Equivalent Semantic Models for a Distributed Dataspace Architecture

Jozef Hooman[1,2] and Jaco van de Pol[2]

[1] University of Nijmegen, Nijmegen, The Netherlands
hooman@cs.kun.nl
http://www.cs.kun.nl/~hooman/
[2] CWI, Amsterdam, The Netherlands
Jaco.van.de.Pol@cwi.nl
http://www.cwi.nl/~vdpol/

Abstract. The general aim of our work is to support formal reasoning about components on top of the distributed dataspace architecture Splice. To investigate the basic properties of Splice and to support compositional verification, we have defined a denotational semantics for a basic Splice-like language. To increase the confidence in this semantics, also an operational semantics has been defined which is shown to be equivalent to the denotational one using the theorem prover PVS. A verification framework based on the denotational semantics is applied to an example of top-down development and transparent replication.

1 Introduction

The general aim of our work is to support the development of complex applications on top of industrial software architectures by means of formal methods. As a particular example, we consider in this paper components on top of the software architecture Splice [Boa93,BdJ97] which has been devised at Thales Nederland (previously called Hollandse Signaalapparaten). It is used to build large and complex embedded systems such as command and control systems, process control systems, and air traffic management systems.

The main goal of Splice is to provide a coordination mechanism between loosely-coupled heterogeneous components. In typical Splice-applications it is essential to deal with large data streams from sensors, such as radars. Hence Splice should support real-time and high-bandwidth distribution of data. Another aim is to support fault-tolerance, e.g., it should be possible to replicate components transparently, i.e. without affecting the overall system behaviour.

Splice is data-oriented, with distributed local databases based on keys. It uses the publish-subscribe paradigm to get loosely-coupled data producers and consumers. An important design decision is to have minimal overhead for data management, allowing a fast and cheap implementation that indeed allows huge data streams. For instance, Splice has no standard built-in mechanisms to ensure global consistency or global synchronization. If needed, this can be constructed for particular data types on top of the Splice primitives. Note that this is quite

F.S. de Boer et al. (Eds.): FMCO 2002, LNCS 2852, pp. 182–201, 2003.

different from Linda [Gel85] and JavaSpaces [FHA99], which have a central data storage. The latter also has a transaction mechanism to handle distributed access.

Our aim is to reason about components of the distributed dataspace architecture Splice in a *compositional* way. This means that we want to deduce properties of the parallel composition of Splice-components using only the specifications of the externally visible behaviour of these components. Compositionality supports verification of development steps *during* the development process.

Many examples in the literature show that it is convenient to specify components using explicit assumptions about the environment. Concerning Splice, in [HH02] we propose a framework with an explicit assumption about the quality of data streams published by environment and a similar commitment of the component about its produced data. When putting components in parallel, assumptions can be discharged if they are guaranteed by other components. Reasoning with assumption/commitment [MC81] or rely/guarantee [Jon83] pairs, however, easily leads to unsound reasoning. There is a danger of circular reasoning, two components which mutually discharge each others assumptions, leading to incorrect conclusions. Hence it is important to prove the soundness of the verification techniques. Correctness of compositional verification rules is usually based on a denotational semantics which assigns a meaning to compound constructs based on the meaning of its constituents. Earlier work on the verification of Splice-systems [HvdP02] was based on a complex semantics with environment actions and its correctness was not obvious.

In this paper we define a denotational semantics for a simple Splice-like language which is more convenient as a basis for compositional reasoning using assumptions about the environment. It is, however, far from trivial that this semantics captures the intuitive understanding of the Splice architecture. Hence, we also provide a more intuitive operational semantics and prove formally that it is equivalent to the denotational one. Moreover, we show by a small example of transparent replication how the denotational framework can be used to support verification.

Both versions of the semantics have been formulated in the language of the interactive theorem prover PVS [OSRSC01]. The equivalence result has been checked completely using PVS. Also the example application (transparent replication) has been verified in detail using PVS.

Related to our semantic study is earlier work on the semantics of Splice-like languages such as a transition system semantics for a basic language of write and read statements (without query) [BKBdJ98] and a comparison of semantic choices using an operational semantics [BKBdJ98,BKZ99]. In previous work on a denotational semantics for Splice [BHdJ00] the semantics of local storages is not very convenient for compositional verification; it is based on process identifiers and a partial order of read and write events with complex global conditions.

New in this paper is an operational semantics that deals with local time stamps and their use for updating local databases. We define an equivalent denotational semantics which includes assumptions about the environment of a

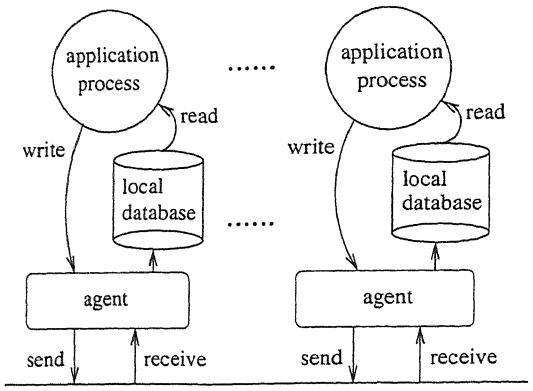

Fig. 1. Splice applications.

component. Moreover, we show that this denotational semantics forms the basis of a formal framework for specifying and verifying Splice applications.

This paper is structured as follows. Section 2 contains a brief informal explanation of Splice. In Sect. 3 we present a formal syntax of the Splice primitives considered in this paper. The operational and denotational semantics of this language are defined in Sect. 4 and Sect. 5, respectively. The main outline of the equivalence proof can be found in Sect. 6. A framework for specification and verification is described in Sect. 7 and applied to an example with top-down design and transparent replication in Sect. 8. Finally, Sect. 9 contains a number of concluding remarks.

2 Informal Splice Introduction

The Splice architecture provides a coordination mechanism for concurrent components. Producers and consumers of data are decoupled. They need not know each other, and communicate indirectly via the Splice primitives; basically *read*- and *write*- operations on a distributed dataspace. This type of anonymous communication between components is strongly related to coordination languages such as Linda [Gel85] and JavaSpaces [FHA99]. These languages, however, have a single shared dataspace, whereas in Splice each component has its own dataspace, see Fig. 1. Communication between components takes place by means of local agents. A data producer writes data records to the other dataspaces via its agent. A data consumer uses its agent to subscribe to the required types of data; only data which matches this subscription is stored. Data items may be delayed and re-ordered and sometimes may even get lost. It is possible to associate certain quality-of-service policies with data delivery and data storage. For instance, for a particular data type, delivery maybe guaranteed (each item is delivered at least once) or best effort (zero or more times). Data storage can be volatile, transient, or persistent.

