Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Towards Automated Verification Of SPLICE In muCRL

P.F.G. Dechering, I.A. van Langevelde

Software Engineering (SEN)

SEN-R0015 May 31, 2000

Report SEN-R0015
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Towards Automated Verification Of SPLICE In μCRL

Paul Dechering
email: dechering@signaal.nl
Hollandse Signaalapparaten B.V.
P.O. Box 42, 7550 GD Hengelo, The Netherlands

Izak van Langevelde
email: izak@cwi.nl
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

A considerable fragment of the coordination architecture SPLICE, including ETHERNET, is specified in the process-algebraic language μCRL. This specification is used to generate transition systems for a number of simple SPLICE applications which are verified by model checking using the CÆSAR/ALDÉBARAN tool set. For these cases the properties of deadlock freeness, soundness and weak completeness are proven.

The primary result reported is a detailed formal model of SPLICE that makes possible automated verification. In practice, however, it is only for very simple SPLICE applications feasible to generate a transition system. Nevertheless, model checking applied to a large number of small applications, or scenarios, can be used to gather evidence for the validity of properties that is more general than testing in that it considers all possible system traces for a given scenario instead of just one trace. For applications with a high degree of non-determinism this can be an interesting advantage.

2000 Mathematics Subject Classification: 68N15 Programming languages, 68N30 Specification and verification, 68Q60 Specification and verification of programs
1998 ACM Computing Classification System: D.2.1 Requirements/Specifications; D.2.4 Program Verification; D.2.11 Software architectures
Keywords and Phrases: μCRL, process algebra, specification, SPLICE, coordination languages, CÆSAR/ALDÉBARAN, verification
Note: The research reported here was part of the project Onderzoek naar Randvoorwaarden voor de Konstruktie van Embedded SysTemen (ORKEST)

1. INTRODUCTION

The complexity of today's computer systems leads to severe difficulties in proving their correctness. In the future, this complexity is likely to increase. Therefore, it is necessary to improve the design process of these systems, such that it can be shown (as convincing as possible), based on well-formed design steps, that certain system requirements are realised.

The project ORKEST[1] aims at such an improvement by applying formal methods. The part of the project described in this report focuses on creating a model of a SPLICE system that can be checked by a model checker in order to verify system-level properties.

The software architecture SPLICE is used by Hollandse Signaalapparaten to build distributed control systems [3]. It forms the core of many of their products, especially in the field of command and control systems. Large streams of data arriving from a number of sensors play a role in several computations, performed by a set of processes running in parallel. SPLICE enables the construction of

[1] In Dutch: Onderzoek naar de Randvoorwaarden voor de Konstruktie van Embedded SysTemen

complex systems based on highly independent application processes that together form a consistent and coherent mechanism. This level of independence clearly has advantages over architectures where more dependency between the processes has been built-in. Most evident is the ease of dynamically reconfiguring and extending the system as well as the availability of a suitable basis for fault-tolerance techniques and the ability to flexibily adapt to externally changing circumstances. Less visible, but a key issue in the context of the project ORKEST, is the potential for reducing the complexity of the design process.

The basic concept for this architecture is a programming model in which processes perform transformations only on shared data. Based on explicit naming of the data elements a subscription paradigm has been constructed that takes care of the necessary actions to manage and distribute all data. This way, the needed communication that is necessary for a coordinated behaviour of the processes can be deduced by the system itself from the subscription paradigm and the involved processes.

In this report it is described how SPLICE can be modeled in the specification language μCRL. Once this μCRL-model is given, properties for SPLICE systems can be checked by means of the available model checkers for this language. To be able to check such properties formally is an important step in being able to improve the design process.

The report is organised as follows. Section 2 outlines the approach to be followed and Section 3 compares this approach to related work. Sections 4, 5, and 6 give overviews of the preliminaries of SPLICE, the underlying network architecture and the language μCRL, respectively. Sections 7 and 8 describe the restrictions that were imposed on SPLICE and ETHERNET, respectively, in order to make modeling feasible. Then, Sections 9 and 10 describe in some detail the μCRL models of SPLICE and ETHERNET. The actual verification is presented in Section 11 and evaluated in Section 12. In conclusion, Section 13 describes the conclusions and pointers to future research. The Appendices I to IV contain the full code of the μCRL specification of ETHERNET, SPLICE, the properties verified and the scenarios .

