Formal Design of Real-Time Components on a Shared Data Space Architecture*

Ulrich Hannemann and Jozef Hooman
Computing Science Institute, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
{ulrichh,hooman}@cs.kun.nl

Abstract

We present a formal approach to the top-down design of real-time components that communicate using a shared data space. The approach is compositional, that is, only the formal specifications of the components are used to reason about their combined behaviour. Formal reasoning is supported by the interactive theorem prover PVS. Our shared data space model is based on the software architecture SPLICE, that allows loosely-coupled components. Our formalism is illustrated by the top-down design of a small flight-tracking-and-display system, which contains an event-driven and a time-driven component. Formal correctness is established, given suitable assumptions about the environment of the system and relations between timing parameters.

1 Overview

We study the formal design of distributed real-time systems on top of the software architecture SPLICE [2]. SPLICE has been devised at the company Thales (previously known as Thomson-CSF Signaal); it has been used – and is currently used – to build large and complex systems, such as command and control systems. The SPLICE architecture provides a coordination mechanism for concurrent components, 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 some shared data space. This makes it possible to add and remove components at run-time. This type of anonymous communication between components is strongly related to coordination languages such as Linda [8] and JavaSpaces [7].

The basic SPLICE mechanism is intentionally rather weak, in the sense that it provides an asynchronous, unordered, communication mechanism. That is, there are no guarantees about the delivery time of data or the order of delivery. In this way, the implementation overhead of SPLICE is minimal, thus offering good performance characteristics for applications for which the basic SPLICE services are sufficient. This is for instance often the case when large time-stamped data streams coming from sensors are processed. When stronger properties are needed, such as message ordering of global consistency, these must be implemented explicitly using additional data fields and specific queries on the data.

An important point during the design of systems on top of SPLICE is the question how the minimal, cheap, SPLICE primitives can be used effectively to construct a correct system. Especially for large and complex systems it is far from trivial to prove correctness formally, using the rather weak properties of basic SPLICE.

In general, for coordination languages only little work has been done on formal verification; the main emphasis in the literature is on semantics. This is also the case for SPLICE, as most of the related work focuses on formal descriptions, e.g., using process algebra [6, 13] or denotational semantics [1], and comparisons between semantic variations [4, 5]. The formal specification and analysis of command and control systems that are designed on top of SPLICE has been studied in [14]. None of the papers mentioned above addresses timing issues. Both [1] and [14] use the interactive theorem prover PVS [11, 12] as a kind of tool support.

The general aim of our work is to investigate formal top-down design of real-time applications on top of the SPLICE architecture, using loosely-coupled, re-usable, replicable components. The idea is to express requirement specifications and abstract specifications of real-time components in the higher-order logic of PVS. This makes it possible to ver-

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

ify design steps during the process of top-down design using the proof checker of PVS. In Section 2.1 we give a brief overview of PVS and introduce the basic notation employed in this paper.

As a first general step, the basic data-space model of SPLICE is formalized in PVS as a separate theory in which the essential properties of the basic coordination mechanism are axiomatized. In Section 2.2, we specify only the required properties of SPLICE in a declarative, assertional way, leaving room for many possible implementations.

As particular application, we demonstrate our approach by a simple, but illustrative, example of a flight-tracking-and-display system, representing a typical application of a distributed control system. The high-level specification, relating flying objects and their display on a monitor is given in Section 3 at a high level of abstraction, allowing the use of continuous, physical, notions. Using explicit assumptions about the environment of the computer system, often including specifications of actuators and sensors, the high-level specification is transformed into a discrete requirements specification. The correctness of this discretization is proved formally in Section 4. Interestingly, the physical assumptions about the objects turned out to be needed also during later stages of the verification, and hence they are also incorporated in lower-level specifications, translated to the appropriate interface.

