Verifying Replication on a Distributed Shared Data Space with Time Stamps

Jozef Hooman[1,2] and Jaco van de Pol[1]
[1] CWI, Amsterdam, The Netherlands
[2] KUN, Nijmegen, The Netherlands
Email: hooman@cs.kun.nl,vdpol@cwi.nl

Abstract— **We investigate transparent replication of components on top of the distributed data space architecture Splice. In Splice each component has its own local data space which can be kept small using keys, time stamps and selective overwriting. Since Splice applications are often safety-critical, we use two complementary formal tools to ensure correctness: the μCRL tool set is used for a rapid investigation of alternatives by a limited verification with state space exploration techniques; next the most promising solutions are verified in general by means of the interactive theorem prover of PVS. With these formal techniques we showed that replication of transformation components can be achieved using sequence numbers. We also prove the correctness of a nicer, more transparent solution which requires a slight extension of the write primitive of Splice.**

I. Introduction

In this paper we study replication and formal verification of components on top of the real-time distributed data space architecture Splice [4]. This architecture has been devised at the company Thales (previously known as Thomson-CSF Signaal). It provides a coordination mechanism for loosely-coupled components, similar to Linda [7] and JavaSpaces [13]. The main difference is that these last two languages have one central data space, to which all processes may write and from which they can all read or take items. Such a central data space is absent in Splice, where the data space is distributed; each application has its own local data storage that is updated according to a publish-subscribe mechanism. Whereas JavaSpaces uses a leasing mechanism to express the temporal validity of data items (and allow garbage collection), the local storages of Splice are kept small using keys and time-stamps: recent data just overwrites old data with the same key. More details about Splice are given in section II-A.

The Splice architecture is being used to build large and complex systems, such as command and control systems. Typically there are sensors, a number of internal processes that perform calculations on the sensor data, and components that decide on appropriate actions such as commands to actuators. Thanks to the efficient implementation of Splice, large streams of sensor data can be processed at real time. Another aim of Splice is to provide a platform that makes it easy to replicate components in a transparent way, i.e. achieving fault-tolerance for certain components without affecting other components. In this paper we investigate to which extent transparent replication on top of Splice is possible. A question is, for instance, whether the implicit time-stamp mechanism of Splice can be exploited and whether we need any conditions on the context of a replicated component.

In general, replication might affect the real-time and the functional behavior of the system. Since most Splice-applications are highly safety-critical, it is important to ensure the correctness of these applications. A possible way to increase the confidence in the correctness of systems is the use of formal methods, i.e. methods and techniques that have a precise, mathematically defined meaning. In our study we use formal methods to capture the precise meaning of Splice, to experiment with versions of replication in a case study, and for the formal verification of correctness.

Several formalizations of (fragments of) Splice [4] already exist. We mention work on the process algebra SPA [8], the algebraic μCRL tool set [10], [20] and a formalization in the higher-order logic of the theorem prover PVS [3]. The idea of achieving transparent replication by requiring from the architecture that, for any process P, we have $P \,||\, P = P$ was put forward in [9]. Related work on the operational semantics of Linda and JavaSpaces has been presented in [6]. Here the possibility for verification is left as future work. A comparison between various shared data space versions was given in [5].

Our semantic models of Splice are based on the models described in [3], [20]. It is less detailed

Partially supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grants EIF.3959 and CES.5009.

than [10] in order to facilitate verification. Moreover, the semantics presented here is based on more recent information about the use of keys and time-stamps to keep the local storages small.

We study replication on top of Splice using two complementary formal approaches:

- The μCRL-approach builds an operational algebraic model that is suitable for quickly prototyping and debugging applications. Small finite instances of the application can be verified automatically.
- The PVS-approach allows the verification of general applications using the interactive theorem prover PVS. A denotational semantics of Splice programs and a compositional proof method based on property-oriented specifications have been defined in the higher-order logic of PVS.

Clearly, each approach has its strengths and weaknesses. For instance, the PVS-approach provides more general results than the μCRL-approach, but it is much more labor-intensive, especially when there are still many errors in the application. Hence we apply this approach after a good intuition has been obtained using μCRL.

Hence, the aim of our work is not only to obtain results on how to achieve correct, transparent replication on top of a distributed data space architecture, but also to identify whether and how the two formal approaches could be used in combination, such that the advantages can be exploited in a methodology for the verification of data space applications.

This paper is structured as follows. In section II we give an informal description of Splice, a case study, our approach and an overview of the results. The μCRL-approach and the PVS-approach are presented in Sections III and IV, respectively. Concluding remarks can be found in Section V.

II. Informal overview

In section II-A we briefly introduce the main concepts of Splice and some details of the underlying implementation. Section II-B describes a small application that is used as a case study. Our formal approaches are briefly introduced in section II-C. The general results of our study are presented in section II-D.

A. Splice

The Splice architecture provides a coordination mechanism based on a publish-subscribe paradigm. Producers and consumers of data are decoupled; they need not know each other, but communicate indirectly via the Splice primitives, basically *read*- and *write*-operations on a distributed data space. This makes it possible to add and remove components at run-time. The data space is distributed in the sense that each component maintains its local version of the data space. Read requests from an application process are served from this local storage.

Looking at the implementation of Splice, each application component has an agent that takes care of the communication between components. When an application process writes a data item of a particular sort, the corresponding agent forwards this item asynchronously via some underlying network to all agents of processes that subscribed to this sort. There are no assumptions on message delay and items may arrive at the agents in different order. Each agent uses received items to update its local storage, as described below, where it might be read by its application.

To explain the update mechanism of local storages, we first describe the entries in the data storage. Each entry consists of three parts: a key, a value and a time stamp. In each local data space, there will be at most one item with a given key. When an application writes a (key,value) pair, its local agent adds the current *local* clock value to obtain a (key,value,time stamp) triple. This triple is sent asynchronously to all subscribed agents.

Next assume that a (key,value,time stamp) triple arrives at some other agent. If no item with the same key exists, the triple is simply added to the local data space. Otherwise, the item with the same key is overwritten by the new item, *provided* the new item is strictly newer than the current item in the local data space, as indicated by their respective time stamps. This prevents data items to be overwritten by older items that suffered from a large network delay. Note that these old items are simply ignored.

