Formalization of a Software Architecture for Embedded Systems: 
a Process Algebra for SPLICE*

Paul Dechering1  Rix Groenboom2
1Applied Systems Research Hollandse Signaalapparaten B.V. Hengelo, the Netherlands {dechering, edejong}@signaal.nl

Edwin de Jong1  Jan Tijmen Udding2
2Department of Computing Science University of Groningen Groningen, the Netherlands {rix, jtu}@cs.rug.nl

Abstract

SPLICE is an industrially developed and applied software architecture for large-scale distributed embedded systems. The key feature of SPLICE is asynchronous communication between processes. The characteristics of systems built with SPLICE include dynamic data distribution, fault-tolerance, and real-time performance.

The objective of the work presented in this paper was to use a formal model to reason about the behaviour of SPLICE systems and to support the design process in system decomposition and refinement. We formalize SPLICE using a process algebra called SPLICE Process Algebra. This process algebra allows us to derive properties of the SPLICE architecture. From these properties a set of guidelines can be produced that aid in the development of SPLICE applications.

1. Introduction

As the software component of control systems becomes larger and more complex, the choice of software architecture becomes more crucial within the overall development process. This is particularly true for the class of distributed control systems, including traffic management systems, process control systems, and command-and-control systems, which place stringent requirements on real-time behaviour, fault-tolerance, and safety. The tasks performed by these systems typically include: (1) processing of measurements obtained from the environment through sensing devices, (2) determination of model parameters describing the environment, (3) tracking discrepancies between desired state and perceived state, (4) taking corrective action, and (5) informing the operator or team of operators about the current and predicted state of affairs. All tasks are closely related and intertwined, and particularly in large-scale systems, there is a huge number of model parameters which are often intricately linked through numerous dependencies.

It is therefore important to have a concise description of the software architecture underlying these systems. An architecture defines the overall structure of the system in terms of components and an organizational principle that defines possible interconnections between these components. In addition, an architecture prescribes a set of rules and constraints governing the behaviour of components and their interactions [4]. Traditionally, software architectures have been primarily concerned with structural organization and static interfaces [17, 24]. With the growing interest in coordination models [1, 9, 12], more emphasis is placed on the organizational aspects of behaviour and interaction. Coordination is the process of building programs by gluing together active pieces [7].

At the company “Hollandse Signaalapparaten” a software architecture for distributed control systems has been developed [3], and has been applied in the construction of commercially available traffic management and command-and-control systems. The architecture, named SPLICE, employs a data-oriented coordination model; in this respect SPLICE bears close resemblance to coordination models and languages like Linda [6], Gamma [2], μLog [18], Imp-Unity [13], and Swarm [23] (the last two being based on the UNITY language [8]). The semantics and implementation of the coordination primitives of SPLICE, however, are strongly tailored towards the specific requirements of distributed control systems.

Building large-scale systems (consisting of several interacting processes) is still a difficult task, and it is even more difficult to modify these systems should their purpose...
be changed, or the description of the environment be re-
refined. In this paper we present a formal framework that
aids and supports the task of designing a system according
to the SPLICE architecture. The proposed framework is a
process algebra for SPLICE processes, called SPA (SPLICE
Process Algebra), which is based on Data Flow Networks
(DFN [16, 19]). The behaviour and interaction of processes
in terms of communication actions can be computed. SPA
gives the formal context in which one can reason about
the composition of these processes. This formal model is
based on observational semantics; we are interested in the
effects of processes as can be monitored by an observer.
Although operational semantics of SPLICE have been con-
structed [5], we are interested in a set of rewrite rules for
program transformations. SPA defines rewrite rules for pro-
cess constructs, in terms of algebraic laws, that do not alter
the process’s semantics. As such, one can deduce properties
of process constructs, which, in turn, give us design rules
and guidelines that streamline the task of decomposing and
synthesizing multiple processes (such that these perform the
required computation). Design rules put constraints on the
possible set of process construct compositions, such that
the system as a whole exhibits certain desirable properties.
Typical system properties in the domain of distributed em-
bedded systems involve transparent and dynamic data dis-
tribution, fault-tolerance (process and data replication), and
real-time performance. Guidelines provide ways to develop
a system that satisfies the design rules. The derived design
rules and guidelines in our formal model SPA can be used
to improve the development of SPLICE-based systems.

This paper shows the usage of an extension to DFN (and
its existing theory) to describe the software architecture
SPLICE and the embedded systems constructed in accor-
dance to SPLICE. In Section 2 we give a more detailed in-
troduction to SPLICE; we present the semantics of its com-
munication primitives and distinguish volatile and persis-
tent data. Section 3 outlines the construction of SPA, and
relates it with SPLICE. SPA is defined formally in Section
4 by enumerating (a subset of) its laws. We apply these
laws in Section 5 to short program constructs, and explore
the subjects functional dependency, process replication, and
process creation. Finally, we discuss the relation between
SPLICE and SPA in Section 6 and present ideas for future
work.

