A software architecture for distributed control systems and its transition system semantics

Marcello M. Bonsangue* and Joost N. Kok
Department of Computer Science, Leiden University, The Netherlands
marcello@cs.leidenuniv.nl and joost@cs.leidenuniv.nl

Maarten Boasson and Edwin de Jong
Hollandse Signaalapparaten B.V., The Netherlands
boasson@signaal.nl and edejong@signaal.nl

Keywords: Coordination models, software architecture, SPLICE, control systems, shared data space, distributed data space, transition system semantics.

Abstract

A software architecture for distributed control systems is presented that is based on a shared data space coordination model. The architecture, named SPLICE, is introduced in two steps. First we give the general structure of the coordination model including the shared data space and the basic operations and we define its semantics by means of a transition system. Second we present a transition system for a refinement of the coordination model: data is distributed and replicated across a communication network using a publication-subscription protocol. We also discuss methods in SPLICE for fault-tolerance and the possibility for on-line system modification and extensions.

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 put stringent requirements to real-time behaviour, fault-tolerance and safety.

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 behavior of components and their interaction [8]. Traditionally, software architectures have been primarily concerned with structural organization and static interfaces [17]. With the growing interest in coordination models [1, 13, 15], more emphasis is placed on the organizational aspects of behavior and interaction. Coordination is the process of building programs by gluing together active pieces [11]. Coordination models can be classified as either data-driven or control-driven depending on how coordination is achieved: by manipulating data values shared among all active processes or by dynamically evolving the interconnections among the active processes as a consequence of observations of their state changes [19]. Processes in a data-driven coordination model typically communicate via a shared data space. The structure of a data space strongly depends on how data is treated by the coordination model. A multiset structure of the data space corresponds to the view of data as resources (multiplicity of data items is significant), a stream structure of the data space is related to the treatment of data as events (ordering and causal relations among data items are significant), while a set structure of the data space implies that data represents information (no causal relation among data items and multiplicity of data items is insignificant).

Applications in the control systems domain mostly deal with data instances that represent continuous quantities: data is either an observation sampled from the system's environment, or derived from such samples through a process of data association, correlation, and classification. The data itself is relatively simple in structure; there are only a few data types, and given the volatile nature of the samples, only recent values are of interest. However, samples may enter the system at very short intervals, so sufficient throughput and low latency are crucial properties. In addition, but to a lesser extent, control systems maintain discrete information, which is either directly related to external events or derived through qualitative reasoning from the sampled input.

At Hollandse Signaalapparaten a software architecture for distributed control systems has been developed, and has been applied in the construction of commercially available traffic management and command-and-control systems. The architecture, named SPLICE, employs a data-driven coordination model that is based on a shared data space; in this respect SPLICE bears close resemblance to coordination models and languages like Linda [10], Gamma [5], μLog [18] ImpUnity [16], and Swarm [21] (the last two being based on the UNITY language [12]). The semantics and implementation of the coordination primitives of SPLICE, however, are strongly tailored towards the specific requirements of distributed control systems.

In this paper we present the semantics of SPLICE by

*The research of Marcello M. Bonsangue was supported by the *Stichting Informatica Onderzoek in Nederland* within the context of the project no. 612-33-007 'Formal methods and refinement for *C*oordination *L*anguages'.

means of transition systems. The resulting semantics is operational in the sense that it is based on an operational intuition: each transition in a transition system represents a state transformation that can be obtained by executing an action of a SPLICE process. We will use a structural approach to the definition of transition systems, meaning that the transition relation will be described by induction on the structure of SPLICE processes [20].

The plan of the paper is as follows. First we define the general structure of the SPLICE coordination model, including the shared data space and the coordination primitives. A transition system is presented which formally describes the behaviour of a SPLICE process. In Section 3 we present a refinement of the coordination model: data is distributed and replicated across a communication network using a publication-subscription protocol. The behaviour of a SPLICE process with a distributed data space is also described by a transition system. In Section 4 we discuss typical problems of system design which are not captured by a transition system semantics, and we informally present a solution within the SPLICE software architecture.

2 SPLICE: A software architecture for control systems

The coordination model of SPLICE consists of two basic types of components: processes and a shared data space. Processes concurrently act on the shared data space. There is no direct interaction between processes; all communication takes place through the shared data space by reading and writing data elements. In this sense SPLICE bears strong resemblance to the coordination model Linda [10] where active entities are coordinated by means of a shared data space. However, in SPLICE, data is not treated as resource but as information, similarly to the view adopted by the concurrent constraint paradigm [22]. Thus a process cannot delete data elements from the shared data space.

The elements that populate the shared data space are structured as labeled pairs of the form $\langle \alpha{:}v, w\rangle$, where v and w range over a fixed set *Val* of values. The label α of a data element $\langle \alpha{:}v, w\rangle$ is associated with a system-wide unique interpretation of the corresponding data by the processes that use it; α is also referred to as the *sort* of a data element. Sorts enable processes to distinguish between different types of information. We assume that sorts range over a fixed set *Sort* and denote the set of all data elements by

$$\mathit{Data} \quad = \quad \{\langle \alpha{:}v, w\rangle \mid \alpha \in \mathit{Sort},\ v, w \in \mathit{Val}\}\,.$$

The v component of a data element $\langle \alpha{:}v, w\rangle$ represents the *key field* or *index*. Each data element in the shared data space is uniquely determined by its sort and the value of its key field. The second component w represents the remaining (non-key) field(s) of the data element.

To illustrate, consider a simplified example taken from the domain of air traffic control. Typically a system in this domain would be concerned with various aspects of flights, such as flight plans and the progress of flights as tracked from the reports received from the system's surveillance radar. Hence, we introduce sorts 'flightplan', 'report', and 'track'.

An element of sort 'flightplan', for instance, could be of the form

$$\langle \texttt{flightplan}{:}\ \mathit{flight}, (\mathit{dept}, \mathit{arr})\rangle\,,$$

consisting of a unique flight number *flight*, which is the key field, and the scheduled times for departure *dept* and arrival *arr*. Examples of data elements of sort 'flightplan' are:

$$\langle \texttt{flightplan}{:}\ \mathit{KL}206, (10{:}30, 12{:}43)\rangle,$$

$$\langle \texttt{flightplan}{:}\ \mathit{AZ}125, (13{:}50, 15{:}30)\rangle.$$

The sort 'report' is structured as

$$\langle \texttt{report}{:}\ \mathit{idx}, (\mathit{pos}, \mathit{time})\rangle\,,$$

where *pos* stands for a position vector of some object as measured at a specific time *time* by the system's surveillance radar, and *idx* is a unique index attached by the radar to be able to distinguish between different reports. Through a correlation and identification process, the progress of individual flights is recorded in the sort

$$\langle \texttt{track}{:}\ \mathit{flight}, (\mathit{state}, \mathit{time})\rangle\,,$$

where *state* is a vector, which typically contains position and velocity information on the associated flight number *flight* at time *time*.