Each data item within Splice has a unique *sort*, specifying the fields the sort consists of and defining the *key* fields [BdJ97]. In each local dataspace, at most one data item is present for each key. Basically, a newly received data item overwrites the current item with the same key (if any). To avoid that old data items overwrite newer information (recall that data may be delayed and re-ordered), data records include a *time stamp* field. A time stamp of a data item is obtained from the local clock of the data producer when the item is published. At the local storage of the consumer, data items are only overwritten if their time stamp is smaller than that of a newly arrived item (with the same key). This overwriting technique reduces memory requirements and allows a decoupling of frequencies between producers and consumers. It also reduces the number of updates to be performed on the dataspace, as not all received records get stored. The timestamps improve the quality of the data stored, as no record can be overwritten by older data.

To program components on top of Splice, a Splice API can be called within conventional programming languages such as C and Java. Splice provides, for instance, constructs for retrieving (reading) data from the local dataspace, and for publishing (writing) data. Read actions contain a query on the dataspace, selecting data items that satisfy certain criteria. Data has a life-cycle attribute, such as "read", "unread", "new", and "update", which may be used for additional filtering.

3 Syntax of a Simple Splice-Like Language

In this section we define the formal syntax of a very simple Splice-like language. We have embedded the basic Splice primitives in a minimal programming language to be able to high-light the essential features and to prove equivalences between various semantic definitions in a formal way. In Sect. 7.1 we show that it is easy to extend the language with, e.g. if-then-else and loop constructs. Life-cycle attributes are not included in the current version, because we detected some problems with the informal definition (as mentioned in Sect. 9).

We consider only one sort. Let $Data$ be some data domain, with a set $KeyData$ of key data and a function key: $Data \to KeyData$. Assume a given type $LocalTime$ to represent values of local clocks (in our PVS representation we choose the natural numbers).

The type $DataItems$ of time-stamped data items, consists of records with two fields: dat of type $Data$ and ts of type $LocalTime$. A record of type $DataItems$ can be written as $(\#dat := ..., ts := ...\#)$, following the PVS notation. Hence, for $di \in DataItems$, we have $dat(di) \in Data$ and $ts(di) \in LocalTime$. The function key is extended to $DataItems$ by $key(di) = key(dat(di))$.

Let $SVars$ be a set of variables ranging over sets of elements from $DataItems$. A *data expression* e yields an element from $Data$ (dependent on the current value of the program variables). Henceforth we typically use the following variables ranging over the types mentioned above:

- $d, d_0, d_1, \ldots$ over $Data$
- $lt, lt_0, lt_1, \ldots$ over $LocalTime$
- $dati, dati_0, dati_1, \ldots$ over $DataItems$
- $diset, diset_0, diset_1, \ldots$ over sets of $DataItems$
- $x, x_0, x_1, \ldots, y, y_0, y_1, \ldots$ over $SVars$

A *query* $q \subseteq \mathcal{P}(diset)$ specifies sets of (time-stamped) data items (it may depend on program variables). A few examples:

- $q_1 = \{diset \mid$ for all $dati \in diset$: $ts(dati) > 100\}$
- $q_2 = \{diset \mid$ for all $dati \in diset$: $dat(dati) \neq 0\}$
- $q_2 = \{diset \mid diset \neq \emptyset$ and $diset \neq x\}$

For simplicity, we do not give the syntax of data expressions and queries h
The syntax of our programming language is given in Table 1.

Table 1. Syntax Programming Language.

Sequential program	$S ::= \texttt{Write}(e) \mid \texttt{Read}(x, q) \mid S_1 \,;\, S_2$
Process	$P ::= S \mid P_1 \parallel P_2$

Informally, the statements of this language have the following meaning:

- `Write`(e) publishes the data item with value e (in the current state) and current time stamp (from the local clock).
  We model *best effort* delivery; a data item arrives 0 or more times at e process, where it might be used to update the local storage if there is current value with the same key which has a larger or equal time stamp
- `Read`(x, q) assigns to x some *set* of data items from local storage that satis query q (if there are several sets satisfying q, the choice is non-determinist For instance, for query q_1 above, we assign to x a set of data items fi local storage such that each data item has time stamp greater than 10(there are no such items in local storage, x becomes the empty set. Note t query q_3 above does not allow the empty set; then the read statement blc until the local storage contains a set of items satisfying the query. Henc read statement may be blocking, depending on the query.
- $S_1 \,;\, S_2$: sequential composition of sequential programs S_1 and S_2.
- $P_1 \parallel P_2$: parallel composition of processes. A process is a sequential progr or a parallel composition of processes.

As a very simple example, consider a few producers and consumers of fli data. Let $Data$ be a record with two fields: *flightnr* (a string, e.g. $KL309$) *pos* a position in some form, here a natural number for simplicity. The fli number is the key, that is, $key(dati) = flightnr(dati)$. Consider a produce flight data

$P_1 = \texttt{Write}((\#\mathit{flightnr} := KL567, \mathit{pos} := 1\#))$;
$\texttt{Write}((\#\mathit{flightnr} := LU321, \mathit{pos} := 6\#))$;
$\texttt{Write}((\#\mathit{flightnr} := KL567, \mathit{pos} := 2\#))$;
$\texttt{Write}((\#\mathit{flightnr} := KL567, \mathit{pos} := 3\#))$
and two consumers:
$C_1 = \texttt{Read}(x_1, \mathit{true})$; $\texttt{Read}(y_1, q_1)$; $\texttt{Read}(z_1, q_1)$
$C_2 = \texttt{Read}(x_2, q_1)$; $\texttt{Read}(y_2, q_2)$
whose queries are specified as follows:

$q_1 = \{\mathit{diset} \mid \mathit{diset} \neq \emptyset$ and for all $\mathit{dati} \in \mathit{diset} : \mathit{flightnr}(\mathit{dati}) = KL567\}$
$q_2 = \{\mathit{diset} \mid \mathit{diset} \neq \emptyset$ and for all $\mathit{dati} \in \mathit{diset} : \mathit{flightnr}(\mathit{dati}) = KL567$ and for all $\mathit{dati}_1 \in x_2 : ts(\mathit{dati}) > ts(\mathit{dati}_1)\}$

Consider the process $P_1 \parallel C_1 \parallel C_2$ and assume there are no other producers of data. Note that the producer does not specify the local time stamp explicitly; this is added implicitly. Recall that the items produced by P_1 may arrive at a different order at the consumers, and they may arrive several times.