2. SPLICE

The software architecture SPLICE defines three types of
components: application processes, agents, and a commu-
nication network. Each process interacts with exactly one
agent. An agent embodies processing facilities for han-
dling all communication needs of the application process
it serves. For each process the accompanying agent keeps
a local data space. All agents are identical and need no
prior information about either the application processes or
their communication requirements. Communication be-
tween agents is established by an asynchronous message
passing mechanism. Messages between agents are handled
by the communication network that interconnects them.
An application process interacts with its assigned agent by
means of coordination primitives.

SPLICE extends an existing programming language, re-
ferred to as the host language, with coordination primitives
for the creation of processes and for the interaction with
the distributed data space; i.e. the collection of local data
spaces. Each data element in the data space has a sort (a
label) attached. Different data elements can have the same
sort; sorts are typically used to identify groups of data. The
coordination primitives can be described informally as fol-

- $\text{write}(a,v)$: Insert a data element $v$ of sort $a$ into
  the distributed data space.
- $\text{read}(a,q)$: (non-destructive read) Read an element
  of sort $a$ from the local data space that satisfies query
  $q$. In case a requested instance is not available the pro-
  cess blocks until one becomes available. If, on the
  other hand, multiple instances satisfying $q$ are avail-
  able, one is selected nondeterministically.
- $\text{get}(a,q)$: Behaves like $\text{read}(a,q)$ except that
  the returned element is removed from the local data
  space.
- $\text{new}(P)$: Start a new process $P$ in parallel with the
  processes already active.

The protocol used by the agents to manage communica-
tion is based on a subscription paradigm (hence the acronym
SPLICE which stands for Subscription Paradigm for the

![Figure 1. Different components in SPLICE: a num-
ber of processes, each with an agent and a local data
space, connected via a network layer](image)
Logical Interconnection of Concurrent Engines). A process is a publisher and/or subscriber of one or more sorts. When a publisher writes a data element into its local data space, its agent will forward a copy of the element to all subscribers of the corresponding sort. If a new subscriber is introduced, every publisher will forward a copy of all its published data to this new subscriber.

SPLICE distinguishes volatile and persistent data. When volatile data is published, all currently active processes can receive this data; it is not guaranteed that processes created after this data has been published will receive it. Persistent data remains available, such that newly created processes will receive a copy of previously published persistent data.

3. SPLICE Process Algebra

Process algebras are a well-known formal method to reason about distributed systems [15, 22]. Processes in SPLICE are sequential processes, simply because the host languages of SPLICE (e.g. C and Ada) are sequential, and interact with each other by means of read and write actions. It seems natural to study SPLICE in the context of Communicating Sequential Processes (CSP [15, 16]). Since in CSP-like algebras the semantics of processes is given in terms of traces, the behaviour of a process is represented by sequences of its communication actions. Unlike CSP, communication actions in SPLICE are not synchronized. Therefore, we use a process algebra for Data-Flow Networks (DFN), an asynchronous variant of CSP, as the basis for SPA. A general explanation of DFN in the context of CSP can be found in [16]. Output of SPLICE processes is never blocked: the environment is always ready to accept data. Processes with this property are called receptive and a general theory is given in [19]). This theory forms the basis of our analysis. DFN is a sub-theory of receptiveness process theory and the development of SPA is inspired by [21] where another sub-theory of based on delay-insensitive communications.

In SPA, SPLICE processes are modeled at a high level of abstraction, while the communication between processes is modeled in great detail. Processes in SPA are typically denoted by \( P, Q \), and \( R \); different data sorts are typically denoted by \( a, b, \) and \( c \). Processes have an input and an output alphabet, denoted with \( \text{i}(P) \) and \( \text{o}(P) \) respectively \( (\text{i}(P) = \text{i}(P) \cup \text{o}(P) \text{ is the total alphabet of } P) \). These alphabets define the data sorts that can be read or written by \( P \) according to the subscription paradigm of SPLICE. We abstract from the agents; their role as manager of communication is integrated in the semantics of the operators in SPA. The local data space is modeled with an after operator. Communication between different processes is expressed by parallel composition. The principle idea is that data \( a \_ v \) (sort \( a \) and value \( v \)) written by \( P \) is read by \( Q \) given that \( a \in \text{o}(P) \) and \( a \in \text{i}(Q) \) (i.e. \( P \) is a publisher of data sort \( a \) and \( Q \) is subscribed to data sort \( a \)). We do not make assertions about when \( Q \) reads \( a \_ v \), but we guarantee that \( Q \) eventually will receive a copy of \( a \_ v \) in its local data space. In general, there is a single publisher of data sort \( a \), while there are multiple subscribers of data sort \( a \).