Processes interact with the shared data space by writing and reading data elements. In SPLICE data elements can be removed either implicitly, using an overwriting mechanism, or explicitly, through a destructive reading operation which makes data invisible to a specific process. These deletion mechanisms reflect the fact that in control systems data gradually loses its value as the environment changes and time evolves. Basically we can distinguish between two cases: either data becomes useless to all processes because more recent data of the same sort is available from the environment, or data becomes useless to a specific process because it falls outside its temporal view. While in the first case data can be safely removed from the shared data space, since it is no longer of interest to any process, in the second case data can only be removed from the view of the particular process, but not from the shared data space itself, since it can be of interest to other processes.

Formally, an element $\langle \alpha{:}v, w\rangle \in \mathit{Data}$ when inserted into the shared data space overwrites all data of the same sort α and index v. The newly inserted data item should be visible to all processes interacting with the shared data space, even to those which have deleted data of sort α and index v from their view. This problem calls for a data versioning mechanism which should be invisible at the programming level. Therefore we define a shared data space d as a subset of the set of so-called versioned data *VData*, defined by

$$\mathit{VData} \quad = \quad \mathit{Data} \times \mathbb{N}\,,$$

where the set of natural numbers $\mathbb{N}$ is the set of version identifiers. A data item $\langle \alpha{:}v, w\rangle \in \mathit{Data}$ can be inserted into a shared data space d by the following operation:

$$d + \langle \alpha{:}v, w\rangle = (d \setminus \{(\langle \alpha{:}v, w'\rangle, n) \mid w' \in \mathit{Val},\ n \in \mathbb{N}\}) \cup \{(\langle \alpha{:}v, w\rangle, m)\}\,,$$

where $m = 1 + \max\{n \mid (\langle \alpha{:}v, w\rangle, n) \in d\}$ and $\max \emptyset = 0$. Next we explain this definition. Through the key field we can distinguish data of the same sort: if a process inserts an element $\langle \alpha{:}v, w\rangle$ into the shared data space, then it is first verified whether an instance of the same sort α with the same key value v currently exists in the data space. If it does then the currently stored instance is overwritten by the newly inserted, more recent value. To reflect that the newly inserted data is more recent than any equal data already present in the data space, the associated version number is incremented.

SPLICE processes read data from the shared data space by means of queries. Formally, we assume given an abstract set $(q \in)$ *Query* of queries together with a binary relation $\models \subseteq (\mathit{Val} \times \mathit{Val}) \times \mathit{Query}$ giving all pairs of values $\langle v, w \rangle$ which satisfy a query q. We use $\langle v, w \rangle \not\models q$ to indicate that the pair of values $\langle v, w \rangle$ does not satisfy the query q.

SPLICE extends an existing programming language, referred to as the host language, with coordination primitives for the creation of processes and for the interaction with the shared data space. Below we first present an informal explanation of the primitives. A formal semantics is given in the next section by means of a transition system.

The coordination primitives are as follows:

- $new(P)$: Start a new process P in parallel with the processes already active.
- $wrt(\langle \alpha{:}v, w \rangle)$: Insert a data element $\langle \alpha{:}v, w \rangle$ into the shared data space.
- $rd(\alpha, q)$: Read an element of sort α from the shared data space that satisfies query q. In case a requested instance does not exist or is outside the process view, the operation blocks until one becomes available. If, on the other hand, multiple instances satisfying q are available, one is selected non-deterministically.
- $get(\alpha, q)$: Remove an element of sort α and satisfying query q from the process view of the shared data space. In case a requested instance does not exist or is outside the process view, the operation blocks until a new one becomes available. If multiple instances satisfying the query are available, one is selected non-deterministically and removed.

As an illustration we briefly return to the air traffic control example. Consider a process that tracks the progress of flight number *flight*. This process reads new reports from the surveillance radar and updates the corresponding track information accordingly. It is defined as follows:

$$
\begin{aligned}
&\mathit{Tracking}(\mathit{flight}, \mathit{state}, \mathit{time}) \widehat{=} \\
&\quad \langle \mathit{idx}, (\mathit{pos}, \mathit{time}') \rangle := get(\texttt{report}, \mathit{correlates}(\mathit{state}, \mathit{time})); \\
&\quad \mathit{state}' := \mathit{update}(\mathit{state}, \mathit{pos}, \mathit{time}' - \mathit{time}); \\
&\quad wrt(\langle \texttt{track}{:}\ \mathit{flight}, (\mathit{state}', \mathit{time}') \rangle); \\
&\quad \mathit{Tracking}(\mathit{flight}, \mathit{state}', \mathit{time}').
\end{aligned}
$$

The process $\mathit{Tracking}(\mathit{flight}, \mathit{state}, \mathit{time})$ initiates tracking of flight number *flight* with a state vector *state* and at time *time*. It then reads a new report $\langle \mathit{idx}, (\mathit{pos}, \mathit{time}') \rangle$ from the data space, and correlates it with the current state vector *state*. This is expressed by the query $\mathit{correlates}(\mathit{state}, \mathit{time})$, which (for simplicity) is not specified here in further detail. The process then updates the state vector *state* with measurement *pos* over a time step $\mathit{time}' - \mathit{time}$ (we assume here that this statement is available from the host programming language). The result is inserted into the shared data space, overwriting the previous state vector of the same flight, after which the process is repeated using the updated state vector.

The SPLICE coordination primitives are embedded into a host program. Since we do not want to commit ourselves to a particular host language, we can think of a process embedding the SPLICE primitives as a process which either executes some internal steps or executes a coordination primitive. However, $rd(\alpha, q)$ and $get(\alpha, q)$ actions are only fully specified if we assume that the pair of values selected by the query q is known. Hence we consider processes that can execute actions from the following set:

$$
\begin{aligned}
\mathit{Act} = \{\tau\} \cup& \\
&\{new(\sigma) \mid \sigma \in \Sigma\} \cup \\
&\{wrt(\langle \alpha{:}v, w \rangle) \mid \alpha \in \mathit{Sort}, v, w \in \mathit{Val}\} \cup \\
&\{(rd(\alpha, q), \langle v, w \rangle) \mid \alpha \in \mathit{Sort}, q \in \mathit{Query}, v, w \in \mathit{Val}\} \cup \\
&\{(get(\alpha, q), \langle v, w \rangle) \mid \alpha \in \mathit{Sort}, q \in \mathit{Query}, v, w \in \mathit{Val}\}\,.
\end{aligned}
$$

Processes embedding the SPLICE primitives may be seen as a transition system

$$T \;=\; \langle \Sigma, \mathit{Act}, \longrightarrow_T \rangle\,,$$

where $(\sigma \in) \Sigma$ is an abstract set of intermediate or final states of computations, and $\longrightarrow_T \subseteq \Sigma \times \mathit{Act} \times \Sigma$ is a transition relation which satisfies the following two conditions, for all $\sigma \in \Sigma$:

i) if $\sigma \xrightarrow{(rd(\alpha,q),\langle v,w \rangle)}_T \sigma'$ then for all $\langle v', w' \rangle \models q$ there exists $\sigma'' \in \Sigma$ such that $\sigma \xrightarrow{(rd(\alpha,q),\langle v',w' \rangle)}_T \sigma''$;

ii) if $\sigma \xrightarrow{(get(\alpha,q),\langle v,w \rangle)}_T \sigma'$ then for all $\langle v', w' \rangle \models q$ there exists $\sigma'' \in \Sigma$ such that $\sigma \xrightarrow{(get(\alpha,q),\langle v',w' \rangle)}_T \sigma''$;