An application can read items satisfying certain queries. It is, for instance, possible to read a value with a given key. Reads can be either blocking or non-blocking (possibly with some time-out). Also, Splice admits both destructive and non-destructive read. In the former case, an application process can read each data item only once. As opposed to the global "take" operation of JavaSpaces, this destructive read only operates on the local data space. Note that an item cannot simply be removed, because it is still needed by the agent to check whether arriving data items are newer than this item.

In our formal study, we only modeled the basic features of the Splice architecture, especially concentrating on read and write operations on the data storages. We did not model, for instance, time-outs on read operations, synchronization of local clocks, the (dynamic) publish/subscribe mechanism, dynamic reconfiguration, data sorts, and different kinds of data such as persistent and context data.

B. The Case Study

As a case study, we consider a simple system with three types of components:

- `Producer`: provides data (with key `input`) to the rest of the system. It can be seen as an abstraction of sensors such as radar, thermometer, altitude measurement device, etc., that provide the system with an approximation of the physical reality.
- `Transformer`: performs internal data computations; here data with key `input` is simply transformed into data with key `output`. In reality, such a process performs some computation on data, such as computing tracks out of plots, making an hypothesis about future movement of objects, etc.
- `Consumer`: consumes data with key `output` and forwards it to the external environment. In a real system, this component might include some decision making, leading to commands to external devices such as motors, pumps, screens, etc.

Although the example is very simple, by abstracting from internal computations, it represents a typical Splice-application in which replication is relevant. The aim is to obtain a higher degree of fault-tolerance by replicating the transformer; the system becomes more robust against crashes of transformers and against network errors. General question is whether the transformer can be replicated in a *transparent* way, i.e. without modifying producer and consumer. Does this, e.g., depend on certain conditions for the components, are the implicit time-stamps useful to support replication, or are other constructs needed for replication?

C. Formal Methods

In this section we give the main ideas of our formal approaches. Details can be found in subsequent sections.

C.1 μCRL

In the μCRL approach, Splice is modeled operationally, by expressing the agents and the network in a form of process algebra. This leads to a `Splice` component. Next also producer, transformer and consumer are modeled as a term in process algebra. Then the aim is to show that

```
Splice || Producer || Transformer || Consumer
```

is equivalent (in some well-defined way) to

```
Splice || Producer || Transformer
                    || Transformer || Consumer
```

We also consider a version with three transformers.

In this approach, it is difficult to split up the verification task; the whole system, with all components, has to be considered. Since the μCRL tool is especially suitable for checking finite systems, we investigated a number of instances of the system. Due to the state-explosion problem, the tool could check a system with at most 5 data items. Still this turned out to be very useful to find errors. We also investigated several types of equivalences, and found surprising differences, depending on the number of data items considered.

To obtain a finite, checkable system that allows rapid prototyping of our ideas, we made some further simplifications in this approach. For instance, we modeled blocking destructive reads, which return only a single value.

C.2 PVS

The PVS-approach aims at general verification of Splice-applications. First a denotational semantics is defined for a programming language with Splice primitives. Here we are not aiming at finite models, but instead formulate a general semantics in terms of the powerful higher-order logic of PVS. Specifications are written in an assertional way, describing properties of the system or its components, by means of pre- and postconditions. Using the compositional character of the semantics, verification can also be done compositionally, allowing reasoning with the specifications of components without knowing their implementation.

In this approach we first define a top-level specification for the whole system `specTL`. Given specifications `specProd` and `specCons` of producer consumer, resp., we determine a specification `specTrans` for the transformer and prove that these three specifications lead to `specTL`. Next we investigate whether `specTrans` can be replicated, i.e. the parallel composition of two transformers implies `specTrans`. Independently, we design programs that are shown to satisfy the specifications of the components. Standard proof rules then easily lead to the fact that `Producer || Transformer || Consumer` conforms to `specTL`, and if `specTrans || specTrans` conforms to `specTrans`, we obtain that `Producer || Transformer || ...`

`|| Transformer || Consumer` conforms to `specTL`, for any positive number of transformers.

Note that this compositional approach supports a strong separation of concerns; one can separately verify the satisfaction of the top-level specification, the replication of transformer specification, and the independent implementation of the components.

D. Results

Experiments in μCRL with the case study, and several other examples, show that in general replication is not transparent; duplication of the transformer typically leads to a different (external) behavior. We investigated two possibilities for obtaining transparent replication:

- The producer adds sequence numbers to data items, which are copied by the transformer(s), and the consumer only accepts items with increasing sequence numbers. This solution has been validated in μCRL.
- The write primitive of Splice has been extended with an additional time-stamp parameter which replaces the implicitly added time-stamp. In this way, time-stamps more accurately reflect the temporal validity of data and the update mechanism in Splice ensures that items are only overwritten by more recent data. With this extended write statement, transparent replication is obtained rather easily, without changing producer or consumer. This solution has been validated in μCRL and its correctness has been proved in general using PVS.

III. The μCRL-approach

The μCRL [15] specification language is a combination of (ACP-style) process algebra (see e.g. [1], [12]) and algebraic datatypes. A system is modeled as a "process", often specified as the parallel composition ($||$) of a number of other processes, the components. Components are often described by recursive equations, using sequential (.) and alternative (+) composition. Consider, e.g., $Buf = in.out.Buf$. Here in and out are so-called atomic actions, which can be externally visible actions, or which synchronize with corresponding actions in different components. Atomic actions can be labeled by data parameters in μCRL. Also recursive specifications can have data parameters, which serve as state variables. Input can be modeled by non-determinism, e.g. $in(0)+in(1)$ models the input of some bit. A generalized choice operator is written with the $\sum$-operator. Another construct is a *guard*: $[b] \to x$, which can execute x provided boolean b is true. Data, like bits and booleans, but also natural numbers, sets etc., are described by means of algebraic data types. A buffer-with-delay can be modeled as:

$$Buf(x : Bit) = \sum_{y:Bit} in(y).out(x).Buf(y)$$

The μCRL tool set [2] supports verification as follows. The operational semantics of a μCRL process is a *labeled transition system* (LTS). This is a rooted directed graph, some of whose edges are labeled with externally visible atomic actions. The remaining edges are labeled with τ, denoting an invisible action. The μCRL tool set allows automatic generation of the LTS from a μCRL specification; this is only possible for finite instances of a system. The resulting LTS can be inspected by means of visualization, model checking, or equivalence checking. For these activities we used the CADP tool set [11].

