Formal Reasoning about Real-time Components on a Data-oriented 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 investigate an approach towards formal reasoning about component-based software architecture. In this paper we develop specification schemes for real-time applications which interact via the data-oriented software architecture SPLICE. Composition of these applications requires not only knowledge about the component specifications, but also information about key parameters of the underlying communication mechanism. We formulate a proof scheme for the interaction of data-oriented components, and illustrate its use with an example of a flight-tracking-and-display system.

Keywords: Component-based software development, coordination languages, real-time applications, formal methods, compositional design.

1. Introduction

The general aim of our work is the formal design of real-time systems which are based on a component-based software architecture. In this specific case we investigate the interaction of components on top of the data-oriented software architecture SPLICE [3].

SPLICE has been used – and is currently used – to build large and complex systems, such as command and control systems. This 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, and communicate indirectly via the SPLICE primitives, basically $read$- and $write$- operations on a distributed data space. This type of anonymous communication between components is strongly related to coordination languages such as Linda [10] and JavaSpaces [9].

The SPLICE architecture offers a timestamp mechanism, but it does not ensure global consistency and the asynchronous communication mechanism does not preserve the order of messages. 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.

Most of the published work on SPLICE focuses on formal descriptions of the semantics of SPLICE [7, 6, 2, 12]. Formal specification and analysis of command and control systems that are designed on top of SPLICE have been studied in [8, 11], using the interactive theorem prover PVS [13] as tool support.

Typical SPLICE application are built out of components which create and process data streams that exhibit some periodic behaviour. We suggest a user-friendly, intuitive specification style for this kind of components with emphasis on their real-time properties.

This is related to the real-time parameters introduced in the real-time specification for Java [5] to describe execution times. A similar concept can be found in the real-time CORBA architecture TAO [14], where periods, average and worst-case delays are used to provide the scheduler with necessary information. For the *data-oriented* SPLICE architecture it seems only natural that the real-time characteristics are associated directly with the data itself. We introduce *data signatures* that include timing information about the respective type of data records.

These data signatures are easy-to read, and yet include all relevant information about the interaction of a component with its environment. We employ them as main ingredients to (formally) reason about the interaction of components on top of SPLICE.

The descriptions of incoming data streams serve as assumptions on the behaviour of the environment, and the central question when reasoning about composition is how they can be satisfied by the remaining system. Composition of components also depends on *quality-of-service* (QoS) aspects for data delivery, like persistence, network delays and delivery guarantees, which can be specified by the SPLICE user.

With this information, we have a formal basis for precise reasoning about when data emitted from one component satisfies an input assumption of another component. This leads to composition patterns for data streams that are used in a composition rule for SPLICE components. This rule is suited for the usual "pipeline" style of data flow within a SPLICE system, where no mutual dependencies on data occur.

The description schemes of data signatures and network characteristics are easily translated to PVS theories as a full formalization of component and network behaviour. These generic translations support reasoning on instances of the proof schemes, where necessary side conditions, relating input and output descriptions, have to be formally introduced.

We give a brief introduction into the concepts behind SPLICE in Section 2, and develop specification patterns for components and network settings in Section 3. Section 4 discusses the general proof scheme for SPLICE components, which is applied to the example of a flight-tracking-and-display system in Section 5. We

*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.

illustrate the use of the theorem prover PVS in Section 6, where we give a detailed formalization of particular network characteristics.

2. SPLICE - a Data-oriented Software Architecture

The software architecture for distributed control systems SPLICE is based on a distributed data space coordination model using loosely-coupled, re-usable, and replicable components. Conceptually, every application on top of SPLICE has its own data space. Communication takes place between the local data spaces by a publish-subscribe mechanism, where a data producer broadcasts data records to the other data spaces, and a data consumer subscribes to the required types of data, and stores only those that match this subscription.