In SPA we write the SPLICE coordination primitives as:

- \( a \_ v ; P \) \( \): Write an element of sort \( a \) and value \( v \) and then behave like \( P \). Writing is never blocked by the environment. We say that \( P \) is prefixed by the output action \( a \_ v \).
- \( a \_ x ; P \) \( \): Read an element \( x \) of sort \( a \) and then behave like \( P \) (nondestructive read). The read operation is blocking when no \( x \) is available. We say \( P \) is prefixed with the input operation \( a \_ x \).
- \( a \_ x ; P \) \( \): Get (read and remove) an element \( x \) of sort \( a \) and then behave like \( P \) (destructive read). We demand that \( a \in \text{i}(P) \). The get operation is blocking when no element of sort \( a \) is available.
- \( P/\_ a \_ v ; P \) \( \): after receipt of data \( a \_ v \) (\( a \in \text{i}(P) \)). In terms of SPLICE, \( a \_ v \) has been stored in the local data space of \( P \).
- \( P \| Q \) Parallel composition of \( P \) and \( Q \).

In SPA we have chosen to replace the ‘dynamic’ process creation operator in SPLICE, \( \text{new}(P) \), with the ‘static’ parallel composition. Parallel composition provides us with a set of algebraic laws that enable process (de-)composition. Process creation can be expressed in terms of the basic operators as we discuss in Section 5.4.

The imperative host languages of SPLICE are replaced by an abstract programming language based on guarded commands [11]. This language contains operators that allow us to express nondeterministic choice, guarded-choice, and recursion.

Using the SPA operators presented above we can express several aspects of SPLICE, such as asynchronously writing, (non)destructively reading, dynamically creating processes, and the replication of processes. Since SPA is defined in terms of DFN, we can use the existing semantics for DFN to validate the laws of our algebra SPA. These laws, presented in the next Sections, provide program transformations for SPLICE programs.

Just as in DFN, we have to impose some syntactic restrictions on processes in SPA. It is required that the input and the output alphabets are finite, that the output alphabet is non-empty, and that the input and output alphabets are disjoint. For an arbitrary process \( P \) we have:

\[ \text{o}(P) \neq \emptyset \quad \text{i}(P) \cap \text{o}(P) = \emptyset \]
The reasons for these restrictions are mainly technical (cf. [20]) and they do not severely limit the expressiveness of SPA. The motivation for these restriction is that SPLICE processes refer to a finite number of data sorts; processes that do not produce output are not interesting for a calculus, and we do not model processes that communicate with themselves.

A process definition has no free occurrences of process variables or data variables. Substitution of a (data) variable \( x \) by value \( v \) in \( P \) is denoted as \( P[v/x] \).

4. Laws of SPA

We present a subset of the algebraic laws for the SPA operators. The laws are grouped by the expressed semantic property, and/or by type of syntactic construction. A full description of the calculus is given in [10].

Basic laws. The basic laws relate the read and get operations with the after operation. The first law expresses that reading a value \( x \) of sort \( a \) binds \( x \) to the value \( v \) provided that \( a.v \) is in the local data space:

**Law 1.** \((a?x; P)/a.v = P[v/x]/a.v\)

The get operation will read and remove an element from the local data space:

**Law 2.** \((a?\overline{x}; P)/a.v = P[v/x]\)

The read operation can be expressed in terms of the get and the after operation. Read \( a.x \) and subsequently behave like \( P \) corresponds to get \( a.x \) and then behave as \( P \) supplied with \( a.x \):

**Law 3.** \(a?x; P = a?\overline{x}; (P/a.x)\)

Law 3 relates the read and get primitive. We can use this relation to deduce the laws for the read operation from the laws for the get operation. For example, we can derive law 1 using laws 2 and 3:

\[
\begin{align*}
(a?x; P)/a.v &= \{ \text{law 3: definition } a?x \} \\
(a?\overline{x}; P/a.x)/a.v &= \{ \text{law 2: instantiating } x \text{ by } v \} \\
(P/a.x)[v/x] &= \{ \text{definition of substitution} \} \\
(P[v/x])/a.v
\end{align*}
\]

Using law 3 we can derive many laws for the \( a?x \) operation from the laws for \( a?\overline{x} \). Some of the laws for \( a?x \) are presented in this paper, the others can be found in [10].

Refinement and choice. Processes are ordered using the refinement ordering of Csp [15]: \( P \subseteq Q \) expresses that \( Q \) is more deterministic than \( P \). The refinement ordering and nondeterministic choice, denoted \( \sqcap \), are tightly coupled. A process \((P \sqcap Q)\) may be implemented either as \( P \) or as \( Q \), therefore:

**Law 4.** \( P \sqcap Q = P \iff P \subseteq Q \)

Just like in DFN, the refinement is an ordering of processes [20]. The chaos process (\( \bot \)) is the most nondeterministic process of all. Every process refines \( \bot \):

**Law 5.** \( \bot \subseteq P \)

Nondeterministic choice is idempotent, commutative, and associative. Prefixing can be distributed over \( \sqcap \). Thus for output prefixing we have:

**Law 6.** \( c!v ; (P \sqcap Q) = (c!v ; P) \sqcap (c!v ; Q) \)

Asynchronous communication. The coordination primitives of SPLICE are asynchronous. Hence, the environment cannot distinguish processes in which successive read or successive write actions are commuted.