2. APPROACH

A SPLICE system consists of a number of autonomous *applications*, each of which communicates to the rest of the system by its local *agent* which is connected to the other agents by a *communications network*. These agents offer their clients a coherent view on the system, abstracting from its distributed aspects.

So far, a number of SPLICE descriptions, with varying levels of abstractions, have been proposed [6, 4] but each of these models is focused on aspects different from automated verification. Each of these falls short both in accuracy and in suitability for automated verification.

The model proposed in this report is meant to surpass existing models in the sense that its abstraction level is lower, in order to better deduce SPLICE characteristics while it is, at the same time, high-level enough to make automatic verification feasible. The scope of the model is limited to the data-exchange primitives of SPLICE, i.e. the API calls for publishing, subscribing, reading and writing.

Although a high level of detail is one of the main features of the current model, it is not yet complete. It should be considered as a first approach, that could be refined in future when experiments or more detailed documentation reveal inconsistencies between the SPLICE implementation and its model. For this reason, it is important that the model is well-documented in that 'modeling choices' are explicitly reported.

The language chosen for the model is μCRL[10], a language based on process algebra with data. The motivation for using this language is threefold: (1) the language has a well-defined semantics; (2) there is a tool set available for generation of labelled transition systems which can be processed by a popular model checker as can be found in CÆSAR/ALDÉBARAN; (3) The second author has direct access to a substantial body of expertise on language and tool set.

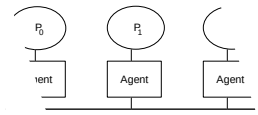

Figure 1: a SPLICE system

3. Related work

The approach taken in this project is to formalise the SPLICE API in a model, such that certain architecture specific properties can be claimed to hold. The chosen model is based on a process algebra with data. Since there are already other formal models that are related to SPLICE we discuss them here.

In [4] SPLICE is modeled by a transition system; it describes the distribution and replication of data across a communication network using the subscription paradigm of SPLICE. The focus of this model is on the basic coordination primitives, which are: `new`, `write`, `read`, and `get`. The precise semantics of these primitives and their compositions is given, where it is assumed that these match with the actual SPLICE implementation.

In [5] the relationship is studied between several different models for a data-oriented coordination language. Three different system architectures are considered: the distributed system where each agent has its own local data space; the centralised ordered system that has a shared data space where the order in which data is produced is preserved; and the centralised unordered system where the order in which data is produced is not necessarily preserved. Furthermore, different structures for the data spaces are taken into account w.r.t. the above mentioned systems. A distinction is made between: a structure where multiplicity of data is significant; a structure where multiplicity of data is insignificant; and a structure where the data is sorted. Different coordination languages, among which is SPLICE, are classified in this spectrum.

The approach most closely related to this project is described in [8] where SPLICE is described in terms of process algebraic laws. This approach is inspired by a process algebra for describing delay-insensitive communications. A set of assumed properties of SPLICE are axiomatised as laws.

The main difference with all these approaches w.r.t. the approach taken in this project is the set of assumptions that is taken, and the method to claim system properties. In our approach the actual implementation of the SPLICE API is formalised in a suitable specification language as precise as possible. From this formalised SPLICE implementation followed the semantics of the SPLICE primitives and their compositions, and consequently the properties that hold for SPLICE applications. No extra assumptions on SPLICE semantics and system-level properties have been used, as is the case with the other approaches. Furthermore, the method of proving properties is model checking, where the other approaches use mathematical structures which are not directly suited to be used in a model checker.

4. An overview of Splice

SPLICE (Subscription Paradigm for the Logical Interconnection of Concurrent Engines) was introduced in [3] as an architecture to support the development of *control systems*. These systems share a number of nonfunctional requirements, such as a certain level of fault-tolerance, real-time constraints, adaptability and distribution over a number of processors that obstruct classical systems development approaches. The SPLICE architecture offers a number of primitives that cover these requirements through a well-defined interface in order to allow system designers and programmers to concentrate on functional requirements. The current section gives a concise overview of the architecture that allows the reader to understand the essentials of the verification presented in this report. For more information on SPLICE the reader is referred to [3, 13, 14, 8, 4].

