Semantical Aspects of an Architecture for Distributed Embedded Systems*

Roel Bloo
Eindhoven University of Technology
Eindhoven, The Netherlands
bloo@win.tue.nl

Jozef Hooman
University of Nijmegen
Nijmegen, The Netherlands
hooman@cs.kun.nl

Edwin de Jong
Hollandse Signaalapparaten B.V.
Hengelo, The Netherlands
edejong@signaal.nl

ABSTRACT

We investigate the formulation of a formal semantics for the industrial software architecture Splice. In this paper, we present a set of basic Splice interaction primitives that is both powerful and easy to implement. We define a semantics for this language based on a conceptual global dataspace. It is shown that the semantics is equivalent to an implementation-biased semantics where each process has its own local dataspace and communication is established by means of asynchronous message passing. Hence, our language allows both convenient reasoning using a global dataspace and efficient implementation by means of distributed dataspaces. The equivalence result is checked mechanically by means of the interactive theorem prover PVS.

1. INTRODUCTION

This introduction contains an overview of the general context of our research, an informal explanation of the Splice architecture and the aims of the work described here.

1.1 Context of the research

Our research concerns the construction of complex distributed software applications that have to satisfy strong requirements concerning functional correctness and reliability. An industrial solution to tackle this complexity is the definition of a suitable underlying architecture which provides high-level communication mechanisms and construction rules for the components. This architecture should be general enough to be suitable for a whole family of products. Our aim is to combine this solution with another technique to increase the quality and reliability of applications, namely the use of formal methods. This means that there are precise mathematical models for specifications and implementations and suitable techniques to relate the two.

As a specific case study we consider the software architecture Splice, as devised by the company Hollandse Signaalapparaten. It forms the core of many of their products, especially in the field of command and control systems. Typically, such systems receive large streams of data from sensors which are used to build a coherent image of the environment and to take appropriate action on the basis of this information. Additionally, the internal representation is visualized for operators which interact with the system. All these tasks are carried out by a large number of parallel processes. Splice can be seen as a coordination language that provides a suitable communication mechanism where processes need not know the producers of the data they need or the consumers of the data they produce. Important is that Splice allows dynamic reconfiguration of the system and, for instance, duplication of processes, thus achieving a high degree of reliability and fault-tolerance.

Other work [9; 10] has addressed the formalization of the requirements of such command and control systems. This has resulted in a structured specification in the language of the interactive theorem prover PVS [8]. Our aim is to define a verification method to prove that a Splice program satisfies such a specification. To be able to deal with large programs, we aim at a compositional framework which allows reasoning by means of the interface specifications of components, without knowing their implementation [6].

As a starting point for such a verification method, a formal definition of the Splice interaction primitives is needed. There is already some work on the semantics of Splice, e.g. a comparison of semantic choices using an operational semantics [3; 4] and a process algebra framework [5]. We aim at a denotational semantics, however, since this is the proper starting point for compositional reasoning. Furthermore, we would like to investigate which choice of Splice primitives is most convenient for high-level reasoning and, at the same time, allows efficient implementation. Finally, to achieve convenient tool support for our work and a clear relation with the requirements specification, our semantics will be formulated in the language of PVS.

1.2 The coordination language Splice

The basic idea behind Splice [2] is that a process which consumes certain data need not know which process produces it. In fact, there may be several producers. The consumer simply registers its interest by subscribing to a certain data sort. Next it will be able to read produced data items of this sort, where it can use keys and queries to filter the data stream on certain aspects (e.g. in an air traffic management system, on flight number or distance to the airport). Similarly, the producers need not know the consumers of their data.

In the rest of this section we consider several examples of simple programs using read and write primitives. The intended meaning of these primitives will vary per example, but the basic intention is that `Write`(d) writes the value of d to the dataspace(s) and

*This research is supported by the Technology Foundation STW (grant EIF.3959), applied science division of NWO and the technology programme of the Dutch Ministry of Economic Affairs.

Read(x, q) asks the dataspace a query q and assigns the answer of the dataspace to its variable x.
Typical for the applications built with Splice is that it is often not needed to ensure that all data will be consumed or that it will be consumed in the order it is produced, since there is usually a continuous stream of data available from the sensor. Moreover, there are often no strong consistency requirements on processes that consume the same data.
Hence, Splice provides a simple but powerful coordination scheme that requires little overhead. Nothing is guaranteed about the speed or order at which data items of producers become available for consumers.
For instance, in the program

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

where the 'read' command has a `true` query – to be able to read all data elements, it is possible that the value 1 is already available for the 'read'-command, whereas the value 0 cannot be read yet.
But if, for instance, ordered messages are needed, sequence numbers can be included in the data and the reader might read with a query on the next sequence number. It is also possible (and often done in practice) to include a time-stamp in the data sort and to express in the query that the data has a recent time-stamp.
Given this basic idea, there are a large number of possible variations, especially concerning the syntax and the semantics of the read operations. We list a number of questions:

1. Which queries can be formulated; a predicate on data items (e.g. "a data item with time stamp greater than 10") or a predicate on the complete set of available data items (e.g. "an available data item with maximal time stamp")?

2. Does the read operation return the set of all available data items that satisfy the query or just one item?

3. Does the read operation block if there is no item satisfying its query or is there an exception/error (or, e.g. the empty set in case a set is returned)?