A process that is waiting for a datum of sort \( a \) and then subsequently for a datum of sort \( b \) before it can continue, is infact waiting for both data, their order of arrival is not important:

**Law 7.** \( a?\overline{x}; b?\overline{y}; P = b?\overline{y}; a?\overline{x}; P \)

**Law 8.** \( a?x; b?y; P = b?y; a?x; P \)

**Law 9.** \( a?x; b?\overline{y}; P = b?\overline{y}; a?x; P \)

By convention, \( a \neq b \). Similarly, the environment cannot distinguish the order in which data is produced:

**Law 10.** \( c!v ; d!w ; P = d!w ; c!v ; P \)

Actually, laws 8 and 9 can be proven with law 7 using law 3 (see example 1 below).

Laws for guarded choice. Guarded choice is a generalization of prefixing and nondeterministic choice. The choice is made between a number of alternatives of the form \( \text{skip} \rightarrow P, a?x \rightarrow P, \) and \( a?\overline{x} \rightarrow P \). The guarded choice is written as \( [g_1 \rightarrow P_1 \sqcap \ldots \sqcap g_n \rightarrow P_n] \) where each alternative consists of a process \( P_i \) guarded by \( g_i \) which are separated by \( \sqcap \). The selection of a \( \text{skip} \)-guard is an internal (for the environment unobservable) action of a process. The relation between a guard \( a?x \) and \( a?\overline{x} \) is given by the next law (cf. law 3):

**Law 11.** \([a?x \rightarrow P \sqcap S] = [a?\overline{x} \rightarrow (P/a.x) \sqcap S]\)
In the remainder of this paper, most of the laws regarding guarded choice are presented in terms of get operations. Using law 11, however, the laws can be transcribed in terms of read operations.

The behaviour of a guarded choice is given by the following laws. A choice with only one alternative is not really a choice, so:

**Law 12.** \( [a ? x \to P] = a ? x \parallel P \)

**Law 13.** \( [\text{skip} \to P] = P \)

The nondeterministic choice can be seen as a shorthand for two skip-guarded processes:

**Law 14.** \( [\text{skip} \to P \land \text{skip} \to Q] = P \cap Q \)

If two alternatives are guarded on \( a \), then either one may be chosen after data of sort \( a \) has been supplied:

**Law 15.** \( [a ? x \to P \land a ? y \to Q \land S] = [a ? x \to P \land S] \cap [a ? y \to Q \land S] \)

**Law 16.** \( [a ? x \to P \land a ? y \to Q \land S] = [a ? x \to (P[z/x] \cap Q[z/y]) \land S] \)

\( z \) not free in \( P \) and \( Q \)

Note that, since we bind variable \( z \) with the guard \( a ? x \), we demand that \( z \) does not occur free in \( P \) or \( Q \).

The internal selection of a skip-guards gives rise to a number of laws. For example in the next law where a nondeterministic choice arises after data of sort \( a \) has been supplied, because the data may arrive before the selection of the skip-guarded alternative:

**Law 17.** \( [a ? x \to P \land \text{skip} \to [a ? y \to Q \land S_0 \land S_1]] = [\text{skip} \to [a ? x \to (P[z/x] \cap Q[z/y]) \land S_0 \land S_1]] \)

\( z \) not free in \( P \) and \( Q \)

More laws for skip-guards can be found in [10].

**Laws for local data spaces.** The process \( P/a . w \) (with \( a \in \bar{i}(P) \)) behaves like \( P \) after the environment has provided a value \( v \) of sort \( a \). It is convenient to adopt the convention that:

\[ P/a . w = P \quad \text{if} \quad a \not\in \bar{i}(P) \]

This equivalence reflects the fact that in SPLICE the agent of a process that is not subscribed to sort \( a \) will ignore all data of sort \( a \).

Since communication is asynchronous, the order in which data is sent to a process does not determine the order in which data is read by that process, so:

**Law 18.** \( P/a . w /b . w = P/b . w /a . w \)

An output-prefix process will produce the output and makes no changes to its local data space:

**Law 19.** \( (c w ; P)/a . w = c w ; (P/a . w) \)

Generalizing laws 1, 2, and 3, \( /a . w \) distributes over the alternatives in a guarded choice, except for alternatives guarded on \( a \); those become skip-guarded. If the guard was a get operation, then \( a . w \) is eliminated (e.g. removed from the local data space), otherwise \( a . w \) is available for future read/get-actions of \( P \):

**Law 20.** \( [S]/a . w = [S] \)

Here \( S \) is formed by substituting for each \( G \in S \) the guarded process \( G/a . w \) which rewrites to:

- \( (a ? x \to P)/a . w \Rightarrow \text{skip} \to P[v/x] \)
- \( (a ? x \to P)/a . w \Rightarrow \text{skip} \to P[v/x]/a \)
- \( (b ? x \to P)/a . w \Rightarrow b ? x \to (P/a . w) \)
- \( (b ? x \to P)/a . w \Rightarrow b ? x \to (P/a . w) \)
- \( (\text{skip} \to P)/a . w \Rightarrow \text{skip} \to (P/a . w) \)