The architecture offers a number of *agents* that communicate over a *network*. An application

db_change_query	db_initial_wait_status	db_read
db_destroy	db_new	db_synchronize
db_destroy_local	db_new_query	db_term
db_destroy_query	db_new_query_a	db_version
db_destroy_wait	db_new_wait	db_wait_status
db_flags	db_new_wait_q	db_wipe
db_get_statistics	db_post_wait	db_write
db_init		

Figure 2: database-to-SPLICE interface

program running on SPLICE is served by its agent through a variety of API calls, which typically require this agent to interact with other agents over the network, but this interaction is transparent to the application (see Figure 1). To an application designer, SPLICE manifests oneself as one well-organised set of API calls.

The SPLICE architecture is centered around the notion of *shared data*, which is a database each client has access to. The data is typed, or *sorted*, and each client can announce itself as a producer or consumer of data of some sort by means of *publications* and *subscriptions*. These announcements are broadcasted over the network, such that each agent knows the production or consumption sorts of the other agents. The agent of a client that is a publisher of a sort distributes over the network all data written by this client to all clients who are subscribed to this sort. In this way, all read actions by a client on a sort it is subscribed to can be handled locally by its agent. Extra attention is needed when a client subscribes to a sort for which data has already been written by some producer: this is where context data offers a solution.

Context data has a relatively static nature, in that it is not frequently updated. Examples are the

sp__advise	sp__inc_refcount	sp__set_waitset_status
sp__cond_broadcast	sp__lock	sp__sleep
sp__cond_destroy	sp__ltime	sp__smalloc
sp__cond_init	sp__malloc	sp__sort_category
sp__cond_timedwait	sp__mutex_destroy	sp__sort_descriptor
sp__cond_wait	sp__mutex_init	sp__sort_keys
sp__dec_refcount	sp__mutex_lock	sp__sort_length
sp__deref_query	sp__mutex_trylock	sp__sort_name
sp__deref_subscription	sp__mutex_unlock	sp__sort_stroffset
sp__die	sp__protect	sp__spliced_thread_crea
sp__flmtime	sp__protect_wait	sp__srealloc
sp__free	sp__random	sp__thread_create
sp__get_enum_value	sp__realloc	sp__thread_create_p
sp__get_field_info	sp__release_instance	sp__unlock
sp__gtime	sp__restore_protection	sp__unprotect
sp__heap_create	sp__rwlock_destroy	sp__update_waitsets
sp__heap_free	sp__rwlock_init	sp__update_waitsets_q
sp__heap_grow	sp__rwlock_lock_read	sp__verbose
sp__heap_malloc	sp__rwlock_lock_write	sp__warn
sp__heap_realloc	sp__rwlock_unlock	

Figure 3: SPLICE-to-database interface

Resource management		
sp_activate_resource	sp_change_query_v	sp_node
sp_application	sp_children	sp_owner
sp_become_simple_res	sp_children_of_type	sp_release_resource
sp_bind_filter	sp_client	sp_remove_world
sp_bind_query	sp_dbsym	sp_resource
sp_bind_query_a	sp_destroy	sp_set_application_s
sp_bind_query_v	sp_event	sp_shcontainer
sp_bind_sort	sp_free_attr	sp_term_superapi
sp_bind_world	sp_free_ds	sp_terminate
sp_bindp_query	sp_init_superapi	sp_type
sp_bindp_query_a	sp_initialize	sp_use_simple_resource
sp_bindp_query_v	sp_limited_applicatio	sp_use_sort
sp_change_query	sp_list_entities	sp_use_world
sp_change_query_a	sp_new_attr	sp_version
Data definition		
sp_add_world	sp_copy_attr	sp_is_self_contained
sp_attr	sp_getattr	sp_parse_ds
sp_attr_attr	sp_getattr_attr	sp_sort_category
sp_attr_d	sp_getattr_d	sp_sort_descriptor
sp_attr_data	sp_getattr_data	sp_sort_keys
sp_attr_i	sp_getattr_i	sp_sort_length
sp_attr_l	sp_getattr_l	sp_sort_name
sp_attr_n	sp_getattr_n	sp_sort_stroffset
sp_attr_type	sp_getattr_v	sp_sorts
Data manipulation		
sp_flush	sp_subscribe_simple	sp_world_category
sp_get_context	sp_subscribe_to_cate	sp_world_descriptor
sp_publish	sp_wait	sp_world_name
sp_read	sp_waitset	sp_worlds
sp_set_buffer	sp_wipe	sp_write
sp_subscribe	sp_wipe_context	sp_write_frame

Figure 4: SPLICE API interface

identification of a given vehicle, or the location of a radar site at an airport. Consider a new subscriber to a sort belonging to context data. From its producer it might receive no data of this sort for quite a long stretch of time, resulting in a deficiency in its database. Such a deficiency can be reduced by accessing the context database, which resides at clients collecting the data of the context sort and distributing this data to the new clients. It is instructive to see why this problem is less pregnant for non-context data, which is regularly updated: soon after a client issues a new subscription, it will receive fresh data of this sort which renders invalid all existing data of this sort.