Variable x_1 may be empty (if no data item has been delivered yet – note that this read is not blocking) or a set with one or two elements, at most one for each flight number. For instance, it may contain position number 3 for $KL567$. Variable y_1 will be a singleton, since the local storage contains at most one item with flight number $KL567$ and the second read is blocking (the query requires a non-empty set). If there is a position for $KL567$ in x_1, then the position in y_1 will be greater or equal (lower values are produced earlier, hence have a smaller local clock value, and thus they cannot overwrite greater values). Similarly for z_1, where the position is greater or equal than the one in y_1. It is possible that $z_1 = y_1$. For consumer C_2 the second read action requires a newer time stamp, hence we always have $y_2 \neq x_2$ and the position in y_2 is at least 2.

4 Operational Semantics

We define an operational semantics for a process $S_1 \parallel ... \parallel S_n$ of the syntax of Sect. 3. where the S_i are sequential programs. We first define an operational status of a sequential process (Def. 1) and a configuration (Def. 3) which represents the state of affairs during operational execution of a process. Next computation steps are defined (Def. 5), leading to the operational semantics (Def. 6).

Let $\mathit{DataBases}$ be the type consisting of sets of data items with at most one item for each key, i.e.
$\mathit{DataBases} = \{\mathit{diset} \subseteq \mathit{DataItems} \mid$ for all $\mathit{dati}_1, \mathit{dati}_2 \in \mathit{diset}$:
$\mathit{key}(\mathit{dati}_1) = \mathit{key}(\mathit{dati}_2) \rightarrow \mathit{dati}_1 = \mathit{dati}_2\}$

Definition 1 (Operational Status). An operational status of a sequential process, denoted os, os_0, os_1, .., is a record with three fields, st, clock and db:

- $st : \mathit{SVars} \rightarrow \wp(\mathit{DataItems})$, represents the local state, assigning to each variable a set of data items;
- $\mathit{clock} \in \mathit{LocalTime}$, the value of the local clock;

- $db \in DataBases$, the local database, a set of data items (at most one for each key) representing local storage.

Definition 2 (Variant). The *variant* of local state st with respect to variable $x \in SVars$ and value $diset \subseteq DataItems$, denoted

$st[x := diset]$, is defined as $(st[x := diset])(y) = \begin{cases} diset \text{ if } y = x \\ st(y) \text{ if } y \neq x \end{cases}$

Similarly, the variant of a record r with fields $f_1 \ldots f_m$ is defined by

$r[f := v](f_i) = \begin{cases} v & \text{if } f_i = f \\ f_i(r) & \text{if } f_i \neq f \end{cases}$

Produced data items are sent to an underlying network. Here this is represented by N, a set of data items, i.e. $N \subseteq DataItems$. Note that we do not use a multiset, although a particular item might be produced several times. The use of a set is justified by the fact that data items may always be delivered more than once and hence are never deleted from N.

Definition 3 (Configuration). The state of affairs of a process $S_1 \parallel ... \parallel S_n$ during execution is represented by a *configuration*:

$\langle (S'_1, os_1), ..., (S'_n, os_n), N \rangle$

It denotes for each sequential process S_i, the status os_i and the remaining part S'_i that still has to be executed, and the current contents N of the network. For convenience, we introduce the empty statement E indicating that the process has terminated.

An execution of $S_1 \parallel ... \parallel S_n$ is represented by sequence of configurations

$C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow ...$

where $C_0 = \langle (S_1 ; E, os_1), ..., (S_n ; E, os_n), \emptyset \rangle$ with, for all i, $db(os_i) = \emptyset$.
The idea is that each step in the sequence $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow ...$ represents the execution of an *atomic* action by some process i. We define the update of a database.

Definition 4 (Update Database). The update of database db using a new database db_1, denoted $UpdateDb(db, db_1)$ is defined by
$dati \in UpdateDb(db, db_1)$ iff

- either $dati \in db$ and for all $dati_1 \in db_1$ with $key(dati_1) = key(dati)$ we have $ts(dati_1) \leq ts(dati)$,
- or $dati \in db_1$ and for all $dati_0 \in db$ with $key(dati_0) = key(dati)$ we have $ts(dati_0) < ts(dati)$.

Computation steps are defined formally as follows.

Definition 5 (Computation Step). We have a step of process i, i.e.,
$\langle (S_1, os_1), .., (S_i, os_i), .., (S_n, os_n), N \rangle \longrightarrow \langle (S_1, os_1), .., (S'_i, os'_i), .., (S_n, os_n), N' \rangle$
iff one of the following clauses holds:

(Update) $S_i \neq E$ and $S'_i = S_i$, $N' = N$, $st(os'_i) = st(os_i)$, $clock(os'_i) = clock(os_i)$, and there exists a database $db_1 \subseteq N$ that is used to update

$db(os_i)$ such that an element of db_1 is added if its key is not yet present and, otherwise, it replaces an element of $db(os_i)$ with the same key if its local time-stamp is strictly greater. Formally, using Def. 4, there exists a database $db_1 \subseteq N$ such that $db(os_i') = UpdateDb(db(os_i), db_1)$.
For simplicity, we did not change the local clock (it is not needed), but alternatively we might require $clock(os_i') \geq clock(os_i)$.
Note that the network has not been changed, since data items might be used several times for an update (modeling the fact that an item might be delivered by the network several times).

(**Write**) $S_i = \texttt{Write}(e)\ ;\ S_i'$, $st(os_i') = st(os_i)$, $db(os_i') = db(os_i)$, $clock(os_i') > clock(os_i)$, and $N' = N \cup \{(v, clock(os_i))\}$ where v is the value of e in the current state.
Note that, given a syntax of expressions, it would be easy to define the values of expressions and queries in the current status (see, e.g. [dRdBH+01]).

(**Read**) $S_i = \texttt{Read}(x, q)$; S_i', $N' = N$, $db(os_i') = db(os_i)$, $clock(os_i') = clock(os_i)$, and there exists a set of data items $diset \subseteq db(os_i)$ satisfying query q and such that $st(os_i') = st(os_i)[x := diset]$, i.e. assigning $diset$ to x.

For two configurations C_1 and C_2, define $C_1 \longrightarrow^k C_2$, for $k \in \mathbb{N}$, by $C_1 \longrightarrow^0 C_1$ and $C_1 \longrightarrow^{k+1} C_2$ iff there exists a configuration C such that $C_1 \longrightarrow^k C$ and $C \longrightarrow C_2$. Define $C_1 \longrightarrow^* C_2$ iff there exists a $k \in \mathbb{N}$ such that $C_1 \longrightarrow^k C_2$.