Intuitively, a transition $\sigma \xrightarrow{a}_T \sigma'$ indicates the possibility of executing the action $a \in \mathit{Act}$ changing the state σ into the state σ'. However, the state transformation related to the execution of a $rd(\alpha, q)$ or a $get(\alpha, q)$ action may depend on the pair of values selected by the query q among those pairs present in the environment. Via the conditions i) and ii) above we abstract from the interaction with the environment: state transformations are specified for each pair of values satisfying the query q which may be present in the environment, leaving "open" what pair of values is actually received. Two different transition systems which take into account the actual interactions among several SPLICE processes via a shared and a distributed data space will be presented in the next two sections.

This abstract formulation allows us to consider different languages in which the SPLICE primitives can be embedded. Besides imperative languages it is also possible, for example, to consider logic or functional languages. Next we give an example of a transition system describing a process of a simple imperative language which incorporates the SPLICE primitives. Let $(x, y \in)$ *Var* be a set of variables and let $(s \in)$ *State* be the set of functions mapping variables in *Var* to values in *Val*. Given $s \in \mathit{State}$ we denote by $s[x/v]$ the function in *State* mapping x to v and every $y \neq x$ to $s(y)$. Let *Prog* be a set of simple imperative SPLICE programs given by the following syntax:

$$
\begin{aligned}
P \;::=\;& \mathit{Nil} \mid x := v.P \mid new(P).P \mid wrt(\langle \alpha{:}v, w \rangle).P \mid \\
& \langle x, y \rangle := rd(\alpha, q).P \mid \langle x, y \rangle := get(\alpha, q).P \mid P \Box P\,.
\end{aligned}
$$

Programs in *Prog* may terminate, assign a value to a variable or execute a SPLICE primitive and then continue with the execution of another program, or make a non-deterministic choice '$\Box$' between two programs.

The behaviour of programs in *Prog* is described by the transition system $\langle \Sigma, \mathit{Act}, \longrightarrow_T \rangle$ where $\Sigma = \mathit{State} \times \mathit{Prog}$, and $\longrightarrow_T$ is the least relation on $\Sigma \times \mathit{Act} \times \Sigma$ satisfying the following axioms and rules:

- $\langle s, x := v.P \rangle \xrightarrow{\tau}_T \langle s[x/v], P \rangle$,
- $\langle s, new(P_1).P_2 \rangle \xrightarrow{new(\langle s, P_1 \rangle)}_T \langle s, P_2 \rangle$,
- $\langle s, wrt(\langle \alpha{:}v, w \rangle).P \rangle \xrightarrow{wrt(\langle \alpha:v,w \rangle)}_T \langle s, P \rangle$,
- for all $v, w \in Val$, if $\langle v, w \rangle \models q$ then

$$\langle s, \langle x, y \rangle := rd(\alpha, q).P \rangle \xrightarrow{(rd(\alpha,q),\langle v,w \rangle)}_T \langle s[x/v][y/w], P \rangle\,,$$

- for all $v, w \in Val$, if $\langle v, w \rangle \models q$ then

$$\langle s, \langle x, y \rangle := get(\alpha, q).P \rangle \xrightarrow{(get(\alpha,q),\langle v,w \rangle)}_T \langle s[x/v][y/w], P \rangle\,,$$

- $\langle s, P_1 \Box P_2 \rangle \xrightarrow{\tau}_T \langle s, P_1 \rangle$ and $\langle s, P_1 \Box P_2 \rangle \xrightarrow{\tau}_T \langle s, P_2 \rangle$.

Note that the transition system satisfies both conditions i) and ii) above.

2.1 A transition system for SPLICE

Next we describe the semantics of SPLICE by means of a transition system. The transition relation defines the interaction of SPLICE processes with the shared data space. We assume that, when created, a SPLICE process becomes active and receives a unique identifier from a set of process identifiers ($\iota \in$) PId. Process identifiers should be assigned locally by the process executing the *new* operation. Therefore each process needs a counter n for the processes already created and a mechanism which, given the process identifier and the above counter, returns a new globally unique identifier, that is, we assume given a function

$$\nu \colon (PId \times \mathbb{N}) \to PId\,.$$

The function ν should be such that

i) $\nu(\iota, n) \neq \iota$ for all $n \in \mathbb{N}$ and $\iota \in PId$,

ii) for all $\iota_1, \iota_2 \in PId$ and $n_1, n_2 \in \mathbb{N}$, if $(\iota_1, n_1) \neq (\iota_2, n_2)$ then $\nu(\iota_1, n_1) \neq \nu(\iota_2, n_2)$.

Informally, the identifier of a newly created process must be different from the identifier of the process creating it, and from all the other identifiers used by any other process. For example, if we take identifiers to be finite strings of natural numbers, that is, $PId = \mathbb{N}^*$, then we could define

$$\nu(w, n) = w \cdot (n + 1)\,,$$

for $w \in PId$ and $n \in \mathbb{N}$.

An active SPLICE process is defined as a tuple $\langle \iota, n, r, \sigma \rangle$, where $\iota \in PId$ is a unique identifier associated with the process, $n \in \mathbb{N}$ is the counter of the processes created by ι, $r \subseteq VData$ is a store containing the data items not visible to the process ι, and $\sigma \in \Sigma$ is the state of the process embedding the SPLICE primitives. We denote by $AProc$ the set of active SPLICE processes. The behaviour of a SPLICE is defined by a transition relation

$$\longrightarrow \subseteq Conf \times Conf\,,$$

where $Conf = \mathcal{P}(AProc) \times \mathcal{P}(VData)$ and $\mathcal{P}(-)$ is the power set construction. An element $[A, d] \in Conf$ represents the set A of active SPLICE processes interacting with the shared data space d. A transition $[A, d] \longrightarrow [A', d']$ means that the interaction of (some of) the active SPLICE processes in A interacting with the shared data space d results in a set A' of active SPLICE processes interacting with the shared data space d'.

The transition relation $\longrightarrow$ is the least relation satisfying the following rules (we denote the disjoint union of sets by $\uplus$). Below each rule there is some explanation.

Local computation:

$$\frac{\sigma \xrightarrow{\tau}_T \sigma'}{[\{\langle \iota, n, r, \sigma \rangle\}, d] \longrightarrow [\{\langle \iota, n, r, \sigma' \rangle\}, d]}$$

If a process performs some local computation then so can the SPLICE process in which it resides. Notice that this rule determines the termination of a SPLICE process when the process which resides in it terminates.