The management of context data is more complex than that of other data, but the differences are all confined to the clients: all agents have exactly the same functionality, but some clients issue special SPLICE calls that allow them to maintain context data. To this end, data sorts are grouped into *categories* one of which is designated as *context*. However, this grouping is dynamical, i.e. it is not known in advance to all clients, which makes management of context non-trivial. If a client issues a *sp_subscribe_to_category(context)* call, then this category subscription is broadcasted as usual, and all receiving agents will report back what sorts of context data they are consuming or producing.

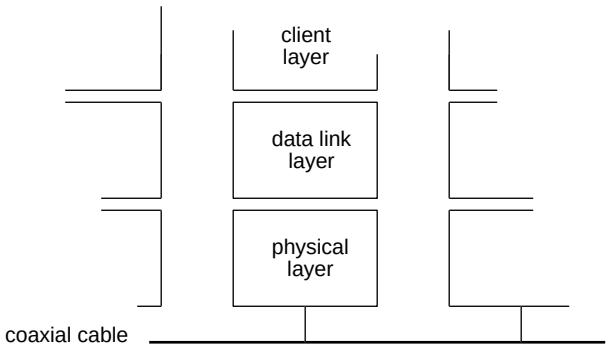

Figure 5: Ethernet layering

Once the agent of the client who subscribed to *context* has a full account of all publications of and subscriptions to context sorts, it will issue a request for data to all publishers of a context sort. After all requests have been answered, the agent has a full copy of all context data in the system, and will be kept up-to-date through the handling of subscriptions that was described above.

The Splice standard [13] includes three interfaces, one of which is available to its 'users', i.e. the Splice application programs. The remaining two emanate from the fact that Splice does not force application programmers into a given database environment. Instead, it facilitates the development of *database modules* that implement the required database functionality. These modules are accessed by the Splice kernel through an interface any module is bound to offer and, conversely, Splice accesses the database modules through a dedicated interface not available to applications.

The Splice API calls can be divided into three groups. First, there is the data definition group, which manages the definition of the data model in terms of sorts and categories. Second, there is the data flow group, which allows one to identify which applications produce and consume which sorts, through publications and subscriptions. Third and final, there is the resource group, which manages the creation, allocation, freeing and destruction of entities such as publications, subscriptions and applications. Just for the sake of giving an idea of the complexity of Splice, the calls from the several categories are summed up in Figures 2, 3, and 4.

Of these three interfaces, the actual API is the one relevant for the current verification project defining Splice as it is visible to its clients. Strictly speaking, the Splice architecture is a layer imposed onto a given database architecture, but in a broader sense Splice includes this database. Therefore, the database-to-Splice and the Splice-to-database interfaces will be left out of this study.

5. An overview of Ethernet

Ethernet was developed at Xerox in the early 1970s as a communication protocol for local area networks [16]. Since then, it has developed into an open standard [15], where it is specified as a *carrier sense multiple access with collision detection*. The current section gives an overview.

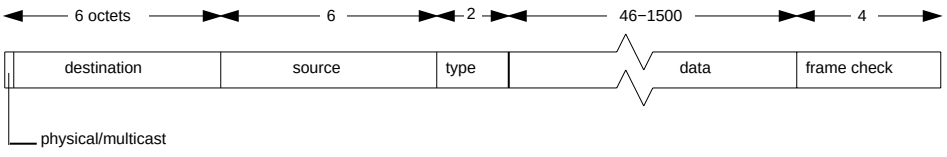

Figure 6: Ethernet frame

The main features of ETHERNET are that it allows communication over a coaxial cable among a number of clients (multiple access) each of which is able to perceive whether a signal is being transmitted over this cable (carrier sense) and whether signals are interfering with each other on the cable (collision detection). This allows a sender to wait until the cable becomes available, to transmit its data and to check whether the transmission was successful. In case of a collision, the sender defers its retransmission for an arbitrary time and tries again, repeating its attempts until a successful retransmission is sensed. Another feature of ETHERNET is its addressing flexibility. It is possible to address one node (pointcast), a group of nodes (multicast) or all nodes (broadcast).

