direction:

$$\begin{array}{rcl} A \triangle\!\triangle B & \equiv & A \text{ and } B \text{ are type compatible} \\ & \wedge & (\mathbf{p}?A \cap \mathbf{p}!B = \emptyset) \wedge (\mathbf{p}!A \cap \mathbf{p}?B = \emptyset) \end{array}$$

   The conformance of alphabet structures has a similar role as the conformance of port structures. For example, we require that operands of the Tangram's sequential composition have conformant alphabet structures.

3. The *conformant union* of two conformant alphabet structures A and B is denoted by $A \cup_{\triangle\!\triangle} B$ and is defined as the componentwise union of A and B.

4. The *conformant difference* of two conformant alphabet structures A and B is denoted by $A \setminus_{\triangle\!\triangle} B$ and is defined as the componentwise set difference of A and B.

5. A and B are connectable, denoted by $A \bowtie B$, if they are type compatible, have no common output ports, and no variable with write access occurs in both structures:

$$\begin{array}{rcl} A \bowtie B & \equiv & A \text{ and } B \text{ are type compatible} \\ & \wedge & (\mathbf{p}!A \cap \mathbf{p}!B = \emptyset) \wedge (\mathbf{v}A \cap \mathbf{v}!B = \emptyset) \wedge (\mathbf{v}B \cap \mathbf{v}!A = \emptyset) \end{array}$$

   The connectability of alphabet structures has a similar role as the connectability of port structures. For example, we require that operands of Tangram's parallel composition have connectable alphabet structures.

6. The *connectable union* of two connectable alphabet structures A and B is denoted by $A \cup_{\bowtie} B$ and is defined as the componentwise union of A and B, *except* for $\mathbf{p}?(A \cup_{\bowtie} B)$, which is defined as

$$\mathbf{p}?(A \cup_{\bowtie} B) = (\mathbf{p}?A \cup \mathbf{p}?B) \setminus (\mathbf{p}!A \cup \mathbf{p}!B),$$

i.e., output ports of common channels are not concealed.

7. Let D denote a list of port definitions that define mutually conformant alphabet structures. Then the conformant union of these alphabet structures defines a new alphabet structure, denoted by $\mathbf{A}D$.

6.2 Language constructs

In this section we introduce Tangram's constructs and define their alphabet structures. Although the alphabet constructs are defined for general alphabet structures, [vB92] provides the formal semantics only for Core Tangram, a subset of the language, where the only data type used is $\{\tilde{\ }\}$, i.e., there is no data transfer. Because there is no formal semantics for general Tangram, we introduce the constructs rather informally.

6.2.1 Program

Let S be a Tangram command and D be a list of port definitions. Then, a Tangram program is denoted by $(D){\cdot}S$, its alphabet structure is $\mathbf{A}D$, and we require $\mathbf{A}S \subseteq \mathbf{A}D$ and $\mathbf{vA}S = \emptyset$. The rule for the alphabet structure states that S does not have any external variables and that all external ports of S must be defined in D. Command S defines the behavior of the program.

6.2.2 Primitive commands

The following table lists Tangram's primitive commands. We assume a is declared as a port of type U (declarations are discussed later), x is declared as a variable of type V, and E is an expression with alphabet structure $\mathbf{A}E$.

construct	alphabet structure	requirements
stop	$\emptyset$	
skip	$\emptyset$	
a	$a\tilde{}$	
$a?x$	$a?U \cup_{\bowtie} \langle\emptyset, \emptyset, \emptyset, \{x\}, \{(x, V)\}\rangle$	$a \neq x$
$a!E$	$a!U \cup_{\bowtie} \mathbf{A}E$	$a \notin \mathbf{v}?\mathbf{A}E$
$x := E$	$\langle\emptyset, \emptyset, \emptyset, \{x\}, \{(x, V)\}\rangle \cup_{\bowtie} \mathbf{A}E$	

The intuition behind the commands above is

- **stop** does not engage in any action; it corresponds to an unconditional deadlock. Note there is formally a difference between *command* **stop** and sequential handshake process **stop**. Though, in Core Tangram, the semantics of command **stop** is exactly sequential process **stop**.