Process creation:

$$\frac{\sigma \xrightarrow{new(\sigma_0)}_T \sigma'}{[\{\langle \iota, n, r, \sigma \rangle\}, d] \longrightarrow [\{\langle \iota, n+1, r, \sigma' \rangle, \langle \nu(\iota, n), 0, \emptyset, \sigma_0 \rangle\}, d]}$$

When a process executes the action $new(\sigma_0)$, a new active SPLICE process is created with identifier $\nu(\iota, n)$. Its counter of created processes is set to 0, all data in d is visible to it, and it can start executing from the initial state σ_0. The SPLICE process ι executing the action $new(\sigma_0)$ increases its counter of created processes by one and changes its state σ into σ'.

Data writing:

$$\frac{\sigma \xrightarrow{wrt(\langle \alpha:v,w \rangle)}_T \sigma'}{[\{\langle \iota, n, r, \sigma \rangle\}, d] \longrightarrow [\{\langle \iota, n, r, \sigma' \rangle\}, d + \langle \alpha{:}v, w \rangle]}$$

When a process executes the action $wrt(\langle \alpha{:}v, w \rangle)$, the data element $\langle \alpha{:}v, w \rangle$ is added to the shared data space d using the $+$ operator. The data is immediately visible to every process which interacts with the shared data space.

Data reading: If $\langle v, w \rangle \models q$ and there exists $m \in \mathbb{N}$ such that $(\langle \alpha{:}v, w \rangle, m) \in d \setminus r$ then

$$\frac{\sigma \xrightarrow{(rd(\alpha,q),\langle v,w \rangle)}_T \sigma'}{[\{\langle \iota, n, r, \sigma \rangle\}, d] \longrightarrow [\{\langle \iota, n, r, \sigma' \rangle\}, d]}$$

If a process executes the action $rd(\alpha, q)$ and there exists a data element of sort α visible to the process ι and such that its value components v and w satisfy the query q, then the state σ is changed into the state σ'.

Data removal: If $\langle v, w \rangle \models q$ and there exists $m \in \mathbb{N}$ such that $(\langle \alpha{:}v, w \rangle, m) \in d \setminus r$ then

$$\frac{\sigma \xrightarrow{(get(\alpha,q),\langle v,w \rangle)}_T \sigma'}{[\{\langle \iota, n, r, \sigma \rangle\}, d] \longrightarrow [\{\langle \iota, n, r', \sigma' \rangle\}, d]}$$

where $r' = r \cup (\langle \alpha{:}v, w \rangle, m)$. If the host process executes the action $get(\alpha, q)$ then one visible data element of sort α from the shared data space which satisfies the query q becomes now invisible to process ι. The selected data element is removed from the view of ι by adding it to the set of locally invisible data r. Also, the state σ is changed into σ'.

Process combination:

$$\frac{[A_1, d] \longrightarrow [A_2, d']}{[A \uplus A_1, d] \longrightarrow [A \uplus A_2, d']}$$

Through this rule we can add processes to the set of active processes executing in parallel. Notice that termination does not remove a SPLICE process from the set of active processes.

The above transition system gives an interleaving semantics where processes interact with the shared data space only one at a time. This might suggest that the transition system can only handle a centralized thread of control, but this is not the case, as we will show next.

Before introducing a more realistic parallel access to the shared data space, we need to define an operation for inserting multiple data elements into a data space. For V a finite subset of $Data$ and $d \subseteq VData$ a data space we denote by $d \oplus V$ the insertion, in every possible order, of the elements in V into the data space d, that is

$$d \oplus V \quad = \quad \begin{cases} \{d\} & \text{if } V = \emptyset \\ \bigcup\{(d + x) \oplus (V \setminus x) \mid x \in V\} & \text{otherwise.} \end{cases}$$

The set $d \oplus V$ contains all possible data spaces resulting from the insertion of elements from V into d. To avoid corruption of the data space, data items are inserted only one at a time. Note that if d does not contain data elements of the same sort and with the same key field then neither does every d' in $d \oplus V$. Furthermore, if $V = \{\langle \alpha{:}v, w\rangle\}$ then $d \oplus V = \{d + \langle \alpha{:}v, w\rangle\}$. More generally, if V does not contain data elements of the same sort and with the same key field then $d \oplus V$ is a set with one element.

We can now add the following rule which says that there may be as many control threads as processes [14]. If at the same time a process reads a data element from the shared data space and another process overwrites this data element, then the first has precedence over the second.

Multistep transitions:

$$\frac{[A_1, d] \longrightarrow [A_1', d'] \text{ and } [A_2, d] \longrightarrow [A_2', d'']}{[A_1 \uplus A_2, d] \longrightarrow [A_1' \uplus A_2', \overline{d}]}$$

where the data space $\overline{d} \in d \oplus V$, and

$$V \quad = \quad \{\langle \alpha{:}v, w\rangle \mid (\langle \alpha{:}v, w\rangle, n) \in (d' \setminus d) \cup (d'' \setminus d)\}\,.$$

In other words, $\overline{d}$ is a data space obtained by writing in an arbitrary order the data items which have been written in the data space d by the transitions $[A_1, d] \longrightarrow [A_1', d']$ and $[A_2, d] \longrightarrow [A_2', d'']$, respectively.

The order in which the data items are inserted in the data space is of importance because two processes can insert two different data items of the same sort and with the same key value, as shown in the next example. Let $v, w, v', w' \in Val$ be such that $v \neq v'$ and $w \neq w'$. If

$$[A_1, \emptyset] \longrightarrow [A_1', \{(\langle \alpha{:}v, w\rangle, n)\}]$$

and

$$[A_2, \emptyset] \longrightarrow [A_2', \{(\langle \alpha{:}v, w'\rangle, n), (\langle \alpha{:}v', w'\rangle, n)\}]$$

then either

$$[A_1 \uplus A_2, \emptyset] \longrightarrow [A_1' \uplus A_2', \{(\langle \alpha{:}v, w\rangle, n), (\langle \alpha{:}v', w'\rangle, n)\}]\,,$$

meaning that $\langle \alpha{:}v, w\rangle$ is inserted overwriting $\langle \alpha{:}v, w'\rangle$, or

$$[A_1 \uplus A_2, \emptyset] \longrightarrow [A_1' \uplus A_2', \{(\langle \alpha{:}v, w', n), (\langle \alpha{:}v', w'\rangle, n)\}]\,,$$

meaning that $\langle \alpha{:}v, w'\rangle$ is inserted overwriting $\langle \alpha{:}v, w\rangle$.

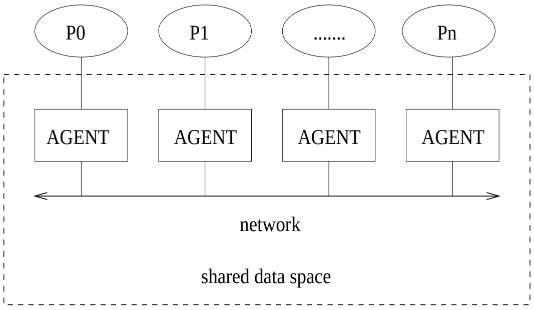

Figure 1: A refinement of the coordination model for distributed systems.

