Formal Verification of Replication on a Distributed Data Space Architecture[*]

Jozef Hooman
University of Nijmegen CWI
Nijmegen Amsterdam
http://www.cs.kun.nl/~hooman/
hooman@cs.kun.nl

Jaco van de Pol
CWI
Amsterdam
http://www.cwi.nl/~vdpol/
vdpol@cwi.nl

ABSTRACT

We investigate the formal verification of safety-critical systems 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. We use two complementary formal tools: first the μCRL tool set 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. These formal techniques are used to investigate transparent replication of certain components on top of Splice. We prove that a convenient solution can be obtained by means of a slight extension of the write primitive of Splice.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, Reliability

Keywords

Coordination, data space architecture, formal verification, model-checking, theorem proving

1. INTRODUCTION

We study formal specification and verification of safety-critical systems that are implemented on top of the distributed data space architecture Splice [4], which has been devised at the company Thales Nederland. 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.

Splice 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.

An important aspect for most Splice applications is fault-tolerance, and often this is achieved by replicating components. In this paper, we investigate transparent replication, i.e. the possibility to duplicate components without affecting other components in the system. This has also been studied in [8], aiming at a Splice-like architecture in which each component can be replicated. Our aim is to investigate how certain components can be replicated transparently on top of Splice. A question is, for instance, whether the implicit time-stamp mechanism of Splice can be exploited.

A second aim is a rigorous verification of the correctness of replicated Splice components using formal methods, i.e. methods and techniques that have a precise, mathematically defined meaning. Our approach uses two rather complementary formal approaches:

- The μCRL tool set [2] is based on 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 tool [20] contains a logic that allows the formulation of a denotational semantics of Splice programs and property-oriented specifications. Verification of general, unbounded, applications is possible using interactive theorem proving.

The PVS-approach provides more general results than the μCRL-approach, but it is much more labor-intensive, espe-

[*] 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.

cially when there are still many errors in the application. Hence we apply this approach after a good intuition has been obtained using μCRL.

To be able to apply our tools to the verification of Splice systems, we need a formal semantics of Splice. Several formalizations of (fragments of) Splice already exist. We mention work on the process algebra SPA [9], the μCRL tool set [10, 21] and a formalization in the higher-order logic of the theorem prover PVS [3]. Related work on the operational semantics of Linda and JavaSpaces has been presented in [6]. 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, 21]. They are less detailed 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.

This paper is structured as follows. In section 2 we give an informal description of our research. The μCRL-approach and the PVS-approach are presented in sections 3 and 4, respectively. Concluding remarks can be found in section 5. We refer to [18] for more details.

2. INFORMAL OVERVIEW

In section 2.1 we introduce the main concepts of Splice and some details of the underlying implementation. Section 2.2 describes a small application that is used as a case study. Our formal approaches are briefly introduced in section 2.3. The general results of our study are presented in section 2.4.

2.1 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, so 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.

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.

Our formal model mainly contains the features of Splice that were needed for our case study (see section 2.2). 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.

2.2 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, some computation on data is performed, such as computing tracks out of plots, making an hypothesis about future movement of objects, etc.
- `Consumer`: reads data with key `output` and forwards it to the external environment. In a real system, this component might include decision making, leading to commands to external devices such as motors, pumps, screens, etc.

Although abstracting from internal computations makes the example quite simple, 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. We try to do this transparently, without modifying producer and consumer.

2.3 Formal Methods

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

2.3.1 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 the system `Splice || Producer || Transformer || Consumer` is equivalent (in some well-defined way) to `Splice || Producer || Transformer || Transformer || Consumer`.

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 only modeled blocking destructive reads which return a single data item.

2.3.2 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.

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.

2.4 Results

Experiments in μCRL with the case study, and several other examples, show that in general replication leads to different external behaviour. Duplication of the transformer typically leads to a duplication in the values output by the consumer. The reason for this was found by inspecting the automatically generated error traces for the case study. The problem is that the replicated transformers have their own local clocks, which are used to time stamp the transformed data items. Due to clock-differences, slightly different copies of the same item are produced. Combined with the asynchronous network, this results in duplication of data by the consumer. We found two possibilities for obtaining correct 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.
- The write primitive of Splice has been extended with an additional time-stamp parameter which replaces the implicitly added time-stamp. The transformer now uses the new primitives, by copying the time stamp. In this way, time-stamps reflect the temporal validity of data more accurately. The original producer and consumer are not changed. Now the update mechanism in Splice ensures that items are only overwritten by more recent data.

Only the latter solution is transparent, because replication is obtained without changing producer or consumer. This solution has been validated in μCRL and its correctness has been proved in general using PVS.

3. 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 atomic actions. The μCRL tool set allows automatic generation of the LTS from a μCRL specification. 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 bottle-neck: the LTS suffers from a combinatorial state explosion, due to the many possible interleavings. To overcome this, the LTS can be minimized modulo some equivalence relation. The equivalence relation used in μCRL is *branching bisimulation* [14]. To avoid the generation of a too large LTS entirely, the μCRL tool applies some reductions already at the symbolic level of the specification. Here many techniques from compiler optimization and theorem proving are used.

