Semantic Models of a Timed Distributed Dataspace Architecture

Jozef Hooman [a,b,c]

[a] *University of Nijmegen, The Netherlands*
[b] *Embedded Systems Institute, Eindhoven, The Netherlands*

Jaco van de Pol [c]

[c] *CWI, Amsterdam, The Netherlands*

Abstract

We investigate various formal aspects of a distributed dataspace architecture in which data storage is based on time stamps. An operational and a denotational semantics have been defined and the equivalence of these two formulations has been proved. Moreover, the denotational semantics is fully abstract with respect to the observation of produced data items. It is used as a basis for compositional reasoning about components, supported by the interactive theorem prover PVS. We use this framework for a small example where components make mutual assumptions about each other's output.

1 Introduction

In this paper, we investigate the application of formal techniques in the context of an industrial software architecture which is based on distributed data storages. In particular, we consider the software architecture Splice [1,2] 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.

Email addresses: `jozef.hooman@embeddedsystems.nl` (Jozef Hooman), `Jaco.van.de.Pol@cwi.nl` (Jaco van de Pol).

Splice is data-oriented, with distributed local databases based on keys. It provides a coordination mechanism between loosely-coupled heterogeneous components by means of the publish-subscribe paradigm. An important design decision is to have minimal overhead for data management, allowing a fast and cheap implementation that allows huge data streams from sensors, such as radars. 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. A brief informal explanation of Splice can be found in Section 2. Section 3 contains a formal syntax of a very simple Splice-like language and informally describes the meaning of this language. There is a slight difference with the semantics presented in [3,4]; the current paper contains a weak and realistic assumption about the synchronization of local clocks which simplifies the formalization significantly. The semantic difference is explained at the end of Sect. 3.

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. In addition, the goal is to allow specifications that may include explicit assumptions about the environment of a component, as described in [5].

Such a compositional verification framework should be based on a solid formal foundation, in particular on a denotational semantics, which defines the meaning of compound constructs in terms of the meaning of the parts [6]. Moreover, to increase the confidence in this denotational semantics, it is important to define an independent operational semantics and relate it formally to the denotational one. This leads to the four main topics of this paper which are briefly described in the next subsections: operational semantics (Sect. 1.1) and denotational semantics (Sect. 1.2), their relation (Sect. 1.3), and the verification framework (Sect. 1.4).

1.1 Operational Semantics

To formalize the meaning of our simple Splice-like language, Sect. 4 contains an operational semantics which is close to the operational intuition. Earlier work on the operational semantics of Splice-like languages includes a transition system semantics for a basic language of write and read statements (without query) [7]. In [8], an operational semantics is provided by a translation to process algebra. That paper focuses on a global dataspace view for a simple fragment of Splice. Related is also a recent comparison of semantic choices using an operational semantics and embeddings [9]. New in our work is the treatment of local time stamps and their use for updating local databases.

Some other operational semantics have been given for shared dataspaces with time, e.g. extensions of Linda or JavaSpaces with delays, time-outs and leasing (i.e.: non-permanent data that expires after a specified time) [10–12]. All these papers study timed extensions of coordination language primitives. Time plays a different role in our paper. Our motivation has not been to make time explicit in the coordination primitives, but time is used internally in the (semantics of the) data space, in order to decide which data items to overwrite, and to make causal relationships explicit.

1.2 Denotational Semantics

The denotational semantics of the Splice primitives is defined in Sect. 5. It forms the basis of our verification framework and includes explicit assumptions about the data items produced by the environment of a component. It is well-known that a denotational semantics provides a good basis for a compositional verification framework [6].

In previous work on a denotational semantics for Splice [13], the semantics of local storages was inconvenient for compositional verification; it uses process identifiers and a partial order of read and write events with complex global conditions. In more recent work on the verification of Splice-systems [3], we used a complex denotational semantics with environment actions.

1.3 Relating Operational and Denotational Semantics

Although the current denotational semantics is a good basis for compositional reasoning using assumptions about the environment, it is far from trivial that it captures the intuitive understanding of the Splice architecture. Hence, in Sect. 6 we prove formally that it is equivalent to the operational semantics. For the slightly more complex semantics defined in [4], a very similar proof has been checked mechanically by means of the interactive theorem prover PVS [14,15].

Another interesting topic concerns the equivalence classes induced by the semantics, grouping the programs that obtain the same semantics. Typically, a denotational semantics has more classes, containing less programs, than an operational semantics. The reason is that the denotational semantics of a program should define its meaning in any context, whereas in an operational semantics the complete system is given. So a denotational semantics has to distinguish more programs, but it might distinguish more than needed (the extreme is a semantics where each syntactically different program gets a different denotation).

This leads to the question 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? In Sect. 7, we claim that our denotational semantics is indeed fully abstract with respect to the observations of the operational semantics, namely the set of produced data items.

Full abstraction has been considered for a large variety of programming concepts, e.g. for the timed semantics of synchronously or asynchronously communicating processes [16,17] to, recently, information exchange in multi-agent systems [18]. Typically, a form of failure sets has been used to obtain full abstraction, but in our case this was not needed.

1.4 Verification Framework

Many examples in the literature (cf. [6]) show that it is convenient to specify components using explicit assumptions about the environment. Concerning Splice, in [5] 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 [19] or rely/guarantee [20] 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 that the reasoning is based on a sound formal foundation.

In Sect. 8, we present a framework for the specification and verification of Splice components. The framework has been defined in terms of the higher-order logic of PVS, thus allowing the use of the interactive theorem prover of PVS to verify applications. In [4] we have used a similar framework to verify an example of transparent replication.

In this paper, we illustrate the approach by a small example with two components that generate even and odd numbers; this example originates from an early paper [21] on assumption/commitment reasoning. In this example, there is a mutual dependency between the components and we observe that simple implication between assumptions and commitments is not suitable. We solve this by requiring that a commitment at a certain point of time may only use assumptions for earlier points of time.

In our solution, we use a discrete notion of time; it is closely related to McMillan's rule which has also been formalized in PVS [22]. It, however, assumes

that parallel composition corresponds to conjunction which is not the case in our framework (many components may produce different data items of a particular sort). Having discrete time is convenient but not strictly needed; in [23] we have shown that sound assumption/commitment reasoning is possible if we require that there exists a $\delta > 0$ such that a commitment at any point t may use assumptions up to point $t - \delta$.

2 Informal Introduction to the Splice Architecture

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 [24] and JavaSpaces [25]. These languages, however, have a single shared dataspace, whereas in Splice each component has its own dataspace, see Fig. 2. Communication between components takes

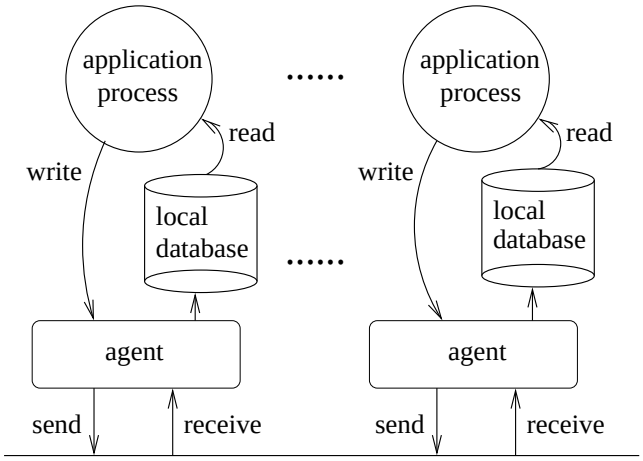

Fig. 1. Splice applications

place by means of local agents. A data producer writes data records to the other dataspaces via its agent. A data consumer uses an 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, see [2] for more details. 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 time stamps improve the quality of the data stored, as no record can be overwritten by older data.

Although it cannot be assumed that the local clocks are synchronized perfectly, many real-time applications require a reasonable tight clock synchronization. In typical Splice environments, this is even supported by special hardware. We will only use the relatively mild assumption that the clock drift is less than the network latency.

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 subscribing to data of a certain sort, 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.

The local dataspace is organized along the well-known relational database model. All information is stored in tables, and sorts determine the names and types of the fields within a table. Fields can be declared to be key fields, which determines the identity relation on data items. Queries may involve typical database operations, such as joining records from various tables based on key-fields, and projecting the records of a table by selecting a subset of the fields. Note that by subscribing to a selection of the fields, a consumer may ignore certain fields of produced data items. Most of this database structure is kept abstract in our formalization; we keep a notion of key-data and assume in Section 7 that producers can extend data sorts by adding new tag fields.

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. It is easy

to extend the language with other constructs; in Sect. 8.1 we show how to add an infinite loop and in [3] we have added assignments and an if-then-else construct.

We consider only one sort. Let *Data* be some data domain, with a set *KeyData* of key data and a function *key: Data* $\rightarrow$ *KeyData*. Assume a given type *LocalTime*, to represent values of local clocks. We assume a total order $>$ on *LocalTime*, and a minimal element $0 \in$ *LocalTime*.

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 $(\#dat := v, ts := c\#)$, following the PVS notation. In long formulas, we may also write this as the ordered pair (v, c). Hence, for $di \in$ *DataItems*, we have $dat(di) \in$ *Data* and $ts(di) \in$ *LocalTime*.

For $X \subseteq$ (*DataItems*) and $t \in$ *LocalTime*, we define $X <_{time} t$ to denote that all time stamps in X are smaller than t. More precisely, $X <_{time} t$ iff for all $di \in X$, $ts(di) < t$. The reverse $t <_{time} X$ is defined similarly.

Using overloading, functions on *Data* can be extended to functions defined on *DataItems*. In particular, the function *key* is extended to *DataItems* by defining $key(di) = key(dat(di))$.

We will use $\perp$ as a special symbol denoting an undefined data item, and $DataItems^{\perp}$ is defined to be $DataItems \cup \{\perp\}$. Let *Vars* be the set of program variables. Program variables range over $DataItems^{\perp}$. For simplicity, we do not give the concrete syntax of data expressions and queries here. Instead, we use standard set notation for expressions and queries. A data expression e : *Data* denotes a data value, possibly depending on the program variables. A *query* is a predicate on $DataItems^{\perp}$, possibly depending on program variables. We will use $\perp$ in queries to specify non-blocking read operations.