- **skip** does not engage in any action, but it does terminate successfully.

- $a?x$ is an *input command*. Upon execution of $a?x$ a value is received through port a and stored in variable x. If the value is not in type V, the effect of $a?x$ is left unspecified. We comment that there is no requirement that types U and V must be related. If their intersection is empty, the result of the input command is never specified. We believe that an additional requirement, for example $U \subseteq V$, would be helpful.

- $a!E$ is an *output command*. First, expression E is evaluated. It the evaluation terminates and the result is in U, the result is send through port a. Otherwise, the effect of $a!E$ is left unspecified. Note that the evaluation of E may require additional handshakes, in particular, reads from its external variables. Therefore, if we want to assure that a is used consistently as an output port, it must not appear in the set of E's read ports.

- $x := E$ is an assignment command. If the evaluation of E terminates successfully and the result is in V, then the result is stored in variable x. Otherwise, the effect of $x := E$ is left unspecified.

6.2.3 Composite commands

The following table lists all composite commands except guarded commands. We assume R and S are commands, N a natural number, and D a list of port definitions.

construct	alphabet structure	requirements
$R \sqcap S$	$\mathbf{A}R \cup_{\triangle\!\triangle} \mathbf{A}S$	$\mathbf{A}R \triangle\!\triangle \mathbf{A}S$
$R; S$	$\mathbf{A}R \cup_{\triangle\!\triangle} \mathbf{A}S$	$\mathbf{A}R \triangle\!\triangle \mathbf{A}S$
$\#N[S]$	$\mathbf{A}S$	
$\#[S]$	$\mathbf{A}S$	
$R \Vert S$	$\mathbf{A}R \cup_{\bowtie} \mathbf{A}S$	$\mathbf{A}R \bowtie \mathbf{A}S$
$\vert[D \mid S]\vert$	$\mathbf{A}S \setminus_{\triangle\!\triangle} \mathbf{A}D$	$\mathbf{A}D \triangle\!\triangle \mathbf{A}S$
$(D)\cdot S$	$\mathbf{A}D \cup_{\triangle\!\triangle} \mathbf{A}S$	$\mathbf{A}D \triangle\!\triangle \mathbf{A}S$

We remark that the same operators that are used in Tangram constructs are also used in the calculus of sequential processes. The context should help us to recognize which operator is indeed in use.

Intuition behind the commands above is

- $R \sqcap S$ is the *nondeterministic composition* of R and S. The composition behaves either like R or like S, with the selection between them being nondeterministic. We require alphabet structures of R and S be conformant (type compatible, common ports agree on direction) — the selection of either behavior must not affect how a port is seen from the environment.

- $R; S$ is the *sequential composition*. It first behaves like R, and, upon successful termination of R, it behaves like R. If R does not terminate successfully, neither does $R; S$. For the same reasons as above, we require alphabet structures of R and S be conformant.

- $\#N[S]$ is the *N-fold repetition* of S. Command $\#0[S]$ behaves like **skip**, and, for $N > 0$, the behavior of $\#N[S]$ is that of S; $\#(N-1)[S]$.

- $\#[S]$ is the *infinite repetition* of S. It never terminates successfully.

- $R \| S$ is the *parallel composition*. The behavior of $R \| S$ must be be in accordance with the behaviors of both operands, which means that communication on common ports must occur simultaneously in R and S. Also note that R and S do not both have write access to common variables. We comment that, because common ports are not concealed for the environment, the parallel composition introduced in the calculus of sequential processes becomes inappropriate for the semantics of Tangram's parallel composition.

- $|[D \mid S]|$ is a *block command*. There are two roles of a port definition in D: (1) the type function of an alphabet structure in D applies also to ports in S, and (2) the names declared in D are hidden from the environment of the block. The behavior of $|[D \mid S]|$ is the same as the behavior of S, but all interactions on ports and variables declared in D are hidden from the environment.

- $(D){\cdot}S$ extends the alphabet structure of S by port definitions in D. The extension does not engage in any communication through newly added ports; otherwise, it behaves like S.

The priority of the binary operators in Tangram commands is, in descending order, ;, $\|$, and $\sqcap$. Unary operators have precedence over binary operators.