**Example 1.** We can use the laws for the after operation to prove some laws for \( a ? x \). Below, we show how laws 7, 18, and 20 are used to prove law 8:

\[ a ? x ; b ? y ; P = \{ \text{law 7: commutativity of } a ? x \} \]

\[ a ? x ; b ? y ; (P/a . x) = \{ \text{law 7: commutativity of } a ? x \} \]

\[ b ? y ; a ? x ; (P/a . x) /b . y = \{ \text{law 3: definition of } b ? y \} \]

This example shows the derivation a law for \( a ? x \) using a similar law for \( a ? x \). In this case commutativity of \( a ? x \) (law 8: \( a ? x ; b ? y ; P = b ? y ; a ? x ; P \)) based on commutativity of \( a ? x \) (law 7: \( a ? x ; b ? y ; (P/a . x) /b . y = b ? y ; a ? x ; (P/a . x) /b . y \)).
Laws for parallel composition. Parallel composition allows us to decompose a system into a number of components (or SPICE processes) operating in parallel. Parallel composition of processes $P$ and $Q$ is denoted as $P \parallel Q$. $P \parallel Q$ is only defined if:

$$o(P) \cap o(Q) = \emptyset$$

This restriction reflects that we allow only one publisher of each sort. We impose this restriction to ensure that $|$ is compositional. This property makes it possible to calculate the parallel composition between two processes independent from the behavior of their environment. In other words, the composition is not “open” for other processes to interfere. Suppose we have $P \parallel Q$ with $a \in i(P)$. If $a \in o(Q)$, then data of sort $a$ must be provided by $Q$, if $a \notin o(Q)$ this data of sort $a$ must be provided by the environment. These requirements allow us to determine that a composition of two processes makes progress and to detect deadlock (see example 2 below).

All the operators we have presented so far do not affect the input and output alphabets of their operands. For a parallel composition $P \parallel Q$, however, we will have different alphabets for the two operands $P$ and $Q$. The alphabets of $P \parallel Q$ are defined as:

$$i(P \parallel Q) = (i(P) \setminus o(Q)) \cup (i(Q) \setminus o(P))$$

$$o(P \parallel Q) = o(P) \cup o(Q)$$

Thus $P \parallel Q$ satisfies the alphabet restrictions for processes:

$$i(P \parallel Q) \cap o(P \parallel Q) = \emptyset$$

$$o(P \parallel Q) \cap o(P \parallel Q) = \emptyset$$

These restrictions on alphabets ensure that parallel composition is commutative and associative.

The three laws for $P \parallel Q$ are as follows. Chaos ($\perp$) is the zero element of the parallel composition:

**Law 21.** $P \parallel \perp = \perp$

Stated differently, if one of the processes of the parallel composition is chaos then the overall process is chaos. If one of the processes of a parallel composition is prepared to produce an output, it may do so immediately. The data will be consumed by the other process if the sort is in its input alphabet and it will be ignored otherwise. Here we implicitly use our convention that $P/\alpha v = P$ if $\alpha \notin i(P)$:

**Law 22.** $(c \vec{v} ; P) \parallel Q = c \vec{v} ; (P \parallel (Q/c \vec{v}))$

The last law deals with the parallel composition of two guarded choices, and specifies the alternatives in the resulting guarded choice:

**Law 23.** $[S_0] \parallel [S_1] = [S]$

Where the alternatives of $S$ are obtained from $S_0$ in the following way. For each alternative $g \rightarrow P$ in $S_0$ we substitute $g' \rightarrow P'$ whenever $g \rightarrow P \Rightarrow g' \rightarrow P'$, specified as follows:

- $a ? x \rightarrow P \Rightarrow a ? x \rightarrow (P/\alpha x \parallel [S_1]/\alpha x)$ if $\alpha \notin o([S_1])$
- $a ? \vec{x} \rightarrow P \Rightarrow a ? \vec{x} \rightarrow (P \parallel [S_1]/\alpha x)$ if $\alpha \notin o([S_1])$
- skip $\rightarrow P \Rightarrow$ skip $\rightarrow (P \parallel [S_1])$

The alternatives of $S_1$ are added to $S$ in a similar way.

In law 23 the alternatives in $S_1$ guarded on $a ? x$ and $a ? \vec{x}$ with $a \in o([S_1/f])$ do not contribute to the resulting composition. This is illustrated in the following example.

**Example 2.** With law 23 we can prove that the parallel composition of two processes that are waiting on each other’s data, results in a deadlock. Let:

$$P = a ? \vec{x} ; c \vec{v} ; P'$$
$$Q = c ? \vec{y} ; a \vec{v} ; Q'$$

We derive:

$$P \parallel Q = \{ \text{definition of } P \text{ and } Q \}$$

$$a ? \vec{x} ; c \vec{v} ; P' \parallel c ? \vec{y} ; a \vec{v} ; Q'$$

$$= \{ \text{law 12: prefixing as guarded choice} \}$$

$$[a ? \vec{x} \rightarrow (c \vec{v} ; P')] \parallel [c ? \vec{y} \rightarrow (a \vec{v} ; Q')]$$

$$= \{ \text{law 23: with } a \in o(Q) \land c \in o(P) \}$$

Which indicates that no events can occur, and thus this parallel composition results in a deadlock.