Typically, the operational semantics yields some abstraction of the execution sequence, depending on what is *observable*. Here we postulate that only the set of produced data items in the last configuration of an execution sequence is (externally) observable.

Definition 6 (Operational Semantics). The operational semantics of a program $S_1 \parallel ... \parallel S_n$, given an initial operational status os_0, is defined by
$\mathcal{O}(S_1 \parallel ... \parallel S_n)(os_0) = \{N \subseteq DataItems \mid$
$\langle (S_1\ ;\ E, os_0), .., (S_n\ ;\ E, os_0), \emptyset\rangle \longrightarrow^* \langle (E, os_1), .., (E, os_n), N\rangle,$
with $db(os_0) = \emptyset\ \}$

Thus the operational semantics of a program yields a set of sets of produced data items, where each set of produced data items represents a possible execution of the program.

Example 1. Let 0 be the query specifying that the value 0 should be read, similarly for 1. Observe that, for any os_0,
$\mathcal{O}((\texttt{Read}(x, 0)\ ;\ \texttt{Write}(1)) \parallel (\texttt{Read}(x, 1)\ ;\ \texttt{Write}(0)))(os_0) = \emptyset$.

5 Denotational Semantics

We define the denotational semantics of a program, given an initial status, i.e. the state of affairs when execution starts. To support our aim to reason with assumptions about the items produced by the environment, such assumptions are

included in the status. The semantics yields a set of statuses, each representing a possible execution of the program.

To achieve compositionality and to describe a process in isolation, it is quite common that information has to be added to the status to express relations with the environment explicitly. Here we add, e.g. the set of written data items and the set of items that are assumed to be produced by the environment.

Moreover, we need a way to represent causality between the written items. As shown below, this is needed to assign a correct meaning to the program $(\texttt{Read}(x,0);\texttt{Write}(1)) \parallel (\texttt{Read}(x,1);\texttt{Write}(0))$. Here this is achieved by the use of conceptual logical clocks that are added to the status of each process. They are updated similarly to Lamport's logical clocks [Lam78], which ensures that there exists a global, total order on the produced items.

Let $Time$ be the domain of logical clock values, here we use the natural numbers. We add a logical clock value to the data items produced. Type $ExtDataItems$, with typical variables $edi, edi_0, edi_1, \ldots$, consists of records with two fields:

- di of type $DataItems$, and
- tm of type $Time$.

Field tm represents the logical moment of publication. It can be used to construct a global partial order on the produced data items that reflects causality.

Field selector di is extended to remove the logical clock values from a set of extended data items. For a set $ediset \subseteq ExtDataItems$ define
$di(ediset) = \{di(edi) \mid edi \in ediset\}$.

A status, typically denoted by s, s_0, s_1, .., representing the current state of affairs of a program, is a record with six fields. In addition to the three fields of the operational status:

- $st : SVars \to \wp(DataItems)$, the local state (values of variables);
- $clock \in LocalTime$, the value of the local clock;
- $db \in DataBases$, the local database (a set of data items, unique per key);

we have three new fields:

- $time \in Time$, the logical clock, to represent causality;
- $ownw \subseteq ExtDataItems$, the set of extended data items written by the program itself in the past;
- $envw \subseteq ExtDataItems$, the set of extended data items written by the environment of the program; this is an assumption about all items produced (including present and future).

Below we define a meaning function $\mathcal{M}$ for programs by induction on their structure. The possible behaviour of a program $prog$, i.e. a set of statuses, is defined by $\mathcal{M}(prog)(s_0)$, where s_0 is the initial status at the start of program execution. Note that this includes an assumption about all data items that have been or will be produced by the environment. The semantics will be such that if $s \in \mathcal{M}(prog)(s_0)$ then

- $ownw(s)$ equals the union of $ownw(s_0)$ and the items written by *prog*.
- $envw(s) = envw(s_0)$; the field *envw* is only used by *prog* to update its local storage. Although all items are available initially, constraints on logical clocks prevent the use of items "too early".

Next we define $\mathcal{M}(prog)$ by induction on the structure of *prog*.

Write. In the semantics of the write statement the published item, extended with the current value of the local clock, is added to the *ownw* field. The local clock is increased to ensure that subsequent written items get a later time stamp. In a similar way, a logical clock value has been added.

$\mathcal{M}(\texttt{Write}(e))(s_0) =$

$\{s \mid clock(s) > clock(s_0), time(s) > time(s_0),$
$ownw(s) = ownw(s_0) \cup \{(\#di := dati, tm := time(s)\#)\}$, where
$dati = (\#dat := v, ts := clock(s_0)\#)$, with v the value of e in s_0,
and s equals s_0 for the other fields (*st*, *db* and *envw*) $\}$

Read. To define the meaning of a read statement, we first introduce an auxiliary *Update* function which may update the local database with data items written (by the process itself or by its environment), using *UpdateDb* of Def. 4.

$Update(s_0) =$

$\{s \mid$ there exists an $ediset \subseteq ownw(s_0) \cup envw(s_0)$
and a database $db_1 \subseteq di(ediset)$, such that
$db(s) = UpdateDb(db(s_0), db_1)$,
$time(s) >= time(s_0)$,
for all $edi \in ediset$, we have $tm(edi) < time(s)$, and
s equals s_0 for the other fields (*st*, *clock*, *ownw* and *envw*) $\}$

Then the read statement $\texttt{Read}(x, q)$ first updates the local storage and next assigns to x a set of data items that satisfies the query q.

$\mathcal{M}(\texttt{Read}(x, q))(s_0) =$

$\{s \mid$ exists $s_1 \in Update(s_0)$ and $diset \subseteq db(s_1)$ such that
diset satisfies query q and $st(s) = st(s_1)[x := diset]$, and
s equals s_1 for the other fields (*clock*, *db*, *time*, *ownw* and *envw*) $\}$

Note that we only represent terminating executions; blocking has not been modeled explicitly.

Sequential Composition. Since we only model terminating executions, the meaning of the sequential composition $S_1 ; S_2$ is defined by applying the meaning of S_2 to any status that results from executing S_1. In Sect. 7.1 we show how this can be extended to deal with non-terminating programs.

$\mathcal{M}(S_1 ; S_2)(s_0) =$

$\{s \mid$ exists s_1 with $s_1 \in \mathcal{M}(S_1)(s_0) \wedge s \in \mathcal{M}(S_2)(s_1)\}$

Parallel Composition. To define parallel composition, let $init(s_0)$ be the condition $db(s_0) = \emptyset \wedge ownw(s_0) = \emptyset$. Moreover, we use $s + ediset$ to add a set

$ediset \subseteq ExtDataItems$ to the environment writes of s, i.e. $envw(s + ediset) = envw(s) \cup ediset$ and all other fields of s remain the same.