Data records within the SPLICE system are described in the relational data model. All data elements are associated with a unique *sort* that defines their structure. A sort definition declares the name of this sort, the record fields the sort consists of, and the *key* fields [4]. Data records include a *timestamp* field which allows some filtering on the consumer's side. The basic idea is that these timestamps indicate an order on the records sent, and that a record can be thrown away by the receiver if its timestamp is smaller (or equal) than a record received before. This technique reduces the number of updates to be performed on the data space, as not all received records get stored, and improves the quality of the data stored, as no record can be overwritten by older data.

Applications

The *application* language is a programming language enhanced with coordination primitives, i.e., SPLICE constructs for retrieving data from the local data space, and for publishing data. The only way to have processes acting in parallel is to invoke them separately; there is no parallel composition within the language other than the process notion. One process may invoke another process, but the new process then acts independently. Application programs are sequential programs, where only the SPLICE constructs interact with the local data space, by reading and writing data elements [7].

Components

In this context, a *component* is an application program together with its associated data space. The interaction of a component with its environment takes place by the SPLICE communication mechanism and by *external events*, which subsume, e.g., receiving data from sensors, or sending signals to external devices. We specify components by a description of their inputs and outputs.

These descriptions are given by listing the respective data sorts or events together with relevant characteristic parameters in a structured way. This unified interface format for both data sorts and external events helps to give a simple and precise specification of a component.

For the component specification, the description of the input is seen as assumption on the behaviour of the environment, and the output description is the commitment of the component towards the environment, provided the environment satisfies the input assumptions. Note that the input description of a data sort refers to the set of data records that are actually stored in the local data space, not to the set that is actually produced by all components. All attributes of other components (like their data emission rate) are not visible, and should not influence the design, implementation and verification of a particular component.

Composing Components

As components interact only via the data records which are transmitted by the data space architecture, any representation of the implemented system should be seen as composition of the various components by means of SPLICE.

Reasoning about the system as a whole merely concerns the decomposition of system tasks into components. To this end, we have to specify the individual components, and prove that their composition guarantees the desired system properties. While the system's description is formulated in terms of external events only, the component description also includes the interface towards the SPLICE mechanism, the definition of data sorts and their attributes. Since a system is inherently based on the composition of data producers and data consumers, reasoning about the interplay of components thus involves the characteristics of the data, as well as the parameters which influence the data record transport. Delivery parameters, representing features of the underlying network(s), are assigned to all data sorts as QoS attributes, and they play a role in the composition rules.

3. Specification Schemes

The general specification scheme of a component lists assumptions on input and the characteristics of the output that is generated on the base of the input. This information is structured as a *data signature*, associating with each data sort a set of quality attributes. It extends the data sort definition, listing the name and data fields, by adding real-time characteristics for input and output.

As typical SPLICE applications, we investigate examples where the components create and process data streams that exhibit some periodic behaviour. A variety of parameters has been suggested for real-time applications[5, 14]; here we choose:

1. *starting time* to denote the time of the first data item of the steam, possibly represented by constraints;
2. *ending time* to denote the time of the last data item (if any), where also constraints can be used, e.g., for a relation with the start time;
3. *period* for the emitting/receiving rate, representing the maximal distance between two subsequent records;
4. *age* as measure for actuality of the data with respect to some other data or event stream that serves as a reference.

For a data sort, we have the following data signature:

```
Sort  : datasort(Key,Values)
Start : s_0    ; [constraints]
Period: period ; [constraints]
End   : t_0    ; [constraints]
Age   : delta    [w.r.t. reference]
```

A typical requirement for data stream processing is "use only data that has been generated at most `Age` before". `Age` might represent the accumulated time in which some information is processed through a number of operations by different components.

External Events

As components of a system on top of SPLICE can also interact with the environment by external events, we characterize these in the very same style as the internal data streams by *event signatures*, replacing *sort* and key/value fields by *eventname* and appropriate parameters.

Component Specifications

We describe components by listing their input and output descriptions (IN, OUT), where both IN and OUT are a set of data and/or event signatures. Provided a component P receives data and events such that these match the constraints of IN, P satisfies (IN, OUT) if the output of P matches the constraints of OUT. We denote this by P **sat** (IN, OUT). We also refer to the input or output part of a specification S as $S.IN$ and, resp., $S.OUT$.