4. Is the set of available data items the same for all consumers at any moment or can this be different?

1.3 The goal of this work

The last question above addresses an important point; conceptually Splice can be seen as a large global dataspace, where producers write data items and subscribers can select items to read, see figure 1.

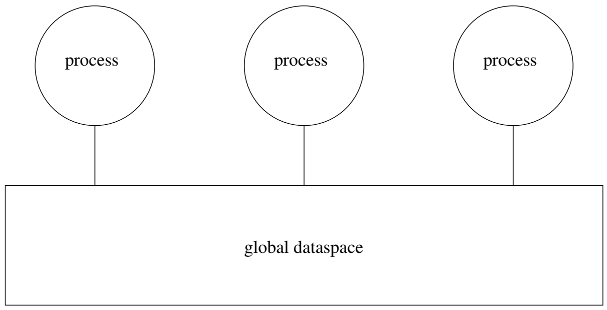

Figure 1: Processes communicating via a global dataspace.

However, in a distributed implementation each process has a local dataspace with its locally available items, see figure 2. Then, a write action is non-atomically distributed to all local dataspaces

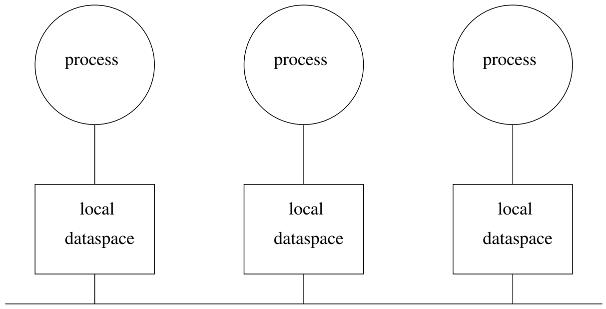

Figure 2: Processes communicating via local dataspaces.

(as is done in the current distributed implementation of Splice); consequently the order of the updates at local dataspaces might be different.
Often there is a large stack of protocols to guarantee consistency of the local dataspaces. A design choice of Splice is to avoid these consistency protocols by carefully designing syntax and semantics such that the conceptual model of a single global dataspace can be implemented by distributed local dataspaces without much overhead.
The aim of our work is to formalize these ideas, that is, to define a simple but powerful Splice language with a denotational semantics based on a global dataspace and to prove that this semantics is equivalent to an interpretation with distributed local dataspaces.

1.4 Examples of non-equivalent variations

Note that the goal above is not trivial, since there are many variations of Splice-like languages where the meaning differs, depending on the dataspace model. We present a few cases where the equivalence does not hold:

- Suppose a read statement returns the set of all data items that satisfy the query. Moreover, assume read statements are blocking, so the result of a read is never the empty set. Consider the following program, where X and Y range over sets of data items.

  $\texttt{Write}(0)\ ||\ \texttt{Read}(Y, \texttt{true})$
  $\quad ||\ (\texttt{Read}(X, \texttt{true})\ ;\ \texttt{Write}(1))$

  In this and the following examples in this section, we assume that all programs are "closed" in the sense that there are no other write actions than the ones shown – this is formalized later. In the distributed implementation (where write actions are distributed to the local dataspaces non-atomically), Y might get the value $\{1\}$, since it is possible that 0 arrives much later in the local dataspace of the second process.

  In the conceptual model with a single global dataspace, however, Y never becomes $\{1\}$ since 0 should be available in the dataspace before 1 can be written, so the second process can read $\{0\}$ or $\{0, 1\}$.

- Now again assume that a read statement returns the set of all data items that satisfy the query, but suppose read statements are non-blocking (returning the empty set if there are no data items satisfying the query). First observe that the program above yields the same results in both models; X might get the values $\emptyset$ or $\{0\}$ (i.e. item 1 is never available for the first read) and Y can read any combination of 0 and 1 (i.e. $\emptyset$, $\{0\}$, $\{1\}$, or $\{0, 1\}$). However, then the following program shows a difference.

$\texttt{Write}(0)$ || $\texttt{Read}(Y, \texttt{true})$
|| $(\texttt{Read}(X, \texttt{true})$; **if** $X \neq \emptyset$ **then** $\texttt{Write}(1)$ **fi**$)$

In the conceptual model with a global dataspace, value 1 is written when 0 is already available in the dataspace and, as in the previous example, Y cannot get the value $\{1\}$. The implementation with local dataspaces allows an execution where Y becomes $\{1\}$; then the value 0 becomes available in the dataspace of the third process much earlier than in that of the second process, where it arrives even after the value 1 written by the third process.

The problem here is that in this example we have constructs by which we can observe whether a data item is present or not. This can be exploited to observe the difference between the conceptual global dataspace model and the implementation with distributed dataspaces.

- As another example, suppose read statements are blocking and return a single data item that satisfies the query. If queries are restricted to predicates on state-variables and data, this is the language for which we show equivalence of the two semantics. However, the equivalence does not hold if the queries are predicates on the dataspace. Consider, for instance, the possibility of reading minimal or maximal values (e.g., $\texttt{Read}(x, \texttt{min})$ yields the minimal value available in the (local) dataspace at the moment), and the following program.