Renaming. The parallel composition of SPA demands a unique publisher for each data sort. However, in SPICE multiple publishers of the same sort might exist (in parallel), e.g. different sensors produce instances of the sort measurement. We introduce a renaming operator to map outputs of the same sort to a new sort name.

To rename the sorts $A \subseteq o(P)$ of a process $P$ to a fresh sort $b$ we introduce the following operator:

$$\langle P \rangle^A_b$$

where $A \subseteq o(P) \land b \notin a(P)$

We demand $b \notin a(P)$ for the following reason: With $b \in i(P)$ we would violate $i(P) \cap o(P) = \emptyset$; while with $b \in o(P)$ we would change the communication behaviour of $P$. The output alphabet for a renamed process is defined as $o(\langle P \rangle^A_b) = o(P) \setminus A \cup \{b\}$. If $A$ is a singleton set, we write $\langle P \rangle^A_b$ for $\langle P \rangle^A_b$. We have:

**Law 24.** $\langle \langle P \rangle^A_b \rangle^A_b = P$

Furthermore, we have the following laws:
Law 25. \([\langle e \nu \rangle; P_b^A] = \beta \nu; \langle P_b^A \rangle \quad \text{if } c \in A\)
\([\langle e \nu \rangle; P_b^A] = e \nu; \langle P_b^A \rangle \quad \text{if } c \notin A\)

Law 26. \([\alpha ? x; P_b] = \alpha ? x; \langle P_b \rangle \]

Law 27. \([\langle P / a \nu \rangle \rangle_b^A = ((P_b^A) / a \nu \]

The renaming of a guarded choice is a straightforward generalization:

Law 28. \([\langle S \rangle_b^A = \langle S' \]

Where each alternative \(g \rightarrow P\) in \(S\) has a corresponding alternative \(g \rightarrow \langle P \rangle_b^A\) in \(S'\).

Hiding. SPLICE distributes all the produced data to the (subscribed) processes connected to the communication network (as specified by law 23). To support process refinement, SPA provides a hiding operation which is written as \(P / C\) with \(C \subseteq o(P)\). Hiding conceals outputs and therefore it modifies the output alphabet of processes: \(o(P \setminus C) = o(P) \setminus C\). We do not allow \(C = o(P)\), because then we would have \(o(P \setminus C) = \emptyset\) which is not a valid SPA process.

The laws for hiding in SPA are similar to the laws for hiding in DFN. Hiding distributes over \(\Pi\) and has \(\bot\) as fixed point. The effect of hiding on output is specified as:

Law 29. \((e \nu; P) \setminus C = P \setminus C \quad \text{if } c \in C\)
\((e \nu; P) \setminus C = e \nu; \langle P \setminus C \rangle \quad \text{if } c \notin C\)

Hiding distributes over input prefixing and the after operation:

Law 30. \((a ? x; P) \setminus C = a ? x; \langle P \setminus C \rangle\)

Law 31. \((P/ a \nu) \setminus C = (P \setminus C) / a \nu \]

Hiding also distributes over guarded choice:

Law 32. \([S] \setminus C = [S']\]

Where each alternative \(g \rightarrow P\) in \(S\) has a corresponding alternative \(g \rightarrow P \setminus C\) in \(S'\).

Note that \([\ ]\) is also a fixed point of hiding.

5. Applications

Application of the laws of SPA to certain program constructs leads to a number of interesting consequences for the development of SPLICE applications. For a validation of SPA, we have specified an existing SPLICE application: a radar simulator combined with a multi-target tracking process. This case-study gives an indication of the expressiveness and applicability of the calculus for a real-world application. We will refrain from presenting it in this paper. There are many application specific technical details that make this case-study less interesting to present it here; the interested reader is referred to [14]. Instead, we will concentrate on the derivation of properties of SPLICE applications by using SPA. These properties will serve as the starting point for the development of guidelines and design rules.

5.1. Functional dependency

An interesting consequence of the asynchronous communication, and therefore the commutativity of outputs, is given in the following derivation. Suppose we have a process \(a \nu; P\) and a process \(a ? x; b \nu(f(x)); Q\) that publishes the result of applying function \(f\) on an input \(x\) of sort \(a\). Their composition reads:

\[
(a \nu; P) || (a ? x; b \nu(f(x)); Q) = \begin{cases}
\text{law 22: definition || for } a \nu \} \\
\text{law 20: } /a \nu \text{ for } a ? x \} \\
\text{law 10: commutativity of outputs } \}
\end{cases}
\]

This derivation is an example of the observational equivalence between processes. From the viewpoint of the environment, we cannot distinguish a process \(a \nu; b \nu(f(v)); (P || Q)\) from \(b \nu(f(v)); a \nu; (P || Q)\). The two (syntactic) different expressions have the same (observational) semantics.