Henceforth, we typically use the following variables ranging over the types mentioned above:

- v over *Data* values. In examples we also use A, B, C as concrete values.
- $di, di_0, di_1, \ldots$ over *DataItems*
- $Di, Di_0, Di_1, \ldots$ over $DataItems^{\perp}$
- $diset, diset_0, diset_1, \ldots$ over sets of *DataItems*
- $x, x_0, x_1, \ldots, y, y_0, y_1, \ldots$ over *Vars*
- $q, q_0, q_1, \ldots$ over queries

The syntax of our programming language is given in Table 1.

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

Table 1

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

- $\texttt{Write}(e)$ publishes a data item with value e (in the current state) and the current time stamp (from the local clock). The local clock is increased.

  We model *best effort* delivery; a data item arrives 0 or more times at each process, where it might be used to update the local storage. It is added to this local storage if there is no item with the same key which has a larger or equal time stamp. As a side-condition of such an update, the value of the local clock should be larger than the time stamp of any data item used for an update of the local database.
- $\texttt{Read}(x, q)$ assigns to x a data item from the local storage that satisfies query q. In particular, if there are data items satisfying q, the choice is non-deterministic. If no data item from the local storage satisfies q, but $\perp$ satisfies q, then $\perp$ can be returned. Otherwise, the execution of read is blocked until the database contains a data item satisfying q.

  For instance, the query $q = \{di \,|\, ts(di) > 100\}$ in $\texttt{Read}(x, q)$, would assign to x a data item from the local storage with time stamp greater than 100. If there are no such items in local storage, the read statement blocks. Note that a query like $q' = q \cup \{\perp\}$ allows $\perp$, so with this query the read may continue, even if the data storage doesn't contain an element satisfying q. Hence a read statement may be blocking or not, 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 either a sequential program or a parallel composition of processes; in the latter case we call it a *parallel program*.

Instead of a concrete syntax for queries, we introduce a number of standard abbreviations:

Definition 1 (Abbreviations)

- We use query "*true*" to denote any data item or $\perp$.
- We use queries of the form "v" to denote $\{di \mid dat(di) = v\}$. These queries require a data value v, and allow an arbitrary time stamp.
- We use queries of the form "$v_\perp$" to denote $\{di \mid dat(di) = v\} \cup \{\perp\}$. These queries require data value v, but allow $\perp$ if v is not present.
- $new(x, v)$ is the query which requires an item with value v and time stamp larger than that of x, if x is defined. Formally:
  $new(x, v) = \{di \mid dat(di) = v \text{ and } (x = \perp \text{ or } ts(di) > ts(x)\}$.
  For instance, if $\texttt{Read}(y, new(x, A))$ terminates then y is a data item with value A and a time stamp larger than all time stamps of the items of x.

Example 2 *As a very simple example, consider a few producers and consumers of flight data. Let Data be a record with two fields:* flightnr *(a string, e.g. KL*309*) and* pos *(a position in some form, here a number for simplicity). The flight number is the key, that is, key*(v) = flightnr(v)*. Consider a producer of flight data*

$P_1 = \texttt{Write}((\#\text{flightnr} := KL567, \text{pos} := 1\#))$;
$\texttt{Write}((\#\text{flightnr} := LU321, \text{pos} := 6\#))$;
$\texttt{Write}((\#\text{flightnr} := KL567, \text{pos} := 2\#))$;
$\texttt{Write}((\#\text{flightnr} := KL567, \text{pos} := 3\#))$

and two consumers:

$C_1 = \texttt{Read}(x_1, true)$; $\texttt{Read}(y_1, q_1)$; $\texttt{Read}(z_1, q_1)$
$C_2 = \texttt{Read}(x_2, q_1)$; $\texttt{Read}(y_2, q_2)$

whose queries are specified as follows:

$q_1 = \{di \mid \text{flightnr}(dat(di)) = KL567\}$
$q_2 = \{di \mid \text{flightnr}(dat(di)) = KL567 \text{ and } ts(di) > ts(x)\}$

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 in a different order at the consumers, and they may arrive several times. However, this only leads to an update of the local database if the time stamp is larger.*

Variable x_1 *may be* $\perp$ *(if no data item has been delivered yet – note that this read is not blocking) or it may contain a produced data item. For instance, it may contain position number* 3 *for KL*567*. The second read is blocking (*q_1 *doesn't allow* $\perp$*), so after that read, variable* y_1 *will contain a data item with flight number KL*567*. If there is a position for KL*567 *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*.*

Difference with earlier versions

The most notable syntactic change compared to [4] is that now variables and queries denote $DataItems^{\perp}$ instead of $\mathcal{P}(DataItems)$. This simplifies the presentation considerably, and the generality of having sets was only used to allow non-blocking reads (corresponding to the empty set) which is now captured by having $\perp$.

The informal meaning of the Splice statement defined here differs slightly from the semantics defined earlier [3,4]. The current semantics contains a slightly stronger - but realistic - requirement on the local clocks, namely that a local

clock is always larger than the time stamps in the data items that have been received. This can be seen as an abstraction of the clock synchronization which is present in the Splice system. In our semantics, these local clocks are updated similar to Lamport's logical clocks [26]. This ensures that the partial order thus obtained is consistent with the causality relation between read and write events.

The following example shows that this semantic difference can be observed by a Splice process. Consider the three processes:

$P_1 = \texttt{Write}(A)$
$P_2 = \texttt{Read}(x, A)\ ;\ \texttt{Write}(B)$
$P_3 = \texttt{Read}(y, B)\ ;\ \texttt{Read}(y, new(y, A))$

Suppose the time stamp of the item with value A is 10. With the original semantics [3,4], the time stamp of the item with value B could be smaller, say 5. Hence P_3 may terminate, because it can first read B and then A with a larger time stamp. With the current semantics, the local clock of P_2 after $\texttt{Read}(x, A)$ will be larger than 10 and, hence, also the time stamp of the item with value B will be larger than 10. This implies that P_3 always blocks after the first read, because there is no new item A with a larger time stamp to read.

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. First, an operational status of a sequential program (Def. 3) and its local computation steps (Def. 6) are defined. Next, we define configurations (Def. 7), which represent the state of affairs during operational execution of a process, and global computation steps (Def. 8), leading to the operational semantics (Def. 9).

For convenience, we slightly rewrite the syntax of the programming language, also introducing the empty statement E which represents a statement that has terminated, as shown in Table 2.

Table 2

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

The state of a program is represented by a function $st : Vars \to DataItems^{\perp}$. An expression is formalized as a function $e : (Vars \to DataItems^{\perp}) \to Data$. We will write $e(st)$ to denote the value of expression e in state st. Similarly, a query q can be represented as $q : (Vars \to DataItems^{\perp}) \to \mathcal{P}(DataItems^{\perp})$.

We write $q(st)(di)$ (resp. $q(st)(\perp)$) to denote that di (resp. $\perp$) satisfies query q in state st.

Let *DataBases* be the type consisting of sets of data items with at most one item for each key, i.e.

$$DataBases = \{diset \subseteq DataItems \mid \text{ for all } di_1, di_2 \in diset: key(di_1) = key(di_2) \rightarrow di_1 = di_2\}$$

Definition 3 (Operational Status) An operational status of a sequential program, denoted os, os_0, os_1, .., is a record with three fields, st, $clock$ and db:

- $st : Vars \rightarrow DataItems^{\perp}$, represents the local state, assigning to each variable a data item (or $\perp$ representing undefined);
- $clock \in LocalTime$, the value of the local clock;
- $db \in DataBases$, with $0 <_{time} db <_{time} clock$, represents the local database as a set of data items representing the local storage. Besides the restriction that each key occurs at at most once, we additionally require that all time stamps in db are smaller than $clock$, but bigger than the minimal element of $LocalTime$ [1] .

Definition 4 (Variant) The *variant* of local state st with respect to variable $x \in Vars$ and value $Di \in DataItems^{\perp}$, denoted by $st[x \mapsto Di]$, is defined as

$$(st[x \mapsto Di])(y) = \begin{cases} Di & \text{if } y = x \\ st(y) & \text{if } y \neq x \end{cases}$$

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

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

Produced data items are sent to an underlying network. 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 by different producers. The use of a set is justified by the fact that the multiplicity of data items cannot be observed: the network is unreliable, and it may deliver this item never, once, or many times. In previous papers [3,4] we showed that this even allows the transparent replication of processes in certain cases.

To avoid problems due to multiple delivery of old data items by the network, the database is only updated with newer data. We define the update of a database, using a new database, i.e. a selected set of data items delivered by the network. An element of the new database is added if its key is not yet present, otherwise it only replaces the element of the old database with the

[1] The additional requirement was not present in [4], but is essential for the full abstraction result in Section 7

same key if its local time stamp is strictly greater.

Definition 5 (Update Database) The update of database db using a new database db_1, denoted $UpdateDb(db, db_1)$ is defined as follows.
$di \in UpdateDb(db, db_1)$ iff

- either $di \in db$ and for all $di_1 \in db_1$ with $key(di_1) = key(di)$ we have $ts(di_1) \leq ts(di)$,
- or $di \in db_1$ and for all $di_0 \in db$ with $key(di_0) = key(di)$ we have $ts(di_0) < ts(di)$.

A local computation step of a sequential program corresponds to a read or write statement, or it can be an update step in which the local database is updated with items delivered by the network.

Definition 6 (Local Computation Step) We denote a local computation step of a sequential program S in an operational status os and given a network N by $\langle S, os, N\rangle \longrightarrow \langle S', os', N'\rangle$. This relation is defined by the three rules of Fig. 2.

- **Update:** In the first – update – rule, X represents the data items that arrive from the network. The condition $X <_{time} cl'$ expresses that the value of the local clock in the end state should be larger than the time stamps of the newly added items; this models our assumption that the local clocks of sender and receiver differ less than the maximal message transmission delay.
  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:** In the second rule, for the write statement, the written data item is given the time stamp of the new clock value, which must be strictly greater than the current clock value. This ensures that subsequent write statements get increasing time stamps. Recall that $e(st)$ denotes the value of the expression e in the current state st.
- **Read:** The last rule expresses that a read statement assigns to x an element from the database that satisfies the query q if it exists. If no such element exists and $\perp$ satisfies the query, then $\perp$ is assigned to x. Otherwise, no rule applies, modeling a blocking read.