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.

3.1 Components and Interfaces

We model a Splice system as the parallel composition of the 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.

Next, we model the interfaces (API) of Splice and the Network in μCRL. Each agent will get a unique address. The read and write actions carry three data parameters: the key, the value and the agent's address. The possibility of synchronizing on write actions and on read actions must also be specified. Here `r` and `w` represent the combined action of the application and the agent of performing a read or write action.

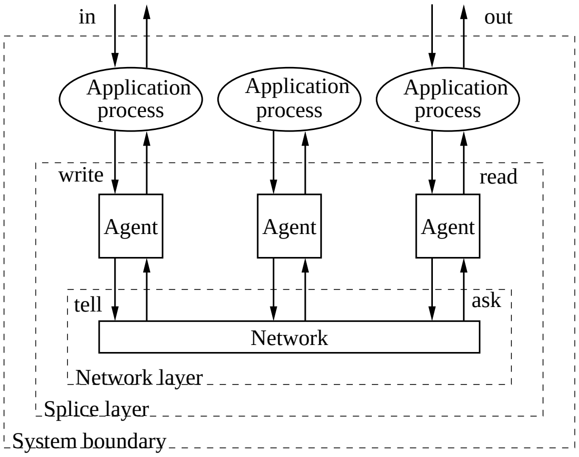


Figure 1: The architecture of a Splice system.

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

A similar interface (not shown) is used to allow synchronization between agents and the network. 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 a particular address.

Having fixed the interfaces, we can be more explicit on the composition. For instance, a system with two components (P, Q) as specified as follows:

$$System = \tau_{\{r,w\}}\partial_{\{read,write\}}(Splice \,||\, P \,||\, Q)$$
$$Splice = \tau_{\{a,t\}}\partial_{\{ask,tell\}}(Network \,||\, Agent \,||\, Agent)\ .$$

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.

3.2 The network

For a high-level description of a reliable network with unbounded delay, we introduce the sort `Multiset`, containing (entry, address)-pairs. The entries will be delivered to the addresses in arbitrary order. Next, some auxiliary functions are needed, such as `union`, `remove` and `in` (membership test). Also the function `send_to_all` is defined in such a way 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.

Process `Network` is defined as a recursive specification, with the current multiset as a state variable (`B`). It has 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.

```
proc 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))))
```

3.3 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. The data base is modeled as a set of (Entry,Bool)-pairs, where the boolean indicates whether the entry has been used. This is needed to model destructive reads.

```
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` and `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. These operations are defined by rewrite rules (not shown), formalizing the intended behaviour of the overwrite mechanism.

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 unread entry from the database can be read, which is then marked as used. 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:

```
proc 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))
```

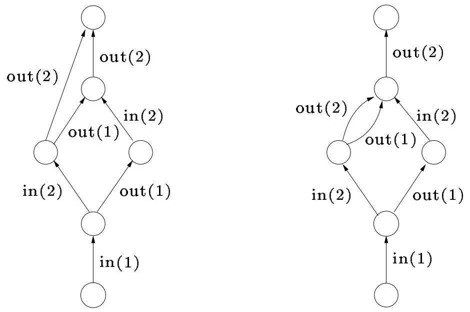
3.4 Application Processes

We 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 `Value`. 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 computes the output values from the input values.

```
func  input,output: ->Key
act   in,out: Value
proc
Producer(i:Address) =
 sum(e:Value,
   in(e). write(input,e,i). Producer(i))
Consumer(i:Address) =
 sum(e:Value,
   read(output,e,i). out(e). Consumer(i))
Transformer(i:Address) =
 sum(e:Value,
   read(input,e,i). write(output,e,i).
   Transformer(i))
```

3.5 Verification

To verify replication, we compare two systems. The first system has a producer, consumer and one transformer. The second system has a producer, consumer and two transformers. We restricted the number of inputs to a parameter n; for fixed n, the system is finite state. The state spaces of both systems are generated and minimized modulo trace equivalence. 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.



For the two systems (with $n = 2$), we obtain the graphs above, showing that they are not the same. The system on the right with two transformers is able to duplicate `out(2)`. By using the CADP model checker, we could expand this to a concrete trace, including all intermediate read/write actions. As explained earlier, the problem can be repaired by extending the write primitive of Splice. This is achieved by adding a parameter to the read- and write actions:

```
act read,r : Key#Value#Nat#Address
    write,w: Key#Value#Nat#Address
proc 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. The system could be verified up to five input items.

3.6 Concluding remarks on μCRL approach

We notice that 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 [19].

The problem sizes that can be dealt with are limited, but some interesting instances can be generated. In figure 2 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 were indispensable to generate these instances.

	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

Figure 2: Size of the generated and reduced LTSs

Finally, 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. We tried several coarser equivalence relations, but many of them fail, when the number of inputs increases. Only trace equivalence appeared to hold (up to 5 input items). This also indicates that a more general tool, dealing with arbitrary many inputs is useful.