Network Attributes

The SPLICE architecture includes the possibility to define a QoS policy for the distribution of each data sort, describing aspects of data management which are realized by some underlying network. A QoS policy is defined in terms of attribute values like persistence and delivery (explained later). These choices allow the developer to tailor the data transfer individually by sort or even by a subset of a sort. We add a parameter `Delay` which indicates the maximal delay of a record between sending and receiving.

```
Sort       : datasort(Key,Values)
Delivery   : {guaranteed | best effort}
Persistence: {Volatile | Transient | Persistent}
Delay      : delay
```

The specification schemes presented above have been defined formally in the specification language of the theorem prover PVS. A few details can be found in Section 6.

4. Composition Principles

A system on top of SPLICE consists of a number of components, and for each of these components we have some input/output specification. The input specification of a component constitutes a set of assumptions on both external events and received data records. Since a top-level system specification consists of event signatures only, a composition rule has to provide some guideline on how to remove input assumptions on data for components from the composed system.

We gradually resolve this by first reasoning about the composition for single data sorts and next using this to define a general pattern for reasoning about the composition of components.

Given specification S_1 of a component P_1 with some data sort `name` as output signature, and an input data signature for `name` of component P_2, we can combine P_1 and P_2 under certain conditions. The asynchronous nature of SPLICE and optimizations like the filtering on timestamps defy a direct correspondence between the signatures. The specifics of the network *together* with the characteristics of the producing component determine which parameters are met on the receiving side.

We say that "OUT establishes IN", if all instances of $S_1.OUT$ for `name` and the network characteristics for `name` resolve to a data stream with characteristics that obey the constraints of $S_2.IN$ for `name`.

If, for instance, S_2 requires that a certain data sort should be received starting at t_1, it is not sufficient to know that the producer's specification guarantees that it will start sending at t_0 and $t_0 \leq t_1$. As the network may delay delivery of records by `delta`, we can only safely assume that the first data record is received in the interval $[t_0, t_0 + \texttt{delta}]$, i.e., that t_1 lies somewhere in this interval.

Composition of components then boils down to a proof that some output data signature of a producer together with the network characteristics associated with that data sort establish the input assumptions for the consuming component.

Application of the proof scheme follows the causal order on data generation and processing, i.e., when one component generates data that is assumed by another one, we may discard this input data signature from the composition. For the moment we do not investigate component combinations with cyclic data flow, since it is well known that this style of assumption-commitment reasoning requires extra care in that case, and the number of applications where this is actually relevant seem to be limited.

For non-cyclic dataflow we have the following proof scheme, explaining the *merge* operation below.

$$\frac{P_1 \textbf{ sat } (IN_1, OUT_1),\quad P_2 \textbf{ sat } (IN_2, OUT_2)}{(P_1 \underset{\text{SPLICE}}{\parallel} P_2) \textbf{ sat } (IN, OUT)}$$

where $OUT \subseteq merge(OUT_1 \cup OUT_2)$,
and $IN = (IN_1 \cup IN_2) \setminus$
$\{in \in IN_i \mid \exists out \in OUT_j : out \text{ establishes } in,\ i \neq j\}$.

The output of the composed system is the output of the respective components, but we may remove individual *data* output characteristics for the composed specification, if we have no further use of them. We also have to be aware of the fact that both components may produce data of the same sort. The generation of records of this sort is anonymous for other components, they can not observe the identity or even the number of producers. Therefore we have to combine the individual characteristics to a *common* description of all producers of a particular sort in the composed system. This *merge* operation calculates new constraints on the output specification of a sort. This can be relevant when discussing fault-tolerance aspects. For instance we may have data producers P_1 and P_2 for the same sort and with the same rate, but the first one producing only in an interval $[t_0, t_1]$ and the second one in an interval $[t_2, t_3]$, with $t_2 \leq t_1$. Neither of this signatures in isolation may be sufficient to establish a certain input signature, but $merge(P_1.OUT, P_2.OUT)$ then provides a stronger output signature which states that there is a data stream in the interval $[t_0, t_3]$. Note that for a component P, with output of only data signatures, i.e., no events, we have transparent replication $P \equiv P \parallel P$.