Definition 7 (Configuration) The state of affairs of a process $S_1 \parallel ... \parallel S_n$ during execution is represented by a *configuration* of the form
$\langle (S'_1, os_1), ..., (S'_n, os_n), N\rangle$
For each sequential program S_i, it denotes the current status os_i and the remaining part S'_i that still has to be executed. Moreover, it contains the current contents N of the network.

$$\frac{cl' \geq cl \quad X : DataBases \quad X \subseteq N \quad X <_{time} cl'}{\langle S\ ,\ (st, cl, db)\ ,\ N\rangle \rightarrow \langle S\ ,\ (st, cl', UpdateDb(db, X))\ ,\ N\rangle}$$

$$\frac{cl' > cl}{\langle \texttt{Write}(e); S\ ,\ (st, cl, db)\ ,\ N\rangle \rightarrow \langle S\ ,\ (st, cl', db)\ ,\ N \cup \{(e(st), cl')\}\rangle}$$

$$\frac{q(st)(Di) \quad cl' \geq cl \quad Di \in db \text{ or } (Di = \bot \text{ and } \neg\exists di \in db, q(st)(di))}{\langle \texttt{Read}(x, q); S\ ,\ (st, cl, db)\ ,\ N\rangle \rightarrow \langle S\ ,\ (st[x \mapsto Di], cl', db)\ ,\ N\rangle}$$

Fig. 2. Local Computation Step

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

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

where $C_0 = \langle (S_1\ ;\ E, os_1), ..., (S_n\ ;\ E, os_n), \emptyset\rangle$ and, for all i, $db(os_i) = \emptyset$. Each step in such a sequence represents the execution of an *atomic* action by some sequential program i, as defined in Def. 8.

Definition 8 (Global Computation Step) The global computation $\Rightarrow_{\mathcal{O}}$ is defined in Fig. 3. The basic rule corresponds to a local computation step of one of the components. The other rules yield the reflexive, transitive closure of the one step computation.

$$\frac{\langle S_i, os_i, N\rangle \rightarrow \langle S', os', N'\rangle, \text{ for some } i,\ 1 \leq i \leq n}{\langle (S_1, os_1), \ldots (S_n, os_n), N\rangle \Rightarrow_{\mathcal{O}} \langle (S_1, os_1), \ldots (S', os'), \ldots (S_n, os_n), N'\rangle}$$

$$\frac{}{C \Rightarrow_{\mathcal{O}} C}\ \text{REFL} \qquad \frac{C_1 \Rightarrow_{\mathcal{O}} C_2 \quad C_2 \Rightarrow_{\mathcal{O}} C_3}{C_1 \Rightarrow_{\mathcal{O}} C_3}\ \text{TRANS}$$

Fig. 3. Global Computation Steps for $\mathcal{O}$

Typically, the operational semantics yields some abstraction of execution sequences, 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 9 (Operational Semantics) The operational semantics of a process $S_1 \parallel ... \parallel S_n$, given an initial operational status os_0, is defined by
$\mathcal{O}(S_1 \parallel ... \parallel S_n)(os_0) =$

$\{N \subseteq DataItems \mid db(os_0) = \emptyset$ and $\exists os_1, \ldots, os_n :$
$\langle (S_1\ ;\ E, os_0), ..., (S_n\ ;\ E, os_0), \emptyset\rangle \Rightarrow_{\mathcal{O}} \langle (E, os_1), ..., (E, os_n), N\rangle\ \}$

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 10 *Observe that, for any* os_0*,*

$$\mathcal{O}((\texttt{Read}(x, A)\ ;\ \texttt{Write}(B)) \parallel (\texttt{Read}(x, B)\ ;\ \texttt{Write}(A)))(os_0) = \emptyset.$$

Indeed, the network and the databases are initially empty and the queries make the read statements blocking, so no component can write the data item needed by the other component. In the next section (Example 11) we show how our denotational semantics avoids that these processes read each other's written items.

5 Denotational Semantics

In this section, we define the denotational semantics of our Splice-like programming language. This means that the semantics of compound constructs (sequential and parallel composition here) is defined in terms of the semantics of its constituents, without referring to the syntax of these parts. The meaning of the atomic statements (read and write here) is defined independently, such that they can be included in any context.

We define the denotational semantics of a program using an initial status which represents the state of affairs at the start of the execution. 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, without knowing the context in which it will operate, it is quite common that information has to be added to the status to express relations with the environment explicitly. Here we add the set of written data items and the set of items that are assumed to be produced by the environment.

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

- $st : \mathit{Vars} \to \mathit{DataItems}^{\perp}$, the local state (values of variables);
- $\mathit{clock} \in \mathit{LocalTime}$, the value of the local clock;
- $\mathit{db} \in \mathit{DataBases}$, $0 <_{\mathit{time}} \mathit{db} <_{\mathit{time}} \mathit{clock}$, the local database (a set of data items, with at most one item per key, and time stamps smaller than *clock* but bigger than the minimal element of *LocalTime*);

there are two new fields:

- $\mathit{ownw} \subseteq \mathit{DataItems}$, with $0 <_{\mathit{time}} \mathit{ownw}$. These are the data items written by the program itself in the past;

- $envw \subseteq DataItems$, with $0 <_{time} envw$. This is the set of data items written by the environment of the program; it is an assumption about all items produced (including present and future). Note that we assume given all items produced by the environment, including those that are assumed to be produced in the future. This simplifies the semantics in the sense that no updates of this *envw*-field have to be taken into account in the semantics. However, we will need a condition to ensure that items are read in the correct causal order (see also Example 11).

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 used in the denotational semantics to update the local storage of *prog* with elements written by its environment. So *prog* itself cannot modify this field. Although all items are available initially, constraints on local clocks prevent the use of items "too early".

Next, we define $\mathcal{M}(prog)$ by induction on the structure of *prog*. The atomic cases use an auxiliary *Update* relation.

Update

The auxiliary *Update* function may update the local database with data items that have been written (by the process itself or by its environment). Its definition uses *UpdateDb* of Def. 5. To ensure the proper use of environment writes, i.e. respecting causal ordering, it is important to require that the local clock becomes larger than the time stamps of the items used for the update.

$Update(s_0) =$

$\{s \mid clock(s) \geq clock(s_0)$ and there exists a $db_1 \subseteq ownw(s_0) \cup envw(s_0)$ such that $db(s) = UpdateDb(db(s_0), db_1)$, $db_1 <_{time} clock(s)$, and s equals s_0 for the other fields (st, $ownw$ and $envw$) $\}$

We will use this relation to occur once before and once after every read and write action. [2] One may wonder how this corresponds to the operational semantics, where arbitrarily many update steps can occur between read and

[2] This is another deviation from the semantics in [4], where the denotational semantics had an update before the read statement only; the change is essential for the full abstraction result.

write events. The fact that $s_0 \in \mathit{Update}(s_0)$ corresponds to the possibility of having no update step, and the fact that $\mathit{Update}(\mathit{Update}(s_0)) = \mathit{Update}(s_0)$ shows that multiple updates can be combined to one. We deliberately reduced the number of explicit update steps in the denotational semantics, in order to simplify the verification framework.

Below we use relation composition, defined as $s_1 \in (R_1 \circ R_2)(s_0)$ iff there exists an s_2 such that $s_2 \in R_1(s_0)$ and $s_1 \in R_2(s_2)$.

Write

In the semantics of the write statement, the published item is time-stamped and added to the *ownw* field. The time stamp will be the clock value in the resulting status. Since the local clock is increased, subsequent written items obtain a larger time stamp.

$\mathit{BasicWrite}(e)(s_0) =$
$\{s \mid \mathit{clock}(s) > \mathit{clock}(s_0)$ and
$\mathit{ownw}(s) = \mathit{ownw}(s_0) \cup \{(v, \mathit{clock}(s))\}$,
where $v = e(\mathit{st}(s_0))$, the value of e in s_0, and
s equals s_0 for the other fields (*st*, *db* and *envw*) $\}$

Next, we define $\mathcal{M}(\texttt{Write}(e)) = \mathit{Update} \circ \mathit{BasicWrite}(e) \circ \mathit{Update}$.

Read

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

$\mathit{BasicRead}(x, q)(s_0) =$
$\{s \mid$ there exists $Di \in \mathit{DataItems}^{\perp}$ such that
$q(\mathit{st}(s_0))(Di)$ and $\mathit{st}(s) = \mathit{st}(s_0)[x \mapsto Di]$ and
either $Di \in \mathit{db}(s_0)$, or $Di = \perp$ and $\forall di \in \mathit{db}(s_0), \neg q(\mathit{st}(s_0))(di)$;
s equals s_0 for the other fields (*clock*, *db*, *ownw* and *envw*) $\}$

Note that we only represent successfully terminating executions; blocking has not been modeled explicitly. Next, we define

$\mathcal{M}(\texttt{Read}(x, q))(s_0) = \mathit{Update} \circ \mathit{BasicRead}(x, q) \circ \mathit{Update}$.

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. 8.1, we show how this can be extended to deal with non-terminating programs.

$$\mathcal{M}(S_1 \,;\, S_2) = \mathcal{M}(S_1) \circ \mathcal{M}(S_2).$$

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 $s + diset$ to add a set $diset \subseteq DataItems$ to the environment writes of s, i.e. $envw(s + diset) = envw(s) \cup diset$ 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 observation is that $envw(s_0)$ contains only the data items produced outside $P_1 \parallel P_2$. Hence the semantic function for P_1 is applied to s_0 where we add the items written by P_2 to the environment writes. Similarly for P_2. Then parallel composition is defined as follows:

$$\begin{array}{l}\mathcal{M}(P_1 \parallel P_2)(s_0) = \\ \quad \{s \mid init(s_0) \text{ and there exist } s_1 \text{ and } s_2 \text{ with} \\ \qquad s_1 \in \mathcal{M}(P_1)(s_0 + ownw(s_2)), \\ \qquad s_2 \in \mathcal{M}(P_2)(s_0 + ownw(s_1)), \\ \qquad ownw(s) = ownw(s_1) \cup ownw(s_2),\ envw(s) = envw(s_0)\}\end{array}$$

Parallel composition is commutative and associative. Observe that there are no constraints on the fields *st*, *clock* and *db* of s; we abstract from these fields when composing processes in parallel and allow them to be arbitrary.

Example 11 *Consider again the program of Example 10:*

$(\texttt{Read}(x, A) \,;\, \texttt{Write}(B)) \parallel (\texttt{Read}(x, B) \,;\, \texttt{Write}(A))$

Without using the condition on the local clock in the Update *function, the semantics would allow for this program a status where $envw = \emptyset$ and $ownw$ contains A and B (each component produces the item required by the other one). This, however, does not correspond to the operational semantics which yields the empty set. But using the condition in* Update*, the first program ensures that the time stamp of the item with value B is larger than the item with value A produced by the other process. Similarly, the second program ensures that the item with value A has a larger time stamp, and hence there are no executions that can be combined at parallel composition. We can indeed show that, for any s_0 with $env(s_0) = \emptyset$,*

$\mathcal{M}((\texttt{Read}(x, A) \,;\, \texttt{Write}(B)) \parallel (\texttt{Read}(x, B) \,;\, \texttt{Write}(A)))(s_0) = \emptyset.$

Since both sequential and parallel composition are associative, we will often omit brackets and write $S_1 \,;\, \ldots \,;\, S_m$ and $S_1 \parallel \ldots \parallel S_n$.