3 Refinements of the coordination model

Starting from a shared data space model of SPLICE we next discuss how a software architecture can be derived that supports a distributed data space. The semantics of the refined architecture is given by a new transition system.

3.1 A distributed data space

The coordination model is refined by introducing two additional components to the basic software architecture from the previous section. As illustrated in Figure 1, the additional components consist of agents and a communication network.

Each process interacts with exactly one agent. An agent embodies processing facilities for handling all communication needs of the application process it serves. All agents are identical and need no prior information about either the application processes or their communication requirements. Communication between agents is established by a message passing mechanism. Messages between agents are handled by the communication network that interconnects them. The network must support broadcasting, but should preferably also support direct addressing of agents, and multicast. An application process interacts with its assigned agent by means of the coordination primitives introduced in Section 2.

Agents are passive servers of the processes, but are actively involved in establishing and maintaining the required inter-agent communication. The communication needs are derived dynamically by the collection of agents from the SPLICE primitives that are issued by the processes. The protocol that is used by the agents to manage communication is based on a subscription paradigm (hence the name SPLICE which stands for Subscription Paradigm for the Logical Interconnection of Concurrent Engines). The assumption underlying this protocol is that the majority of data transfers in a system occurs regularly, and that it is therefore beneficial to forward data to all known users as soon as it is written (cf. a subscription to a newspaper: the newspaper is sent upon printing, without further requests from those subscribed).

More specifically, if an agent receives a data element from another agent on the network, it first verifies whether an element with the same key values currently exists in its local data space. If it does, the newly received instance is more recent, and the currently stored instance is overwritten. Conversely, if a process inserts a data element into its local data space, its agent will forward a copy of the element to all subscribed agents, and it replaces any current instance with the same key values in its local data space with the more recent version. An agent subscribes to all data sorts of which its process is known to be a consumer dynamically. Every producer of data of these sorts will forward a copy of all its stored values to the newly subscribed agent. The newly subscribed agent has a complete visibility of data of the subscribed sort since agents never terminate (even if their host processes may terminate) and produced data instances cannot be removed but only substituted with a more recent instance of the same sort and index value.

As a result of the above protocol, the logically shared data space is selectively replicated across the agents in the network. The local data space of each agent contains instances of only those sorts that are actually read or written by the application process it serves. In practice the approach is viable, particularly for large systems, since the processes are generally interested in only a fraction of all sorts. Moreover, the communication pattern in which agents exchange data is relatively static: it may change when the operational mode of a system changes, or in a number of circumstances in which the configuration of the system changes (such as extensions or failure recovery). Such changes to the pattern are very rare with respect to the number of actual communications using an established pattern. It is therefore beneficial from a performance point of view to maintain a subscription registration. After an initial short phase each time a new sort has been introduced, the agents will have 'learned' the new communication requirement. This knowledge is subsequently used by the agents to distribute newly produced instances of a data sort to all the agents that hold a subscription. Since subscription registration is maintained dynamically by the agents, all changes to the system configuration will automatically lead to adaptation of the communication patterns.

3.2 A transition system for SPLICE with a distributed data space

In order to formalize the subscription protocol of SPLICE we need to consider two sets of messages,

$$\begin{aligned} Sub &= \{\mathtt{sub}(\alpha) \mid \alpha \in Sort\} \qquad \text{and} \\ Req &= \{\alpha \triangleright \iota \mid \alpha \in Sort, \iota \in PId\}\,. \end{aligned}$$

Messages in Sub are used by an agent to indicate its subscription to data of sort α. It is stored in the local data space of the agent the first time it requires a data element of sort α. It is used as a filter to select (by means of their sorts) the data elements to be inserted from those which are broadcasted by other agents. A message $\alpha \triangleright \iota$ in Req is broadcasted by the agent ι and is stored in the local data space of any agent which is a producer of sort α (the agent need not to be ι itself). It indicates that the agent ι requires a copy of all data items of sort α already produced. When the request is satisfied the message $\alpha \triangleright \iota$ is removed from the data space.

We denote by $MData$ the set of request messages and data items:

$$MData \;=\; Req \cup Data\,.$$

An agent embodies two local data spaces: one storing data instances written by the process and request messages, and another one storing the data to be read and the subscription information. We refer to the first one as the *producer data space* and to the second one as the *consumer data space*. A producer data space d is a subset of $Data \cup Req$. The operation for inserting an item $x \in MData$ in a producer data space d is defined as follows:

$$d +_{\mathtt{p}} x = \begin{cases} d \cup \{x\} & \text{if } x = \alpha \triangleright \iota,\ \exists \langle \alpha{:}v, w\rangle \in d \\ d' & \text{if } x = \langle \alpha{:}v, w\rangle \\ d & \text{otherwise,} \end{cases}$$

where $d' = (d \setminus \{\langle \alpha{:}v, w\rangle \in d \mid w \in Val\}) \cup \{x\}$. In other words, a request for data of sort α by a process ι is inserted only if data space d is part of an agent which is has already produced some data instances of the same sort α, while the insertion of a data element in a producer data space overwrites existing data items of the same sort and key value.

A consumer data space d is a subset of $Data \cup Sub$. The operation for inserting an item $x \in MData$ into a consumer data space d is now defined as follows:

$$d +_{\mathtt{c}} x = \begin{cases} d' & \text{if } x = \langle \alpha{:}v, w\rangle,\ \mathtt{sub}(\alpha) \in d \\ d & \text{otherwise,} \end{cases}$$

where $d' = (d \setminus \{\langle \alpha{:}v, w\rangle \in d \mid w \in Val\}) \cup \{x\}$. As before, the insertion of a data element in a local data space overwrites existing data of the same sort and key value. However this happens only if the data space is part of an agent which is subscribed (and hence is a consumer) to data of the same sort α, denoted by $\mathtt{sub}(\alpha)$. The insertion of request messages has no effect on a consumer data space.

Inserting several data items at the same time in a consumer or producer data space may result in different data spaces depending on the order the agent inserts the data items. We denote by $d \oplus_{\mathtt{p}} V$ and $d \oplus_{\mathtt{c}} V$ the sets of all possible consumer data spaces and producer data spaces, respectively, resulting from the insertion of a finite subset V of $Data$ into d. If $d \subseteq Data \cup Req$ then $d \oplus_{\mathtt{p}} V$ is defined inductively by

$$d \oplus_{\mathtt{p}} V = \begin{cases} \{d\} & \text{if } V = \emptyset \\ \bigcup\{(d +_{\mathtt{p}} x) \oplus_{\mathtt{p}} (V \setminus x) \mid x \in V\} & \text{otherwise.} \end{cases}$$

If $d \subseteq Data \cup Sub$ is a consumer data space then $d \oplus_{\mathtt{c}} V$ is defined similarly (substituting '$+_{\mathtt{p}}$' with '$+_{\mathtt{c}}$'). If the set V contains at most one data element for every sort and index value, then both sets $d \oplus_{\mathtt{p}} V$ and $d \oplus_{\mathtt{c}} V$ contain a single element for every sort and index value. Therefore, in this case, the order in which the items in V are inserted in d is not important.