The sketched verification route has a clear bottleneck: the LTS suffers from a combinatorial state explosion, due to the many possible interleavings. To overcome this, one can hide (i.e. rename to τ) as many actions as possible, given a requirement to verify, and subsequently minimize the LTS modulo some equivalence relation. Usually, the equivalence relation used in μCRL is *branching bisimulation* [14].

In order to avoid the generation of the large LTS entirely, the μCRL tool set will first compile the specification to a *linear process equation*, to be viewed as an internal symbolic representation of the state space. Several reduction tools are implemented, which transform a linear process to an equivalent process. Eventually, a much smaller, but branching bisimilar state space will be generated. We mention a few of these techniques:

- Temporarily unused state variables are given a default value, reducing the number of states of the corresponding component.
- The number of interleavings of invisible actions with other actions can be reduced, provided the invisible actions enjoy the confluence property. This property is established by means of an automated theorem prover.
- Invariants are generated and used to evaluate the guards symbolically in order to remove dead (i.e. unreachable) code. This is often a fruitful preparation for the other steps. This technique also uses the automated theorem prover.

The novelty is that these reductions are performed at a symbolic level, which makes the tool set quite flexible: the user can apply the reductions in any desired order.

This verification method has the limitation that it can only be applied on finite state systems. As advantages, we mention that it is completely automatic, and that it also gives useful feedback in case some requirement doesn't hold. For instance, the model checker will return an execution path which violates the requirement. This is very useful for debugging the specification.

A. *Components and their interconnection*

We model a Splice system as the parallel composition of n application processes and a separate Splice-process. Subsequently, the Splice-process itself can be defined as the parallel composition of a number of agents and a separate Network-process. The applications *synchronize* with Splice(-agents) via atomic *read*- and *write*-primitives. Similarly, the agents *synchronize* with the network via *tell*- and *ask*-primitives. See Figure 1 for an overview of the system.

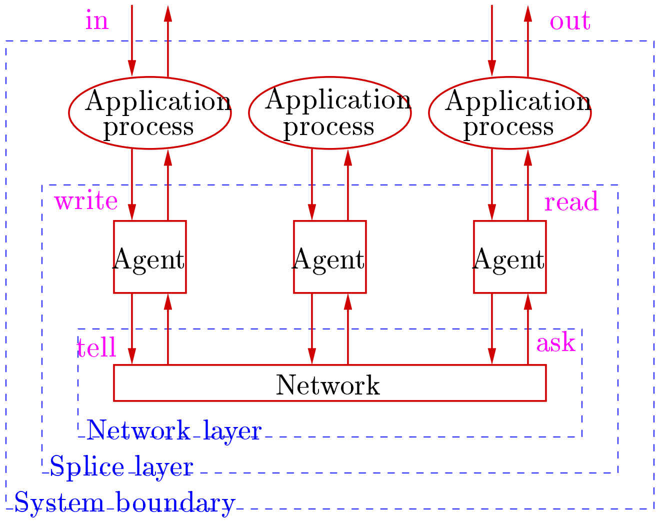

Fig. 1. The architecture of a Splice system.

Next, we model the interfaces (API) of Splice and the Network in μCRL.

- Synchronization between applications and Splice:

```
sort    Key, Value, Address
act     read,r : Key#Value#Address
        write,w: Key#Value#Address
comm    write | write = w
        read | read = r
```

To avoid confusion between agents, we will give them a unique address. Now the read and write actions carry three data parameters: the key, the value and the agent's address. The **comm** section specifies the possibility to synchronize on write actions and on read actions. Here `r` and `w` can be seen as the combined action of the application and the agent of performing a read or write action.

- Synchronization between agents and the network:

```
sort    Entry, AddressList
...
act     tell,t: Address#Entry#AddressList
        ask,a : Address#Entry
comm    tell | tell = t
        ask  | ask  = a
```

A `tell` action corresponds to broadcasting an entry to a number of addresses asynchronously. An `ask` action corresponds to receiving an entry from the network at some address. We omit the standard algebraic specification of lists of addresses.

Having fixed the interfaces, we can be more explicit on the composition (P_i are application processes):

$$\begin{aligned} System &= \tau_{\{r,w\}}\partial_{\{read,write\}}(Splice \,||\, P_1 \,||\, \cdots \,||\, P_n) \\ Splice &= \tau_{\{a,t\}}\partial_{\{ask,tell\}} \\ &\qquad (Network \,||\, Agent \,||\, \cdots \,||\, Agent)\ . \end{aligned}$$

Here parallel composition ($||$) is ACP-style parallelism, corresponding to an interleaving semantics, with the possibility of synchronization between atomic actions, as defined in the preceding **comm**-sections. The encapsulation ∂ is needed to enforce synchronization. The hiding τ is used to hide externally invisible actions: only the actions in P_i different from *read*/*write* are externally visible. In the subsequent sections we define the components Network, Agent, and some application processes.

B. *The network*

As we are not studying details of the underlying network, a high-level description will suffice. In order to model a reliable network, with an unbounded delay, we simply introduce a multi-set of (entry,address)-pairs, which will be delivered in any order. We now first introduce the algebraic definition of multi-sets. Consider the following μCRL-fragment:

```
sort Multiset
func empty_ms: ->Multiset
     add: Entry#Address#Multiset -> Multiset
map  union: Multiset#Multiset -> Multiset
     remove: Entry#Address#Multiset -> Multiset
     send_to_all: AddressList#Entry->Multiset
     in: Entry#Address#Multiset -> Bool
var  x,y: Multiset
     m: Entry
     p: Address
rew  union(empty_ms,x) = x
     union(add(m,p,x),y) = add(m,p,union(x,y))
```

First the sort `Multiset` is introduced (keyword **sort**), and defined by its constructors `empty_ms` and `add` (keyword **func**). Next, some other functions are declared (keyword **map**) and defined algebraically (keyword **rew** for "rewrite rules"). The **var**-section declares the variables used in the following **rew**-section. Above, `union`, `remove` and `in` (membership test) are standard multiset functions. The auxiliary function `send_to_all` is defined such that e.g. `send_to_all([a,b,c],e) =` $\{$`(a,e),(b,e),(c,e)`$\}$, where `a`, `b`, and `c` are addresses and `e` is an entry.