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. For the more complicated semantics described in [4], the equivalence proof has been checked completely using the interactive theorem prover PVS.

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

- The operational semantics allows updates of the local database at any point in time, whereas in the denotational semantics updates occur once before and after each atomic statement.
- The parallel composition of the denotational semantics is defined by a few recursive equations and it is not obvious a priori that this indeed corresponds to the operational semantics.
- The underlying network is modeled in different ways. In the operational semantics, all produced items are collected in a single set. In the denotational semantics, there is a distinction between the produced items of a process and its environment; moreover, these environment writes are all available initially.

Equivalence is based on what is externally *observable*, i.e. two semantic functions are equivalent if they assign the same observable behaviour to any program. Here 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 of D by $Obs(D) = \{ownw(s) \mid s \in D\}$. For a set T of n-tuples $(s_1, \ldots, s_n)$ of statuses, define $Obs(T) = \bigcup\{\cup_{i,1\leq i\leq n} ownw(s_i) \mid (s_1, \ldots, s_n) \in T\}$.

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; $Ext(os)$ is defined by $st(Ext(os)) = st(os)$, $clock(Ext(os)) = clock(os)$, $db(Ext(os)) = db(os)$, $ownw(Ext(os)) = \emptyset$, and $envw(Ext(os)) = \emptyset$.

This leads to the main theorem.

Theorem 12 *If* $db(os) = \emptyset$, *then* $\mathcal{O}(P)(os) = Obs(\mathcal{M}(P)(Ext(os)))$.

Let $P = S_1 \parallel ... \parallel S_n$. We present the main steps of the proof, ignoring for instance details about initial conditions. The proof uses a few intermediate versions of the semantics. First, we define $\mathcal{OD}$, which extends the operational semantics $\mathcal{O}$ to the status of the denotational semantics (adding *ownw* and *envw*). Moreover, the network N is removed. This is achieved by defining the

atomic steps of a single sequential program as $(S, s) \xrightarrow{diset} (S', s')$, where *diset* represents the set of items written in the step (a singleton if S starts with a write statement, the empty set otherwise). $\mathcal{OD}$ also includes update steps that are similar to the updates of the denotational semantics, so including a condition on the value of the local clock. The local steps for a sequential program are shown in Fig. 4.

$$\frac{cl' \geq cl \quad X : DataBases \quad X \subseteq ow \cup ew \quad X <_{time} cl'}{\langle S\ ,\ (st, cl, db, ow, ew)\rangle \xrightarrow{\emptyset} \langle S\ ,\ (st, cl', UpdateDb(db, X), ow, ew)\rangle}$$

$$\frac{cl' > cl}{\langle \texttt{Write}(e); S\ ,\ (st, cl, db, ow, ew)\rangle \xrightarrow{\{(e(st), cl')\}} \langle S\ ,\ (st, cl', db, ow \cup \{(e(st), cl')\}, ew)\rangle}$$

$$\frac{q(st)(Di) \quad cl' \geq cl \quad Di \in db \text{ or } (Di = \perp \text{ and } \neg\exists di \in db, q(st)(di))}{\langle \texttt{Read}(x, q); S\ ,\ (st, cl, db, ow, ew)\rangle \xrightarrow{\emptyset} \langle S\ ,\ (st[x \mapsto Di], cl', db, ow, ew)\rangle}$$

Fig. 4. Local Computation Step Using Denotational Status

$$\frac{\langle S_i, s_i\rangle \xrightarrow{diset} \langle S', s'\rangle, \text{ for some } i,\ 1 \leq i \leq n \quad s'_j = s_j + diset, \text{ for all } j,\ j \neq i}{\langle (S_1, s_1), \ldots (S_n, s_n)\rangle \Rightarrow_{\mathcal{OD}} \langle (S_1, s'_1), \ldots (S', s'), \ldots (S_n, s'_n)\rangle}$$

$$\frac{}{C \Rightarrow_{\mathcal{OD}} C} \qquad \frac{C_1 \Rightarrow_{\mathcal{OD}} C_2 \quad C_2 \Rightarrow_{\mathcal{OD}} C_3}{C_1 \Rightarrow_{\mathcal{OD}} C_3}$$

Fig. 5. Global Computation Steps for $\mathcal{OD}$

The global steps of $\mathcal{OD}$ for a process are defined in Fig. 5. This leads to the definition of $\mathcal{OD}$.

Definition 13 ($\mathcal{OD}$) The operational semantics extended to denotational statuses is defined as a list of final statuses for each of the sequential programs of a process.

$$\mathcal{OD}(S_1 \parallel ... \parallel S_n)(s_0) = \{(s_1, \ldots s_n) \mid init(s_0) \wedge \langle (S_1\ ;\ E, s_0), ..., (S_n\ ;\ E, s_0)\rangle \Rightarrow_{\mathcal{OD}} \langle (E, s_1), ..., (E, s_n)\rangle\ \}$$

We will also use the above definition for a sequential program ($n = 1$), identifying a one-tuple with its element, yielding:

$$\mathcal{OD}(S)(s_0) = \{s \mid \langle S\ ;\ E, s_0\rangle \Rightarrow_{\mathcal{OD}} \langle E, s\rangle\ \}$$

We present the main outline of the proof, showing how suitable lemmas reduce the statement to be proved. The aim is to prove:

$\mathcal{O}(S_1 \parallel ... \parallel S_n)(os) = Obs(\mathcal{M}(S_1 \parallel ... \parallel S_n)(Ext(os)))$
For $\mathcal{OD}$ we can prove the following lemma.

Lemma 14 $\mathcal{O}(S_1 \parallel ... \parallel S_n)(os) = Obs(\mathcal{OD}(S_1 \parallel ... \parallel S_n)(Ext(os)))$

Then it remains to prove:
$Obs(\mathcal{OD}(S_1 \parallel ... \parallel S_n)(Ext(os))) = Obs(\mathcal{M}(S_1 \parallel ... \parallel S_n)(Ext(os)))$

The following lemma expresses the observations of a parallel program, according to the $\mathcal{OD}$ semantics, in terms of the observations of the sequential programs. To express that an individual component uses the items written by all other components as its environment writes, we define
$OtherWrites(os, i) = Ext(os)[envw \mapsto \cup_{j \neq i} ownw(s_j)]$.

Lemma 15 $Obs(\mathcal{OD}(S_1 \parallel ... \parallel S_n)(Ext(os))) =$
$Obs(\{(s_1, \ldots, s_n) \mid$ for all i, $1 \leq i \leq n$, $s_i \in \mathcal{OD}(S_i)(OtherWrites(os, i))\})$

Then it remains to show, assuming $1 \leq i \leq n$,
$Obs(\{(s_1, \ldots, s_n) \mid$ for all i, $s_i \in \mathcal{OD}(S_i)(OtherWrites(os, i))\}) =$
$Obs(\mathcal{M}(S_1 \parallel ... \parallel S_n)(Ext(os)))$

Observe that $\mathcal{M}$ is defined for the parallel composition of two processes; see the definition of $\mathcal{M}$ in Sect. 5. In the next lemma we prove a similar formulation for the application of $\mathcal{M}$ to the parallel composition of n sequential programs.

Lemma 16 $Obs(\mathcal{M}(S_1 \parallel S_2 \parallel ... \parallel S_n)(Ext(os))) =$
$Obs(\{(s_1, \ldots, s_n) \mid$ for all i, $s_i \in \mathcal{M}(S_i)(OtherWrites(os, i))\})$

Then it remains to show
$Obs(\{(s_1, \ldots, s_n) \mid$ for all i, $s_i \in \mathcal{OD}(S_i)(OtherWrites(os, i))\}) =$
$Obs(\{(s_1, \ldots, s_n) \mid$ for all i, $s_i \in \mathcal{M}(S_i)(OtherWrites(os, i))\})$
This follows trivially from the following lemma.

Lemma 17 $\mathcal{OD}(S) = \mathcal{M}(S)$, for any sequential program S.

This completes the outline of the proof of Theorem 12. The lemmas used above have been proved using the proof checker PVS. Here we only present the main ideas for the proof of the most complex lemma, namely Lemma 15. We prove
$Obs(\mathcal{OD}(S_1 \parallel ... \parallel S_n)(Ext(os))) =$
$Obs(\{(s_1, \ldots, s_n) \mid$ for all i, $s_i \in \mathcal{OD}(S_i)(OtherWrites(os, i))\})$

Proof:
We show 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,

$\langle (S_1\ ;\ E, Ext(os)), ..., (S_n\ ;\ E, Ext(os))\rangle \Rightarrow_{\mathcal{OD}} \langle (E, s_1), ..., (E, s_n)\rangle$.
By the definition of $\Rightarrow_{\mathcal{OD}}$, there exist a finite number of atomic steps:
$\langle (S_1\ ;\ E, Ext(os)), ..., (S_n\ ;\ E, Ext(os))\rangle \stackrel{diset_1}{\longrightarrow} \ldots \stackrel{diset_k}{\longrightarrow} \langle (E, s_1), ..., (E, s_n)\rangle$.
We have shown by induction on the number of steps in this execution sequence that it can be used to construct for each sequential program a local execution. Such a local execution starts with status $Ext(os)$ where the *envw*-field contains the items written by all other components, as expressed by $OtherWrites(os, i)$. This leads to $(S_i\,;\, E, OtherWrites(os, i)) \Rightarrow_{\mathcal{OD}} (E, s_i)$, i.e. $s_i \in \mathcal{OD}(S_i)(OtherWrites(os, i))$, for all i, $1 \leq i \leq n$.