We gradually implement the system by designing components that communicate by means of SPLICE. At this level, components are only specified in terms of properties at their interface. Correctness of each design step is verified in Section 5 by means of a compositional approach using these specifications and the formal specification of SPLICE. To minimize errors in the formal reasoning, PVS is used to construct the correctness proof interactively and to check them mechanically. Finally, one could implement each component individually, according to its specification (this last step is not covered in the example presented here).

To reason about the timing behaviour of a system, we introduce a number of timing parameters that represent assumptions about the timing characteristics of components and also about the timing behaviour of SPLICE. The resulting system is shown to be correct provided certain constraints on these parameters hold. This provides insight in the trade-off between the required performance of SPLICE and, for instance, publication rates of producers, consuming frequencies of consumers, and timing parameters of the environment (e.g., maximum changes of physical objects).

In the example, the main problem is to design two loosely-coupled components that on one side are connected to sensor and display, respectively, and on the other side communicate with each other via SPLICE. In our solution, the component interfacing with the sensor is event-triggered, whereas the other (which is connected to the display) is time-triggered. To deal with the asynchronous nature of SPLICE, we introduce time-stamps in the data. All design steps have been proved correct using PVS, resulting in a system design that is correct provided a number of explicitly stated constraints on timing parameters hold.

2 Basics

First we present a few basic notations of our formal framework in the language of the theorem prover PVS in Section 2.1. In Section 2.2 we explain the basic principles of the SPLICE architecture, and present part of the formalization of the data space in PVS, the complete description is available in [9].

2.1 Basic framework in PVS

To obtain mechanized support for our approach, we use the theorem prover PVS[1]. Specifications in PVS are written in higher-order typed logic and they can be structured into parameterized theories. These theories may contain, for instance, declarations, definitions, axioms, lemmas, and theorems. PVS has a large prelude with many predefined theories such as lists, sets, natural numbers, integers, reals, relations, functions, etc. The tool also contains a theorem prover which can be used to construct proofs for lemmas and theories interactively. It can also be used to check previous proofs, e.g., after small changes. We briefly introduce our formal framework in terms of the PVS specification language. The type-checking mechanism of PVS might generate proof obligations (so-called *Type Check Conditions*), which are often already very useful for an early detection of inconsistencies in the specification. During the refinement of the system we reason on the level of predicative specifications. The correctness of these design steps is verified using the interactive proof checker of PVS. This verification system also allows the easy reuse of an already existing framework formalized in PVS, e.g, by using a compositional proof rule, discussed in [10], "off the shelf". In the sequel, the `typewriter` font indicates formalization in the PVS language.

We choose the time domain `Time` to be equal to the set of real numbers `real`, which is predefined in PVS. Our basic observables are `Events`, like sending a value, updating a database, etc. As in [10] we define the notion of an *observation function* `o`, which is a function from the time domain to a set of events. Note that we allow to have both discrete actions and continuous functions (which are observable at any point in time).

```
Time       : TYPE = real
Duration   : TYPE = { t : Time | t > 0 }
```

[1]For more info about PVS see `http://pvs.csl.sri.com/`.

```
t         : VAR Time
Events    : NONEMPTY_TYPE
ObsFuncts : TYPE = [Time -> setof[Events]]
o         : VAR ObsFuncts
```

Usually we won't be able to state that some event occurs after some exact amount of time - we only define time intervals, represented as sets of time points `setof[Time]`, in which events happen or during which events are permanently observable. We use `co` for left-closed right-open intervals, etc. A predicate `pred[T]` of an arbitrary type `T` is a function `[T -> bool]`. We define when a time predicate `P` holds *inside* or *during* an interval `I`. A specification of a (real-time) component is essentially a predicate of observation functions; it expresses constraints on the observations that are allowed by the component.

In the full real-time framework of [10], a component is described by an assertion and an alphabet which describes the set of events that might occur during the execution of the component. Reasoning about parallel components has been supported by a proof rule which expresses that we can take the conjunction of the assertions, provided the assertion of each component only refers to events that occur in the alphabet of the component itself. Since this last condition is easy to check and not relevant for the topic of the current paper, we don't formalize the alphabet here, but simply identify specification and assertion, and just use the conjunction of assertions to verify parallel composition of components.