Next, a process `Network` is defined, having the current multiset as state variable (initially empty). It is always willing to either receive a tell-request, in which case it adds the messages to be delivered to its multiset, or to non-deterministically deliver one of its messages. We first present this process in μCRL and then explain the notation.

```
proc Network = Network(empty_set)

Network(B:Multiset) =
  sum(a:Address,sum(e:Entry,sum(AL:AddressList,
     tell(a,e,AL).
     Network(union(B,send_to_all(AL,e))))))
+ sum(a:Address,sum(e:Entry,
    [in(e,a,B)] ->
       ask(a,e).
       Network(remove(e,a,B))))
```

The above definition is a recursive specification of `Network`, parameterized by its state parameter `B` (the multi-set), having two possible behaviors. At any moment a tell-action can happen from any address `a` of entry `e` to recipients in address list `AL`. Similarly for all `a, e` an ask-action can happen, provided `(e,a)` is an element of `B`. The recursive calls specify the new value of the multi-set in both branches.

C. The Agents

The agents maintain a local data base of current entries. Entries are defined as triples (key,value,time stamp), where we choose the natural numbers as time stamps. To model destructive reads, agents also store whether an entry has been read already or not. Hence, the data base is modeled as a set of (Entry,Bool)-pairs, where the boolean indicates whether the entry has been used. The signature for the data base is as follows:

```
func    entry : Key#Value#Nat->Entry

sort    Database
func    empty : -> Database
        add : Entry#Bool#Database -> Database
map     value: Key#Database -> Value
        time: Key#Database -> Nat
        unused_elt: Key#Database ->Bool
        update: Entry#Database -> Database
        mark_used: Key#Database ->Database
```

Here `empty` and `add` are the (list-like) constructors for `Database`. Furthermore, `value`, `time` are functions to retrieve the value and time stamp of an item with a certain key in the database; `unused_elt(k,S)` holds if and only if key `k` refers to an entry in `S` which is not yet used. Finally, `update` and `mark_used` are modifiers, in order to update the database with a new entry, or to mark the entry with a certain key as used. The definitions of these operations are straightforward, except for `update`, which forms the core of the data base mechanism.

```
var k,l: Key
    e,f: Value
    p,q: Nat
    m: Entry
    x: Database
    b: Bool
rew update(m,empty) = add(m,F,empty)
    update(entry(k,e,p),add(entry(l,f,q),b,x))=
      if(eq(k,l),
         if(leq(p,q),
            add(entry(l,f,q),b,x),
            add(entry(k,e,p),F,x)),
         add(entry(l,f,q),b,
             update(entry(k,e,p),x)))
```

In order to update the database with an entry (k, e, p), a matching entry (l, f, q), i.e. one with $eq(k, l)$, is searched. If a matching (l, f, q) cannot be found, then (k, e, p) is added to the database. If a matching (l, f, q) is found, then the time stamps p and q are compared. If $p \leq q$, then the entry (k, e, p) is simply ignored. Otherwise, if $q > p$, then (l, f, q) is overwritten by (k, e, p), and this item is marked as not yet used.

Next, we define the behavior of the agents. Besides the database (initially empty), an agent has a local clock (`t:Nat`, initially 0), and it is parameterized with its address. The *Agent* process is defined recursively, and consists of three branches. First, an unused entry from the database can be read, which is then marked as already read. Second, a new element can be added, which is then time stamped with the current clock value and broadcasted over the network to all subscribers. We assume that some (application-dependent) function is given to compute the subscribers for some key. In this case the clock is increased by one. Finally, some new entry may arrive from the network, after which the database is updated accordingly. So we get:

```
map subscribers: Key->AddressList

proc Agent(i:Address) = Agent(empty,i,0)

Agent(X:Database,i:Address,t:Nat) =
sum(k:Key,
  [unused_elt(k,X)]->
    read(k,value(k,X),i).
    Agent(mark_used(k,X),i,t))
+ sum(k:Key,sum(e:Value,
    write(k,e,i).
    tell(i,entry(k,e,t),subscribers(k)).
    Agent(X,i,S(t))))
+ sum(e:Entry,
    ask(i,e).
    Agent(update(e,X),i,t))
```

D. Application Processes

We will model a producer and a consumer, which are intermediated by a (number of identical) transformer(s). The system interacts with the external world through *in*- and *out*-actions parameterized by `Data`. These actions model sensor measurements and actuator commands, respectively. Furthermore, we have a conversion `val:Data->Value`.

```
sort  Data
func  val: Data->Value
func  input,output: Key
act   in,out: Data
```

When the producer gets some input, it writes it to the database with key `input`. The consumer tries to read elements with key `output` and outputs them to the external world. The transformer should compute the output values from the input values. See Figure 2.

In order to tie the system together, we also have to define some concrete addresses, and to define the subscription information. The agents and the applications are then instantiated to obtain the system. See Figure 3.

```
proc Producer(i:Address) =
       sum(e:Data,
         in(e).
         write(input,val(e),i).
         Producer(i))
     Consumer(i:Address) =
       sum(e:Data,
         read(output,val(e),i).
         out(e).
         Consumer(i))
     Transformer(i:Address) =
       sum(e:Value,
          read(input,e,i).
          write(output,e,i).
          Transformer(i))
```

Fig. 2. Producer, Consumer and Transformer

```
func  a1,a2,a3,a4 :-> Address
rew   subscribers(input) =
                   cons(a3,cons(a4,nil))
      subscribers(output) = cons(a2,nil)
proc
Splice =
  hide({a,t}, encap({ask,tell},
     Network || Agent(a1) || Agent(a2)
             || Agent(a3) || Agent(a4)))

System =
  hide({w,r}, encap({write,read},
      Splice || Producer(a1) || Consumer(a2)
   || Transformer(a3) || Transformer(a4)))
```

Fig. 3. Composing the complete system

E. Verification

In order to verify replication, we will compare two systems. Both are obtained by replacing P_i (section III-A) by some application processes. The first system has a producer, consumer and one transformer. The second system has a producer, consumer and two transformers. Because the resulting state space is infinite, we have to make a finite instance. This is done by modeling an environment, performing $in(1).in(2)...in(n)$, where n is a parameter of the system. For fixed n, the system is finite state.