  $(\texttt{Write}(0)$; $\texttt{Write}(1))$
  || $(\texttt{Read}(x, \texttt{min})$; $\texttt{Read}(y, \texttt{min}))$
  || $(\texttt{Read}(u, \texttt{max})$; $\texttt{Read}(v, \texttt{max}))$

  In the implementation model with local dataspaces, x, y, u, v may get values 1, 0, 0, 1 respectively, since in the dataspace of the second process 1 may arrive before $\texttt{Read}(x, \texttt{min})$ is executed and 0 may arrive afterwards (but before execution of the statement $\texttt{Read}(y, \texttt{min})$), whereas in the dataspace of the third process 0 may arrive before 1 does. This is of course impossible in the conceptual model with one global dataspace. Observe that this example again depends on the possibility of testing for absence of data (if $\texttt{Read}(x, \texttt{min})$ returns value 1 then the value 0 must be absent in the dataspace).

  Note that if read statements are non-blocking and return an error value when no data item satisfies the query (and otherwise a single data item satisfying the query), then we can also observe whether a data item is present, even for queries that are restricted to predicates on data only.

1.5 Tool support

The verification system PVS [8][1] has been used to support our research. Two versions of the semantics, one for the conceptual view of a global dataspace and one for the implementation based view of local dataspaces, have been defined in PVS. The specification language of PVS is a classical, typed higher-order logic with a large number of built-in types (e.g. booleans, integers reals) and type-constructors (as functions, sets, tuples, records), including a powerful mechanism to define abstract datatypes. Specifications can be structured by hierarchies of parameterized theories.
The PVS system contains a powerful theorem prover that can be used to construct proofs of theorems interactively. Proofs are recorded and can be rerun after changes. All results mentioned in this paper have been checked mechanically by means of PVS[2].

1.6 Structure of this paper

The structure of the paper is as follows. In section 2 we introduce a simple language with Splice interaction primitives. A semantics for this language based on the conceptual model with a single global dataspace is presented in section 3. A semantics based on the implementation model with distributed local dataspaces is defined in section 4. In section 5 we show that these two semantics are equivalent. Section 6 contains a brief discussion of related work and some concluding remarks can be found in section 7.

2. SYNTAX OF A SPLICE-LIKE COORDINATION LANGUAGE

To high-light the essential ideas, we incorporate the basic Splice interaction primitives in a simple imperative coordination language. With respect to the questions listed in section 1.2 the following choices have been made. Queries are predicates on data items and a state, the read operation returns exactly one data item satisfying the query and blocks if there is none. The question whether the set of available data items is the same for all consumers at any time corresponds to the choice between the global or the local model.
We assume a data type `Data` and a set of variables `Vars`. Then the state of a program is a function from variables to data, i.e., $\texttt{States} = \texttt{Vars} \rightarrow \texttt{Data}$, and expressions are functions of type $\texttt{States} \rightarrow \texttt{Data}$. Now programs in our language are defined as follows:

$$P \;::=\; \texttt{Skip} \mid \texttt{Write}(e) \mid \texttt{Read}(x, q) \mid P_1 \;;\; P_2 \mid P_1 \,||\, P_2 \mid \texttt{Close}(P).$$

Some remarks are in order here:

- `Skip` denotes the empty program.
- $\texttt{Write}(e)$ writes the data value denoted by expression e to the dataspace.
- $\texttt{Read}(x, q)$ assigns to the variable x a previously written data element satisfying the query q. It blocks if there is no such element.
- P_1 ; P_2 denotes sequential composition.
- $P_1 \,||\, P_2$ denotes parallel composition. As a syntactic constraint, we require that the set of variables of the two processes in the parallel composition be disjoint.
- $\texttt{Close}(P)$ denotes the closure operation. It specifies that all data items that have been read by statements in P also have been produced by preceding writes inside P. There is no hiding of the produced data items, so they might be read by an external reader.

In PVS, this language is represented as an abstract datatype, which allows us to use recursive definitions and proofs by induction on the structure of programs.
Note that, for simplicity, we do not distinguish between different datasorts. An extension to multiple sorts and adding a sort to the closure operation, is straightforward. Similarly, it is easy to add more programming constructs such as assignment, choice, and while. See, for instance, [7] for the denotational semantics of a real-time imperative programming language in PVS.

[1]PVS is freely available, see the web-site of SRI International `http://www.csl.sri.com/sri-csl-pvs.html`

[2]For a PVS dump file with all theories and proofs, see `http://www.cs.kun.nl/~hooman/SpliceSem.html`

3. SEMANTICS FOR THE MODEL WITH A GLOBAL DATASPACE

In this section we define the semantics of our coordination language under the assumption that there is a single global dataspace. We refer to it as the *global semantics*.