In the semantics of $P_1 \parallel P_2$, starting in initial status s_0, the main observatio is that $envw(s_0)$ contains only the data items produced outside $P_1 \parallel P_2$. Henc the semantic function for P_1 is applied to s_0 where we add the items writte by P_2 to the environment writes. Similarly for P_2. Then parallel composition i defined as follows:

$\mathcal{M}(P_1 \parallel P_2)(s_0) =$

$\{s \mid init(s_0)$ and there exist s_1 and s_2 with
$s_1 \in \mathcal{M}(P_1)(s_0 + ownw(s_2))$,
$s_2 \in \mathcal{M}(P_2)(s_0 + ownw(s_1))$,
$ownw(s) = ownw(s_1) \cup ownw(s_2)$, $envw(s) = envw(s_0)\}$

Example 2. Consider again the program of Example 1:

$(\texttt{Read}(x, 0)\,;\, \texttt{Write}(1)) \parallel (\texttt{Read}(x, 1)\,;\, \texttt{Write}(0))$

Without using logical clocks, the semantics of parallel composition would al low for this program a status where $envw = \emptyset$ and $ownw$ contains 0 and 1 (each component produces the item required by the other one). This, however does not correspond to the operational semantics which yields the empty set In the current version of the semantics, we can indeed show that, for any s_0 $\mathcal{M}((\texttt{Read}(x, 0)\,;\, \texttt{Write}(1)) \parallel (\texttt{Read}(x, 1)\,;\, \texttt{Write}(0)))(s_0) = \emptyset$.

6 Equivalence of Denotational and Operational Semantics

In this section we first define what it means that the operational and the denotational semantics of Sect. 4 and 5, resp., are equivalent. Next we give an outline of how we proved this equivalence formally.

Note that equivalence is far from trivial, since there a number of prominent differences:

- The operational semantics allows updates of the local database at any point in time, whereas in the denotational semantics we minimized the number of updates to keep verification simple, allowing it only once, immediately before reading items.
- The parallel composition of the denotational semantics is defined by a few recursive equations and it is not clear whether this indeed corresponds to operational execution.
- The denotational semantics has additional fields in a status and the read statement contains an additional check on logical clock values.
- The underlying network is modeled in different ways.

Equivalence is based on what is externally *observable*, i.e. two semantics functions are equivalent if they assign the same observable behaviour to any program. For the denotational semantics, we choose the same notion of observable behaviour as has been used in the operational semantics, namely the set of published data items. For a set D of denotational statuses, define the observations

by $Obs(D) = \{di(ownw(s)) \mid s \in D\}$. Define for an n-tuple $(s_1, \ldots, s_n)$ of statuses $Obs(s_1, \ldots, s_n) = di(\cup_{i,1 \leq i \leq n} ownw(s_i))$, and for a set T of such tuples $Obs(T) = \{Obs(s_1, \ldots, s_n) \mid (s_1, \ldots, s_n) \in T\}$.

To relate the operational and the denotational semantics, we use a function Ext to extend an operational status to a status of the denotational semantics; $Ext(os)$ copies the fields *st*, *clock*, and *db* of os and it sets the fields *time* to 0 and *ownw* and *envw* to $\emptyset$. This leads to the main theorem.

Theorem 1. $\mathcal{O}(S_1 \parallel \ldots \parallel S_n)(os) = Obs(\mathcal{M}(S_1 \parallel (S_2 \parallel (\ldots \parallel S_n)\ldots))(Ext(os)))$

We give an outline of the proof that has been checked completely using the interactive theorem prover PVS. Below we only give the main steps, for instance, ignoring details about initial conditions. The proof uses three intermediate versions of the semantics and corresponding lemma's:

- $\mathcal{M}'$ which is the same as $\mathcal{M}$ except that also the write-statement is preceded by an update action.

  Lemma 1. $Obs(\mathcal{M}(S_1 \parallel (S_2 \parallel (\ldots \parallel S_n)\ldots))(Ext(os))) =$
  $Obs(\mathcal{M}'(S_1 \parallel (S_2 \parallel (\ldots \parallel S_n)\ldots))(Ext(os)))$

  Note that $\mathcal{M}$ and $\mathcal{M}'$ are defined for the parallel composition of two processes. But we derive a similar formulation for n processes:

  Lemma 2. $Obs(\mathcal{M}'(S_1 \parallel (S_2 \parallel (\ldots \parallel S_n)\ldots))(Ext(os))) =$
  $Obs(\{(s_1, \ldots, s_n) \mid init(Ext(os)), envw(Ext(os)) = \emptyset$ and
  for all i, $1 \leq i \leq n$, $s_i \in \mathcal{M}'(S_i)(Ext(os)[envw := \cup_{j \neq i} ownw(s_j)])\ \})$

- $\mathcal{OD}$ which extends the operational semantics $\mathcal{O}$ to the status of the denotational semantics (adding *time*, *ownw* and *envw*). Moreover, the network N is removed. This is achieved by defining the atomic steps of a single sequential process as $(S, s) \longrightarrow_{ediset} (S', s')$, where *ediset* represents the set of items written in the step (a singleton if S starts with a write statement, the empty set otherwise). This also includes update steps, similar to the update inside a read statement of the denotational semantics, so including a condition of the values of logical clocks.
  Based on this step relation for one process, we have a step of n parallel processes, denoted $\langle (S_1, s_1), \ldots, (S_n, s_n)\rangle \longrightarrow \langle (S_1', s_1'), \ldots, (S_n', s_n')\rangle$, if there exists an i with $(S_i, s_i) \longrightarrow_{ediset} (S_i', s_i')$, and for all $j \neq i$, $S_j' = S_j$ and $s_j' = s_j + ediset$.
  This leads to the following semantics for n processes:
  $\mathcal{OD}(S_1 \parallel \ldots \parallel S_n)(s_0) = \{(s_1, \ldots s_n) \mid init(s_0) \wedge$
  $\langle (S_1\ ;\ E, s_0), \ldots, (S_n\ ;\ E, s_0)\rangle \longrightarrow^* \langle (E, s_1), \ldots, (E, s_n)\rangle\ \}$

  Lemma 3. $\mathcal{O}(S_1 \parallel \ldots \parallel S_n)(os) = Obs(\mathcal{OD}(S_1 \parallel \ldots \parallel S_n)(Ext(os))$

- $\mathcal{OS}$ which extends the operational semantics of a sequential process to the status of the denotational semantics, using the relation $(S, s) \longrightarrow_{ediset} (S', s')$ mentioned above.