6.2.4 Guarded commands

The table below lists Tangram's selection and repetition constructs, that follow well known guarded commands. We assume B is a boolean expression, S a command, and H and G guarded-command sets.

construct	alphabet structure	requirements
if G **fi**	$\mathbf{A}G$	
do G **od**	$\mathbf{A}G$	
$B \rightarrow S$	$\mathbf{A}B \cup_{\triangle\!\triangle} \mathbf{A}S$	$\mathbf{A}B \triangle\!\triangle \mathbf{A}S$
$G [\!] H$	$\mathbf{A}G \cup_{\triangle\!\triangle} \mathbf{A}H$	$\mathbf{A}G \triangle\!\triangle \mathbf{A}H$

- **if** G **fi** is the *selection command*. The execution of a selection command with a guarded-command set G depends on the disjunction of the guards of G, denoted by $BB{\cdot}G$. If $BB{\cdot}G$ evaluates to `false`, the behavior of **if** G **fi** is left unspecified. Otherwise, the selection command behaves like one of the commands of the guarded-command set for which the guard evaluates to `true`.

- **do** G **od** is Tangram's *guarded repetition* command. As long as $BB \cdot G$ evaluates to `true`, one of the commands for which the guard evaluates to `true` is executed. When $BB \cdot G$ evaluates to `false`, **do** G **od** terminates successfully.

6.2.5 Expressions

The following table lists the four forms of the last syntactic category in Tangram, *expressions*. We assume E and F are expressions, x is a variable declared of type V, and C is a constant.

construct	alphabet structure	requirements
C	$\emptyset$	
x	$\langle \emptyset, \emptyset, \{x\}, \emptyset, \{(x, V)\} \rangle$	
$\Box E$	$\mathbf{A}E$	
$E \Box F$	$\mathbf{A}E \cup_{\triangle\triangle} \mathbf{A}F$	$\mathbf{A}E \triangle\triangle \mathbf{A}F$

- The value of C is simply the value of the constant; the evaluation of C always terminates successfully.
- x, a variable of type V is also an expression of type V. The value of expression x is the value stored in variable x; even initially, the value stored in x is guaranteed to be of type V. The evaluation of x always terminates successfully.
- If it terminates successfully, the value of $\Box E$ is obtained by applying unary operator $\Box$ to the value of expression E. The evaluation of $\Box E$ terminates successfully if the evaluation of E terminates successfully and the operator $\Box$ is defined for the value of E.
- $E \Box F$ is a generalization of $\Box E$ to binary operators.

6.3 Tangram semantics

A translation method, proposed in [vB92] maps Tangram programs to handshake circuits, In order to relate Tangram programs to handshake circuits, [vB92] provides a handshake-process semantics of Tangram programs. The semantics is provided for Core Tangram only. Core Tangram is a subset of Tangram where the set of all values *Val* is reduced to $\{\tilde{\ }\}$, and all constructs that require types richer than a singleton, in particular the selection, the repetition, guarded commands, and the expressions, are removed from the language. Another consequence of reducing *Val* is that with only one possible value the distinction between input and output disappears and the concept of storage becomes meaningless.

All alphabet structures in Core Tangram have the form

$$\langle \mathbf{p}?A, \emptyset, \emptyset, \emptyset, \{\tilde{\ }\} \rangle.$$

We remark that all ports do not have to be "reduced" to input ports. This decision was made because it triggers further simplifications: the notions of type compatibility, conformance, and connectability of alphabet structures becomes void.

A decision that has to be made is which Tangram ports to implement active and which passive. Note that read/write access to variables must be done through active ports, and that the choice operator in the calculus of sequential processes requires a passive port. Additional issues we have to keep in mind are connectability requirements in the case of parallel composition, and broadcast in Tangram that does not have a counterpart in the calculus of sequential processes.

There are basically two simple strategies to implement ports:

- *directed* mappings: implement inputs active and outputs passive, or vice versa;
- *uniform mappings*: implement all ports passive or implement all ports active.

Although it is believed that "inputs-active-outputs-passive" mapping produces cheaper circuits, [vB92] chooses the uniform-mappings strategy, because it results in a simpler translation strategy.