$\supseteq$
Assume, for all i, $1 \leq i \leq n$ that $s_i \in \mathcal{OD}(S_i)(OtherWrites(os, i))$. Thus, for each of the sequential programs, we have an operational execution
$(S_i\ ;\ E, OtherWrites(os, i))) \stackrel{diset_1}{\longrightarrow} \ldots \stackrel{diset_k}{\longrightarrow} (E, s_i)$. We have to show,
$\langle (S_1; E, Ext(os)), ..., (S_n; E, Ext(os))\rangle \Rightarrow_{\mathcal{OD}} \langle (E, s_1), ..., (E, s_n)\rangle$, 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 construction of the global execution sequence is far from trivial, since the local, sequential executions start with all available environment writes, whereas in the global execution a process may only use what has been produced up to the current moment. However, the constraints on the local clocks (that have been included in the extended operational semantics $\mathcal{OD}$), ensure that only items are used that have been produced before its current local time. Formally, this is captured by the property that if $(S, s + diset) \stackrel{diset_1}{\longrightarrow} (S', s')$ (representing a step of a local process) and for all $di \in diset$, $ts(di) \geq clock(s')$ (i.e. *diset* contains only time stamps after the clock value in s') then $(S, s) \longrightarrow_{diset_1} (S', s'')$ where $envw(s'') = envw(s)$ and s'' equals s' for all other fields. Hence the step can be used in the global sequence without environment writes that have been produced later. □

7 Full Abstraction

As mentioned in Sect. 5, to obtain a denotational semantics, the definition of a status had to be extended and the meaning of each atomic statement has been defined in isolation, such that it can be used in any context. Typically, this means that in the denotational semantics more programs are distinguished than in the operational one. That is, in the denotational semantics more programs get a different semantics and less programs are identified. For instance, we have $\mathcal{O}((\texttt{Read}(x, A); \texttt{Write}(B)) \,\|\, (\texttt{Read}(x, B); \texttt{Write}(A)))(os_0) = \mathcal{O}(\texttt{Read}(x, A))(os_0) = \mathcal{O}(\texttt{Read}(x, B))(os_0) = \emptyset$, since the operational semantics considers each of them as the complete program and then they all block.

In the denotational semantics, all three programs have a different semantics; it always includes the possibility that the context in which it will be placed provides the required data items. In fact, they behave differently in a particular context and hence a denotational semantics should distinguish them.

Recall that in the equivalence proof of the previous section we have only proved equivalence for a particular initial status (where *envw* and *db* are empty) and with respect to a particular observation criterion, namely the set of published data items. The question remains whether we did not make too much distinctions to make the semantics compositional. Ideally, in the denotational semantics we should distinguish exactly those programs that behave differently in a particular context. This corresponds to the notion of *full abstraction*, which is defined formally below, using the notion of a *context* as defined by Table 3.

Table 3

Sequential Context	$SC ::=$	$[\,] \mid \texttt{Write}(e) \mid \texttt{Read}(x, q) \mid SC_1 \,;\, SC_2$
Context	$C ::=$	$SC \mid C_1 \parallel C_2$

Observe that the only new construct is [ ] which serves as an "open place" for which we can substitute a program to obtain a complete program. We often denote a context by $C[\,]$ to emphasize that there is an open place, and use $C[P]$ to denote the context $C[\,]$ where every occurrence of [ ] is replaced by P. In fact, we can restrict ourself here to contexts with exactly one open place.

Convention: if $C[P]$ is not syntactically correct (e.g. because a parallel program is inserted in a sequential context) then we define
$\mathcal{O}(C[P])(os_0) = \mathcal{M}(C[P])(s_0) = \emptyset$, for any os_0, s_0.

Definition 18 (Full Abstraction) *Semantic function $\mathcal{M}$ is* fully abstract *with respect to observable behaviour $\mathcal{O}$ if for every two processes P_1 and P_2,*
$\mathcal{M}(P_1) = \mathcal{M}(P_2)$ *iff for every context $C[\,]$ we have $\mathcal{O}(C[P_1]) = \mathcal{O}(C[P_2])$.*

Typically, it requires quite some effort to turn a denotational semantics into a fully abstract one. Here we claim that the denotational semantics is already fully abstract with respect to the operational one. Actually, some technical modifications of the semantics with respect to [4] were required, as we indicated when defining the current semantics. We will show along the way why these modifications were needed. One direction of the proof is easy; it is based on the equivalence result proved before. For the other direction we have to construct a context explicitly. It appears to be convenient to assume some structure on the data sorts, as we will explain later on.

Theorem 19 *$\mathcal{M}$ is fully abstract with respect to $\mathcal{O}$.*

Proof:

$\Rightarrow$
Assume $\mathcal{M}(P_1) = \mathcal{M}(P_2)$ and consider a context $C[\,]$ and an operational semantic primitive os. Then

$\mathcal{M}(P_1) = \mathcal{M}(P_2)$
$\quad \Rightarrow$ { since $\mathcal{M}$ is compositional }
$\mathcal{M}(C[P_1]) = \mathcal{M}(C[P_2])$
$\quad \Rightarrow$ { take $Ext(os)$ as initial status }
$\mathcal{M}(C[P_1])(Ext(os)) = \mathcal{M}(C[P_2])(Ext(os))$
$\quad \Rightarrow$
$Obs(\mathcal{M}(C[P_1])(Ext(os))) = Obs(\mathcal{M}(C[P_2])(Ext(os)))$
$\quad \Rightarrow$ { Theorem 12 }
$\mathcal{O}(C[P_1])(os) = \mathcal{O}(C[P_2])(os)$

$\Leftarrow$
Assume $\mathcal{M}(P_1) \neq \mathcal{M}(P_2)$. Without loss of generality, we can assume there exists s_0 and s_1 such that $s_1 \in \mathcal{M}(P_1)(s_0)$ and $s_1 \notin \mathcal{M}(P_2)(s_0)$. It suffices to construct a context $C[\,]$ such that $\mathcal{O}(C[P_1]) \neq \mathcal{O}(C[P_2])$.

Without loss of generality, we can assume that $envw(s_0)$ is finite. Moreover, it is allowed to assume $ownw(s_0) = \emptyset$, as can be shown as follows. If P_1 is a parallel program, then $ownw(s_0) = \emptyset$ follows from the *init* condition. Otherwise, we shift the elements of $ownw(s_0)$ to the *envw* field. Formally, define $s_0' = s_0[ownw \mapsto \emptyset, envw \mapsto envw(s_0) \cup ownw(s_0)]$. Then it can be proved (by an appropriate induction on P_1) that there exists an $s_1' \in \mathcal{M}(P_1)(s_0')$ with $ownw(s_0) \cup ownw(s_1') = ownw(s_1)$ and $ownw(s_1) \cup envw(s_1) = ownw(s_1') \cup envw(s_1')$. Further, we can prove that $s_1' \in \mathcal{M}(P_2)(s_0')$ implies $s_1 \in \mathcal{M}(P_2)(s_0)$, which contradicts our assumption, so we have $s_1' \notin \mathcal{M}(P_2)(s_0')$.

Similarly, we can assume that $db(s_0) = \emptyset$ by shifting the elements of $db(s_0)$ to the *envw* field. Then the update included in the first read action of P_1 can use these elements to reconstruct $db(s_0)$ as far as the items are not overwritten. [3]

We have to show that there exists a context C and initial status os_0 such that $\mathcal{O}(C[P_1])(os_0) \neq \mathcal{O}(C[P_2])(os_0)$. By the equivalence result, Theorem 12, it is sufficient to show that there exists a context C and an initial status $\hat{s}_0$ with $ownw(\hat{s}_0) = envw(\hat{s}_0) = db(\hat{s}_0) = \emptyset$ such that
$Obs(\mathcal{M}(C[P_1])(\hat{s}_0)) \neq Obs(\mathcal{M}(C[P_2])(\hat{s}_0))$.

Let $\hat{s}_0$ be such that $envw(\hat{s}_0) = \emptyset$ and it equals s_0 for the other fields. Thus $s_0 = \hat{s}_0 + envw(s_0)$, and since $envw(s_1) = envw(s_0)$, also $s_0 = \hat{s}_0 + envw(s_1)$.

The construction of the context depends on whether the programs are sequen-

[3] At this point we use that items in the database must have time stamp smaller than the clock, otherwise we would have to increase the clock value.

tial or parallel. Before going into this case distinction we introduce some extra notation for queries and expressions. In the next three subsections we distinguishes three cases: both programs are sequential, both are parallel, and one is sequential and the other is parallel. We give the proof for the first, most complicated, case.

Notation for tags in expressions

It is sometimes needed to distinguish elements written by the context from elements written by the original program. To this end, we may extend all data sorts with an additional tag field. The *tag* field may have values E (denoting items written by the environment), V (denoting items representing values of variables), and D (denoting items of the database). As noted in the introduction, such extensions are transparent for P_1 and P_2; as P_1 and P_2 are subscribed to the original sorts, their local database performs a suitable selection of relevant fields.

Also, in some cases we want to recognize some data items exactly, including time stamp, and also when they are $\perp$. To this end we allow `Write` actions with expressions of sort $DataItems^{\perp}$, i.e. time stamps are added as additional data fields. We use $\texttt{Write}(\langle x, T\rangle)$ to denote that the data item in x is written with an additional *tag* field with value T. Also $\texttt{Write}(v, T)$ is used to write a data value v tagged with T.

This extension to a multi-sorted language is realistic from the point of view of the real Splice (cf. [2]). It greatly simplifies the proofs. It is not clear if having multiple sorts is strictly needed for full abstraction, but the proof below breaks down if items written by the environment are indistinguishable from items written by P_1 and P_2 (see also Ex. 26)

Two sequential programs

Context C is now constructed based on s_1 such that it produces the elements in $envw(s_1)$ (i.e., $envw(s_0)$), represented by C_e, and after termination of the program the values of all variables and the contents of the database are written. Let $x_1, \ldots, x_n$ be the finite list of all variables occurring in P_1 or P_2, $di_1, \ldots, di_m$ be the finite list of data items occurring in $ownw(s_1) \cup envw(s_1)$ (which is the maximal set of data items that may occur in $db(s_1)$ - note that we assume some ordering on the items), and $e_1, \ldots, e_k$ be the finite list of data values (i.e. the data part of the data items) occurring in $envw(s_1)$. We define the context as follows.

$$C = ([\,]\,;\, C_v\,;\, C_d) \parallel C_e$$

where

$C_v = \texttt{Write}(\langle x_1, \mathsf{V}\rangle)\ ;\ \ldots\ ;\ \texttt{Write}(\langle x_n, \mathsf{V}\rangle)$
$C_d = \texttt{Read}(y, di_{1\perp})\ ;\ \texttt{Write}(\langle y, \mathsf{D}\rangle)\ ;\ \ldots\ ;\ \texttt{Read}(y, di_{m\perp})\ ;\ \texttt{Write}(\langle y, \mathsf{D}\rangle)$
$C_e = \texttt{Write}(\langle e_1, \mathsf{E}\rangle) \parallel \ldots \parallel \texttt{Write}(\langle e_k, \mathsf{E}\rangle)$

Thus, C_v publishes the values of the variables in a particular order with tag V. We use $\langle st(s_1), \mathsf{V}\rangle$ to denote the set of written items that corresponds to the values of the variables in s_1. Observe that C_d tries to read all possible values from the database in a non-blocking way and publishes the result with tag D; hence it writes $\langle \perp, \mathsf{D}\rangle$ iff the item is not in the database. We use $\langle db(s_1), \mathsf{D}\rangle$ to denote the set of writes that corresponds exactly to $db(s_1)$. Context C_e writes the data values that occur in $envw(s_1)$ with tag E. It is essential that C_e is parallel, because it may have to write several data items with the same time stamp. [4] Let $\langle envw(s_1), \mathsf{E}\rangle$ be the set of data items that correspond to $envw(s_1)$, i.e. with the same time stamps.