  Lemma 4. $\mathcal{OS}(S) = \mathcal{M}'(S)$, for sequential programs S.

Now, by the lemmas 1, 2, 3, and 4, theorem 1 reduces to the following lemma.

Lemma 5. $Obs(\mathcal{OD}(S_1 \parallel ... \parallel S_n)(Ext(os))) =$
$Obs(\{(s_1,\ldots,s_n) \mid init(Ext(os)), envw(Ext(os)) = \emptyset$ and
for all i, $1 \leq i \leq n$, $s_i \in \mathcal{OS}(S_i)(Ext(os)[envw := \cup_{j \neq i} ownw(s_j)])\})$

Proof. We give the main ideas of the proof, showing that the sets are contained in each other.
$\subseteq$
Suppose $(s_1,\ldots s_n) \in \mathcal{OD}(S_1 \parallel ... \parallel S_n)(Ext(os))$, that is,
$\langle (S_1\,;\, E, Ext(os)), ..., (S_n\,;\, E, Ext(os))\rangle \longrightarrow^* \langle (E, s_1), ..., (E, s_n)\rangle$.
We have proved by induction on the number of steps in the execution sequence that this implies, for all i, $1 \leq i \leq n$,
$(S_i\,;\, E, Ext(os)[envw := \cup_{j \neq i} ownw(s_j)]) \longrightarrow_{ediset_1} \cdots \longrightarrow_{ediset_k} (E, s_i)$,
that is, $s_i \in \mathcal{OS}(S_i)(Ext(os)[envw := \cup_{j \neq i} ownw(s_j)])$.
$\supseteq$
Assume, for all i, $1 \leq i \leq n$ that $s_i \in \mathcal{OS}(S_i)(Ext(os)[envw := \cup_{j \neq i} ownw(s_j)])$.
Thus we have an operation execution
$(S_i\,;\, E, Ext(os)[envw := \cup_{j \neq i} ownw(s_j)]) \longrightarrow_{ediset_1} \cdots \longrightarrow_{ediset_k} (E, s_i)$
for each of the sequential processes. We have to show,
$\langle (S_1\,;\, E, Ext(os)), ..., (S_n\,;\, E, Ext(os))\rangle \longrightarrow^* \langle (E, s_1), ..., (E, s_n)\rangle$,
i.e., we have to show that these sequential executions can be merged into a global execution sequence for the parallel program.
Basically, this is done by induction on the total number of steps in all sequential executions. The global execution sequence is constructed by selecting for each step the process with the lowest logical time after its next possible step. This construction is far from trivial, since the local, sequential executions start with all available environment writes, whereas in the global execution a process can only use what has been produced up to the current moment. However, the constraints on the logical clocks (that have been included in this extended operational semantics), ensure that only items produced before its current logical time are used. Formally, this is captured by the property that if $(S, s{+}ediset) \longrightarrow_{ediset_1} (S', s')$ (representing a step of a local process) and for all $edi \in ediset$, $tm(edi) \geq time(s')$ then $(S, s) \longrightarrow_{ediset_1} (S', s'[envw := envw(s)])$ (i.e. it can be used in the global sequence). Since we execute the process with the earliest logical time, we can also show that all items produced later cannot have a smaller logical time stamp. □

7 Verification Framework

In this section we provide a framework that can be used to specify and verify processes, as shown in Sect. 8. First, in Sect. 7.1, the programming language is extended with a number of useful constructs. Section 7.2 contains the main specification and verification constructs.

7.1 Language Extensions

To deal with some more interesting examples, the simple programming language of Sect. 3 is extended with an assignment, if-then-else construct, and infinite

loops. Accordingly, the denotational semantics of Sect. 5 is extended. Since infinite loops introduce non-terminating computations, we add one field to the status:

- $term \in \{true, false\}$: indicates termination of the process; if it is false all subsequent statements are ignored.

Henceforth, we assume that s_0 is such that $term(s_0) = true$, i.e. after s_0 we can still execute subsequent statements.

The definition of sequential composition has to be adapted, since it is possible that the first process does not terminate and thus prohibits execution of the second process.

$\mathcal{M}(S_1 \,;\, S_2)(s_0) =$
$\{s \mid s \in \mathcal{M}(S_1)(s_0) \wedge \neg term(s)\} \cup$
$\{s \mid$ there exists an s_1 with $s_1 \in \mathcal{M}(S_1)(s_0) \wedge term(s_1) \wedge s \in \mathcal{M}(S_2)(s_1)\}$

The meaning of the skip statement can be defined easily.

$\mathcal{M}(\texttt{Skip})(s_0) = \{s_0\}$

Similarly we can easily define $x := e$ where $x \in Svars$ and e an expression yielding a set of data items. We also add variables that range over data items and define assignments for such variables. Next we define an if-then-else statement, where b is a boolean expression.

$\mathcal{M}(\texttt{If } b \texttt{ Then } S_1 \texttt{ Else } S_2 \texttt{ Fi})(s_0) =$
$\{s \mid$ if b is true in $st(s_0)$ then $s \in \mathcal{M}(S_1)(s_0)$ else $s \in \mathcal{M}(S_2)(s_0)$ $\}$

Finally, we define an infinite loop by means of an infinite sequence of statuses $s_0, s_1, s_2, \ldots$, where s_i is the result of executing the loop body i times, provided all these executions terminate. Otherwise the $term$-field of s_i is false. The written items are collected by taking the union of the produced items in each execution of the body. Note that only the own and environment writes are relevant.

$\mathcal{M}(\texttt{Do } S \texttt{ Od})(s_0) =$
$\{s \mid$ there exists a sequence $s_0, s_1, s_2, \ldots$ such that for all i,
if $term(s_i)$ then $s_{i+1} \in \mathcal{M}(S)(s_i)$ else $term(s_{i+1}) = false$, and
$ownw(s) = \cup_{\{i \mid term(s_i)\}} ownw(s_{i+1})$ and $envw(s) = envw(s_0)$ $\}$

7.2 Specification and Verification

To obtain a convenient specification and verification framework, we define a mixed formalism in which one can freely mix programs and specifications, based on earlier work [Hoo94].

Specifications are part of the program syntax; let $p, p_0, p_1, \ldots, q, q_0, q_1, \ldots$ be *assertions*, that is, predicates over statuses. A *specification* is a "program" of the form $\texttt{Spec}(p, q)$ with the following meaning.

$\mathcal{M}(\texttt{Spec}(p, q))(s_0) =$
$\{s \mid$ ($p(s_0)$ implies $q(s)$) and $envw(s) = envw(s_0)$ $\}$

Next we define a refinement relation $\Rightarrow$ between programs (which now may include specifications).