The input specification of the composition is a subset of the union of the individual input specifications. We can remove data signatures from this union if they are already established by some output signature of the other component. Note that we can only remove data signatures with this rule, external events remain in IN and OUT.

5. Example

We consider 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.

System Description

In [11], we described a formal top-down design of this flight-tracking-and-display. Such a system represents a typical application of a distributed control system, relating the detection of flying objects to their display on a monitor. We formally proved some refinement steps, leading to the description of a system which satisfies the following properties:

1. If an object is observable for at least a duration `minobs` (such that there is a sensor event at least every `senseps`), there will be an associated display on the screen within some delay (at most `sysdelay`). Moreover, the information represented by that display is accurate w.r.t. the observed position.
2. Every item displayed originates from some recently observed object.
3. Provided an object is observed, the system emits a display event at least every `displayperiod`, as long as the object is observed.

The approach of [11] was based on a single data space and featured complicated PVS specifications. In this paper we use more user friendly specifications and a formal model that much closer reflects SPLICE.

Formalization

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 suitable common data format `Position`. For this system we have the following scheme `SensorDisplay` specifying input and output:

```
SensorDisplay

   IN    : sensor(FlightId,Position)     : Events
   Start : t0
   Period: senseps
   End   : t1 >= t0 + minobs

   OUT   : display(FlightId,Position)    : Events
   Start : [t0, t0 + sysdelay]
   Period: displayperiod
   End   : [t1, t1 + sysdelay]
   Age   : sysdelay w.r.t. sensor
```

This description of the incoming and outgoing events contains all necessary information of the informal specification above. The object has to be observable to the system for at least `minobs`. Then there is a sensor event at least every `senseps`, i.e., we have as input to the system the event `sensor` (with the parameters `FlightId` and `Position`) with period `senseps`. This observation starts at `t0` and ends at `t1` with the additional contraint that `t1 >= t0+minobs`. The output specification describes the emission of display events, starting between `t0` and `t0+sysdel` (The sensor event from `t0` induces a display event within this interval). Display events are emitted at rate `displayperiod` and no data is displayed that is older than `sysdelay`, i.e., originates from data by a sensor event that occurred longer than `sysdelay` ago.

Using our generic interpretation in PVS, we can prove that this specification is a mere reformulation of the formal system specification of [11], which has been extracted out of the informal specification above.

Decomposition

We decompose the system specified by `SensorDisplay` into a sensor component, which gets some periodic discrete input from the the radar, and a display component which emits information to the external monitor. Sensor and display are loosely-coupled components which communicate via the distributed data space concept of SPLICE, where the sensor component emits flight data which might be read by the display component.

Here, we use a sort `sensdat` for data records that have been generated by a radar sensor, consisting of a key field `FlightId`, another attribute `Position`, and a field `Timestamp`.

The `Sensor` component gets `sensor` events (as above in `SensorDisplay.IN`) as input, and generates time-stamped data as output to SPLICE with an output rate of `sensdel`, starting between `t0` and `t0+sensdel`, and ending between `t1` and `t1+sensdel`.

```
Sensor

  IN    : see SensorDisplay.IN

  OUT: sensdat(FlightId,Position, Timestamp)
                                          : SpliceData
  Start    : [t0,t0 + sensdel]
  Period   : sensdel
  End      : [t1, t1 + sensdel]
  Age      : sensdel w.r.t. sensor
```

For the description of the `Display` we also specify the input/output characteristics. This component starts receiving `sensdat` records at `t2` until `t3` at a rate of `delta1`. On arrival the data has an age of at most `delta2`.

```
Display

  IN: sensdat(FlighId,Position, Timestamp)
                                        : SpliceData
  Start    : t2
  Period   : delta1
  End      : t3
  Age   : w.r.t. sensor          : delta2

  OUT : see SensorDisplay.OUT
```