2.2 The SPLICE data space

The software architecture for distributed control systems SPLICE is based on a shared data space coordination model. It has been applied in the construction of commercially available command and control systems. This model consists of two basic types of components: a shared data space and processes which act concurrently on this data space. Communication takes place through the shared data space only by reading and writing data elements. In our simplified SPLICE model, the operations performed on this data space are a $write(key, val)$ operation with the effect of inserting an entry (key, val) for the particular sort (and overwriting a possible existing entry with the same key), and a $read(query, val)$ operation. For simplicity, we reduce the options for $query$ here to be only asking for a certain key, returning the result val instantaneously.

The data space definition is independent of the particular *sort* to be stored. (Thus a data space containing multiple sorts is just a collection of these simple spaces.) In the PVS formalization this is modeled by parameterizing the theory `DataSpace` with the relevant information on keys, values, and a characteristic *delay* constant.

```
DataSpace [ Keys: TYPE+,
            Values: TYPE+,
            delay: Duration ]            : THEORY
```

Data entries are indexed by their key field such that an instance that differs in this key from already stored instances will be stored separately; new instances that are identical in the key-field with an already stored instance will replace the old one. As this key is unique, the contents (or state) of a data space can be modeled as a function from `Keys` to `Values`, denoted `[Keys -> Values]`. To capture the changes of the state of the data space over time, the data space is defined as a function from `Time` to `[Keys -> Values]`.

```
dataspace : TYPE =  [ Time -> [Keys -> Values]]
db :  dataspace
```

The operations involved are

```
write(key,val) : Events
update(key,val): Events
read(key,val)  : Events
```

where `write(key,val)` represents the write action of the producer, `update(key,val)` represents the internal update of the database, and `read(key,val)` represents the retrieval of information from the database. The `write` operation originates from an application process, its effect on the data space is not immediate. We assume a duration `delay` as a limit between executing the `write` operation and the actual `update` of the data space. This `delay` covers internal delays of the data space system as well as delays caused by the communication lines between application process and data space. The specification of the data space is a collection of properties. We list only those needed for the flight tracking and display example below.

In our model every `write` results in at least one `update`. This is in fact a strong assumption on the SPLICE architecture, as the basic architecture does not guarantee a reliable data transmission between a sender and the storage [3]. Moreover we postulate that any `write` event causes an `update` within less than `delay` time.

```
WriteUpdate : bool = FORALL  o, key,val, t1:
       o(write( key,val))(t1)
       IMPLIES
       inside( o( update( key,val)), co(t1, t1 + delay))
```

Symmetrically, every `update` is caused by a `write` event which has been emitted less than`delay` before.

```
UpdateWrite : bool = FORALL  o, key,val, t1:
      o(update( key,val))(t1)
      IMPLIES
      inside( o( write( key,val)),  oc(t1-delay, t1))
```

The actual value of the data space `db` is defined as the value stored by the last `update` event in property `Update_DB`. We can only retrieve the value belonging to an initialized `key` (here, a key for which we had already an `update` event). We use a simple reading mechanism which returns the current value belonging to the indicated `key` from the data space.

```
ReadCurrent: bool = FORALL o, key, val, t:
     o(read(key,val))(t) IMPLIES db(t)(key)= val
```

There is a subtle point in formally identifying the last update of the data space before a given moment, as required by Update_DB. This poses a problem if we allow "Zeno" situations, with infinite updates in a finite amount of time. As update events are generated by the data space system, we assume that they are discrete and we can have only finitely many of them in a given interval. This is captured by a property Finiteupdate. Furthermore there is at most one update of an entry in the database at any moment, formalized in property Exclu_update. The specification of the data space is the conjunction of these properties.

```
DataSpaceSpec: Assertion =
     WriteUpdate & UpdateWrite & Update_DB
     & ReadCurrent & Finiteupdate & Exclu_update
```

As a justification of our model, we prove some properties, representing typical consistency requirements of standard database theory, from the above characterization of the data space.