As a consequence, the environment (due to the asynchronous communication) looses the functional dependency between the argument \(v\) and the result \(f(v)\) of a function \(f\). If this functional dependency is important in the application, then it is necessary to encode such dependencies explicitly in the data model. This will influence the design of processes as illustrated by the radar simulator case-study in [14].

5.2. Process replication

In order to increase the fault tolerance of applications, SPLICE supports process replication. In order to use SPA to analyse the effects of process replication, we need to define another parallel composition that is capable to compose
publishers of the same sort. Renaming can be used to distinguish the output of these publishers, such that these can be composed respecting the conditions of parallel composition. Outputs to the environment are renamed back to the original sort name. We introduce a shortcut for this construction.

**Definition 1** We define a merge operator $P \parallel Q$ as:

$$P \parallel Q = \langle \langle P \rangle_{b_1 \ldots b_n} || \langle Q \rangle_{c_1 \ldots c_n} \rangle_{a_1 \ldots a_m}^{b_1 \ldots c_n}$$

We use $\langle \langle P \rangle_{b_1 \ldots b_n} || \langle Q \rangle_{c_1 \ldots c_n} \rangle_{a_1 \ldots a_m}$ as a shortcut for $\langle \ldots \langle P \rangle_{b_1} \ldots \rangle_{a_n} \langle Q \rangle_{c_1} \ldots \rangle_{a_n}$.

This merge operator allows us to express process replication. One might expect idempotency of $\parallel$ (e.g. $P \parallel P = P$), but replication will lead to a duplication of the number of outputs. Assume $P = a_1 v ; Q$ and $b_i c \notin a(P)$ (so $b_i c$ are fresh data-sorts in $P$). Then we have:

$$P \parallel P = \{ \text{definition of } P \}$$

$$= \{ \text{definition } \parallel ; b \text{ and } c \text{ fresh } \}$$

$$\langle a_1 v ; Q \rangle_{b_i c} || \langle a_1 v ; Q \rangle_{c_i b}^{b_i c} = \{ \text{definition } \}$$

$$\langle b_1 v ; \langle Q \rangle_{b_i c} || \langle c_1 v ; \langle Q \rangle_{c_i b}^{b_i c} \rangle = \{ \text{definition } \}$$

$$\langle b_1 v ; c_1 v ; \langle Q \rangle_{b_i c} || \langle Q \rangle_{c_i b}^{b_i c} \rangle = \{ \text{definition } \}$$

$$a_1 v ; a_1 v ; \langle Q \rangle_{b_i c} || \langle Q \rangle_{c_i b}^{b_i c} = \{ \text{definition } \}$$

$$a_1 v ; a_1 v ; \langle Q \rangle_{b_i c} || \langle Q \rangle_{c_i b}^{b_i c}$$

Since it is advantageous to replicate $P$ (as above) without further consequences for the environment consisting of processes $Q_i$, we would like the $Q_i$ to be insensitive against data replication. So, apart from other requirements, we want to have at least the following property for the processes $Q_i$: $(\forall a \in a(P), v : Q_i(a,w) \downarrow_a = Q_i(a,w))$. It is future work to give guidelines for obtaining $Q_i$ with this property.

### 5.3. Order of data

The laws of SPA are focussed on asynchronous communication between processes that write data of different sorts. However, with the $\parallel$ operator we can also compose processes that produce data of the same sort. Suppose we compose processes $a_1 v ; P$ and $a_2 w ; Q$, then we can derive:

$$a_1 v ; P \parallel a_2 w ; Q$$

then we can derive:

$$\{ \text{definition } \}$$

$$\langle a_1 v ; P \rangle_{a_1} || \langle a_2 w ; Q \rangle_{a_2}^{a_1 a_2}$$

By applying commutativity of output, using law 10, on the sub-expression $a_1 v ; a_2 w ; \langle P \rangle_{a_1} || \langle Q \rangle_{a_2}^{a_1 a_2}$, we can also derive:

$$a_1 v ; a_2 w ; \langle P \rangle_{a_1} || \langle Q \rangle_{a_2}^{a_1 a_2}$$

The above derivation illustrates commutativity of write actions of the same sort, in case multiple publishers of this sort exist.

### 5.4. Process creation

Dynamic process creation is another feature of SPLICE that we would like to model in SPA. In SPLICE, a process can create a new process using the coordination primitive $\text{new}(P)$ while in SPA the processes are static entities composed in parallel. To relate these two approaches, we can construct a special process in SPA that simulates process creation.

In relating these approaches, we pay attention to the *volatile* and *persistent* data types of SPLICE that we introduced in Section 2. Due to dynamic process creation we can semantically distinguish these data types. The moment at which a process becomes alive is rather arbitrary (e.g. it depends on the load of the processors and the network). SPLICE ensures that a newly created process receives all previously sent persistent data to which it is subscribed. For volatile data the situation is different and it is not guaranteed that transmitted data is received by the created processes. Consider a process $P$ that is subscribed to data sort $a$. If a datum $a,w$ is sent before $P$ is created, $P$ might miss the opportunity to read $a,w$; on the other hand, transmission of $a,w$ over the network could be delayed in such a way that $P$ (in the meantime) is capable of receiving $a,w$. Note that sending $a,w$ after $P$ has been created does not guarantee the reception of $a,w$, since the communication is asynchronous.
The only way to guarantee that $P$ will receive $a_0 \varphi$ is to send this datum after some output of $P$ has been received.