ETHERNET fits in the two lowermost levels of the ISO model. That is, it consists of a *physical layer* and a *data link layer*, and the higher levels of the network architecture are referred to as the *client layer* (see Figure 5). The physical layer deals directly with the coaxial cable, offering services like framing, contention resolution and the passing of bit streams to the data link layer. The data link layer offers services like sending and receiving of ethernet frames to the client layer.

The interface of the physical layer consists of the following elements:

ReceiveBit A function to receive one single bit

TransmitBit A function to transmit one single bit

Wait A function to wait for a specified time interval

collisionDetect A boolean variable read by the data link layer to check whether a collision is occurring

carrierSense A boolean variable read by the data link layer to check whether the coaxial cable is in use

transmitting A boolean variable set to *true* by the data link layer before transmitting the first bit of a frame, and set to *false* after the last bit of a frame has been transmitted.

The interface of the data link layer consists of the following elements:

TransmitFrame A function to send one frame, containing the addresses of source and destination, type and data. The status returned is either *transmitOK* or *excessiveCollisionError* to indicate failure due to repeated collisions

ReceiveFrame A function to receive one frame, containing the addresses of source and destination, type and data. The status returned is either *receiveOK*, *frameCheckError* to indicate that a corrupted frame has been received or *alignmentError* when an incomplete frame has been received.

The *frames* that are sent and received by the datalink layer have a layout that is depicted in Figure 6. Here, source and destination are ethernet addresses, where the destination includes a physical/multicast bit. This indicates whether the address refers to one node, to a group of nodes, or to the group of all nodes as specified by one reserved broadcast address. The type field is not interpreted by the data link layer, and reserved to be used by the client layer, the data field speaks for itself and the frame check is a cyclic redundancy check to guard the integrity of the frame.

In order to send data, a client calls *TransmitFrame* with the relevant parameters. The physical layer waits for the variable *collisionDetect* to become false, sets *transmitting* to *true* and calls *TransmitBit* for each bit of the frame and, subsequently, its computed checksum, and concludes with resetting *transmitting* to *false*. While sending the frame, the data link layer monitors the variable *collisionDetect*. If this variable becomes *true* the transmission is continued until at least 32 more bits have been transmitted, to assure that all other nodes are able to detect this collision, after which transmission is aborted and the data link layer waits some time until a new attempt is made. The delay before the nth retransmission is represented by a uniformly distributed random integer in the range

sort Natural
func 0:→Natural
s:Natural→Natural
map add:Natural×Natural→Natural
mul:Natural×Natural→Natural
var m, n:→ Natural
rew add(0,n)=n
add(s(m),n)=s(add(m,n))
mul(0,n)=0
mul(s(m),n)=add(mul(m,n),n)

Figure 7: An example μCRL data type

$\{1, ..., 2^{min(n,10)}\}$; after 16 attempts the data link layer gives up and returns *excessiveCollisionError* to its client.

In order to receive data, a client calls *ReceiveFrame* and the physical layer waits for the boolean *carrierSense* to become *false*. Then, it repeatedly calls *ReceiveBit* to receive the frame bits, until the frame is complete, indicated by *carrierSense* to become *true*. If the frame is destined to this node, possibly taking into account the physical/multicast bit, it is passed on to the client layer, otherwise it is discarded. However, if a collision is detected during the receiving of the frame, it is also discarded. If the frame received does not contain an integer number of octets, the status *alignmentError* is returned and if the frame does not match the CRC attached then the status *frameCheckError* is returned.

The standard specifies a variety of aspects that are too numerous to include them here. Examples are parameters like *interframe spacing*, *slot time*, and *round trip delay*, and the calculation of CRC values. For these and other details the reader is referred to the standard [15].

6. An overview of μCRL

The specification language μCRL was designed as a language expressive enough to cover real-life systems and featuring a well-defined semantics to facilitate mathematical analysis, supported by software tools. It can be characterised as a language based on process algebra with data.

The current text does not aim at a full-fledged introduction in μCRL. Instead, it offers the reader, familiar with algebraic specifications and basic process algebra, an intuitive grasp of the syntax and semantics of the language. A reader with a weak background in these fields is referred to [1, 2] for introductions in process algebra and algebraic specifications. The syntax and semantics of μCRL are given in [10].