3 Top-level system description

In this section we discuss the formalization of the requirements on a flight-tracking-and-display system, consisting of

- a radar sensor, which detects and observes flying objects, and calculates their position, and
- a display device, which reads data from the system and displays it on some visualization device.

While developing the system, we have to incorporate the specifics of the underlying architecture. Here radar and display are distributed devices, connected to loosely-coupled components which communicate via the shared data space, where the sensor component stores flight data which might be read by the display component.

Flying objects have a unique flight number $key \in FlightId$ as identification. Although in reality the representation of a position may be different in the various components of the system, requiring conversions, this has been ignored here and we just assume some common data format Position, and a distance function distance which yields a non-negative real number of type Distance.

3.1 System requirements

We list the properties which constitute our top-level specification TopLevel in PVS notation, relating the observation of a real object and the physical representation on the display device. Typical Events are realobject(key, pos)(t), denoting that object key is at position pos at some global observation time t, and onscreen(key, pos)(t), denoting that flight key is displayed on the screen at position pos at some observation time t.

We introduce an upper time bound globdelta, the maximal duration between the actual observation of an object and its representation on the display. Another duration constant is minobs, the minimal duration for which an object has to be in the observation area if it is to appear on the display. The distance between the position of the real object and its displayed position is bound by parameter globallimit.

The system has to satisfy the following three properties:
1. If an object is observable for at least minobs, there has to be an associated display on the screen within the delay globdelta. Moreover, the information represented by that display is accurate w.r.t. to the observed position, i.e., they only differ from it by at most the distance globallimit. For ease of notation we abstract from the actual position of the object in the interval co(t, t+ minobs) using the abbreviation realob(key)

```
Existonscreen : bool = FORALL o, key, t, pos:
     dur( realob(key)(o), co(t, t+ minobs))
     &  o( realobject(key,pos))(t)
     IMPLIES   EXISTS  pos1 :
     inside(o( onscreen(key, pos1)), co(t,t+ globdelta))
     &  distance(pos, pos1) <= globallimit
```

2. Every item displayed on the screen represents an object which was at that position at most globaldelta ago.

```
Onscreensecond: bool = FORALL o, key, pos, t:
     o( onscreen(key,pos))(t)
     IMPLIES
     inside(o( realobject(key,pos)),oc( t- globdelta,t))
```

3. The screen is displaying permanently; provided an object is observed, it is represented at most globdelta later, and continuously at least as long as the object is observed.

```
Continreq: bool = FORALL o, key, t1, t2:
     dur( realob(key)(o), co(t1, t2))
     & t2 > t1+ minobs
     IMPLIES
     dur( screenob(key)(o), co(t1+ globdelta, t2))
```

3.2 Assumptions about the environment

In the design process of a system, usually some knowledge about its environment is needed. Quite often this knowledge is used implicitly without even being properly aware of its use. When using formal methods, however, this implicit knowledge has to be formalized. This is especially the case when tools such as PVS are used to check all proof steps mechanically.

In the current example, an assumption on the environment is, e.g., that an object can only move within a certain range in a certain time. This maximal movement is limited by a monotonous function epsilon. Therefore, the distance limit globallimit is considered to be derived from the time limit globdelta.

```
Physics : bool = FORALL o, key, pos1, pos, t1, t2:
     o( realobject(key, pos1))(t1) & t1 <= t2
     & o( realobject(key, pos))(t2)
     IMPLIES
     distance(pos1,pos) <=  epsilon(t2- t1)
```