With all ports passive or all ports active, the connectability of sequential processes, required for parallel composition, is satisfied in trivial cases only (in Core Tangram, for example). Therefore, additional "glue" circuitry is needed to overcome the connectability problems. According to [vB92], choosing all ports passive results in expensive handshake circuits, because we have to pay the price for receptiveness. Furthermore, a glue component that synchronizes two passive handshakes cannot be realized, because there is no way to enforce synchronization between two passive two-phase handshakes. On the other hand, according to [vB92] implementing all ports active results in rather cheap circuits. Furthermore, a passive glue can be realized as a *passivator*, and if *JOIN* is used as a glue component, the common ports of parallel composition remain accessible for the environment, providing an equivalent to broadcast. For these reasons, all-ports-active strategy is used in Tangram's semantics. The price we have to pay is that the choice operator from the calculus of sequential handshake processes does not have its counterpart in Tangram.

In the definition of semantics, the direct approach is taken: every Tangram command is associated with a sequential handshake process. If data types are ignored, every Tangram command has a corresponding operation in the calculus of handshake processes presented in Section 5, therefore defining the semantics for Core Tangram is not very difficult.

The semantics of a Core Tangram program is formally defined as the *handshake expansion* of Tangram commands. For alphabet structure $A = \langle \mathbf{p}?A, \emptyset, \emptyset, \emptyset, \{\tilde{\ }\} \rangle$, its handshake expansion is denoted by $\mathcal{H}\cdot A$, and defined as

$$\mathcal{H}\cdot A = (\cup a:\ a \in \mathbf{p}?A : a^\bullet),$$

i.e., all ports are chosen to be active.

Below we list the handshake expansions for commands in Core Tangram. We assume that D is a list of port definitions, N is a natural number, and S and T are Core Tangram commands:

$$\begin{array}{lcllcl}
\mathcal{H}\cdot\mathbf{skip} & = & \mathbf{skip} & \mathcal{H}\cdot(S;T) & = & \mathcal{H}\cdot S;\mathcal{H}\cdot T \\
\mathcal{H}\cdot\mathbf{stop} & = & \mathbf{stop} & \mathcal{H}\cdot(\#N[S]) & = & \#N[\mathcal{H}\cdot S] \\
\mathcal{H}\cdot a & = & a^\bullet & \mathcal{H}\cdot(\#[S]) & = & \#[\mathcal{H}\cdot S] \\
\mathcal{H}\cdot((D)\cdot S) & = & (\mathcal{H}\cdot D)\cdot(\mathcal{H}\cdot S) & \mathcal{H}\cdot(S \parallel T) & = & \mathcal{H}\cdot S \overset{\bullet\bullet}{\parallel} \mathcal{H}\cdot T \\
\mathcal{H}\cdot(S \sqcap T) & = & \mathcal{H}\cdot S \sqcap \mathcal{H}\cdot T & \mathcal{H}\cdot|[D \mid S]| & = & |[\mathcal{H}\cdot D \mid \mathcal{H}\cdot S]|
\end{array}$$

Note that we used a new operator $\overset{\bullet\bullet}{\parallel}$ to denote the handshake expansion of the parallel composition. As we mentioned before, the parallel composition defined in the calculus of sequential handshake processes cannot be used for the semantics of Tangram's parallel composition, because it conceals common ports. For this reason, [vB92] redefines parallel composition of sequential processes with all ports active as follows:

Definition 6.2 (Parallel composition, all-ports-active) *Let S and R be connectable Core Tangram commands. Let $P = \mathcal{H}\cdot S$ and $Q = \mathcal{H}\cdot R$. Then $P \overset{\bullet\bullet}{\parallel} Q$ denotes the parallel composition of all-active processes and is defined as*

$$P \overset{\bullet\bullet}{\parallel} Q = \langle \mathbf{p}P \cup \mathbf{p}Q, \mathbf{t}P^{\mathbf{s}}\,\mathbf{w}\,\mathbf{t}Q \cup \mathbf{t}P\,\mathbf{w}\,\mathbf{t}Q^{\mathbf{s}}, \mathbf{u}P\,\mathbf{w}\,\mathbf{u}Q\rangle,$$

where $P^{\mathbf{s}}$ denotes the $\mathbf{s}$*-closure of P based on preorder* $\mathbf{s}$ *on handshake traces:*

$$s\,\mathbf{s}\,t \equiv (\exists u:\ u \in (\mathbf{o}P)^* :\ su = t),$$

i.e., s is a prefix of t that can be extended to t by output symbols only.