A μCRL specification consists of four parts. A definition of the data types by their signatures and equations, a definition of the actions with, possibly, data parameters, a definition of the processes constructed from these actions and process-algebraic primitives, and finally, a definition of the start-up process, possibly with actions *encapsulated* or *abstracted*, in a process-algebraic sense.

Data types, or *sorts* in μCRL terminology, are specified by *signatures*, defining their language, and *equations*, defining the operations defined on their elements. As an example, Figure 7 defines the sort

act get, $\overline{\text{get}}$, cget: Natural
comm get | $\overline{\text{get}}$ = cget

Figure 8: An example μCRL action definition

procServer(n: Natural) = $\overline{\text{get}}$(n)·Server(s(n))

Figure 9: An example μCRL server process

procClient = sum(n: Natural, get(n) · ◁ eq(n,s(s(s(0)))) ▷ get(n)·Client)

Figure 10: An example μCRL client process

Natural with the operations addition and multiplication.

Actions are defined with zero or more arguments, each of which is defined over some data type. Of special interest are the *communication actions*, which come in pairs: two matching communication actions are said to communicate if they are executed by two processes at exactly the same time. Usually, communicating actions are not allowed to be executed in isolation, that is without their counterpart, and this is the reason why they are deadlocked in the initial process' *encapsulate* clause as will be demonstrated further on. Figure 8 defines two actions *get* and $\overline{get}$, which are meant to rewrite to a third action *cget*. These actions will be used in a simple running example specification of a 'server' that delivers sequence numbers on request.

Processes are constructed from actions by the process-algebraic operators '·', '+' and '||' for sequential composition, non-deterministic choice and parallel composition, respectively. Also, there is a sum operator, offering a non-deterministic choice for one element of a sort and there is an 'if-then-else' construction of the form $then \lhd if \rhd else$. As an example, Figure 9 defines a process for the server mentioned earlier. This process executes a $\overline{get}$ action with its current sequence number and increases its sequence number, executes a $\overline{get}$ for this new sequence number, and so on.

The simple sequence server can be used by a client process that gets sequence numbers from the server until it receives the sequence number $s(s(s(0)))$, after which it terminates (Figure 10) Here, it is essential that the server's $\overline{get}$ and the client's *get* synchronise: the server only delivers a number when requested to do so, and the client only receives a number when the server delivers one. This is effectuated by excluding all $\overline{get}$ and *get* actions that are not synchronised in the *initial process*.

To conclude the running example, Figure 11 shows how the server is started in parallel with two clients. The clients will continuously obtain sequence numbers from the server and the one to first 'draw' a $s(s(s(0)))$ will terminate and the other one will loop forever.

The μCRL language is supported by a tool set featuring a high-performance labelled transition system generator that interfaces with CÆSAR/ALDÉBARAN[9], a tool set which contains several tools for manipulation and analysis of labelled transition systems, and a process simulator. The μCRL tool set can be obtained from [7] and is documented in [11].

7. A LIMITED VERSION OF SPLICE

Section 4 gave some idea of the size and depth of SPLICE. The SPLICE interface is large and under the surface of functionality, the implementation of these calls is far from trivial. Also, the actual architecture has not been laid down in a detailed specification, and some of the details of the architecture are classified. Therefore, it is not realistic to aim at a SPLICE specification in μCRL that covers the original in full detail.

init encap({ $\overline{\text{get}}$, **get** }, Server(0) || Client || Client)

Figure 11: An example μCRL server process

sp_application
sp_init_super_api
sp_publish
sp_read
sp_subscribe
sp_subscribe_to_category
sp_terminate
sp_use_sort
sp_write

Figure 12: The modeled SPLICE calls

The SPLICE model presented in this report is essentially a limited model, which does *not* imply it is a dead end in research. On the contrary, the limits imposed are meant to be pushed back in future and a necessary condition for this to be possible is that these limits are well-documented. This is why attention is paid to what fragment will be modeled and, more important, what fragment will not be modeled.

First, the model concentrates on SPLICE as a *coordination language* which allows a number of clients to coordinate their behaviour by having access to one shared data model. Subordinate aspects, such as the possibility to define *data modules* and *resources* which are used for synchronisation of clients have been left out. This means that the calls modeled are all in the SPLICE API (Figure 4). The selection made is given in Figure 12.