Lemma 20 There exists a status $s \in \mathcal{M}((P_1\ ;\ C_v\ ;\ C_d) \parallel C_e)(\hat{s}_0)$ with
$ownw(s) = \langle envw(s_1), \mathsf{E}\rangle \cup ownw(s_1) \cup \langle st(s_1), \mathsf{V}\rangle \cup \langle db(s_1), \mathsf{D}\rangle$.

Proof:
By the construction of C_e, which cannot block, there is an $s_e \in \mathcal{M}(C_e)(\hat{s}_0)$ with $ownw(s_e) = \langle envw(s_1), \mathsf{E}\rangle$. Since environment writes can always be extended without affecting an existing execution, there is an $s'_e \in \mathcal{M}(C_e)(\hat{s}_0 + ownw(s'_1))$ with $ownw(s'_e) = \langle envw(s_1), \mathsf{E}\rangle$, for any s'_1.

Using $\hat{s}_0 + envw(s_1) = s_0$, we obtain that $s_1 \in \mathcal{M}(P_1)(\hat{s}_0 + envw(s_1))$. Since P_1 is not affected by the additional tags, also $s_1 \in \mathcal{M}(P_1)(\hat{s}_0 + ownw(s'_e))$. Note that C_v and C_d do not block and there exists an $s'_1 \in \mathcal{M}(C_v\ ;\ C_d)(s_1)$ with $ownw(s'_1) = ownw(s_1) \cup \langle st(s_1), \mathsf{V}\rangle \cup \langle db(s_1), \mathsf{D}\rangle$. Hence, by sequential composition, we have that $s'_1 \in \mathcal{M}(P_1\ ;\ C_v\ ;\ C_d)(\hat{s}_0 + ownw(s'_e))$.

Combining s'_e and s'_1 we obtain an $s \in \mathcal{M}((P_1; C_v; C_d) \| C_e)(\hat{s}_0)$ with $ownw(s) = \langle envw(s_1), \mathsf{E}\rangle \cup ownw(s_1) \cup \langle st(s_1), \mathsf{V}\rangle \cup \langle db(s_1), \mathsf{D}\rangle$. □

For the next step in the proof we need a few lemmas about the value of the local clock. The first lemma expresses that the clock is always larger than the time stamps in the database and not smaller than the time stamps in the produced data items.

Lemma 21 For any sequential program S, if $db(s_0) = ownw(s_0) = \emptyset$ and

[4] Observe that C_e cannot produce items with time stamp 0, according to the semantics of a write statement in Section 5. But, in Section 5, we also required that *envw* only contains items with time stamp bigger than 0.

$s \in \mathcal{M}(S)(s_0)$ then

(1) for all $di \in db(s)$, $clock(s) > ts(di)$
(2) for all $di \in ownw(s)$, $clock(s) \geq ts(di)$

The next lemma expresses the other direction; if time t is larger than the time stamps in the database and not smaller than the time stamps in the produced items, then t occurs in the semantics as a possible value of the clock.

Lemma 22 For any sequential programs S, if $s \in \mathcal{M}(S)(s_0)$, $t \geq clock(s_0)$ and

(1) for all $di \in db(s)$, $t > ts(di)$
(2) for all $di \in ownw(s)$, $t \geq ts(di)$

then $s[clock \mapsto t] \in \mathcal{M}(S)(s_0)$

These lemmas are used to prove the following.

Lemma 23 There exists no status $s' \in \mathcal{M}((P_2 \ ; \ C_v \ ; \ C_d) \parallel C_e)(\hat{s}_0)$ with $ownw(s') = \langle envw(s_1), \mathsf{E}\rangle \cup ownw(s_1) \cup \langle st(s_1), \mathsf{V}\rangle \cup \langle db(s_1), \mathsf{D}\rangle$.

Proof:
This is proved by contradiction, so suppose there exists a status s' with $s' \in \mathcal{M}((P_2; C_v; C_d)\|C_e)(\hat{s}_0)$ and $ownw(s') = \langle envw(s_1), \mathsf{E}\rangle \cup ownw(s_1) \cup \langle st(s_1), \mathsf{V}\rangle \cup \langle db(s_1), \mathsf{D}\rangle$. Since the items with tag E must have been produced by C_e, there exists an s'_e in the semantics of C_e with $ownw(s'_e) = \langle envw(s_1), \mathsf{E}\rangle$ and an $s'_2 \in \mathcal{M}(P_2; C_v; C_d)(\hat{s}_0 + ownw(s'_e))$, with $ownw(s'_2) = ownw(s_1) \cup \langle st(s_1), \mathsf{V}\rangle \cup \langle db(s_1), \mathsf{D}\rangle$. Since the E-tags are not used by $P_2; C_v; C_d$ and $\hat{s}_0 + envw(s_1) = s_0$, we obtain $s'_2 \in \mathcal{M}(P_2 \ ; \ C_v \ ; \ C_d)(s_0)$. Hence there exist an s_2 such that $s_2 \in \mathcal{M}(P_2)(s_0)$ and $s'_2 \in \mathcal{M}(C_v; C_d)(s_2)$. Since P_2 does not write the V and D tags and the program $C_v \ ; \ C_d$ only writes tagged items, we have that $ownw(s_2) = ownw(s_1)$. Since C_v produces $\langle st(s_1), \mathsf{V}\rangle$, we obtain $st(s_2) = st(s_1)$.
Observe that the fact that C_d produces $\langle db(s_1), \mathsf{D}\rangle$ doesn't imply that $db(s_2) = db(s_1)$. Actually, some database updates can occur during execution of C_d, and we only know that at the time di_k is written, the database indeed contains di_k. However, from the fact that these updates are possible, we conclude that either db_2 doesn't contain items with the same key, or db_2 contains an item with the same key, but with a smaller time stamp. So all these updates can be combined in a single update, which is glued to the last atomic action of P_2.[5] Hence there exists an $s_3 \in \mathcal{M}(P_2)(s_0)$ with $st(s_3) = st(s_1)$, $db(s_3) = db(s_1)$, $ownw(s_3) = ownw(s_1)$, and $envw(s_3) = envw(s_1)$. This leads to $s_1[clock \mapsto$

[5] Here we use the fact that the atomic actions are followed by an update. The programs $\texttt{Write}(x)$ and $\texttt{Write}(x);\texttt{Read}(x, \{x\})$ would be denotationally different without the update after `Write`, leading to a counter example for full abstraction.

$clock(s_3)] \in \mathcal{M}(P_2)(s_0)$.
Using Lemma 21, we have for all $di \in db(s_1)$, $clock(s_1) > ts(di)$, and for all $di \in ownw(s_1)$, $clock(s_1) \geq ts(di)$. By $s_1[clock \mapsto clock(s_3)] \in \mathcal{M}(P_2)(s_0)$ and Lemma 22, we obtain $s_1[clock \mapsto clock(s_3)][clock \mapsto clock(s_1)] \in \mathcal{M}(P_2)(s_0)$, i.e. $s_1 \in \mathcal{M}(P_2)(s_0)$. Contradiction. □

Finally, observe that by Lemma 20 and Lemma 23 there exists an s such that $ownw(s) \in Obs(\mathcal{M}(C[P_1])(\hat{s}_0))$ and $ownw(s) \notin Obs(\mathcal{M}(C[P_2])(\hat{s}_0))$, so $Obs(\mathcal{M}(C[P_1])(\hat{s}_0)) \neq Obs(\mathcal{M}(C[P_2])(\hat{s}_0))$.

We present a few small examples that show how the context distinguishes sequential programs. In the first example, the final states are different.

Example 24 *The programs P_1:* $\texttt{Read}(x_1, A)$ *and P_2:* $\texttt{Read}(x_2, A)$ *are denotationally different, because the values of x_1 and x_2 might be different, e.g. if the initial status s_0 is such that $st(s_0)(x_1) = st(s_0)(x_2) = 0$ and $envw(s_0)$ contains an item with value A and time stamp 10. The construction above leads to the context*

$$C = ([\,]\,;\ \texttt{Write}(\langle x_1, \mathsf{V}\rangle)\,;\ \texttt{Write}(\langle x_2, \mathsf{V}\rangle)\,;\ \texttt{Read}(y, (\#dat := A, ts := 10\#)_\perp)\,;\ \texttt{Write}(\langle y, \mathsf{D}\rangle)) \parallel \texttt{Write}(\langle A, \mathsf{E}\rangle)$$

Note that the construction of the context depends on a particular status s_1 that shows a difference between the two programs. Since there might be several differences, this may lead to several possible contexts that distinguish the programs. This is illustrated by the next example.

Example 25 *Suppose we have the two programs P_1:* $\texttt{Read}(x, A)\,;\ \texttt{Read}(x, \perp)$ *and P_2:* $\texttt{Read}(x, B)\,;\ \texttt{Read}(x, \perp)$. *There are several possible differences, leading to different contexts.*

- *If $db(s_0) = ownw(s_0) = \emptyset$ and $envw(s_1)$, which equals $envw(s_0)$, only contains an item with value A then the second program blocks and first one does not. This leads to a context of the form $C = ([\,]\,;\ C_v\,;\ C_d) \parallel \texttt{Write}(\langle A, \mathsf{E}\rangle)$*
- *Another possibility is that $db(s_0) = ownw(s_0) = \emptyset$ and $envw(s_1)$ (and hence $envw(s_0)$) contains two data items, with values A and B and local time stamps 6 and 8, respectively. If these items have the same key, then P_1 may have the item with value A in its database, which is not possible for P_2 because it must have B in its database before the read and this cannot be overwritten by value A which has a smaller time stamp. This difference is made visible by the following context:*

$$C = ([\,]\,;\ \texttt{Write}(\langle x, \mathsf{V}\rangle)\,;\ \texttt{Read}(y, (\#dat := A, ts := 6\#)_\perp)\,;\ \texttt{Write}(\langle y, \mathsf{D}\rangle)\,;\ \texttt{Read}(y, (\#dat := B, ts := 8\#)_\perp)\,;\ \texttt{Write}(\langle y, \mathsf{D}\rangle)) \parallel \texttt{Write}(\langle A, \mathsf{E}\rangle) \parallel \texttt{Write}(\langle B, \mathsf{E}\rangle)$$

Two parallel programs

Suppose P_1 and P_2 are parallel programs. Then we do not use the sequential part of the context (this would lead to syntactically invalid programs), but define the context by

$$C = [\,] \parallel C_e$$

The proof that this indeed distinguishes the two programs is a simple version of the proof for two sequential programs.