To compare the systems, we first generate both state spaces, performing reduction modulo branching bisimulation symbolically and on the fly. The resulting LTS is then minimized further modulo branching bisimulation. These activities are performed with the μCRL tool set [2]. The resulting LTS is then minimized modulo trace equivalence and can be visualized or be subjected to model checking. These activities are carried out with the CADP tool set [11]. The goal is to check whether the minimized systems are trace equivalent. So our notion of correctness is trace equivalence between systems with and without replication. This is deliberately weaker (coarser) than branching bisimulation equivalence, for reasons described later.

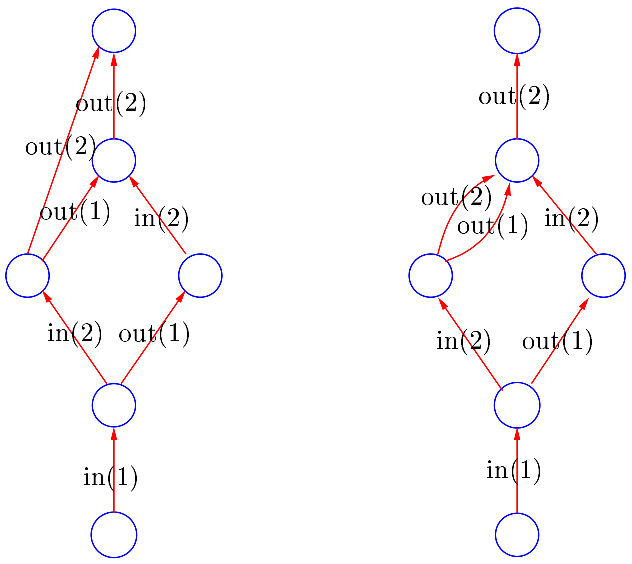

Fig. 4. On the left the system without replication, on the right the system with a replicated transformer

We applied state space generation and minimization modulo trace equivalence on the two systems (for $n = 2$), and got the graphs in Figure 4. Unfortunately, the systems appear not to be the same. The system with two transformers is able to duplicate some output. Apparently, duplication is not completely avoided by the overwrite-mechanism of Splice. To see which sequence of read/write actions leads to this undesirable situation, we have to remove the hiding operator (τ). This will generate a larger state space, which cannot be aesthetically visualized. However, we now know what we are looking for, so we let the model checker find some path with two `out(2)` actions:

```
<true*."out(2)".true*."out(2)">true
```

Now the following concrete trace is automatically generated, in which the second item arrives at the transformer at a4 first; it is forwarded and arrives with time stamp 0 at the consumer, and is subsequently output. Then the first item arrives at the transformer at a3, and is forwarded to the consumer. This item is ignored as it has time stamp 0 too. Finally, the second item arrives at a3, is forwarded to the consumer with time stamp 1, and subsequently output for the second time.

```
in(1)
w(input,val(1),a1)
in(2)
w(input,val(2),a1)
r(input,val(2),a4)
w(output,val(2),a4)
r(input,val(1),a3)
r(output,val(2),a2)
out(2)
w(output,val(1),a3)
r(input,val(2),a3)
w(output,val(2),a3)
r(output,val(2),a2)
out(2)
```

We investigated two possible solutions to this problem. The first solution shifts the problem to the surrounding producer/consumer-processes. The second (preferred) solution slightly extends the Splice primitives.

• The first solution adds logical sequence numbers. This can be done by redefining the constructor for values as: `val: Data#Nat -> Value`. The producer gets an extra parameter `p:Nat`. It writes `val(e,p)` to the database, and increases p. The transformer doesn't change: it just maintains the sequence numbers. The consumer however becomes more complicated. At any moment, it waits for an item with logical sequence number at least `p`. This query is modeled by a guard $q \geq p$. After outputting this data item, it will wait on sequence number at least $q + 1$.

```
proc Consumer'(i:Address,p:Nat)
  = sum(e:Data,sum(q:Nat,
      [geq(q,p)]->
         read(k2,val(e,q),i).
         out(e).
         Consumer'(i,S(q))))
```

We used the μCRL tool set to verify this system. This could be done for systems with up to 4 input items.

• The essence of the previous solution is that the transformer doesn't tamper with the logical sequence numbers. This observation leads to the next, more elegant solution. We allow that the transformer can explicitly read the time stamp of an item, and can choose to write its items *with the same time stamp*, instead of automatically using its local clock as time stamp. Indeed, the clock value of the transformer is rather meaningless in terms of the "temporal validity" of data. This solution is modeled by adding two new actions, slightly modifying the read and write-primitives, and extending the agents accordingly. The transformer now uses the new primitives, by copying the time stamp. The original producer and consumer are not changed.

```
act read,r : Key#Value#Nat#Address
    write,w: Key#Value#Nat#Address
proc Agent(...) = ...
  + sum(k:Key,
     [unused_elt(k,X)]->
        read(k,value(k,X),time(k,X),i).
        Agent(mark_used(k,X),i,t))
  + sum(k:Key,sum(e:Value,sum(t0:Nat,
     write(k,e,t0,i).
     tell(i,entry(k,e,t0),subscribers(k)) .
     Agent(X,i,t))))
```

```
Transformer'(i:Address) =
    sum(e:Value,sum(t:Nat,
        read(input,e,t,i).
        write(output,e,t,i).
        Transformer'(i)))
```

Again, the version with and without replication were generated and compared using the μCRL tool set. This time we could compare them up to 5 input items.