Network Characteristics

The central question in reasoning about composition of SPLICE components is the combination of constraints on data with the characteristics of the network. We instantiate the SPLICE data transfer mechanism for a selected set of options.

```
Sort        : sensdat(FlightId,Position, Timestamp)
Delivery    : guaranteed
Persistence : Volatile
Delay       : netwdel
```

The variant discussed here features *guaranteed* data delivery, i.e., every record sent is actually received at least once. Moreover we postulate a maximal delay `netwdel`. The `sensdat` data sort is `Volatile`, i.e., no copy of the record is made in some secondary storage, e.g., for recovery purposes.

Composing the Parts

Having `Sensor.OUT` and `Display.IN` referring to the same sort, we prove that they indeed implement the system specification. Given the PVS theory which characterizes the network and the description (and interpretation) of both the `Sensor` output and the `Display` input in terms of `sensdat`, we calculate a number of constraints on the relation of the parameters of `Sensor.OUT` and `Display.IN`. Some of them are rather trivial, e.g., `t0 <= t2`, just stating that the `Display` does not receive anything before the `Sensor` starts emitting data; some of them reflect a worst-case scenario of accumulating delays, leading to a receiving period of `netwdel + sensdel`. This leads to the proof that the composition

$$\texttt{Sensor} \underset{\text{SPLICE}}{\|} \texttt{Display}$$

requires a restriction of `Display.IN` to:

```
Display
  IN: sensdat(FlighId,Position, Timestamp)
                                      : SpliceData
  Start    : [t0,t0 +netwdel]
  Period   :  netwdel + sensdel
  End      : [t1, t1+netwdel+sensdel]
  Age      : netwdel+sensdel w.r.t. sensor
```

Then `Sensor.OUT` establishes `Display.IN`, we can apply the proof scheme from Section 4 and remove both `Sensor.OUT` and `Display.IN`, leading to a specification that consist of descriptions of external events only, which constitute the system specification `SensorDisplay`.

6. Formalization in PVS

We formulate the network behaviour as a PVS theory, where e.g., `send(sensdat(key,pos,ts))(t)` denotes that the record `(key,pos,ts)` of sort `sensdat` is sent at time `t`. The term `member(t1,cc(t,t + netwdel))` is an expression for "`t1` is in the interval `[t, t+netwdel]`".

We characterize the network behaviour as specified in Section 5 by the following properties. Property `OutIn` captures the guaranteed delivery and the maximal delay, and property `InOut` states that data received must originate from somewhere.

```
FORMALIZATION Network

OutIn : bool = FORALL key, pos, ts, t:
                send(sensdat(key,pos,ts)) (t)
                IMPLIES
                EXISTS t1: member(t1,cc(t,t + netwdel))
                AND receive(sensdat(key,pos,ts))(t1)

InOut : bool =  FORALL  key, pos, ts, t:
                receive(sensdat(key,pos,ts))(t)
                IMPLIES
                EXISTS t1:  member(t1,cc(t - netwdel,t))
                AND send(sensdat(key,pos,ts)) (t1)
```

Function `filtrec` is used to select (filter) those records that are actually processed by the receiver's data space, i.e., those records that are not removed out because some more recent data is already available. A record is *not* removed if all records received before had a smaller timestamp.

```
Filter: bool = FORALL  key, pos, ts, t:
               filtrec(sensdat(key,pos,ts)) (t)
               IFF ( receive(sensdat(key,pos,ts)) (t) AND
               FORALL t1, pos1, ts1 :( t1 < t
                AND  receive(sensdat(key,pos1,ts1))(t1) )
                      IMPLIES ts1 < ts)
```

We have some more properties that are needed to fully characterize the communication mechanism; the use of a proof tool enforces us to make these explicit. As an example, we have to formalize that a component can only receive one record at a time. This is already sufficient to prove that the remaining records selected by `filtrec` are indeed ordered by timestamps.
Another necessary property is the fact that a component can only receive finitely many records, which is needed, e.g., to determine the *first* of possibly many `receive`s caused by a particular `send`.