We can see that the definitions of $\parallel$ and $\overset{\bullet\bullet}{\parallel}$ differ in the definitions of their quiescent traces. The reasoning behind the definition of the set of quiescent traces of $P \overset{\bullet\bullet}{\parallel} Q$ is as follows. Let a be a *common* (active) port of P and Q and let trace t be a possible behavior of P and Q running in parallel. Furthermore, assume that a_0 (initiation of a handshake through a, sent by P and Q simultaneously) would bring P into a quiescent state, but Q is not ready to participate in a_0. In such situation, the parallel composition of P and Q can be quiescent after trace t, even if $t\lceil \mathbf{p}P \notin \mathbf{t}P$ or $t\lceil \mathbf{p}Q \notin \mathbf{t}Q$. Therefore, all traces that belong to the prefix closure of the weave of quiescent sets $\mathbf{t}P$ and $\mathbf{t}Q$ and can be extended to a quiescent set for either P or Q by output symbols only, are quiescent traces for the parallel composition of P and Q.

We comment that the reasoning above in a sense redefines the notion of a quiescent trace. In Section 3.1 we defined quiescent trace t as a trace after which the environment cannot obtain further output from process P. This may happen for two reasons: (1) the behavior of P does not allow extension of t by an output, or (2) P may decide not to extend t by an output. If t is quiescent because of the second reason, we did not define at which point P can decide not to extend t by an output. The definition of $\overset{\bullet\bullet}{\|}$, however, implies, that this decision can, or even should, be made in advance.

Finally, we remark that in defining $\mathbf{t}(P \overset{\bullet\bullet}{\|} Q)$ we do not have to worry about divergences, because the common ports are not concealed.

7 Translation

In this section we briefly discuss translation of Tangram programs into handshake circuits. The translation is defined as *compilation function* $\mathcal{C}$ that maps handshake expansions of Tangram constructs into handshake circuits:

$$\triangleright^* \mathcal{H}{\cdot}T = \| {\cdot} \mathcal{C}{\cdot}T,$$

where $\triangleright^* P$ is the sequential process $\#[\triangleright^\circ : P]$ and the port name $\triangleright$ is pronounced as "go". Since implementing the complete behavior of the program may be expensive, [vB92] suggests it is sufficient to realize

$$\triangleright^* \mathcal{H}{\cdot}T \sqsubseteq \| {\cdot} \mathcal{C}{\cdot}T,$$

where the behavior of the handshake circuit is a refinement of the handshake behavior of the Tangram program.

The translation of Tangram programs into handshake circuits is *syntax-directed*, i.e., we "recursively" translate each construct of the program. As an example, we discuss how the compilation function is defined for parallel composition of Tangram commands R and S. The schematics of the result of the compilation is shown in Figure 1. The double circled subcircuits $\mathbf{l}R$ and $\mathbf{r}S$ represent the compiled commands R and S, and the *JOIN* subcircuit is the "glue" circuitry that has to be added because all ports of $\mathbf{l}R$ and $\mathbf{r}S$ are active. The addition of the glue circuitry requires introduction of new names to denote the new connections. Because of this, operators $\mathbf{l}$ and $\mathbf{r}$ are used to systematically rename ports of