F. Concluding remarks on the μCRL approach

- μCRL is quite expressive. Especially the combination of choice operators and guards allows the modeling of restricted non-deterministic input *and output*, in contrast to e.g. value passing CCS [18].
- The problem sizes that can be dealt with are limited, but some interesting instances can be generated. In Figure 5 we show the size of the state space for m transformers and n input items, denoted `SYSmn`. It appears that we can easily generate situations with up to 3 transformers, or 5 input items (slightly larger instances can be generated, but this is time and memory consuming).
- The symbolic reduction tools are indispensable, and allow to generate systems of considerable size. Without applying these tools, the limit lies around two transformers and four input items.
- The used equivalence relation matters. It appears that the systems with and without replication are not equal modulo branching equivalence with more than two input items. Apparently, this equivalence relation is too fine. The red entries in Figure 6 show that by increasing the length of the input, also the coarser equivalence relations weak (=observational) bisimulation [18] and τ^*a-equivalence fail. Only trace equivalence remains. This also indicates that a more general tool, dealing with arbitrary many inputs is useful.

	generated		reduced	
	states	transitions	states	transitions
SYS12	35	56	6	7
SYS22	419	1278	6	7
SYS32	4547	20465	6	7
SYS13	152	350	10	16
SYS23	5052	22305	10	16
SYS33	142472	925429	10	16
SYS14	611	1825	15	30
SYS24	55041	315712	15	30
SYS15	2339	8565	21	50
SYS25	566640	3984157	21	50

Fig. 5. Size of the generated and reduced LTSs

	BRANCH. states	WEAK states	τ^*a states	TRACE states
SYS12	8	8	7	6
SYS22	8	8	7	6
SYS13	22	19	13	10
SYS23	23	19	13	10
SYS14	55	45	27	15
SYS24	69	48	27	15
SYS15	127	105	63	21
SYS25	198	128	67	21

Fig. 6. Fine and coarse equivalences

IV. The PVS-approach

The tool PVS (Prototype Verification System) [19] is used to give general verifications of Splice-based systems, for instance with an unbounded number of data items or any arbitrary number of transformers. The logic of PVS is a typed higher-order logic in which we express the semantics of Splice. In earlier work on a denotational semantics for Splice [3] the equivalence of a global data space view and an implementation with local data spaces was proved for a carefully selected set of Splice-primitives. This result, however, does not hold for the full Splice architecture, which is essentially based on distributed storages.

The semantics for local storages of [3] seems not very convenient for verification; it is based on a partial order of read and write events, with complex global conditions at closure. It also uses process identifiers, which we would like to avoid if possible (to stay close to the high-level concepts of Splice). Here we aim at a more intuitive denotational semantics, which enables local reasoning as much as possible, and which also incorporates more recent information about the characteristics of Splice, especially concerning the time stamps.

A new denotational semantics is presented in section IV-A. The specifications and verification techniques are based on earlier work on compositional program verification in PVS [17] and are described in section IV-B. Section IV-C contains the PVS-work on the case study.

A. Denotational semantics

The PVS theories that describe the general Splice semantics are parameterized by (non-empty) types `Data` , `KeyData`, and a key function from data to key data `key : [Data -> KeyData]`. Moreover, there are parameters for sets of variables, ranging over data and sets of data. As usual, there is a type `States` which assigns values to variables.

As time domain we use the real numbers, which are predefined in PVS. By adding a time stamp to data we obtain data items, represented in PVS as a record with two fields, `dat` and `ts`. Extended data items contain an additional boolean `used`. A (local) data base is a set of these extended items, where `used` indicates if the item has been read destructively, hence cannot be read by subsequent reads.

```
Time          : TYPE = real
DataItems     : TYPE = [# dat : Data,
                          ts  : Time #]
ExtDataItems : TYPE = [# di   : DataItems,
                          used : bool #]
DataBases     : TYPE = setof[ExtDataItems]
```

The basic idea of the semantics, see Figure 7, is that for each sequential program we record the current contents of the local data base, the set of data written by the program itself, and the data items assumed to be written by its yet unknown environment.

The written items are used to update the local data base; this may happen non-deterministically, at any point in time. In the sets of written items, the field `used` indicates whether an item has already been used for an update.

At any point during program execution it is possible to add items written by the environment. For a process in isolation, all possibilities are included; these assumptions are checked later at parallel composition

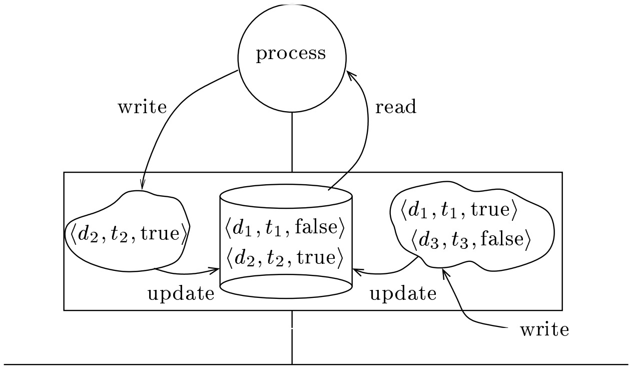

Fig. 7. Basic concepts of the semantics.

and closure.

This leads to the following semantic primitives (type `SemPrim`), which are modeled as a record in PVS with five fields: the current state, value of the local clock, local data storage, own written items and items written by the environment.

```
WriteSets : TYPE = setof[ExtDataItems]
SemPrim  : TYPE =
    [# st     : States,
       clock  : Time,
       db     : DataBases,
       ownw   : WriteSets,
       envw   : WriteSets
     #]

sp, sp0, sp1, sp2 : VAR SemPrim
```

Then the denotational semantics of each statement is a function from an initial semantic primitive (representing the effect of preceding statements) to a set of resulting primitives, that denote all possible non-blocking executions. Based on earlier experience [17], we identify a program and its semantics, since that provides the most flexible framework. So here a Splice program is simply defined as its semantics, a function which assigns to each initial semantic primitive a set of semantic primitives denoting the outcome of its executions.

```
SpliceProgs : TYPE = [SemPrim -> setof[SemPrim]]

prog, prog1, prog2 : VAR SpliceProgs
```

For instance, a basic skip statement simply yields a set containing only the initial state `sp0`. The full skip statement is more complicated, because it also includes a so called UPDATE statement which allows arbitrary environment writes and non-deterministic updates of the data base using the write sets. The update of the data base formalizes the mechanism described before, using keys and time stamps. To combine update and basic statement, we first introduce sequential composition.

```
SKIPB(sp0) : setof[SemPrim] = singleton(sp0)

UPDATE(sp0) : setof[SemPrim] = ...

Seq(prog1,prog2)(sp0) :  =
{ sp | EXISTS sp1 : member(sp1,prog1(sp0)) AND
                    member(sp,prog2(sp1)) }

Skip : [SemPrim -> setof[SemPrim]] =
   Seq(UPDATE,SKIPB)
```

In this way, we define all basic statements, such as assignment, read, and write; they all include `UPDATE`.