In SPA we do not have a primitive for process creation. It is, however, possible to simulate process creation in terms of existing primitives of SPA; we assume the number of to be created processes has a known upperbound. Let $Q$ be a process that needs to be ‘created’. We then construct a process $P$ (called a wrapper) similar to $Q$, except that $P$ is waiting for a special datum to arrive first. When it has arrived, $P$ behaves exactly like $Q$. To formalize this we define the following alphabets:

\[
\begin{align*}
i(P) &= i(Q) \cup \{\text{create}\} \\
o(P) &= o(Q) \\
i(Q') &= i(P) \\
o(Q') &= o(P)
\end{align*}
\]

where create is the sort of the special datum, and $Q'$ is equal to $Q$ except for its alphabets. For simplicity we distinguish two cases: (1) $Q$ receives persistent data and (2) $Q$ receives volatile data. It is, however, easy to verify that mixtures of these two are possible as well.

1. A process $P$ is constructed that will wait for a datum of sort create, which will then be read destructively, and $P$ starts behaving like $Q$:

\[
P = [\text{create} ? \varphi \rightarrow Q']
\]

Note that all data that is sent before a create datum is received remains available to $Q'$ (via the after operation).

2. A process $P$ is constructed that will wait for all data to which it is subscribed, and will wait for a datum of sort create. Whatever is received is read destructively. Thereafter, it behaves again like $P$, except when a create datum is received; it will then behave like $Q$. As such, all data that is received by $P$ before a create datum is received is not available to $Q'$. Suppose $i(Q) = \{a_1, \ldots, a_n\}$, then:

\[
P = \begin{cases} 
\text{create} ? \varphi \rightarrow Q' \\
\varnothing a_1 \varphi \rightarrow P \\
\ldots \\
\varnothing a_n \varphi \rightarrow P
\end{cases}
\]

When the inputs of $Q$ are partly volatile and partly persistent data, we just add destructive read operations to $P$ for the volatile inputs.

Now suppose a process $R$ will create the process $Q$. $R$ will look like:

\[
R = \ldots; \text{create}!0; \ldots
\]

where 0 acts as a dummy value of sort create. Due to the asynchronicity of writing data it is easy to see that it is uncertain at what point in the program execution $Q'$ is actually created.

6. Discussion

We have presented the process algebra SPA that offers a formal method for development of SPLICE applications. This process algebra is based on data-flow networks (DFN), as a suitable model for the software architecture SPLICE. We have applied DFN and can use (innumerable) data sets and data variables, a nondestructive read operation in terms of existing constructs: $a?x; P = a?\varphi; (P/a,x)$, and a renaming operator. In formalizing SPLICE, we have imposed several constraints on processes. In particular, we demand input and output alphabets to be disjoint, and a unique publisher of each data sort when processes are composed in parallel. This uniqueness constraint makes it impossible to express process replication with parallel composition only. However, we have shown that with renaming we can define a merge operator with which process replication can be expressed. Dynamic process creation can be expressed in SPA by means of a wrapper. Using this wrapper, we can distinguish between volatile and persistent data.

The following table summarizes the formalization of the coordination primitives of SPLICE by SPA:

<table>
<thead>
<tr>
<th>Coordination primitives:</th>
<th>SPLICE:</th>
<th>SPA:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous write</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Destructive read</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Nondestructive read</td>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>Dynamic process creation</td>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>Parallel composition</td>
<td>–</td>
<td>X</td>
</tr>
<tr>
<td>Process replication</td>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>Renaming and hiding</td>
<td>–</td>
<td>X</td>
</tr>
</tbody>
</table>

Where, $X$ stands for primitive operator, $C$ stands for compound operator (i.e. defined in terms of primitive operators), and – stands for not available.

The goal of the formalization is to support the design of SPLICE based systems within a process algebra. Design rules and guidelines can be formally derived and used to improve the development process. Design rules impose (extra) constraints on the composition of program constructs; guidelines streamline the task to develop programs that satisfy the design rules. Therefore, the next task is to focus on the definition of design rules and guidelines. Some initial steps in defining these rules and guidelines are presented in this paper and the direction of our research is expected to lead to a more comprehensive set of rules and guidelines.

Another task is to extend the read and get operations with the query mechanism of SPLICE; that is, an element is read when it is available and when the given query is satisfied for this element; otherwise, the operation blocks. With such a mechanism we can distinguish the order in which data is read. Reordering read operations in the presence of queries can introduce deadlock. Therefore, we need to impose restrictions on the use of the query mechanism.
In addition to the coordination primitives, there are other important features of SPLICE that need to be expressed, such as real-time and fault-tolerant behaviour. Currently, we can express data and process replication in SPA; future work will focus on design rules for fault-tolerant behaviour and the introduction of real-time processing in our formal framework.

References