The values of internal variables of a process are represented by a state, which is a function from variables to `Data`. Furthermore, let `Events` be a set of events. The meaning of a program P, given an initial state representing the values of the variables at the start of the program, is a set of semantic primitives. Each semantic primitive corresponds to a terminated execution. For simplicity we do not model blocking or infinite computations. Semantic primitives, denoted sp, are records in PVS with three fields:

- st: the final state of the program, representing the situation after the execution.
- *evs*: a subset of `Events` containing those events that occur during the execution of P. An event represents the occurrence of an action. Let *act* be a global function that yields for each event in the set `Events` the corresponding action. Here we have actions of the form $\mathtt{R}(d)$ and $\mathtt{W}(d)$ denoting the reading and writing, resp., of data element d. Note that if we replace *evs* and *act* by a multiset of actions, then we cannot order events with the same action and hence it is not possible to distinguish them.
- *ord*: a strict (irreflexive & transitive) partial order on the set of events (*evs*). Intuitively, we have $ord(E_1, E_2)$ iff E_1 is a read event occurring before E_2; if E_2 is a read event then anything available in the dataspace at the moment of execution of E_1 will also be available at the moment of execution of E_2, and if E_2 is a write event then the value written by E_2 can not be available in the dataspace at the moment of execution of E_1 (unless there is another write action writing the same value).

Next we present the formal semantics of the language constructs in the setting of the conceptual model with a single global dataspace. This is done by defining a function $[\![\cdots]\!]$ which maps a program `P`, given an initial state `s`, to a set of semantic primitives, denoted $[\![P]\!](s)$. Here, a semantic primitive, represented in PVS by a record, is denoted by a triple $(s, eset, \prec)$, where s is a state, *eset* a set of events and $\prec$ a strict partial order on *eset*. The functions st, *evs*, and *ord* are used to access the first, second and third field of these triples.

$$[\![\mathtt{Skip}]\!](s) = \{(s, \emptyset, \emptyset)\}$$

The skip statement performs no visible actions. Hence, the set of events (and its order) are empty, and the final state is equal to the initial state s.

$$\begin{aligned}&[\![\mathtt{Write}(e)]\!](s)\\ &\quad = \big\{(s, \{E\}, \emptyset) \mid E \in \mathtt{Events} \wedge act(E) = \mathtt{W}(e(s))\big\}\end{aligned}$$

A write statement $\mathtt{Write}(e)$ performs a single event E that corresponds to a write action with the value of e evaluated in the state s. The initial state s is unaffected.

$$\begin{aligned}&[\![\mathtt{Read}(x, q)]\!](s)\\ &\quad = \big\{(s[x \mapsto d], \{E\}, \emptyset) \mid E \in \mathtt{Events}\\ &\qquad\quad \wedge\ d \in \mathtt{Data} \wedge q(d, s) \wedge act(E) = \mathtt{R}(d)\big\}\end{aligned}$$

To understand the semantics of statement $\mathtt{Read}(x, q)$, it is important to recall that we aim at a denotational, i.e. compositional, framework. This means that the semantics of a read statement should allow the combination with any context, where in principle every possible data item could be written. So we have to take an arbitrary environment into account. This is done by including semantic primitives for all data items $d \in \mathtt{Data}$ that satisfy the query in the initial state (i.e., $q(d, s)$ holds). So the semantics includes all possibilities and only at the point of applying a closure operation we decide about the actual values read. We use $s[x \mapsto d]$ to denote the state that equals s except that the value of x has been replaced by d.

$$\begin{aligned}&[\![P_1\ ;\ P_2]\!](s)\\ &\quad = \big\{sp \mid \exists sp_1, sp_2\big[sp_1 \in [\![P_1]\!](s)\\ &\qquad \wedge\ sp_2 \in [\![P_2]\!](st(sp_1)) \wedge st(sp) = st(sp_2)\\ &\qquad \wedge\ evs(sp_1) \cap evs(sp_2) = \emptyset\\ &\qquad \wedge\ evs(sp) = evs(sp_1) \cup evs(sp_2)\\ &\qquad \wedge\ \forall E_1, E_2 \in evs(sp)\big[ord(sp)(E_1, E_2) \leftrightarrow\\ &\qquad\qquad \big(ord(sp_1)(E_1, E_2)\\ &\qquad\qquad\ \vee\ ord(sp_2)(E_1, E_2) \vee \mathtt{R}?(E_1)\big)\big]\big]\big\}\end{aligned}$$

Here $\mathtt{R}?(E_1)$ is an abbreviation for $\exists d[act(E_1) = \mathtt{R}(d)]$. The meaning of $P_1\ ;\ P_2$ expresses that the final state of this construct is produced by P_2, starting in the final state of P_1. The latter executes from the original initial state s. We require that the event sets of the two components be different. This requirement can always be met simply by renaming common events, assuming that there are sufficiently many events to make this possible. The events of $P_1; P_2$ are those performed by P_1 or P_2. The order on the resulting event set respects the orders of P_1 and P_2; additionally, the order requires that each read action by P_1 occurs before any event of P_2.

$$\begin{aligned}&[\![P_1 \mid\mid P_2]\!](s)\\ &\quad = \big\{sp \mid \exists sp_1, sp_2\big[sp_1 \in [\![P_1]\!](s) \wedge sp_2 \in [\![P_2]\!](s)\\ &\qquad \wedge\ \forall x[st(sp)(x) = \text{if } x \in vars(P_1) \text{ then } st(sp_1)(x) \text{ else}\\ &\qquad\quad \text{if } x \in vars(P_2) \text{ then } st(sp_2)(x) \text{ else } s(x)]\\ &\qquad \wedge\ evs(sp_1) \cap evs(sp_2) = \emptyset\\ &\qquad \wedge\ evs(sp) = evs(sp_1) \cup evs(sp_2)\\ &\qquad \wedge\ ord(sp) = ord(sp_1) \vee ord(sp_2)\big]\big\}\end{aligned}$$