Definition 7 (Refinement). For any two programs P_1, P_2, we define $P_1 \Rightarrow P_2$ iff for all s_0, we have $\mathcal{M}(P_1)(s_0) \subseteq \mathcal{M}(P_2)(s_0)$.

Note that it is easy to prove that the refinement relation is reflexive and transitive. We have the usual consequence rule.

Lemma 6 (Consequence). If $p \rightarrow p_0$ and $q_0 \rightarrow q$ then $\mathtt{Spec}(p_0, q_0) \Rightarrow \mathtt{Spec}(p, q)$.

Based on the denotational semantics for Splice, we checked in PVS the soundness of a number of proof rules for programming constructs. For instance, for sequential composition we have a composition rule and a monotonicity rule which allows refinements in a sequential context.

Lemma 7 (Sequential Composition). $(\mathtt{Spec}(p, r); \mathtt{Spec}(r, q)) \Rightarrow \mathtt{Spec}(p, q)$.

Lemma 8 (Monotonicity of Sequential Composition). If $P_3 \Rightarrow P_1$ and $P_4 \Rightarrow P_2$ then $(P_3 \,;\, P_4) \Rightarrow (P_1 \,;\, P_2)$.

The reasoning about parallel composition in PVS mainly uses the semantics directly. We only have a monotonicity rule, which forms the basis of stepwise refinement of components.

Lemma 9 (Monotonicity of Parallel Composition). If $P_3 \Rightarrow P_1$ and $P_4 \Rightarrow P_2$ then $(P_3 \parallel P_4) \Rightarrow (P_1 \parallel P_2)$.

8 Verification Example

To illustrate our reasoning framework, we first show top-down development of a typical application consisting of a producer, a transformer, and a consumer in Sect. 8.1. Next, in Sect. 8.2, we consider transparent replication. The general question is whether we can replace a single process by two, or more, copies of the same process without having to change the environment. So, given the monotonicity rules mentioned above, we look for sufficient conditions for having $P \| P \Rightarrow P$. We investigate this for the concrete case study developed in Sect. 8.1.

8.1 Top-Down Development

Here we consider a concrete and simple case study, which is prototypical for the application area, with the following processes:

- *Producer*: provides monotonically increasing data (here simply natural numbers) with name *SensData*.
- *Transformer*: assuming it gets increasing *SensData* items, it provides monotonically increasing data with name *Intern*.
- *Consumer*: assuming the environment provides increasing data with name *Intern*, it produces monotonically increasing *Display* items.

To formalize this, we define the following types and functions:

- $DataName = \{SensData, Intern, Display\}$, with typical variable dn.
- $DataVal = \mathbb{N}$.
- $Data$ is a type of records with two fields: $name$ of type $DataName$ and val of type $DataVal$. Let $dvar$ be a variable of type $Data$.
- $KeyData = DataName$ and $key(dvar) = name(dvar)$.

To formulate the specifications, first a few preliminary definitions are needed, where $wset$ is a set of extended data items:

- $NameOwnw(dn)(s) = \forall edi \in ownw(s) : name(edi) = dn$
- $SensData(wset) = \{edi \mid edi \in wset \wedge name(edi) = SensData\}$
  Similarly, we have $Intern(wset)$ and $Display(wset)$.
- $Increasing(wset) =$
  $\forall edi1 \in wset, edi2 \in wset : (val(edi1) < val(edi2) \leftrightarrow ts(edi1) < ts(edi2))$

Let pre be an assertion expressing that $db = \emptyset$, $ownw = \emptyset$, and all variables ranging over sets of data items are initialized to the empty set. It is used as a precondition for the processes.

The top-level specification of the overall system is defined as follows.

$$postTopLevel(s) = envw(s) = \emptyset \rightarrow Increasing(Display(ownw(s)))$$
$$TopLevel = \mathtt{Spec}(pre, postTopLevel)$$

We implement this top-level specification by the parallel composition of the three processes mentioned above: $Producer \parallel Transformer \parallel Consumer$
We first specify the processes, without considering their implementation, and show that these specifications lead to the top-level specification.

The producer writes an increasing sequence of sensor data.

$$postProd(s) = NameOwnw(SensData)(s) \wedge Increasing(ownw(s))$$
$$Prod = \mathtt{Spec}(pre, postProd)$$

The consumer should produce an increasing sequence of $Display$ data, assuming the environment provides increasing internal values.

$$postCons = NameOwnw(Display)(s) \wedge Increasing(Intern(envw(s))) \rightarrow Increasing(Display(ownw(s)))$$
$$Cons = \mathtt{Spec}(pre, postCons)$$

To connect producer and consumer, we specify the transformer as follows.

$$postTrans = NameOwnw(Intern)(s) \wedge Increasing(SensData(envw(s))) \rightarrow Increasing(Intern(ownw(s)))$$
$$Trans = \mathtt{Spec}(pre, postTrans)$$

We have proved in PVS that this leads to a correct refinement of the top-level specification.

Theorem 2. $(Prod \parallel (Trans \parallel Cons)) \Rightarrow TopLevel$

Next we consider the implementation of the three components. Let d be a variable ranging over $Data$, whereas $dset$ and $dold$ are variables ranging over sets of data items.

$Producer = d := (\#name := SensData, val := 0\#)$;
Do Write(d) ; $d := (\#name := SensData, val := val(d) + 1\#)$ Od

Note that the producer writes all natural numbers, which is not required by the specification. Also note that subscribers to data with name $SensData$ will usually read only a (increasing) subsequence of these items. We have proved in PVS that this is indeed a correct refinement.

Lemma 10. $Producer \Rightarrow Prod$

Similarly we provide a program for the transformer and show that it is a correct implementation. Let $q(dn, old)$ be the query that specifies a set of data items with name dn and local time stamp different from those in variable old (which is initially empty). The set is allowed to be empty, to avoid blocking computations; otherwise it will be a singleton with a new, unread, item. The item that has been read is transformed using a function $Tr(dn, dset)$ which, for simplicity, just changes the name of the record of the item in $dset$ into dn.

$Transformer =$ Do Read($dset, q(SensData, old)$) ;
If $dset \neq \emptyset$
Then Write($Tr(Intern, dset)$) ; $old := dset$
Else Skip Fi Od

Lemma 11. $Transformer \Rightarrow Trans$

Similarly for the consumer. For simplicity, we used the same name transformation, which makes it possible to derive the correctness of transformer and consumer from a single proof that was parameterized by the name of the data sort.