4. THE PVS-APPROACH

The tool PVS [20] 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. Earlier work on a denotational semantics for Splice [3] showed the equivalence of a global data space view and an implementation with local data spaces 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. It also uses process identifiers, which we would like to avoid if possible. Here we aim at a more intuitive denotational semantics, which enables local reasoning as much as possible and also incorporates more recent information about the characteristics of Splice, especially concerning the time stamps.

A new denotational semantics is presented in section 4.1. The specifications and verification techniques are based on earlier work on compositional program verification in PVS [17] and are described in section 4.2. Section 4.3 contains the PVS-work on the case study.

4.1 Denotational semantics

The PVS theories that describe the general Splice semantics are parameterized by 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. 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 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 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. In the semantics, these written data items have an additional logical clock value that is used to avoid causal inconsistencies. Since these logical clocks are not relevant for the current case study, we omit them in the rest of the presentation here.

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. For a process in isolation, the environment may write any data item; in each execution they are recorded as an assumption that is checked later at parallel composition and closure.

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

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, denoting all possible non-blocking executions. Similar to [17], a program and its semantics are identified, 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
```

A basic skip statement simply yields a set containing only the initial state. The full skip statement is more complicated; 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. 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. 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.

We also define a number of compound constructs, such as sequential composition `Seq(prog1,prog2)`, choice construct `IfThenElse(b,prog1,prog2)` and infinite loop `Loop(prog)`. At parallel composition `prog1 // prog2` we check whether the environment writes of one program are equal to the union of the own writes of the other program and the remaining environment writes of the compound construct. 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.

4.2 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.

```
Assertions : TYPE = pred[SemPrim]
p, q       : VAR Assertions
sp, sp0    : VAR SemPrim

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

We define when `prog1` refines `prog2`, denoted `prog1 => prog2`, as the subset relation. It 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 that have been proved using the interactive theorem prover of PVS. As an example, we show the monotonicity rule for parallel composition, formulated in PVS as a theorem with label `mono_par`:

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

4.3 Case study

To model the case study in PVS, we import the general PVS theories described above with the following parameters. Data consists of name and 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)
```

4.3.1 *Top-level specification*

The top-level specification, `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)
```

4.3.2 *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)
```

4.3.3 *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, using a data variable `d` and appropriate definitions for `dinit`, `dval`, and `dnext`, we can obtain a correct program for the producer:

```
Producer : SpliceProgs =
  Seq(Assign(d,dinit),
      Loop(Seq(Write(dval), Assign(d,dnext))))
ProdCor : LEMMA Producer => Prod
```

Similarly for the transformer and the consumer.

4.3.4 *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, as expressed by assertion `MaintainTs`.

```
postTransNew : Assertions = LAMBDA sp :
     NameOwnw(output)(sp) AND MaintainTs(sp)
TransNew : SpliceProgs = spec(pre, postTransNew)
```

We prove that the new transformer refines the old one, so it still conforms 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.

5. 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. This made 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. 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.

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

6. REFERENCES

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

[2] S. Blom, W. Fokkink, J. Groote, I. Langevelde, B. Lisser, and J. v. d. Pol. μCRL: a toolset for analysing algebraic specifications. In *Proc. of CAV*, LNCS 2102, pages 250–254. Springer, 2001.

[3] R. Bloo, J. Hooman, and E. de Jong. Semantical aspects of an architecture for distributed embedded systems. In *Proc. of SAC*, pages 149–155. ACM, 2000.

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

[5] M. Bonsangue, J. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proc. of SAC*, pages 146–155. ACM, 1999.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In *Proc. of AMAST*, LNCS 1816, pages 198–212. Springer, 2000.

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

[8] P. Dechering and E. de Jong. Transparent object replication: A formal model. In *Proc. of WORDS'99.* IEEE, 2000.

[9] P. Dechering, R. Groenboom, E. de Jong, and J. Udding. Formalization of a Software Architecture for Embedded Systems: a Process Algebra for Splice. In *Proc. of HICSS-32.* IEEE, 1999.

[10] P. Dechering and I. v. Langevelde. The verification of coordination. In *Proc. of COORDINATION*, LNCS 1906, pages 335–340. Springer, 2000.

[11] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proc. of CAV*, LNCS 1102, pages 437–440. Springer, 1996.

[12] W. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer, 2000.

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

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

[15] J. Groote and M. Reniers. Algebraic process verification. In J. Bergstra et al., editor, *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 *Proc. of COMPSAC*, pages 143–150. IEEE, 2001.

[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, 1994.

[18] J. Hooman and J. van de Pol. Verifying replication on a distributed shared data space with time stamps. In *Proc. 2nd Workshop on Embedded Systems*, pages 107–120. STW, Utrecht, NL, 2001.

[19] R. Milner. A calculus on communicating systems. LNCS 92. Springer, 1980.

[20] 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.

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