When activated, each SPLICE process P is associated with a unique agent $[\iota, n, \sigma, c, p]$, where $\iota \in PId$ is a (globally unique) identifier, $n \in \mathbb{N}$ is a counter of the processes already activated by the agent ι, $\sigma \in \Sigma$ is the local state of the process, and $c \subseteq Data \cup Sub$ and $p \subseteq Data \cup Req$ are the consumer and producer data spaces. The data space c contains a copy of the data items of those sorts to which the agent ι is subscribed and the information about subscriptions to data sorts, whereas the data space p contains a copy of the data items produced by ι and the requests for data by other agents. We denote the set of all agents by *Agent*.

For $[\iota, n, \sigma, c, p] \in Agent$ and a finite set V of $MData$, define the set $[\iota, n, \sigma, c, p] \oplus V$ as follows:

$$\begin{aligned} [\iota, n, \sigma, c, p] \oplus V \;=\; \{[\iota, n, \sigma, c', p'] \mid\ & c' \in c \oplus_{\mathtt{c}} (V \cap Data), \\ & p' \in p \oplus_{\mathtt{p}} (V \cap Req)\}. \end{aligned}$$

An agent in the set $[\iota, n, \sigma, c, p] \oplus V$ is therefore the result of inserting the request messages in V into the producer data space p, and the data elements in V into the consumer data space c of the agent ι. Notice that data elements of sort α are inserted into c only if the agent is subscribed to α, whereas requests for data of sort α are inserted into p only if the agent is a producer of data of sort α. Also, if the set V contains at most one data element for every sort and index value, then the set $[\iota, n, \sigma, c, p] \oplus V$ contains only one agent.

For a given $A \subseteq \mathit{Agent}$ and a finite subset V of MData we define $A_V \subseteq \mathcal{P}(\mathit{Agent})$, as follows:

$$\begin{aligned} A_V \;=\; \{\{f(a) \mid a \in A\} \mid\, & f\colon A \to \bigcup\{a \oplus V \mid a \in A\}, \\ & f(a) \in a \oplus V\}\,. \end{aligned}$$

The set $\bigcup\{a \oplus V \mid a \in A\}$ represents the set of all possible results of inserting items from V into the local data spaces of the agents in A. Each agent can insert the items in V in a different order, possibly resulting in different local data spaces even if they were equal before the insertion. All possibilities are considered by taking a choice function f which transforms every agent a in A into an agent in $a \oplus V$. By doing some calculation we see that if V contains at most one data element $\langle \alpha{:}v, w\rangle$ for every sort α and index value v then

$$A_V = \{\{a' \mid a \in A, \{a'\} = a \oplus V\}\}\,.$$

For example, if no data is inserted (i.e. $V = \emptyset$), then the agents do not change, and hence $A_V = \{A\}$.

We can now define the behaviour of a SPLICE process with a distributed data space by a transition relation

$$\longrightarrow \subseteq \mathcal{P}(\mathit{Agent}) \times \mathcal{P}(\mathit{MData}) \times \mathcal{P}(\mathit{Agent})\,.$$

The idea is that a transition $A \stackrel{V}{\rightarrow} A'$ indicates that the set of agents A broadcasts items in V to the environment and evolves into the set of agents A'. The data elements or messages broadcasted are received by other agents which insert them in their local data spaces (via the operation described above).

The transition relation $\longrightarrow$ is defined as the least relation satisfying the following axioms and rules:

Local computation:

$$\frac{\sigma \stackrel{\tau}{\rightarrow}_T \sigma'}{\{[\iota, n, \sigma, c, p]\} \stackrel{\emptyset}{\rightarrow} \{[\iota, n, \sigma', c, p]\}}$$

As in the shared data space view, if a process performs some local computation, then so can the SPLICE process in which it resides, possibly determining its termination. Notice that if a process terminates then its agent is not removed in order to avoid the loss of data elements it has produced.

Process creation:

$$\frac{\sigma \xrightarrow{\mathit{new}(\sigma_0)}_T \sigma'}{\{[\iota, n, \sigma, c, p]\} \stackrel{\emptyset}{\rightarrow} \{[\iota, n+1, \sigma', c, p], [\nu(\iota, n), 0, \sigma_0, \emptyset, \emptyset]\}}$$

If a process executes the action $\mathit{new}(\sigma_0)$ then a new agent is created named $\nu(\iota, n)$ which will execute the process starting from the state σ_0. Initially both the consumer and producer data spaces of the newly created agent are empty as it is not known which sorts it will produce or consume.

Data writing:

$$\frac{\sigma \xrightarrow{\mathit{wrt}(\langle \alpha:v,w\rangle)}_T \sigma'}{\{[\iota, n, \sigma, c, p]\} \xrightarrow{\{\langle \alpha:v,w\rangle\}} \{[\iota, n, \sigma', c +_{\mathtt{c}} \langle \alpha{:}v, w\rangle, p +_{\mathtt{p}} \langle \alpha{:}v, w\rangle]\}}$$

If a process executes the action $\mathit{wrt}(\langle \alpha{:}v, w\rangle)$ then the data element is inserted in its producer data space and its consumer data space if ι is subscribed to α and is broadcasted to the environment. All agents subscribed to α will receive the data element and insert it into their consumer data spaces (see the rule for process combination below).

Data subscription: If $\mathtt{sub}(\alpha) \notin c$ then

$$\frac{\sigma \xrightarrow{(\mathit{rd}(\alpha,q),\langle v,w\rangle)}_T \sigma' \text{ or } \sigma \xrightarrow{(\mathit{get}(\alpha,q),\langle v,w\rangle)}_T \sigma'}{\{[\iota, n, \sigma, c, p]\} \xrightarrow{\{\alpha \rhd \iota\}} \{[\iota, n, \sigma, c \cup \{\mathtt{sub}(\alpha)\}, p +_{\mathtt{p}} \alpha \rhd \iota]\}}$$

If a process executes the action $\mathit{rd}(\alpha, q)$ or $\mathit{get}(\alpha, q)$ but its agent is not yet subscribed to sort α then it subscribes to α and requires 'old' data elements of sort α from their producers. Notice that the agent does not change the state σ of the host process.

Data reading: If $\mathtt{sub}(\alpha) \in c$, $\langle v, w\rangle \models q$ and $\langle \alpha{:}v, w\rangle \in c$ then

$$\frac{\sigma \xrightarrow{(\mathit{rd}(\alpha,q),\langle v,w\rangle)}_T \sigma'}{\{[\iota, n, \sigma, c, p]\} \stackrel{\emptyset}{\rightarrow} \{[\iota, n, \sigma', c, p]\}}$$

If a process executes the action $\mathit{rd}(\alpha, q)$ and its agent is already subscribed to sort α and a data element of sort α which satisfies the query q is present in its consumer data space then it changes its state σ into σ'.