In $P_1 \mid\mid P_2$ both processes start from the initial state s and, since there are no shared variables, the final state of the parallel construct is obtained by simply combining the two final states. Similar to sequential composition, the event sets of P_1 and P_2 are assumed to be disjoint, and the events of $P_1 \mid\mid P_2$ are those that occur in P_1 or P_2. No additional ordering is imposed by the parallel composition.

$$\begin{aligned}&[\![\mathtt{Close}(P)]\!](s)\\ &\quad = \big\{sp \mid \exists sp_1\big[sp_1 \in [\![P]\!](s) \wedge st(sp) = st(sp_1)\\ &\qquad \wedge\ ord(sp) = \emptyset\\ &\qquad \wedge\ evs(sp) = \{E \mid E \in evs(sp_1) \wedge \neg\mathtt{R}?(E)\}\\ &\qquad \wedge\ \exists \prec \big[\prec \text{ is a strict and total order on } evs(sp_1)\\ &\qquad\quad \wedge\ ord(sp_1) \subseteq \prec\\ &\qquad\quad \wedge\ \forall E \in evs(sp_1), d \in \mathtt{Data}[act(E) = \mathtt{R}(d)\\ &\qquad\quad \Longrightarrow\ \exists E_1 \in evs(sp_1)[E_1 \prec E\\ &\qquad\qquad\qquad \wedge\ act(E_1) = \mathtt{W}(d)]]\big]\big]\big\}\end{aligned}$$

The semantics of the `Close` operation formalizes the relation between read and write actions. Recall that the semantics of a read statement includes all possible data items and only at closure it can be checked which data items are actually available. This is done by imposing a strict total order on the events of P, extending the existing order *ord*, such that each read action $\mathtt{R}(d)$ is preceded by a corresponding write action $\mathtt{W}(d)$. Only the semantic primitives from P for which such a total order exists are retained in the semantics of $\mathtt{Close}(P)$.

Examples We present a few simple examples to illustrate the semantics. In the following program, the variable x might become 0 or 1, which is reflected in its denotational semantics.

$$\begin{aligned}
&[\![\, \mathtt{Close}\big((\mathtt{Write}(0)\ ;\ \mathtt{Write}(1)) \mid\mid \mathtt{Read}(x, \mathtt{true})\big)\,]\!](s) \\
&\quad = \big\{ sp \mid (st(sp) = s[x \mapsto 0] \vee st(sp) = s[x \mapsto 1]) \\
&\qquad \wedge\ ord(sp) = \emptyset \\
&\qquad \wedge\ \exists E_1, E_2 : \mathtt{Events}[evs(sp) = \{E_1, E_2\} \\
&\qquad \wedge\ act(E_1) = \mathtt{W}(0) \wedge act(E_2) = \mathtt{W}(1)] \big\}
\end{aligned}$$

The following example illustrates how read actions are delayed until the requested data item arrives in the shared dataspace. Let P be the program

$$(\mathtt{Write}(0)\ ;\ \mathtt{Write}(1)) \mid\mid (\mathtt{Read}(x, x > 0)\ ;\ \mathtt{Read}(y, y < x)).$$

Then

$$\begin{aligned}
&[\![\, \mathtt{Close}(P)\,]\!](s) \\
&\quad = \big\{ sp \mid (st(sp) = s[x \mapsto 1][y \mapsto 0] \wedge ord(sp) = \emptyset \\
&\qquad \wedge\ \exists E_1, E_2 : \mathtt{Events}[evs(sp) = \{E_1, E_2\} \\
&\qquad \wedge\ act(E_1) = \mathtt{W}(0) \wedge act(E_2) = \mathtt{W}(1)] \big\}
\end{aligned}$$

4. SEMANTICS FOR THE MODEL WITH DISTRIBUTED LOCAL DATASPACES

Here we present the semantics of an implementation-biased model with distributed local dataspaces, henceforth called the *local semantics*. To formalize that in this model, read statements of different components might observe the write events in different order, we introduce a set $\mathtt{ProcIds}$ of process identifiers and extend the semantic primitives of the previous section. In addition to the three fields st, *evs* and *ord*, we add a fourth *pid* field, which is a function that assigns a process identifier to each event in *evs*. This leads to so-called *local semantic primitives* (denoted *lsp*).

For simplicity, we restrict ourselves to so-called *non-nested* Splice programs, denoted by S, where parallel composition does not occur inside sequential composition. In this way, we avoid a complex definition of assigning pids to processes. Consider, e.g., a program of the form $(P_1 \mid\mid P_2)\ ;\ P_3$. Here it is unclear whether the pid of program P_3 should be equal to the pid of P_1, P_2 or none of them.

The semantics of a non-nested Splice program S, given an initial state s, to a set of local semantic primitives, denoted $\mathtt{SemL}(S)(s)$, is obtained by adapting the semantics of the previous section as follows.