The above assumption turns out to be needed, since in the implementation of the system using SPLICE, some data items might be overwritten before they are read. Now assume that an object was detected at a certain position at time `t`, after which it quickly "jumped" to some other unbounded distant position, and also assume it exists during at least `minobs` after `t`. This might lead to data `pos` at `t` and some data `pos'` later at `t'` such that `distance(pos,pos') > globlimit`. If the first data item was not read and we only got the second one displayed (within `t + globdelta`), requirement `Existonscreen` would have been violated. The next assumption `Uniqueobject` is that an object is a solid, indivisible entity. Consequently any object has a unique position at any fixed moment `t`. The top-level specification `TopLevelSpec` is given by a conjunction of the three properties, provided the assumptions hold:

```
TopLevelSpec : Assertion =
      Physics & Uniqueobject
      IMPLIES Existonscreen
      & Onscreensecond & Continreq
```

4 Specification refinement

This section contains the refinement steps that transfer the top-level specification, which is in terms of continuous observations, into a discrete specification, as indicated in Figure 1. We present only the main ideas of the specifications and refer to [9] for more details.

We start with the top-level specification `TopLevel` which describes the relation between the continuous observations on the `realobject` and the actual presentation `onscreen`. The details of this specification are explained in Section 3.

In the first step, we abstract from the continuous aspects of the screen by postulating that our system emits `display` messages with a sufficiently high rate. The specification is thus reduced to a relation between the `realobject` observations and the discrete `display` messages, as formulated in specification `DisplayTrack`, in combination with the `Screen` specification (see Section 4.1). In a second step, the continuous observation of the `realobject` is discretized into a stream of `sensor` events.

The specification `SensDisplay` then serves as description of the discrete system to be implemented. It is split up into components in Section 5.

4.1 First refinement step

The first step separates the `Screen`, as specified in Section 4.1.1, from the rest of the system, called `DisplayTrack`, which is specified in Section 4.1.2.

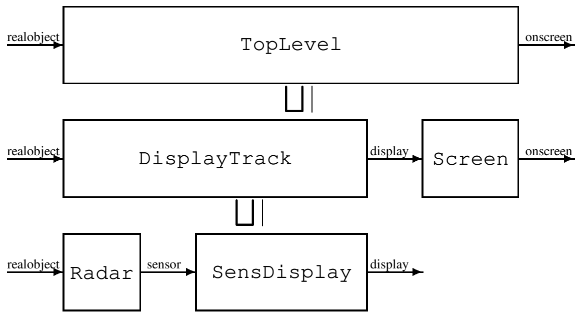

Figure 1. Refinement

4.1.1 Screen

For simplicity we assume that the screen has a refresh rate, given by `screenperiod`. For each object `key` to be displayed we regularly get a `display` event with a position. The screen immediately displays the current position, and it displays this information until the next `display` event occurs. This also means that a display icon will vanish if there is no new `display` event within `screenperiod`. These mechanisms are formalized in property `Displayonscreen`.

Similar to the `update` operations in the data space, we have to formalize that our system can only generate finitely many `display` events within a limited interval, this is formulated analogously in property `Finitedisplay`. The `display` events are only consumed by the `Screen`, therefore this property has to be seen as an *assumption* when specifying the `Screen`. It is the remaining system `DisplayTrack` that has to satisfy `Finitedisplay`.

```
AScreen : Assertion = Finitedisplay
        IMPLIES Displayonscreen
```

4.1.2 DisplayTrack

For the remainder of the system we establish a specification `DisplayTrack` in terms of the `realobject` and the discrete `display` events of the system. Besides `Finitedisplay`, this specification consists of three properties `Existdisplay`, `Displaycorrect`, and`Displayinterval`, which correspond to the three properties of the original specification. E.g., `Existdisplay` expresses that after a `realobject` event a corresponding `display` event occurs within `sysdelta` with a position that differs at most `epsilon(sysdelta)`, similar to property `Existonscreen`. Another timing parameter of `DisplayTrack` is `displayperiod`, the rate of `display` events sent to the display.