7. Conclusions

We have investigated the interaction of components on top of the data-oriented software architecture SPLICE, and founded a base for the formal treatment of the composition operation by identifying relevant parameters for specification patterns for real-time data streams. For these patterns we generate an automatic interpretation as a PVS theory. This yields tool support for the analysis of the timing constraints.

Timing constraints are extracted from the combination of parameters of both the data stream and the SPLICE architecture as transport medium. Given a set of constraints, we can determine whether data production and data consumption match and apply a proof scheme for real-time applications on top of SPLICE.

Recall that network characteristics are *assumptions* only, as SPLICE has no built-in mechanism that could guarantee, e.g., a maximal network delay yielding a maximal delivery time from a data emitter to a receiver. The network parameters of Section 6 provide an example for the general construction of reasoning patterns for networks. In practice, these attributes can be replaced by some stochastic description, e.g., by listing *average delay*, *worst case delay*, etc. The validation that these assumption actually *are* met by the real system has to be provided by other means beyond the pure mathematical analysis of specifications (e.g., performance testing).

In the formulation of the proof scheme, the *merge* operation on data streams requires more attention: for a component P we may have P **sat** (IN_1, OUT_1) and P **sat** (IN_2, OUT_2), where the characteristics of the output depend on the characteristics of the input. Here, effects of changing data rates have to be taken into account, when calculating the parameters of the receiving data space. The same problem actually occurs when merging producers of the same sort with different output characteristics. The combination of them results in a sequence of intervals with different characteristics each.

The general reasoning pattern presented in this paper is restricted to a data flow without cycles. Yet, the model of time underlying our formalization induces a causal order on data transfer, which allows to break up cyclic reasoning. Further investigation, using appropriate examples, should lead to a more general composition rule.

Future work also includes reasoning about application programs. Note that input data signatures define an invariant on the data space state of a component. Together with some initialization condition, this establishes some assumption for formal reasoning about the actual application program, determining the values returned for data retrieval operations. Since application programs are sequential programs, one could apply some Hoare-logic style proof system [1] with extensions to cope with the data retrieval operations.

References

[1] K.R. Apt. Ten years of Hoare logic: a survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[2] R. Bloo, J. 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.

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

[4] 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.

[5] Greg Bollela, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Harding, Mark Turnbull, and Rudy Belliardi. *The Real-Time Specification for Java*. The Real-Time for Java expert group, final specification, December 2001. `http://www.rtj.org`.

[6] M. M. Bonsangue, J. N. Kok, and G. Zavattaro. Comparing software architectures for coordination languages. In *Proceeding of the Third International Conference on Coordination Models and Languages, COORDINATION'99.*, volume 1594 of *Lecture Notes in Computer Science*, pages 150–165. Springer-Verlag, April 1999.

[7] M.M. Bonsangue, J.N. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In J. Carroll, G.B. Lamont, D. Oppenheim, K.M. George, and B. Bryant, editors, *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98)*, pages 159 – 168, Atlanta, Georgia, USA, February 27-March 1 1998. ACM press.

[8] E. de Jong, J. van de Pol, and J. Hooman. Refinement in requirements specification and analysis: a case study. In *Proceedings of the 7th IEEE Conference and Workshop on teh Engineering of Computer-Based Systems (ECBS)*, pages 290–298, 2000.

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

[10] D. Gelernter. Genarative communication in linda. *Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[11] U. Hannemann and J. Hooman. Formal design of real-time components on a shared data space architecture. In *Proceedings of the Twenty-Fifth Annual International Computer Software & Applications Conference COMPSAC 01*, pages 143 – 150. IEEE Computer Society, 2001.

[12] J. Hooman and J. van de Pol. Formal verification of replication on a distributed data space archiutecture. In *Proceedings of the 17th ACM Symposium on Applied Computing (SAC 2002)*, pages 351–358, 2002.

[13] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. SRI International, Computer Science Laboratory, Menlo Park, CA, version 2.4, December 2001.

[14] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications, Elsevier Science*, 21(4), 1998.