- The local semantics of the three atomic statements $\mathtt{Skip}$, $\mathtt{Write}(e)$ and $\mathtt{Read}(x, q)$ is basically equal to the global semantics extended with a *pid* field. The value of this field is not restricted and may take any possible value, e.g.:

  $$\begin{aligned}
  &\mathtt{SemL}(\mathtt{Write}(e))(s) \\
  &\quad = \big\{ (s, \{E\}, \emptyset, p) \mid E \in \mathtt{Events} \\
  &\qquad \wedge\ p : \{E\} \to \mathtt{ProcIds} \wedge act(E) = \mathtt{W}(e(s)) \big\}.
  \end{aligned}$$

- For sequential composition we require, in addition to the global semantics, that the events of both components be assigned the same pid.

- The semantics of parallel composition extends the global semantics by requiring that the events of both components are assigned different pids.

- The new semantics of $\mathtt{Close}$ becomes more complex, as explained below.

  $$\begin{aligned}
  &\mathtt{SemL}(\mathtt{Close}(S))(s) \\
  &\quad = \big\{ lsp \mid \exists lsp_1 \big[lsp_1 \in \mathtt{SemL}(P)(s) \\
  &\quad \wedge\ st(lsp) = st(lsp_1)\ \wedge\ ord(lsp) = \emptyset \\
  &\quad \wedge\ evs(lsp) = \{E \mid E \in evs(lsp_1) \wedge \neg \mathtt{R}?(E)\} \\
  &\quad \wedge\ \forall E \in evs(lsp)[pid(lsp)(E) = pid(lsp_1)(E)] \\
  &\quad \wedge\ \exists \prec_1 \big[\prec_1 \text{ is a strict order on } evs(lsp_1) \\
  &\qquad \text{and } \prec_1 \text{ is total on } \{E \in evs(lsp_1) \mid \mathtt{R}?(E)\} \\
  &\quad \wedge\ ord(lsp_1) \subseteq \prec_1 \\
  &\quad \wedge\ \forall E_1, E_2 \in evs(lsp_1)[E_1 \prec_1 E_2 \implies \mathtt{R}?(E_1)] \\
  &\quad \wedge\ \forall p\, \exists \prec_2 \big[\prec_2 \text{ is a strict and total order on } evs(lsp_1) \\
  &\qquad \wedge\ \prec_1 \subseteq \prec_2 \\
  &\qquad \wedge\ \forall E \in evs(lsp_1), d \in \mathtt{Data}[act(E) = \mathtt{R}(d) \\
  &\qquad\qquad \wedge\ pid(lsp_1)(E) = p \\
  &\qquad\qquad \implies \exists E_1 \in evs(lsp_1)[E_1 \prec_2 E \\
  &\qquad\qquad \wedge\ act(E_1) = \mathtt{W}(d)]]\big]\big]\big]\big\}
  \end{aligned}$$

  Observe that the events of $\mathtt{Close}(S)$ are assigned the same pid as in S (line 5) and that the main difference with the semantics of the previous section concerns the order on the events. The requirement of a single, total order on all events is replaced by two conditions:

  1. There should be a global total order $\prec_1$ on the read events which extends *ord* but does not order write events before others (lines 6 to 9).
  2. For each process identifier p there exists a local total order $\prec_2$ which extends $\prec_1$ (lines 10 and 11) and is such that each read event with process identifier p has a preceding (w.r.t. $\prec_2$) write event with the same value (lines 12 to 15). This local order allows write actions to be ordered differently for each process, modelling the non-atomicity of write actions.

  To understand the reason for the complexity of this definition, it might be instructive to mention that in an early attempt we did not include the first condition and used only the second condition, with $\prec_2$ extending *ord*. Then the proof of equivalence in PVS failed and we found that the definition was wrong. Consider, for instance,

  $$\begin{aligned}
  &\mathtt{Close}\big((\mathtt{Read}(x, x = 0)\ ;\ \mathtt{Write}(1)) \\
  &\qquad \mid\mid\ (\mathtt{Read}(y, y = 1)\ ;\ \mathtt{Write}(0))\big).
  \end{aligned}$$

  As we have shown in PVS, the global semantics of this program leads to the empty set. This conforms to the intuition that for both processes the read statement blocks, i.e. the complete program leads to deadlock. Our erroneous local semantics however did not yield the empty set because for each process we could order the two read events differently.

5. EQUIVALENCE OF THE TWO MODELS

To express the equivalence of the two semantics defined above, we remove the pids from the local semantics. This is done by a function $\mathtt{RmPids}$ which removes the pids from the local semantic primitives to obtain the semantic primitives of the global semantics. Then we prove, for a non-nested Splice program S, that

$$[\![S]\!] = \texttt{RmPids}(\texttt{SemL}(S)).$$

The main issues in the proof concern:

1. The equivalence for the `Close`-operator, especially showing that we can construct a global order for the global semantics, given the local orders for the processes in the local semantics.
2. The existence of a *pid* assignment, satisfying the constraints, for the local semantics, given a semantic primitive in the global semantics of a non-nested Splice program.

Since both issues involve quite some technical detail and proof effort, they are separated by introducing an intermediate semantics for non-nested Splice programs S with initial state s, denoted $\texttt{SemG}(S)(s)$, where we have added a *pid*-field to the global semantics. For $\texttt{SemG}(S)(s)$, the semantics of the `Close`-operator equals the one for the global semantics plus the constraint on pids of the local semantics. All other statements have the local semantics, which equals the global one plus constraints on pids. Then the proof is split into the following two lemmas, corresponding to the two issues above:

1. $\texttt{SemL} = \texttt{SemG}$. An outline of the proof can be found in section 5.1.
2. $[\![S]\!] = \texttt{RmPids}(\texttt{SemG}(S))$. The proof of this property is sketched in section 5.2.