Second, the model is limited to SPLICE in fault-free circumstances. One of the strong points of SPLICE is that it is to some extent resistant to hardware failure and network failures. For instance, as soon as a client appears to be unreachable, it is replicated on a reachable host. This sophisticated behaviour has not been modeled in μCRL, in the first place since it would increase the model's complexity far beyond the currently known limits of verification technology and, in the second place, because the μCRL tool set lacks as yet support for a necessary language construct as dynamic process creation.

Third, the model aims at the *observable behaviour* of SPLICE as opposed to the internal behaviour of the architecture, which consists of a number of daemons interacting through internal protocols. The reason for this limitation is merely pragmatic, as the internal details of SPLICE are not publicly available.

Fourth, SPLICE is embedded in C, which means that the SPLICE applications are implemented in C and interact with the SPLICE system by library routines. In the μCRL model, SPLICE is embedded in μCRL which means that the SPLICE applications are written in μCRL. With the expressiveness of μCRL and the simplicity of the examples to be studied in mind this is no real limitation.

Fifth and final, the temporal aspects of SPLICE have been ignored, because the μCRL tool set does not support the temporal features of the underlying specification language. In some sense, this restriction has more weight than the other four, in that it not just excludes aspects from the model, but it also introduces some artificial features not present in the original. As an example, in reality a message queue does hardly ever overflow, since SPLICE agents will only in erroneous circumstances glut another agent with more messages it can handle. If timing is ignored, however, overflowing queues are very well possible.

8. A LIMITED MODEL OF ETHERNET

With the objective of modeling ETHERNET using μCRL and its tool set, a number of restriction had to be imposed on the model of ETHERNET used. The prime reason for this is that the μCRL tool set provides no support to specifications where time plays a role. A second reason is that the size of the

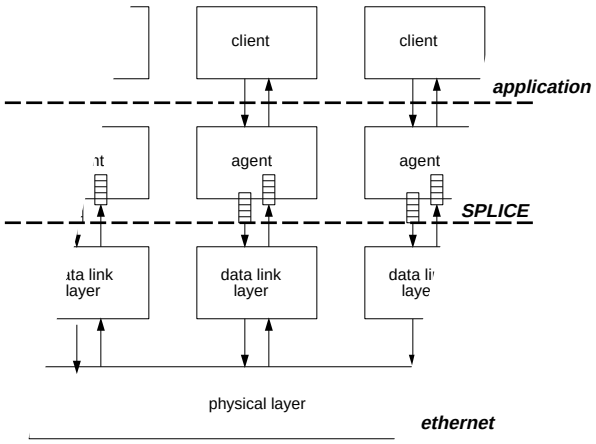

Figure 13: SPLICE model

state space is restricted. Remember that the model of SPLICE was restricted for the same reason (see Section 7).

Timing in ETHERNET is restricted to parameters like *round trip delay*, i.e. the time it takes for a signal to propagate over the coaxial cable and back, and the time a sender waits between two retransmits. Leaving these aspects out results in a model that behaves like the original model with the difference that no facts about timing can be inferred from it. All other aspects are left intact. For instance, the way a sender waits longer and longer between subsequent retransmits guarantees that in practice it is highly unlikely, but not impossible, that a transmit fails due to excessive collisions, so after all, what remains is either success or failure.

In order to keep the size of the state space within manageable proportions, frames will not be transmitted as a stream of individual bits, but as one abstract signal instead. As a consequence, a frame is either completely transmitted or completely lost. Features related to fragments of frames sent over the cable, such as the jamming after a collision and carrier sense, are not modeled. Also, no size limit is imposed on frames.

Since timing aspects are excluded from the model, the use of retransmits is marginal, so retransmits have not been modeled. Other, minor aspects excluded are the exact configuration of nodes attached to the coaxial cable: in reality, signals travel over the cable starting at the sender and propagating in both directions. In the model, signals propagate over the cable along a fixed, predefined route. In practice any number of transmissions can collide on the cable; this has been restricted to two simultaneous transmissions.

Finally, the direction in which signals propagate along the coaxial cable has been fixed. In reality, a signal transmitted from one transceiver will first reach the direct neighbours of its source, then reach the neighbours of these neighbours, and so on. The model imposes a strict order on transceivers and all signals first reach the first transceiver, then the second and so on. It can be easily understood that this restriction simplifies the specification while it has no consequence of any weight for the system's characteristics.