A read statement `Read(svar,q,destr)` has three parameters: a variable `svar`, ranging over sets of items, a query `q`, and a boolean `destr` which indicates whether the read should be destructive. The query is a predicate over the current state and database, specifying subsets of the data base that might be read. If such a subset exists, it is assigned to `svar`, otherwise the read statement blocks. Note that the query may disallow the empty set, specifying a blocking read.

A write statement `Write(e)` adds a data item specified by expression `e` and extended with the current value of the clock to the set of own writes. This statement also increases the local clock. Since all other statements do not decrease the clock, this ensures that all writes of a sequential program have different time stamps.

Besides sequential composition, we also define a number of other compound constructs, such as `IfThenElse(b,prog1,prog2)` and an infinite loop `Loop(prog)`. At parallel composition `prog1 // prog2` we check whether the own writes of one program are included in the environment writes of the other, and whether the remaining external writes are the same. The written items of both components are removed from the environment write actions of the composition. Finally, there is a closure operation `Close(prog)` which requires that there are no environment writes; hence all consumed items must have been produced inside the program itself.

B. Specification and verification

To obtain a very flexible framework, suitable for top-down program design, we freely mix specifications and program constructs. Starting from a specification, gradually more programming constructs can be introduced, until finally all specifications are removed. Hence we define a specification also as a program. Here we use a pre- and postcondition style specification, where an assertion is a predicate over the semantic primitives, i.e. a function of type `[SemPrim->bool]`.

```
Assertions : TYPE = pred[SemPrim]
p, q, r           : VAR Assertions

spec(p,q) : SpliceProgs =
  LAMBDA sp0 : { sp | p(sp0) IMPLIES q(sp) }
```

To express when one program refines another, we define `=>`, which is the subset relation here. Clearly, this relation is reflexive and transitive.

```
=>(prog1,prog2) : bool =
   FORALL sp0 : subset?(prog1(sp0),prog2(sp0))
```

Verification of this refinement relation is supported by a number of proof rules. As an example, we show the rule for sequential composition, formulated in PVS as a theorem with label `rule_seq`. For parallel composition we present the monotonicity rule, which shows that we can refine in a parallel context. These rules, and many others, have been proved using the interactive theorem prover of PVS.

```
rule_seq : THEOREM
  Seq( spec( p, r ), spec( r, q ) )
   =>  spec( p, q )

mono_par : THEOREM
  (prog3 => prog1) AND (prog4 => prog2)
   IMPLIES
  ((prog3 // prog4) => (prog1 // prog2))
```

C. Case study

To model the case study in PVS, we import the general PVS theories described above with the following parameters. Data consists of a name and a value, where the name acts as key.

```
DataName : TYPE = {input,output,out}
DataVal  : TYPE = nat
Data     : TYPE = [# name : DataName,
                     val  : DataVal  #]
KeyData :  TYPE = DataName
key(dvar: Data) : KeyData = name(dvar)
```

Moreover, we introduce program variables `d` and `dset` ranging over data and sets of data items, respectively.

C.1 Top-level specification

The top-level specification of the case study, called `TopLevel`, expresses that if there are no writes outside the system then the `out`-values are increasing, i.e. for two items `edi1` and `edi2` in `ownw` with name `out` we have that `val(edi1) < val(edi2) IFF ts(edi1) < ts(edi2)`. Using suitable abbreviations, this can be written as follows.

```
pre : Assertions = LAMBDA sp0 :
  db(sp0)   = emptyset AND
  ownw(sp0) = emptyset AND
  envw(sp0) = emptyset

postTopLevel : Assertions = LAMBDA sp :
     empty?(envw(sp))
       IMPLIES
          Increasing(Out(ownw(sp)))

TopLevel : SpliceProgs = spec(pre, postTopLevel)
```

C.2 Specifying components

The aim is to implement the above specification by a producer, one or more transformers, and a consumer. For the producer we specify that it produces only `input`-values, and its writes should be increasing. The consumer produces only `out`-items and it just maintains the order of items, i.e. if the environment writes increasing `output`-items, then it will also write increasing `out`-items. In PVS, omitting many details:

```
postProd : Assertions = LAMBDA sp :
  NameOwnw(input)(sp) AND Increasing(ownw(sp))

Prod : SpliceProgs = spec(pre, postProd)

postCons : Assertions = LAMBDA sp :
  NameOwnw(out)(sp) AND
     MaintainOrder(Output(envw(sp)),
                    Out(ownw(sp)))

Cons : SpliceProgs = spec(pre, postCons)
```

To satisfy the top-level specification, we introduce the following specification for the transformer:

```
postTrans : Assertions = LAMBDA sp :
  NameOwnw(output)(sp) AND
     MaintainOrder(Input(envw(sp)),
                    Output(ownw(sp)))

Trans : SpliceProgs = spec(pre, postTrans)
```

C.3 Verifying the design

Using the specifications above, it is relatively easy to verify that the three components in parallel lead to the top-level specification.

```
DesignCorrect: THEOREM
    (Prod // (Trans // Cons)) => TopLevel
```

Next, the components can be implemented independently. By the monotonicity property (and transitivity of `=>`), conformance to the top-level specification is still guaranteed. For instance, let `s` be a state variable, where `dvars(s)` yields the values of the data variables (such as `d`), then we have the following program for the producer:

```
dinit : Exprs = LAMBDA s :
                    (# name := input, val := 0 #)
dval  : Exprs = LAMBDA s : dvars(s)(d)
dnext : Exprs = LAMBDA s :
    (# name := input,
       val  := val(dvars(s)(d)) + 1 #)

Producer : SpliceProgs =
  Seq(Assign(d,dinit),
      Loop(Seq(Write(dval), Assign(d,dnext))))

ProdCor : LEMMA Producer => Prod
```

For the transformer we have a program of the following form (omitting details):

```
Transformer : SpliceProgs =
 Loop(Seq(Read(dset,q(input),TRUE),
       IfThenElse(NonEmpty,Write(mk(output)),
                           Skip)))
```

Similarly for the consumer.