For more details we refer to the technical report [1].

5.1 Equivalence of the two semantics of the Close-operator

To prove $\texttt{SemL}(S) = \texttt{SemG}(S)$, for any non-nested Splice program S, first observe that it is rather straightforward to prove that if $lsp \in \texttt{SemG}(S)(s)$ then $lsp \in \texttt{SemL}(S)(s)$. The proof proceeds by induction on the structure of S, which is possible in PVS because we have defined programs as an abstract data type. Most cases are trivial, only for the `Close` statement one has to observe that given a strict total order on the events, this order can be used as the required order on the read events and also as the local order for each process identity.

The proof that $lsp \in \texttt{SemL}(S)(s)$ implies $lsp \in \texttt{SemG}(S)(s)$ also proceeds by induction on S. The main issue here is again the proof for the `Close` statement; given a strict total order on all read events ($\prec_1$) and for each process identifier a total order on all events ($\prec_2$—see the definition of $\texttt{SemL}(\texttt{Close}(S))(s)$), we have to construct a global strict total order on all events.

To do this, we introduced in PVS an axiom stating that there exists some total order $\prec_{\text{events}}$ on the set of all events `Events`. This order is used to construct a so-called minimal extension $\prec_{me}$ of the order $\prec_1$ on the read events:

$$\begin{aligned}
&E_1 \prec_{me} E_2 \iff \\
&\quad \big(\mathrm{R?}(E_1) \wedge E_1 \prec_1 E_2\big) \\
&\quad \vee \big(\neg \mathrm{R?}(E_1) \wedge \mathrm{R?}(E_2) \wedge E_2 \not\prec_1 E_1\big) \\
&\quad \vee \exists E_3\big[(\mathrm{R?}(E_3) \wedge E_3 \prec_1 E_2 \wedge E_3 \not\prec_1 E_1\big] \\
&\quad \vee \big(E_1 \prec_{\text{events}} E_2 \wedge \forall E_3\big[E_3 \prec_1 E_1 \iff E_3 \prec_1 E_2\big]\big).
\end{aligned}$$

The idea is that write events are ordered as early as possible in the sense that any read event that does not preceed a write event E in *ord* is ordered after E by $\prec_{me}$. Hence this minimal extension orders all write events not later than any of the local orders. We have proved that this minimal extension satisfies the condition on the read events in the definition of the global semantics $\texttt{SemG}(\texttt{Close}(S))(s)$ (which is equal to the one in the definition of $[\![\texttt{Close}(S)]\!](s)$), because this condition is invariant under the operation of ordering write events earlier.

5.2 Existence of pids

To prove $[\![S]\!] = \texttt{RmPids}(\texttt{SemG}(S))$, first observe that one direction is quite obvious. Recall that `RmPids` removes the pid field of a local semantic primitive *lsp*. Hence, $lsp \in \texttt{SemG}(S)(s)$ implies $\texttt{RmPids}(lsp) \in [\![S]\!](s)$. This has been proved in PVS by induction on S; all cases for the language constructs could be proved automatically.

Next consider the other direction, i.e., $sp \in [\![S]\!](s)$ implies that there exists a local semantic primitive *lsp* such that $\texttt{RmPids}(lsp) = sp$ and $lsp \in \texttt{SemG}(S)(s)$. The problem here is to construct a *pid*-field that can be added to sp, i.e., a function from the events to process identifiers, which satisfies the constraints of the semantics. Recall that for sequential composition, the pids of events of the components should be equal, whereas for parallel composition they have to be different. Hence, as long as S does not contain any parallel composition, all events should have the same pid. Otherwise, since S is non-nested, we only have to deal with parallel composition and closure operations and then it is important to be able to choose different pids.

To formalize this, we have proved a stronger lemma expressing that if $sp \in [\![S]\!](s)$ then

a) if S does not contain parallel composition then for each pid p there exists an *lsp* such that $\texttt{RmPids}(lsp) = sp$, $lsp \in \texttt{SemG}(S)(s)$ and $\forall E \in evs(lsp)[pid(lsp)(E) = p]$;

b) otherwise, for each set of pids *pset* there exists a local semantic primitive *lsp* such that $\texttt{RmPids}(lsp) = sp$, $lsp \in \texttt{SemG}(S)(s)$ and $\forall E \in evs(lsp)[pid(lsp)(E) \notin pset]$.

Additionally, we have to assume that there are sufficiently many process identifiers. Then the lemma is strong enough to be proved by induction on the structure of program S.

6. RELATED WORK

Since our aim is to develop a compositional proof system for proving properties about Splice programs, we defined a denotational semantics for the Splice interaction primitives. There are already a few other semantic models for closely related languages. We discuss the operational semantics of [3; 4] and an algebraic framework [5].

In [3] a transition system semantics for a subset of Splice interaction primitives is presented. The resulting semantics is similar to our implementation model semantics. Differences are the use of a *new* operation to fork processes instead of parallel composition, a *get* operation and the absence of a state so that write operations can only write values. The queries are predicates on data only.