Data removal: If $\mathtt{sub}(\alpha) \in c$, $\langle v, w\rangle \models q$ and $\langle \alpha{:}v, w\rangle \in c$ then

$$\frac{\sigma \xrightarrow{(\mathit{get}(\alpha,q),\langle v,w\rangle)}_T \sigma'}{\{[\iota, n, \sigma, c, p]\} \stackrel{\emptyset}{\rightarrow} \{[\iota, n, \sigma', c \setminus \{\langle \alpha{:}v, w\rangle\}, p]\}}$$

If a process executes the action $\mathit{get}(\alpha, q)$ and the consumer data space of its agent contains a data element $\langle \alpha{:}v, w\rangle$ such that v and w satisfy the query q then the agent may change its state σ into the state σ'. The data element $\langle \alpha{:}v, w\rangle$ is removed from the consumer data space.

Request satisfaction: If $\alpha \rhd \iota' \in p$ then

$$\begin{aligned} &\{[\iota, n, \sigma, c, p], [\iota', n', \sigma', c', p']\} \stackrel{\emptyset}{\rightarrow} \\ &\qquad\qquad \{[\iota, n, \sigma, c, p \setminus \{\alpha \rhd \iota'\}], [\iota', n', \sigma', c'', p']\} \end{aligned}$$

where $c'' \in c' \oplus_{\mathtt{c}} \{\langle \alpha{:}v, w\rangle \mid \langle \alpha{:}v, w\rangle \in p\}$. An agent ι satisfies a request for data of sort α from another agent ι' by copying into the consumer data space of ι' all data of sort α stored in its producer data space. The request, being satisfied, is deleted from the producer data space of ι. Note that the agent ι may coincide with the agent ι'.

Process combination:

$$\frac{A_1 \stackrel{V}{\rightarrow} A_2}{A \uplus A_1 \stackrel{V}{\rightarrow} \overline{A} \uplus A_2}$$

where $\overline{A} \in A_V$. In this rule the data items and messages broadcasted by the agents in A_1 are inserted into the data spaces of the agents in A.

The above transition system assumes interleaving between agents. Therefore the set of broadcasted data is either empty or contains a single element. In a more realistic view there may be as many control threads as processes, and hence more data items and messages may be broadcasted at a time. This can be formalized by adding the following rule:

Multistep transitions:

$$\frac{A_1 \stackrel{V_1}{\rightarrow} A_1' \text{ and } A_2 \stackrel{V_2}{\rightarrow} A_2'}{A_1 \uplus A_2 \xrightarrow{V_1 \cup V_2} \overline{A_1'} \uplus \overline{A_2'}}$$

where $\overline{A_1'} \in (A_1')_{V_2}$ and $\overline{A_2'} \in (A_2')_{V_1}$. Notice that the above rule guarantees the exclusive access of data in the consumer data space c of an agent $[\iota, n, \sigma, c, p]$ by the host process since the insertion of data items and messages broadcasted by the other agents is executed only afterward the agent ι terminates its transition step.

The shared and the distributed view of the data space differ with respect to the multiple insertion of data items: conflicts are solved globally in the shared data space model while they are solved locally to each agent in the distributed data space model. For example, let V be a finite subset of *MData* containing more than one data element of the same sort and with the same index value, and assume A_2 contains two agents with an equal local data space. If $A_1 \stackrel{V}{\rightarrow} A_1'$ and $A_2 \stackrel{\emptyset}{\rightarrow} A_2$ then the local data space of the agents in A_2 may be different depending on the order they insert the data items of the same sort and index value in V.

Another difference between the shared and the distributed view of the data space may be caused by the rule 'request satisfaction' in case several producers of data with the same sort and index value are active at the same time: data elements of one producer may overwrite more recent ones of another producer if these are copied first into a consumer data space when satisfying a request for data.

Both the above problem and conflicts in the insertion of several data items may be solved if we require that at any moment no pair of data items of the same sort and with the same index value is broadcasted. This can be obtained if there is at most one process producing data items of a given sort and index value.

4 Non-functional system specifications

System design is a multidimensional problem of which functionality is only one dimension. In the above transition system specifications of SPLICE we took into account only the functionality and the distribution aspects of the system. A transition system semantics does not say much about others aspects like fault-tolerance and system upgrades. Therefore, in the following two subsections we briefly and informally discuss how SPLICE deals with them.

4.1 Fault-tolerance

In safety-critical systems, such as aircraft flight control systems, there is the need for redundancy in order to mask hardware failures during operation. Fault-tolerance in general is a very complex requirement to meet and can, of course, only be partially solved in software. In SPLICE, the agents have been refined to provide a mechanism for fault-tolerant behavior. By making fault-tolerance a property of the coordination model, the design complexity of applications is significantly reduced.

A full treatment of fault tolerance is beyond the scope of this paper, only two techniques will be briefly mentioned. The first makes use of the treatment of data as information in SPLICE. A process performing samples from continuous quantities of information may be replicated several time while the consumers of the multiply-produced data sorts need not to be aware of this multiplicity. If one producer fails the others will continue to produce the necessary data without the consumers will need to stop their executions. Because data items of the same sorts and index value overwrite each other, there is no interference among the replicated processes. Notice that, for this mechanism to work properly, the following assumptions should holds: data must be continuously produced and must not be catastrophic for the consumer if it is occasionally lost.

A second technique for fault tolerance in SPLICE makes use of the distributed data space view. If data elements of sort α need to be duplicated and used as back-up copies, then a process can simply be created which subscribes to data of sort α. In this way when another process inserts a data element of sort α, its agent additionally stores a copy of the element in more local data spaces than strictly necessary. When, in case of failure, a data element gets lost from a local data space, its agent can restore that data from one of the local data spaces which contain a copy of it.

An optimization to this scheme is possible to minimize the amount of data remotely stored. As mentioned before, control systems are mainly concerned with data that represents continuous quantities: data is either an observation sampled from the system's environment or derived from such samples. Consequently, the environment itself can be regarded as a back-up of the system's data. Simply through new observations the system is able to recover from a failure. Not all data in a control system represent continuous quantities, however. By distinguishing two classes of data, representing continuous and discrete quantities respectively, storage and communication overhead can be significantly reduced: the back-up processes only read data belonging to the discrete class.

4.2 System Modifications and Extensions

A major problem in the design of systems of the kind considered here is the need to provide for upgrades and other modifications, preferably while the existing system remains on-line. There are two distinct cases to be considered.

- The upgrade is an extension to the system, introducing new application processes and data sorts but without further modifications to the existing system.
- The upgrade includes modification of existing application processes.

As a consequence of our transition rule *process combination*, both for the shared and the distributed data space models, it is obvious that SPLICE can deal with the first case without further refinements. Simply starting a new application will be enough for it to automatically integrate in the already running system.