We maintain the assumptions on the environment from `TopLevel` and have as specification of the system without the screen

```
ADisTrack : Assertion =  Physics & Uniqueobject
        IMPLIES
        Existdisplay & Displaycorrect
        & Displayinterval & Finitedisplay
```

The combination of `ADisTrack` and `AScreen` then satisfies the original specification `TopLevelSpec`.

```
RefStep1  : THEOREM ADisTrack  & AScreen
        IMPLIES TopLevelSpec
```

In order to prove correctness of this refinement we formulate and prove a number of lemmas in PVS. During these proofs we have to introduce constraints on time parameters by adding additional `AXIOMS` to the PVS-theory. Theorem `RefStep1` holds under the following assumptions on the durations which occur in these specifications.

```
TimeConstraint1 :  AXIOM
       sysdelta + screenperiod <= globdelta
TimeConstraint2 :  AXIOM displayperiod < screenperiod
```

To satisfy the requirement in the top-level specification that the screen has to display a representation of a object within the duration `globdelta`, we have to ensure that the sum of the duration of the refined system together with the maximal delay the screen may exhibit is less than the global duration. `TimeConstraint2` requires the rate of `display` events to be higher (i.e., to have a smaller periodicity) than the screen refresh rate as given by `screenperiod`.

4.2 Second refinement step

The second step separates the radar unit from the system, leading to the system specification `SensDisplay`. The radar sensor samples the position of an object at least every `senseps` time units. We have a property `Evensensor` stating that if an object is present for a certain duration `senseps`, it will be detected within this period.

A second property `Correctsensor` indicates that the sampled data provided by the radar is precise. These two properties define a relation that transforms the continuous observation `realobject` into a stream of discrete `sensor` events.

Furthermore we define a relation between incoming `sensor` events and the outgoing `display` events of the system to describe the remaining system, now in terms of discrete events only. This new specification `SensDisplay` has local time limit `sysdelta2` as maximal duration between a `sensor` event and the associated `display` event. The properties listed in the specification `ASensDis` are similar to those of `DisplayTrack` before.

Similar to the top-level specification, we need some assumption `Slimid` on the incoming sensor data, stating that the observed object moves within the limits given by the function `epsilon`, now in terms of the interface of `SensDisplay`. This property is implied by the assumptions on the environment from the top-level specification together with the sensor specification. We prove that the sensor specification `ASensor` and the new system specification `ASensDis` imply the previous specification `ADisTrack`.

```
RefStep2  : THEOREM ASensDis & ASensor IMPLIES ADisTrack
```

This is possible only by introducing the following constraints.

```
TimeConstraint3 : AXIOM senseps <= minobs
TimeConstraint4 : AXIOM
      2 * senseps + sysdelta2 < sysdelta
TimeConstraint5 : AXIOM senseps <= displayperiod
```

Here we have, e.g., the requirement that the `minobs` duration has to be greater than `senseps`, the time the radar needs to ensure detection. The relations between the other time parameters is less obvious. These constraints were found necessary during the PVS-supported verification of theorem `RefStep2`. Even though they are not generated automatically, these additional requirements can be extracted during the proof process without much effort.

5 Decomposing the system

Next the discrete specification `SensDisplay` is decomposed into two application components and a data space, as shown in Figure 2. Component `Sensor` is event-driven; it gets information by a `sensor` event from the radar and sends this information to the data space by a `write` event. Component `Display` is time-driven; it reads periodically the present information of each object's position, calculates the display data and sends a `display` event to its output device. Observe that specification `SensDisplay` does not

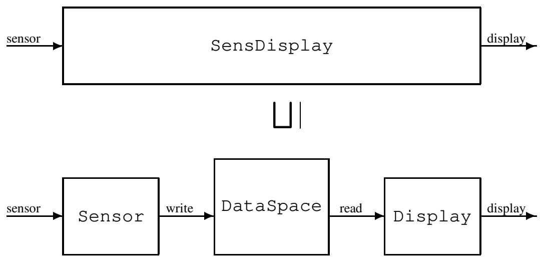

Figure 2. Implementation

allow an endless display of the last data space item after the real object has disappeared for a significant duration. To obtain loosely-coupled components, they should only communicate using the data space, and hence a stop of the display should also be ensured using the data space. Therefore, we introduce an additional data field, generated by the `Sensor`, serving as *time-stamp*.