In [4], Bonsangue et al. study the relationship between several different models for a data-oriented coordination language. Although we do not have an explicit database in the semantics, it is possible to construct one corresponding to each semantic primitive. Then our conceptual model corresponds to the unordered centralized system with a set dataspace in [4], and our implementation model is related to their distributed system with a set dataspace. Bonsangue et al. show that these two systems are equivalent for programs using read, write and test-for-absence operators[3]. Apart from the fact

[3] The test-for-absence operator blocks if the specified value is available in the dataspace.

that we give a denotational semantics where Bonsangue et al. give an operational semantics and that our semantics and proofs have been formalized in PVS, a difference between their work and ours is that read operations do not have queries in [4]. Simple queries on the requested data item only can be simulated in the system of Bonsangue by using a (possibly infinite) choice, but we allow stronger queries, namely on the current state as well as the data value. These queries can not be simulated in the system of Bonsangue et al. This explains why in [4] the systems are equivalent even in the presence of a test-for-absence.

Another approach for giving a semantics to part of the Splice language has been taken in [5], where Splice is described in terms of process algebraic laws. The approach is inspired by a process algebra for describing delay-insensitive communications. Whereas the aim of [5] is to be able to reason and compute algebraically with programs, we aim at compositional assertional verification of programs. Indeed we can prove properties formally that have are axiomatized as laws in the Splice Process Algebra of [5]. A few examples of general process algebraic laws that have been proven to hold in our semantics:

- Symmetry of $||$, i.e., $[\![P_1 \;||\; P_2]\!] = [\![P_2 \;||\; P_1]\!]$ for all programs P_1, P_2.
- Associativity of $||$ and $;$, that is, for all programs P_1, P_2, P_3, $[\![P_1 \;||\; (P_2 \;||\; P_3)]\!] = [\![(P_1 \;||\; P_2) \;||\; P_3]\!]$ and $[\![P_1 \;;\; (P_2 \;;\; P_3)]\!] = [\![(P_1 \;;\; P_2) \;;\; P_3]\!]$.

The following general law has also been proven.

- Sequential composition of write statements is equivalent to their parallel composition: $[\![\mathtt{Write}(e_1) \,;\, \mathtt{Write}(e_2)]\!] = [\![\mathtt{Write}(e_1) \;||\; \mathtt{Write}(e_2)]\!]$, for all expressions e_1, e_2.

Now we have, as a consequence of the first and third law above, law 10 from [5] (we translate the action-prefix of the Splice process algebra to sequential composition):

- Symmetry of sequential writes:
  $[\![\mathtt{Write}(e_1) \,;\, \mathtt{Write}(e_2)]\!] = [\![\mathtt{Write}(e_2) \,;\, \mathtt{Write}(e_1)]\!]$, again for all expressions e_1, e_2,

7. CONCLUDING REMARKS

We have proven that for the coordination language introduced in section 2, the semantics according to the conceptual model of a single global dataspace (presented in section 3) is equivalent to the semantics corresponding to the distributed implementation model with a local dataspace for each process (presented in section 4). This proof has been completely formalized and checked mechanically by means of the interactive theorem prover of PVS. We found this kind of tool support very useful for checking the correctness of the semantics definition as well as the proof itself. Without mechanized support, the error in the definition of the semantics of the `Close`-operator as explained in section 4 might not have been found.

Due to the equivalence it is now possible to use the global dataspace semantics for formally verifying properties of a system implemented on an architecture with local dataspaces.

Interesting subjects for further research are a formal equivalence proof of our semantics and a more operational transition system semantics in the style of [3]. We expect that it is rather straightforward to extend the language with loops and non-blocking read operations. Furthermore, one of our goals is to study the use of formal methods and techniques in the design process leading from the formal requirements specification of a system towards its correct implementation. In other work (cf. [9; 10]), a method for writing formal requirements specifications in PVS has been developed. It is our goal to give a similar semantics as the one presented in this paper to these requirements specifications, and next devise a method to formally verify properties of an implementation of such a specification.

8. ACKNOWLEDGEMENTS

We would like to thank Marcello Bonsangue, Paul Dechering, Rix Groenboom and the anonymous referees for their useful comments on a draft of this paper.

9. REFERENCES

[1] R. Bloo, J. Hooman, and E. de Jong. Semantical aspects of an architecture for distributed embedded systems. Technical report, Eindhoven University of Technology, 2000.

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

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

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

[5] P. Dechering, R. Groenboom, E. de Jong, and J. Udding. Formalization of a Software Architecture for Embedded Systems: a Process Algebra for Splice. In *Proceedings of the Hawaiian International Conference on System Sciences (HICSS-32)*, Maui, Hawaii, January 5-8 1999. IEEE Computer Society.

[6] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*. LNCS 558, Springer-Verlag, 1991.

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

[8] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. Softw. Eng.*, 21(2):107–125, 1995.

[9] J. van de Pol, J. Hooman, and E. de Jong. Formal requirements specification for command and control systems. In *Proceedings of the Conference on Engineering of Computer-Based Systems*, pages 37–44. IEEE, 1998.

[10] J. van de Pol, J. Hooman, and E. de Jong. Modular formal specification of data and behaviour. In K. T. K. Araki, A. Galloway, editor, *Proceedings 1st Conference on Integrated Formal Methods (IFM'99)*, pages 109–128. Springer-Verlag, 1999.