$Consumer =$ Do Read($dset, q(Intern, old)$) ;
If $dset \neq \emptyset$
Then Write($Tr(Display, dset)$) ; $old := dset$
Else Skip Fi Od

Lemma 12. $Consumer \Rightarrow Cons$

Finally observe that top-level theorem 2 and the lemmas 10, 11, and 12 for the components lead by the monotonicity property (lemma 9) to

$$(Producer \parallel (Transformer \parallel Consumer)) \Rightarrow TopLevel$$

8.2 Transparent Replication

While investigating transparent replication, we observed that the current version of the transformer specification is not suitable. The next lemma expresses that we cannot replicate the transformer in a transparent way.

Lemma 13. $\neg((Trans \parallel Trans) \Rightarrow Trans)$

The lemma has been proved by a concrete counter example in PVS. The basic idea is that the two transformers each have increasing output, but - because they may write this output at different moment - the local time stamps in these sequences might be different and hence merging these output streams need not lead to an increasing sequence.

The main problem is that the items produced get their local time stamp from the local clock when the item is written. This need not be related to the temporal validity of the data. Hence we propose an alternative specification for the transformer where the local time stamp is just copied from the incoming data item (as we will see later, this also requires a modified write statement).

$MaintainTs(s) = \forall edi \in ownw(s) : \exists edi_1 \in envw(s) : ts(edi) = ts(edi_1) \wedge$
$name(edi_1) = SensData \wedge val(edi) = val(edi_1)$

$postTransNew = NameOwnw(Intern) \wedge MaintainTs$

$TransNew = \mathtt{Spec}(pre, postTransNew)$

This indeed refines the transformer specification:

Lemma 14. $TransNew \Rightarrow Trans$

Moreover, this specification can indeed be replicated.

Lemma 15 (Transformer Replication).
$(TransNew \parallel TransNew) \Rightarrow TransNew$

Hence we have (by theorem 2, lemma 14 and monotonicity):

$(Prod \parallel (TransNew \parallel Cons)) \Rightarrow TopLevel$

and by lemma 15 and monotonicity:

$(Prod \parallel ((TransNew \parallel TransNew) \parallel Cons)) \Rightarrow TopLevel$

It remains to implement $TransNew$. Important part of this specification is that it just copies the local time stamp (the ts-field). This, however, cannot be implemented with the current write statement, since it always use the current value of the local clock as its time stamp (the ts field is set to $clock(s_0)$). Therefore we introduce a new write statement $\mathtt{Write}(e, texp)$ which has an additional parameter, a time expression $texp$, which specifies the ts value, i.e. in the semantics of this extended write statement the ts-field in the data item gets the value of $texp$.

Query $Qtrans$ specifies a set of data items that is either empty or a singleton containing an item with name $SensData$. Expression $MkIntern(dset)$ changes the name of the item in $dset$ to $Intern$ and copies it value. Now the write statement has a time expression $Ts(dset)$ which just yields the ts-field of the data item in $dset$.

$NewTransformer =$ `Do` `Read`$(dset, Qtrans)$;
`If` $dset \neq \emptyset$
`Then` `Write`$(MkIntern(dset), Ts(dset))$
`Else Skip Fi Od`

Lemma 16. $NewTransformer \Rightarrow TransNew$

So, finally, we have $(Producer \| (NewTransformer \| Consumer)) \Rightarrow TopLevel$ and $(Producer \parallel ((NewTransformer \parallel NewTransformer) \parallel Consumer)) \Rightarrow TopLevel$.

9 Concluding Remarks

We have defined a new denotational semantics for the main primitives of the industrial software architecture Splice. This architecture is data-oriented and based on local storages of data items. Communication between components is anonymous, based on the publish-subscribe paradigm.

New is especially the modeling of time stamps, based on local clocks, and the update mechanism of local storages based on these time stamps. Moreover, the denotational semantics supports convenient compositional verification, based on assumptions about the data items produced by the environment of a component. Causality between data items is represented by logical time stamps. To simplify verification, we minimized the number of updates of the local storages and tried a short, but non-trivial, formulation of parallel composition.

To increase the confidence in this denotational semantics, we also formulated a rather straightforward operational semantics. Using the interactive theorem prover PVS, we formally showed that the operational semantics is equivalent to the denotational one. The proof revealed a number of errors in earlier versions of the semantics, e.g. concerning the precise interpretation of logical time stamps representing causality.

As a side-effect, the proof resulted in a number of useful properties. Classical ones, such as associativity of sequential and parallel composition, but also more Splice-oriented properties such as idempotence of updates. We also observed that the current definition of the notion of a lifecycle in Splice breaks idempotence of updates. Since it indicates a conceptual error in the definition, we have removed the lifecycle from the current formalization.

A topic of current work concerns the equivalence classes induced by the semantics, i.e. which programs obtain the same semantics? The question is whether the denotational semantics is fully abstract with respect to the operational one, that is, does it only distinguish those programs that are observably different in some context?

We have shown how the semantics can be used as a basis for a formal framework for specification and compositional verification. The study of transparent replication clarified the use of local time stamps and led to an improvement of the write primitive. In future work we intend to investigate the use of clock synchronization.

References

[BdJ97] M. Boasson and E. de Jong. Software architecture for large embedded systems. IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, 1997.

[BHdJ00] R. Bloo, J. Hooman, and E. de Jong. Semantical aspects of an architecture for distributed embedded systems. In *Proc. of the 2000 ACM Symposium on Applied Computing (SAC 2000)*, volume 1, pages 149–155. ACM press, 2000.

[BKBdJ98] M.M. Bonsangue, J.N. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In *Proc. of the 1998 ACM Symposium on Applied Computing (SAC '98)*, pages 159 – 168. ACM press, 1998.

[BKZ99] M. M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proc. of the 1999 ACM Symposium on Applied Computing (SAC'99)*. ACM Press, 1999.

[Boa93] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.

[dRdBH+01] W.P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification, Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Addison-Wesley, Reading, MA, USA, 1999.

[Gel85] D. Gelernter. Genarative communication in Linda. *Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[HH02] U. Hannemann and J. Hooman. Formal reasoning about real-time components on a data-oriented architecture. In *Proc. of 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI02)*, volume XI, pages 313–318, 2002.

[Hoo94] J. Hooman. Correctness of real time systems by construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, Springer-Verlag, 1994.

[HvdP02] J. Hooman and J. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proc. of the 2002 ACM Symposium on Applied Computing (SAC 2002)*, pages 351–358, 2002.

[Jon83] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, 1981.

[OSRSC01] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. SRI International, Computer Science Laboratory, Menlo Park, CA, version 2.4 edition, December 2001. `http://pvs.csl.sri.com`.