The second case, clearly, is more difficult. One special, but important, category of modifications can be handled by a simple refinement of the agents. Consider the problem of upgrading a system by replacing an existing application process with one that implements the same function, but using a better algorithm, leading to higher quality results. In many systems it is not possible to physically replace the old application with the new one, since this would require the system to be taken off-line.

By a refinement of the agents it is possible to support on-line replacement of application processes as follows. If an application process performs a *wrt* operation, its agent attaches an additional key field to the data sort instance representing the application's version number. Upon a read request, an agent now first checks whether multiple versions of the requested instance are available in the local data space. If this is the case, the instance having the highest version number is delivered to the application. From that moment on, all instances with lower version numbers, received from the same agent, are discarded. In this way an application process can be dynamically upgraded, simply by starting the new version of the application, after which it will automatically take over the role of the earlier version, and then stopping the older version.

5 Conclusion

We have presented a software architecture for large control systems that incorporates an explicit coordination model. We started from a relatively simple model based on a shared data space to arrive at a completely distributed model in order to meet the requirements that are typical for this class of systems.

SPLICE has been applied in the development of commercially available command-and-control, and traffic management systems. These systems consist of some 1000 application processes running on close to 100 processors interconnected by a hybrid communication network [6].

We formally defined the behaviour of SPLICE using transition systems in the style of Plotkin [20] for the shared data space model and the distributed data space model. Both transition systems are defined independently from the host languages in which computations take place. We have used an approach similar to the one adopted in [4] and different from the the syntax-based approach in [14]. The advantage is that SPLICE primitives can be embedded in several programming languages, possibly members of different classes of programming paradigms.

The configuration of both transition systems represent a set of several concurrently executing processes. This representation, introduced for a transition system semantics of a parallel object oriented language [2, 3], guarantees commutativity and associativity of the parallel composition. We used this representation because it allows for an easy description of dynamic process creation.

The rules specifying our transition systems reflect the fact that SPLICE is a distributed system since a transition can take place without any knowledge of the other processes active in the systems. The additional mathematical complexity of the second transition systems is compensated by a clear and full distribution of both data and processes. The treatment of data as information by SPLICE is specified by both the use of sets as data spaces and the semantics of the *rd* and *get* operations. These operations do not specify any mutual exclusion property for the selected data item.

There may be different ways to collect the information about the behaviour of a program from a transition system. Further work is needed to formalize an observational equivalence between SPLICE processes. The general semantic theory of asynchronous communication as formulated in [9] can be used to define an operational semantics which records the internal states of each active process. The semantics can abstract from finite repetition of states, that is, we can abstract from the concept of a global clock that is present in the transition system model. A global clock is unrealistic in a distributed implementation.

Acknowledgments: The authors are grateful to Jaco de Bakker, Jan Rutten, Farhad Arbab, Eric Monfroy, Adriano Scutellá, and all members of the Amsterdam Coordination Group for several fruitful discussions and suggestions on the contents of this paper. We would also like to thank Paul Dechering of Hollandse Signaalapparaten and the referees of this paper for their useful suggestions.

References

[1] J.-M. Andreoli, C.L. Hankin, and D. Le Métayer (eds.) *Coordination Programming, mechanisms, models and semantics.* Imperial College Press, London, 1996.

[2] P.H.M. America and J.W. de Bakker. Designing equivalent semantic models for process creation. In *Theoretical Computer Science* 60, pages 109–176, 1988.

[3] P.H.M. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Operational semantics of a parallel object-oriented language. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 194–208, 1986.

[4] F. Arbab, J.W. de Bakker, M.M. Bonsangue, J.J.M.M. Rutten, and A. Scutellá. A transition system semantics for a control-driven coordination language. In preparation.

[5] J.-P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. In *Science of comput¡er programming* 15, pages 55–77, 1990.

[6] M. Boasson. Control systems software. In *IEEE Transactions on Automatic Control* 38:7, pages 1094–1107, 1993.

[7] M. Boasson. Complexity may be our own fault. In *IEEE Software*, March 1993.

[8] M. Boasson (ed.). *Software Architecture.* In *IEEE Software*, November 1995.

[9] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. A paradigm for asynchronous communication and its application to concurrent constraint programming. In K.R. Apt, J.W. de Bakker and J.J.M.M. Rutten, editors, *Logic programming languages: constraints, functions, and objects*, Logic Programming Series, pages 82–114, MIT Press, 1993.

[10] N. Carriero and D. Gelernter. Linda in context. In *Communications of the ACM* 32:4, pages 444–458, 1989.

[11] N. Carriero and D. Gelernter. Coordination languages and their significance. In *Communications of the ACM* 35:2, page 97–107, 1992.

[12] K.M. Chandy and J. Misra. *Parallel Program Design: a Foundation.* Addison-Wesley, 1988.

[13] P. Ciancarini and C. Hankin (eds.). *Coordination Languages and Models, Proceedings of the first international conference COORDINATION'96*, Volume 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, 1996.

[14] P. Ciancarini, K. Jensen, and D. Yankelevich. On the operational semantics of a coordination language. In P. Ciancarini, O.Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995.

[15] D. Garlan and D. Le Métayer (eds.). *Coordination Languages and Models, Proceedings of the second international conference COORDINATION'97*, Volume 1282 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997.

[16] H. Goeman, J.N. Kok, K. Sere and R. Udink. Coordination in the ImpUnity framework. In [13], pages 215-230, 1996.

[17] K. Jackson and M. Boasson. The importance of good architectural style. In *Proceedings of the workshop of the IEEE on Engineering of Computer Based Systems*, Tucson, 1995.

[18] J.-M. Jacquet and K. De Bosschere. On the semantics of *μLog*. In *Future Generation Computer Systems*, 1994.

[19] G.A. Papadopoulos and F. Arbab. Coordination models and languages. In *Academic Press* 48, 1998.

[20] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[21] G.-C. Roman and H.C. Cunningham. Mixed programming metaphors in a shared data space model of concurrency. In *IEEE Transactions of Software Engineering* 16:12, pages 1361–1373, 1990.

[22] V.A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, 1993.

Biographical notes

Marcello Bonsangue is a post doctoral researcher in Computer Science at Leiden University. His research interests include operational, denotational, and axiomatic semantics of programming languages, program refinement, mathematical structures for semantics, and topology.

Joost Kok is Full Professor of Computer Science at Leiden University. His research interests include intelligent data analysis, algorithms for optimization and prediction, neural networks, evolutionary computation, program development, and semantics of parallel and distributed systems. He is currently the president of the Dutch/Belgium AI society.

Maarten Boasson is head of the Applied Systems Research department at Signaal. He is the intellectual father of the SPLICE architecture, development of which began roughly 20 years ago. In addition to his industrial activities, he is Full Professor of Computer Science at the University of Amsterdam, where he specializes in distributed reactive systems.

Edwin de Jong is research manager at the Applied Systems Research department at Signaal. His research interests include distributed reactive systems, software architecture, coordination and computation models, and application of formal methods in requirements specification, design, and implementation.