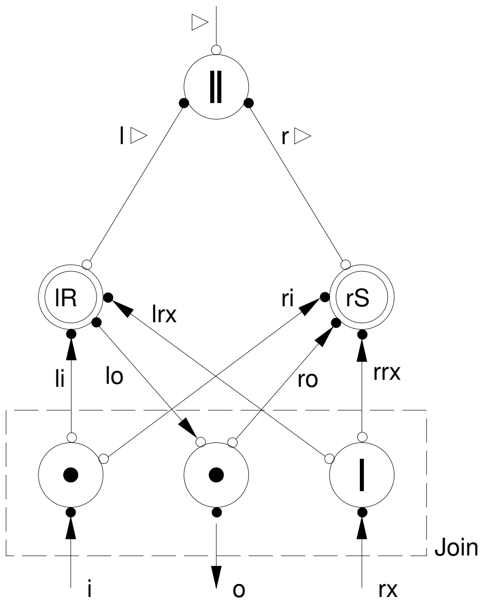

Figure 1: Translation of the parallel composition

both subcircuits and thus avoid naming problems. After receiving the go ($\triangleright$) signal, the *PAR* component (marked with $\|$) starts both subcircuits. The communication of $\mathbf{l}R$ and $\mathbf{r}S$ with the environment must be synchronized. Therefore, the *JOIN* components (marked with $\bullet$) are added to synchronize the common inputs and outputs of R and S, and the *MIX* component (marked by $|$) is added to synchronize reads from the variables shared by R and S.

Similarly to above, [vB92] defines the compilation function for every construct in Tangram, and then, for Core Tangram program T, proves that

$$\triangleright^* \mathcal{H}\cdot T = \| \cdot \mathcal{C}\cdot T,$$

There are a number of other issues concerning translation of handshake circuits into physical circuits: possible optimizations on the handshake-circuit level (peephole optimization, for example), replacing expensive receptive components by cheaper nonreceptive ones, choosing the signaling scheme for the implementations ([vB92] chooses 4-phase signaling, because simple implementations of basic components are known), data encoding scheme ([vB92] does not explicitly state which encoding is chosen), circuit initialization, and testing. The discussion of these issues, all of them are research topics, is beyond the scope of this report.

8 Conclusions

[vB92] proposes a theory of handshake circuits that forms a foundation for a VLSI programming language Tangram. The major decision made in [vB92] is to consider handshake behaviors only. Since the ultimate goal of the formalism is to provide a method for designing asynchronous circuits, we believe this early restriction does not contribute to the flexibility of the formalism.

Another important feature of the formalism is that it does not concentrate on the specification of the behavior, but is mostly concerned with design issues. In particular, there is no convenient method to specify a parallel behavior of *one* process, since the parallel composition is, as the name suggests, intended to describe parallel connections of several processes. However, the formalism does not impose any conditions that should hold if the processes are connected in parallel. Rather, the formalism follows the line of least effort and requires processes be receptive. Furthermore, the parallel composition of sequential handshake processes, as defined originally, turns out to be insufficient for capturing Tangram's parallel composition. For this reason, [vB92] provides an additional definition of parallel composition that applies only to handshake processes with all ports active.

The decision to map Tangram programs to handshake processes with all ports active was made because of a simple translation method, and because of a rather cheap implementation. A consequence of this decision is, however, that additional, though simple, glue circuitry is needed in order to properly connect the handshake processes. Another consequence is that the choice construct from the calculus of sequential handshake processes does not have its counterpart in Tangram.

The theory of sequential handshake processes is intended to provide a semantics for Tangram. Unfortunately, [vB92] provides a semantics for a very limited subset of Tangram only; the large part of the theory developed is actually never used; an example is all theory that deals with data types. The translation from Tangram to handshake circuits is syntax-directed. This means that each Tangram command is translated individually and the glue circuitry is added to ensure proper connections between the processes. Furthermore, the order of constructs, for example, the order of guards in a guarded command, determines the result of the compilation. Such a translation strategy makes it possible for a designer to influence a design directly by rewriting a program. This strategy was also present in early programming languages (with C being the most visible example), because there was little knowledge of program analysis and optimization. As the knowledge level grows, the syntax-directed translation strategy is becoming less important. We expect a similar course of events in the area of silicon compilation.

References

[Ebe91] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[Jos92] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992.

[Pee90] Ad Peeters. Decomposition of delay-insensitive circuits. Computing Science Notes 90/04, Dept. of Math. and C.S., Eindhoven Univ. of Technology, April 1990.

[Udd86] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[vB92] Kees van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.

[vdS85] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.