This allows the consumer `Display` to ignore data which is considered to be "too old" to be accurate. All decisions w.r.t. the time-out mechanism are solely taken by the consumer.

5.1 The sensor component

The `Sensor` computing unit is an input driven data converter, getting information from the physical radar and forwarding it with an additional time-stamp within less than `sctdel`. To add a time-stamp to a position, we use the record notion of PVS, denoted as
`[# pos : Position, t: Time #]`.

We choose a simple mechanism, where every data tuple `(key,val)` received is forwarded to the storage, now as a pair consisting of the original data and a representation of the time point when this data was received by the process, i.e., a data entry of the form `(key,[val,timestamp])`. As the producer provides the time-stamp w.r.t. its local clock, this guarantees that data d_2 produced later than data d_1 also has a bigger time-stamp. The specification of the sensor `Scompt` describes the above forwarding mechanism, and guarantees that every `write` operation originates from a recent `sensor` operation.

5.2 The display component

The display component `Display` is required to stop emitting `display` messages, whenever we have no recent value available from the data space. We have no recent value whenever we read a defined value at time `t1` and we are not able to read a value with higher time-stamp for a period longer than a constant `valperiod`.

The behaviour of the display component is given by the following three cases on what may happen after a `read` operation.

`PDisTime1`: If the read value is undefined, we will `read` again, but there is no `display`.

`PDisTime2`: If we have no recent value, no `display` will be send until the next `read` operation.

`PDisTime3`: If the value is defined and recent, a `display` message will be emitted before the next `read` operation.

These three properties include the requirement that the next `read` operation should occur within `displayperiod`. Furthermore, property `Sounddisplay` requires that every `display` originates from a `read` with recent data. Moreover, this component guarantees property `Finitedisplay`, as required by the `Screen` (see Section 4.1.1). This leads to the specification of the display process:

```
DisSpec: Assertion =  Finitedisplay & Sounddisplay
             & PDisTime1 & PDisTime2 & PDisTime3
```

5.3 Process Composition

We combine the components `Sensor` and `Display` with the `DataSpace` defined in Section 2.2. As we concentrated our analysis on a running system, we ignored all initialization aspects of the system so far. Now that we are at the level of individual independent processes, we have to assume some properties that are due to some initialization.

For the `DataSpace`, we have to capture the case that a value might be undefined before the first update occurs, this is done by the condistion `Initialdb`. For the `Display`, we assume that for each interval of length greater or equal than `displayperiod` there is a `read` operation at least every `displayperiod` time units. This assumption `Alwaysread` guarantees that we have some `read` operation that qualifies as “first” w.r.t. our observation interval.

Note that the `Sensor` needs no initialization condition because its output is triggered by the presence of objects. Together with these initialization properties, the three component specifications `Scompt`, `DataSpaceSpec` and `APDis` imply the discrete system specification `ASensDis`.

```
Refstep3 : THEOREM Scompt & DataSpaceSpec
       & DisSpec  & Alwaysread & Initialdb
       IMPLIES  ASensDis
```

Again, the correctness proof of this step leads to more time constraints such as

```
TimeConstraint6 :  AXIOM
       2 * senseps + 2 * sctdel + 2 * delay
       + displayperiod    <= valperiod
TimeConstraint7 : AXIOM  sctdel + delay + valperiod
       + 2 * displayperiod <= sysdelta2
```

`TimeConstraint6` requires the timeout value `valperiod` to be chosen big enough to avoid classifying a value which just took all possible allowed delays as “too old”. `TimeConstraint7` is generated as the worst case for some sensor data to be forwarded as a display event with maximal delay in every component without triggering the timeout mechanism.