C.4 Introducing replication

Note, that the previous transformer specification cannot be replicated. This has been proved in PVS by constructing a counter example manually.

```
NoRepl : LEMMA NOT ((Trans // Trans) => Trans)
```

To obtain a transformer that can be replicated, we modify the specification such that it also maintains the time stamp of the `input` item.

```
MaintainTs : Assertions = LAMBDA sp :
 FORALL edi :member(edi,ownw(sp)) IMPLIES
   EXISTS edi1 : member(edi1,envw(sp)) AND
             name(edi1) = input AND
             val(edi) = val(edi1) AND
             ts(edi) = ts(edi1)

postTransNew : Assertions = LAMBDA sp :
  NameOwnw(output)(sp) AND MaintainTs(sp)

TransNew : SpliceProgs = spec(pre, postTransNew)
```

Note that the new transformer refines the old one, so we still conform to the top-level specification.

```
NewImpliesOld : LEMMA TransNew => Trans

NewCorrect: THEOREM
    (Prod // (TransNew // Cons)) => TopLevel
```

Now we can prove replication of the new transformer and insert it into the system (as many times as we want).

```
TransNewRepl : THEOREM
 (TransNew // TransNew) => TransNew

NewReplCorrect: THEOREM
 (Prod // ((TransNew // TransNew) // Cons))
    => TopLevel
```

With the current Splice primitives, however, the new transformer specification cannot be implemented; there is no possibility to specify the value of the time stamp. Hence we propose to add a write primitive `Write(e,texp)` which has as additional parameter a time expression `texp` that is used in the time stamp field of the data item written.

V. Conclusion

To achieve transparent replication of components on top of the distributed data space architecture Splice, we propose a slightly extended write command. By adding a time expression that replaces the default time stamp of data, the temporal validity of data can be expressed more accurately. Together with the update mechanism of Splice, where data with old time stamps cannot overwrite newer values, this leads to a more logical use of time stamps. It turned out this makes replication much easier, avoiding for instance the need for additional sequence numbers. Note that the extended write command can also be used for predicted or interpolated data items. Also other examples with explicit time stamps, e.g. [16], could have been simplified with this new write primitive.

The use of formal tools and techniques turned out to be very useful during our study of replication on top of Splice. Informal reasoning is difficult, because there are many possible variations in the components and the use of the underlying architecture. For instance, for each read statement there are already a number of choices concerning the precise query, and whether it should be blocking and/or destructive. There are also many variations concerning the structure of the data and the choice of keys which influence the overwriting of data. Moreover, the fact that Splice allows arbitrary delays and reordering of messages leads to a large number of possible executions.

Due to the combination of these aspects, it is already for very simple systems difficult to predict whether they are correct of not. Using the μCRL tool set we often found errors in our initial solutions. We also discovered subtle points such as the fact that, for transparent replication, overwriting data items should only be done if the time-stamp is strictly greater, not if it is equal. We also discovered differences between small systems that in a subtle way depend on the equivalence used and the number of data items considered.

The μCRL tool set and PVS turned out to be complementary. Debugging initial ideas and building an intuition about the correctness of applications is much easier in μCRL than with PVS where it is usually difficult to see why a proof does not work. The μCRL tool set automatically generates counter examples, whereas they have to be constructed in PVS manually. On the other hand, by the well-known state explosion problem, the μCRL tool set can only check small instances of the system and our case study showed that adding one more data item might already break an equivalence. Hence the need for a tool like PVS that makes it possible to perform general verifications. Our PVS framework also supports compositional reasoning, allowing a separation of concerns and scalability of the approach.

Also note that our two approaches use a different specification paradigm; the μCRL approach provides a more operational description, whereas the PVS approach is property-oriented. By comparing these approaches, we increase our confidence in the correctness of the formalization. Note, however, that we do not yet have a precise formal relation between the two approaches. Here the aim was to investigate whether it could be useful to use these approaches in combination. Now that the answer is positive, a precise formal connection becomes a topic of future research.

We would like to thank Edwin de Jong and Ronald Lutje Spelberg from Thales (The Netherlands) for their detailed explanation of Splice.

References

[1] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[2] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. Langevelde, B. Lisser, and J.C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In *Proc. of CAV 2001*, 2001.

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

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

[5] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In J. Carroll, H. Haddad, D. Oppenheim, B. Bryant, and G.B. Lamont, editors, *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99)*, pages 146 – 155, San Antonio, Texas, USA, February 1999. ACM press.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In T. Rus, editor, *8th International Conference on Algebraic Methodology and Software Technology*, number 1816 in LNCS, Iowa, USA, 2000. Springer-Verlag.

[7] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course.* MIT Press, 1990.

[8] P. Dechering, R. Groenboom, E. de Jong, and J.T. 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.

[9] P.F.G. Dechering and E. de Jong. Transparent object replication: A formal model. In *Fifth Workshop on Object-oriented Real-Time Dependable Systems (WORDS'99F)*, Monterey, California, USA, 2000. IEEE Computer Society.

[10] P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In A. Porto and Roman C., editors, *Proceedings of the Fourth International Conference on Coordination Models and Languages*, number 1906 in LNCS, Limassol, Cyprus, 2000. Springer-Verlag.

[11] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proc. 8th Conference on Computer-Aided Verification*, number 1102 in LNCS, pages 437–440. Springer, 1996.

[12] W. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science (EATCS). Springer-Verlag, 2000.

[13] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley, Reading, MA, USA, 1999.

[14] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.

[15] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17. Elsevier, 2001.

[16] U. Hannemann and J. Hooman. Formal design of real-time components on a shared data space architecture. In *Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC 2001)*. IEEE, 2001 (to appear).

[17] 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.

[18] R. Milner. A calculus on communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

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

[20] J.C. van de Pol. Expressiveness of basic SPLICE. Technical Report SEN-R0033, CWI, Amsterdam, 2000.