We present a small example that indicates why we have used the tags.

Example 26 *Consider the programs*
P_1*:* $\texttt{Write}(A) \parallel (\texttt{Read}(x, A)\,;\, \texttt{Write}(B)) \parallel \texttt{Read}(y, \bot)$
P_2*:* $(\texttt{Write}(A)\,;\, \texttt{Write}(B)) \parallel \texttt{Read}(y, \bot)$
Statement $\texttt{Read}(y, \bot)$ *has been added to obtain two parallel programs. If* $db(s_0) = ownw(s_0) = \emptyset$ *and* $envw(s_1)$ *only contains an item with value* A *then* P_1 *may read this and write its* A *after the* B*. Program* P_2 *will always write* B *after* A*.*

If we use a context without tags, i.e. $C = [\,] \parallel \texttt{Write}(A)$ *then the own writes of* $C[P_1]$ *may contain an* A *(produced by the context), followed by a* B *and another* A *(ordering them by time stamp). But this is also possible for* $C[P_2]$*, since there the own writes may contain an* A *followed by a* B *(produced by* P_2*), followed by an* A *produced by the context. The problem is that without tags we cannot observe which item was produced by the context.*

A sequential and a parallel program

To prove that we can make a distinction in general, we distinguish three cases:

- If P_1 is sequential and P_2 is a parallel program, then use some sequential context, say $C = [\,]\,;\, C_v$. Since $s_1 \in \mathcal{M}(P_1)(s_0)$, we can prove that $\mathcal{M}(C[P_1])(\hat{s}_0) \neq \emptyset$. But $C[P_2]$ is a syntactically incorrect program, so by convention we have $\mathcal{M}(C[P_2])(\hat{s}_0) = \emptyset$.
- Similarly, if P_1 is parallel and P_2 is sequential with $\mathcal{M}(P_2)(s_0) \neq \emptyset$, we can also use context $C = [\,]\,;\, C_v$.
- If P_1 is parallel and P_2 is a sequential program with $\mathcal{M}(P_2)(s_0) = \emptyset$, then use context $C = [\,] \parallel C_e$, since $\mathcal{M}(P_1)(s_0) \neq \emptyset$ implies $\mathcal{M}(P_1 \parallel C_e)(\hat{s}_0) \neq \emptyset$. $\square$

8 Verification Framework

In this section, we provide a framework that can be used to specify and verify processes, as shown in Sect. 8.3. First, in Sect. 8.1, the programming language is extended with an infinite loop. Section 8.2 contains the main specification and verification constructs. The same framework has been used in [3] to verify transparent replication in another example.

8.1 Language Extensions

The simple programming language of Sect. 3 is extended with 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.

$$\begin{aligned}&\mathcal{M}(S_1\ ;\ S_2)(s_0) = \\ &\quad \{s \mid s \in \mathcal{M}(S_1)(s_0) \wedge \neg term(s)\} \cup \\ &\quad \{s \mid \text{there exists an } s_1 \text{ with } s_1 \in \mathcal{M}(S_1)(s_0) \wedge\ term(s_1) \wedge s \in \mathcal{M}(S_2)(s_1)\}\end{aligned}$$

We define the meaning of 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, as long as *term* is true for the start state of this execution (we should also include the data items produced when the body does not terminate).

$$\begin{aligned}&\mathcal{M}(\texttt{Do}\ S\ \texttt{Od})(s_0) = \\ &\quad \{s \mid \neg term(s) \text{ and there exists a sequence } s_1, s_2, \ldots \text{ such that for all } i \geq 0, \\ &\qquad \text{if } term(s_i) \text{ then } s_{i+1} \in \mathcal{M}(S)(s_i) \text{ else } term(s_{i+1}) = false, \\ &\qquad ownw(s) = \cup_{\{i \geq 0 \mid term(s_i)\}} ownw(s_{i+1}), \text{ and } envw(s) = envw(s_0)\ \}\end{aligned}$$

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 [27].

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. We will use the usual Boolean connectives (e.g. $\rightarrow$, $\leftrightarrow$, $\wedge$) on assertions. A *specification* is a "program" of the form $\texttt{Spec}(p, q)$ with the following meaning.

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

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

Definition 27 (Refinement) For any two programs P_1, P_2, we define that P_1 is a refinement of P_2 (denoted by $P_1 \Rightarrow P_2$) as follows:
$P_1 \Rightarrow P_2$, 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, which expresses that we can refine a specification by strengthening the precondition and weakening the postcondition.

Lemma 28 (Consequence)
If $p \rightarrow p_0$ and $q_0 \rightarrow q$ then $\texttt{Spec}(p_0, q_0) \Rightarrow \texttt{Spec}(p, q)$.

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 29 (Sequential Composition)
$(\texttt{Spec}(p, r)\ ;\ \texttt{Spec}(r, q)) \Rightarrow \texttt{Spec}(p, q)$.

Lemma 30 (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. But we do have a monotonicity rule for parallel composition, which forms the basis of stepwise refinement of components. Note that our main motivation to develop a denotational semantics has been to obtain the following

rule.

Lemma 31 (Monotonicity of Parallel Composition)
If $P_3 \Rightarrow P_1$ and $P_4 \Rightarrow P_2$ then $(P_3 \parallel P_4) \Rightarrow (P_1 \parallel P_2)$.

8.3 Verification Example

To illustrate the reasoning about Splice components in PVS, we consider a simple system with two processes that produce data items based on previously written data by the other component. So they mutually depend on each other. As a simple example, we consider two components that produce even and odd numbers, based on each others output.

In Sect. 8.3.1, we define the top-level specification of the system. A failed decomposition attempt is shown in Sect. 8.3.2. The main problem of the correctness of this decomposition is mutual dependency: both of the components is correct if the other is. We show how our formalization blocks this cyclic reasoning. Based on the reasoning problems encountered there, we rewrite the specifications in Sect. 8.3.3 and prove the correctness of the new decomposition. The components are implemented in Sect. 8.3.4.

8.3.1 Top-level Specification

To formalize the top-level specification of the system, we define the following types and functions:

- $DataName = \{Even, Odd\}$, 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*.
- $KeyData = DataName$ and $key(v) = name(v)$.
- $Vars = \{evenvar, oddvar\}$, variables over data items that are used later in the implementation of the components (for simplicity, we have not introduced hiding or scoping rules).

To formulate the specifications, first a few preliminary definitions are needed, where *diset* is a set of data items, and *even*? and *odd*? are predicates that hold when a number is even or odd, respectively.

- $Even(diset) = \{di \mid di \in diset \wedge name(di) = Even\}$
- $Odd(diset) = \{di \mid di \in diset \wedge name(di) = Odd\}$
- *Increasing*(*diset*) holds iff
$$\forall di_1 \in diset, di_2 \in diset : (val(di_1) < val(di_2) \leftrightarrow ts(di_1) < ts(di_2))$$

- $EvenNrs(diset)$ holds iff $\forall di \in diset : even?(val(di))$
- $OddNrs(diset)$ holds iff $\forall di \in diset : odd?(val(di))$
- $EvenWrites(diset)$ holds iff $Increasing(Even(diset)) \wedge EvenNrs(Even(diset))$
- $OddWrites(diset)$ holds iff $Increasing(Odd(diset)) \wedge OddNrs(Even(diset))$

Let *pre* be an assertion expressing that $envw = \emptyset$ and the variables *evenvar* and *oddvar* are initialized to the empty set. The top-level specification of the overall system is defined as follows.

$$postTopLevel(s) = EvenWrites(ownw(s)) \wedge OddWrites(ownw(s))$$

$$TopLevel = \texttt{Spec}(pre, postTopLevel)$$

8.3.2 Failed Decomposition Attempt

The aim is to specify two components, say *EvenComp* and *OddComp*, such that $EvenComp \parallel OddComp \Rightarrow TopLevel$.

Since it is important to express which components produce certain data items, we define

- $NameOwnw(dn)(s) = \forall di \in ownw(s) : name(di) = dn$

The main idea is that component *EvenComp* first writes 0 and next reads an item with name *Odd*, stores it in variable *oddvar*, and uses this item – assuming it is odd indeed – to produce a new even number. Similarly, *OddComp* reads *Even* items, stores them in *evenvar* and uses them to produce odd numbers.

Let *preEvenComp* be an assertion expressing that *db*, *ownw* and *oddvar* are all empty. Similarly, *preOddComp* expresses that *db*, *ownw* and *evenvar* are all empty. Then we try the following specifications of the components:

$$postEven(s) = NameOwnw(Even)(s) \wedge (OddWrites(envw(s)) \rightarrow EvenWrites(ownw(s)))$$

$$EvenCompTry = \texttt{Spec}(preEvenComp, postEven)$$

$$postOdd(s) = NameOwnw(Odd)(s) \wedge (EvenWrites(envw(s)) \rightarrow OddWrites(ownw(s)))$$

$$OddCompTry = \texttt{Spec}(preOddComp, postOdd)$$

The aim is to prove $EvenCompTry \parallel OddCompTry \Rightarrow TopLevel$. Since the precondition of *TopLevel* requires that $envw = \emptyset$ for the complete system, this reduces to the obligation to prove for s, s_1 and s_2 that
$(OddWrites(ownw(s_2)) \rightarrow EvenWrites(ownw(s_1)))$ and

$(EvenWrites(ownw(s_1)) \rightarrow OddWrites(ownw(s_2)))$ implies
$EvenWrites(ownw(s)) \wedge OddWrites(ownw(s))$, where
$ownw(s) = ownw(s_1) \cup ownw(s_2)$.