5.4 Combining the Time Constraints

The collection of time constraints allow further investigation on the trade-offs between the various parameters. As an example we have

```
TimeReq1: LEMMA   2 * senseps +  sctdel + delay
       + valperiod + 2 * displayperiod
       + screenperiod    <= globdelta
```

and by applying `TimeConstraint6` to this, we conclude that we have to restrict the following term by the global upper time bound `globdelta`.

```
TimeReq2: LEMMA  4 * senseps +  3 * sctdel + 3 * delay
       + 3 * displayperiod + screenperiod  <= globdelta
```

The durations senseps and screenperiod (which belong to Radar and, respectively, Screen) are externally given. This leaves us with the possibility to tune the parameters of the software system: sctdel (the maximal time between writes), delay (the maximal time between a write and the corresponding update), and displayperiod (the maximal time between reads). Observe that the value of displayperiod is limited further, e.g., by TimeConstraint2. Note also that the choice of external components (i.e., their timing constraints) influences the requirements on the individual components of this flight-tracking-and-display system.

6 Conclusions

We have presented a formal approach to the top-down design of distributed real-time applications on top of a shared data space application. To this end, we formalized the properties of a data space architecture which has been inspired by the industrial software architecture SPLICE. Components only communicate via a shared data space, which strongly influences the design. In our compositional approach, every design step is proved correct, using the verification system PVS. The use of this proof tool is especially useful to identify various constraints on the timing parameters of components. The resulting timing constraints often give an indication for the trade-off between timing characteristics of the components .

Observe that our formalization of SPLICE decouples a write event by a producer of data and the corresponding change in the data space (by an event, reflecting the asynchronous nature of this architecture. In the current version, there is no such decoupling for read events; all read events immediately obtain one of the globally available data items. In the actual implementation of SPLICE, however, each node in a network has its own local storages [3]. Even if these local data spaces are to contain the same data sorts, they do not necessarily contain the same data items at a fixed moment. In future work, we intend to formalize this situation, by adding the above mentioned decoupling for read events to our model. Then it becomes interesting to re-use our flight-tracking-and-display example and study a version with several data producers and consumers.

The flight-tracking-and-display system is a typical example of a system with producers and consumers, and a certain coordination pattern (e.g., the fact that consumers only accept "fresh" data items). An interesting topic for future work is the formulation of such patterns on a general level, using parameterized PVS theories, and to investigate whether already some general properties can be verified in PVS. Concrete applications could then be considered as instantiations of these general patterns.

Acknowledgement

We would like to thank Edwin de Jong for suggesting the flight-tracking-and-display system as a basic example with both event-driven and time-driven components.

References

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

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

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

[4] M. Bonsangue, J. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In *Symposium on Applied Computing (SAC '98)*, pages 159 – 168, Atlanta, Georgia, USA, February 27-March 1 1998. ACM press.

[5] M. M. Bonsangue, J. N. Kok, and G. Zavattaro. Comparing software architectures for coordination languages. In *Conference on Coordination Models and Languages, COORDINATION'99.*, volume 1594 of LNCS, pages 150–165. Springer-Verlag, April 1999.

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

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

[8] D. Gelernter. Generative communication in Linda. *ACW Toplas*, 7(1):80–112, 1985.

[9] U. Hannemann and J. Hooman. Top-down design of a command and control system with timing assumptions. Technical report, CSI-R0116, University of Nijmegen, 2001.

[10] J. Hooman. Compositional verification of real-time applications. COMPOS'97, volume 1536 of LNCS, pages 276–300. Springer-Verlag, 1998.

[11] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of LNAI, pages 748–752. Springer, 1992.

[12] 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*, 21(2):107–125, 1995.

[13] J. van de Pol. Expressiveness of basic Splice. Technical Report SEN-R0033, Centrum voor Wiskunde en Informatica, Amsterdam, 2000.

[14] J. van de Pol, J. Hooman, and E. de Jong. Requirements specification and analysis of command and control systems. Technical Report 99-09, TU Eindhoven, 1999.