Unfortunately, it is not possible to draw any suitable conclusion from the mutually dependent specifications of the components. Hence we rewrite these specifications in the next section to allow some form of inductive reasoning.

8.3.3 Correct Components

To obtain specifications that can be used for inductive reasoning at parallel composition, we rewrite them such that a property of a component up to some point of time $n+1$ has to be established using the assumptions up to time n. This means that we use local clock values as a basis for induction and we take $LocalTime = \mathbb{N}$.

Let n be a variable ranging over $\mathbb{N}$ and define:

- $(diset < n) = \{di \mid di \in diset \wedge ts(di) < n\}$

Observe that $(diset < 0) = \emptyset$.

Then we specify the components as follows:

$$postEvenComp(s) = NameOwnw(Even)(s) \wedge (\forall n : OddWrites(ownw(s) < n) \rightarrow EvenWrites(ownw(s) < n+1))$$

$$EvenComp = \texttt{Spec}(preEvenComp, postEvenComp)$$

$$postOddComp(s) = NameOwnw(Odd)(s) \wedge (\forall n : EvenWrites(ownw(s) < n) \rightarrow OddWrites(ownw(s) < n+1))$$

$$OddComp = \texttt{Spec}(preOddComp, postOddComp)$$

Note that we have not formalized that *EvenComp* first writes value 0 because we only consider safety here, i.e., we show that no wrong numbers are produced.

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

Theorem 32 $EvenComp \parallel OddComp \Rightarrow TopLevel$

Proof:
Since $envw = \emptyset$, by the precondition of *TopLevel*, we have to show that for s, s_1 and s_2 with $NameOwnw(Even)(s_1)$, $NameOwnw(Odd)(s_2)$,

$\forall n : OddWrites(ownw(s_2) < n) \rightarrow EvenWrites(ownw(s_1) < n+1)$ and
$\forall n : EvenWrites(ownw(s_1) < n) \rightarrow OddWrites(ownw(s_2) < n+1)$ imply
$EvenWrites(ownw(s_1) \cup ownw(s_2)) \wedge OddWrites(ownw(s_1) \cup ownw(s_2))$.
This follows immediately from the following three properties:

- $\forall n : OddWrites(ownw(s_2) < n) \rightarrow EvenWrites(ownw(s_1) < n+1)$ and
  $\forall n : EvenWrites(ownw(s_1) < n) \rightarrow OddWrites(ownw(s_2) < n+1)$
  imply $\forall i : EvenWrites(ownw(s_1) < i) \wedge OddWrites(ownw(s_2) < i)$.
  We have proved this property by induction on i. The case for $i = 0$ depends on the fact that the predicates *EvenWrites* and *OddWrites* hold for the empty set. The inductive case is almost trivial and does not depend on the predicates.
- $\forall i : EvenWrites(ownw(s_1) < i) \wedge OddWrites(ownw(s_2) < i)$
  implies $EvenWrites(ownw(s_1)) \wedge OddWrites(ownw(s_2))$.
  This property depends on the predicates used (*EvenWrites* and *OddWrites*), but is not difficult here; given particular data items, choose i larger than their time stamps.
- $NameOwnw(Even)(s_1)$, $NameOwnw(Odd)(s_2)$, $EvenWrites(ownw(s_1))$, and $OddWrites(ownw(s_2))$ imply $EvenWrites(ownw(s_1) \cup ownw(s_2))$ and $OddWrites(ownw(s_1) \cup ownw(s_2))$.
  This property can be proved almost automatically by PVS.

  □

8.3.4 Implementing the Components

To implement the components in our Splice-like programming language, we define the following notation.

- $NewNamed(x, dn)$ is a query that requires a data item with name dn and a time stamp which is larger than the current time stamp of x (provided $x \neq \perp$).

Then implement *EvenComp* by the program

$$\begin{aligned} EvenProg = \; & \texttt{Write}((\#name := Even, val := 0\#)) \; ; \\ & \texttt{Do Read}(oddvar, NewNamed(oddvar, Odd)) \; ; \\ & \quad \texttt{Write}((\#name := Even, val := val(oddvar) + 1\#)) \; \texttt{Od} \end{aligned}$$

We have proved in PVS that this is indeed a correct refinement.

Lemma 33 *EvenProg* ⇒ *EvenComp*

Similarly, *OddComp* is implemented by

$$\begin{aligned} OddProg = \; & \texttt{Do Read}(evenvar, NewNamed(evenvar, Even)) \; ; \\ & \quad \texttt{Write}((\#name := Odd, val := val(evenvar) + 1\#)) \; \texttt{Od} \end{aligned}$$

and we have proved the following lemma

Lemma 34 $OddProg \Rightarrow OddComp$

Finally observe that Theorem 32, expressing the correctness of the decomposition, and the Lemmas 33 and 34, which concern the correctness of the components, lead by the monotonicity property (Lemma 31) to

$$(EvenProg \parallel OddProg) \Rightarrow TopLevel$$

9 Concluding Remarks

We have proposed a compositional verification framework for reasoning about components 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. Verification is supported by the interactive theorem prover PVS.

This formal framework is based on a new denotational semantics for Splice which includes 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 includes assumptions about the data items produced by the environment of a component. To simplify verification, we reduced 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 and proved that it is equivalent to the denotational one. This revealed a number of errors in earlier versions of the semantics. In general, our study of the semantics of Splice led to many discussions about the precise meaning of this software architecture. As a final justification of the semantics, we have proved that the denotational semantics is fully abstract with respect to the operational semantics.

Acknowledgment

We are grateful to Edwin de Jong and Paul Dechering for many extensive discussions on the meaning of the Splice architecture. We also like to thank Simona Orzan for discussions on full abstraction, and the anonymous referees for their many valuable remarks. The research was carried out in the project CES.5009 funded by PROGRESS.

References

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

[2] M. Boasson, Software architecture for distributed reactive systems, in: B. Rovan (Ed.), SOFSEM '98: Theory and Practice of Informatics, LNCS 1521, Springer-Verlag, 1998, pp. 1–18.

[3] J. Hooman, J. C. van de Pol, Formal verification of replication on a distributed data space architecture, in: Proc. of the 2000 ACM Symp. on Applied Computing, (SAC '02), 2002, pp. 351–358.

[4] J. Hooman, J. C. van de Pol, Equivalent semantic models for a distributed dataspace architecture, in: F. S. de Boer, M. Bonsangue, S. Graf, W. P. de Roever (Eds.), Formal Methods for Components and Objects, First Int. Symp. (FMCO '02), LNCS 2852, Springer-Verlag, 2003, pp. 182–201.

[5] U. Hannemann, J. Hooman, Formal reasoning about real-time components on a data-oriented architecture, in: Systemics, Cybernetics and Informatics, 6th World Multiconf. (SCI '02), Vol. XI, 2002, pp. 313–318.

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

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

[8] S. M. Orzan, J. C. van de Pol, Distribution of a simple shared dataspace architecture, in: A. Brogi, J.-M. Jacquet (Eds.), Foundations of Coordination Languages and Software Architectures, First Int. Workshop (FLOCASA'02), Electronic Notes in Theoretical Computer Science 68(3), 2003.

[9] M. M. Bonsangue, J. N. Kok, G. Zavattaro, Comparing coordination models and architectures using embeddings, Science of Computer Programming 46 (1-2) (2003) 31–69.

[10] F. S. de Boer, M. Gabbrielli, M. C. Meo, A timed Linda language, in: A. Porto, G.-C. Roman (Eds.), Coordination Models and Languages, Fourth Int. Conf. (COORDINATION '00), LNCS 1906, Springer-Verlag, 2000, pp. 299–304.

[11] J.-M. Jacquet, K. De Bosschere, A. Brogi, On timed coordination languages, in: A. Porto, G.-C. Roman (Eds.), Coordination Models and Languages, Fourth Int. Conf. (COORDINATION '00), LNCS 1906, Springer-Verlag, 2000, pp. 81–98.

[12] N. Busi, R. Gorrieri, G. Zavattaro, Temporary data in shared dataspace coordination languages, in: F. Honsell, M. Miculan (Eds.), Foundations of Software Science and Computation Structures, 4th Int. Conf. (FOSSACS '01), LNCS 2030, 2001, pp. 121–136.

[13] R. Bloo, J. Hooman, E. de Jong, Semantical aspects of an architecture for distributed embedded systems, in: Proc. of the 2000 ACM Symp. on Applied Computing, (SAC '00), Vol. 1, ACM press, 2000, pp. 149–155.

[14] S. Owre, J. Rushby, N. Shankar, PVS: A prototype verification system, in: 11th Conference on Automated Deduction, LNAI 607, Springer-Verlag, 1992, pp. 748–752.

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

[16] R. Gerth, A. Boucher, A timed failures model for extending communicating processes, in: T. Ottmann (Ed.), Automata, Languages and Programming, 14th Int. Coll. (ICALP '87), LNCS 267, Springer-Verlag, 1987, pp. 95–114.

[17] F. S. de Boer, J. Hooman, The real-time behaviour of asynchronously communicating processes, in: J. Vytopil (Ed.), Formal Techniques in Real-Time and Fault-Tolerant Systems, Second Int. Symp., LNCS 571, Springer-Verlag, 1992, pp. 451–472.

[18] F. S. de Boer, R. van Eijk, W. van der Hoek, J.-J. Meyer, A fully abstract model for the exchange of information in multi-agent systems, Theoretical Computer Science 290 (3) (2003) 1753–1773.

[19] J. Misra, K. M. Chandy, Proofs of networks of processes, IEEE Trans. Softw. Eng. 7 (7) (1981) 417–426.

[20] C. B. Jones, Tentative steps towards a development method for interfering programs, ACM Trans. Prog. Lang. Syst. 5 (4) (1983) 596–619.

[21] J. Zwiers, A. de Bruin, W. P. de Roever, A proof system for partial correctness of dynamic networks of processes, in: E. Clarke, D. Kozen (Eds.), Logic of Programs, Workshop ('83), LNCS 164, Springer-Verlag, 1984, pp. 513–522.

[22] J. Rushby, Formal verification of McMillan's compositional assume-guarantee rule, Tech. Rep. CSL Technical Report, SRI International (September 2001).

[23] J. Hooman, Compositional verification of distributed real-time systems, in: Proceedings Workshop on Real-Time Systems - Theory and Applications, North-Holland, 1990, pp. 1–20.

[24] D. Gelernter, Generative communication in Linda, Transactions on Programming Languages and Systems 7 (1) (1985) 80–112.

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

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

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