9. A μCRL model of a SPLICE fragment

The μCRL model of SPLICE consists of about 1600 lines of code. In this section, some of the characteristic constructs of the specification will be highlighted in order to give the reader some notion of

the complexity of the code and the μCRL way of modeling. The full μCRL specification is included in Appendix II.

The structure of the model is given in Figure 13. The model consists of three parts, i.e. the SPLICE *clients*, which are applications that use SPLICE, the actual SPLICE *architecture*, which consists of a number of *agents*, and, finally, the ETHERNET through which the agents communicate. Each client is associated with an agent, through which it accesses the architecture and each agent has access to the network by its protocol stack. This section describes the μCRL model of agents and clients; the next section describes the model of the ETHERNET.

9.1 the database model

The architecture is not dependent on one specific database structure; instead it allows for all kind of databases to be 'plugged in'. The μCRL model, however, is restricted to one simple kind of database which features a number of essential properties.

A database in the model is a *set* of typed, or *sorted*, *records*. Each record has a *key* associated with it which identifies it within its sort: if a record of a sort is added to the database while a record of the same sort with the same key is already existent, then the existing record is overwritten.

In order to be able to order values in a natural way, an index function i is defined which maps each value onto a natural number. Now, the ordering of values can be simply expressed in terms of the standard order of the naturals. In μCRL the definition of values looks like the following.

sort Value
func Value0, Value1, Value2, Value3, Value4:→Value
map i:Value → Natural
eq: Value × Value → Bool
lt: Value × Value → Bool
gt: Value × Value → Bool
var v1, v2: Value
rew i(Value0)=0
i(Value1)=s(i(Value0))
i(Value2)=s(i(Value1))
i(Value3)=s(i(Value2))
i(Value4)=s(i(Value3))
eq(v1,v2)=eq(i(v1),i(v2))
lt(v1,v2)=lt(i(v1),i(v2))
gt(v1,v2)=gt(i(v1),i(v2))

The structure of records remains abstract in that all that is specified is that there are a number of records for which a key is defined which is some kind of value. Note that the ordering of records is based on their key value alone, which is essential for the definition of data sets which will follow.

sort Record
func Record0, Record1, Record2, Record3, Record4:→ Record
map i:Record→ Natural
eq: Record × Record→ Bool
lt: Record × Record→ Bool
gt: Record × Record→ Bool
key:Record→ Value
var v1, v2: Record
rew i(Record0)=0
i(Record1)=s(i(Record0))
i(Record2)=s(i(Record1))

```
i(Record3)=s(i(Record2))
i(Record4)=s(i(Record3))
eq(v1,v2)=eq(i(v1),i(v2))
lt(v1,v2)=lt(key(v1),key(v2))
gt(v1,v2)=gt(key(v1),key(v2))
key(Record0)=Value0
key(Record1)=Value0
key(Record2)=Value0
key(Record3)=Value1
key(Record4)=Value1
```

Data consists of a record with a sort, and sorts are just constants. The μCRL language constructs for these have been explained by now, so the explicit definitions will not be written out here. The specification of a set of data is more interesting.

A set will be represented as a list of data, and in order to make sure that the order in which data is added to the set is irrelevant, the list is kept ordered. Also, it is defined that a set cannot contain two records of the same sort and the same key. The following fragment only shows the definition of adding data, but the full specification contains more set-theoretic operators.

```
sort  DataSet
func  vnil: → DataSet
      cons:Data × DataSet → DataSet
map   add: DataSet × Data → DataSet
var   v1, v2: Data
      s1, s2: DataSet
rew   eq(vnil,vnil)=T
      eq(vnil,cons(v2,s2))=F
      eq(cons(v1,s1),vnil)=F
      eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
      add(vnil,v1)=cons(v1,vnil)
      add(cons(v1,s1),v2)=if(lt(v1,v2),
        cons(v1,add(s1,v2)),
        if(lt(v2,v1),
          cons(v2,cons(v1,s1)),
          cons(v2,s1)))
```

A query is defined as a combination of a *qualifier* which defines what records are to be retrieved and a quantifier, which defines how may records are to be retrieved. In SPLICE both of these can be complex, but in the model a qualifier is a predicate over records and a quantifier is either *any* or *each*, meaning that one or all records that satisfy the qualifier are to be retrieved.

```
func select:DataSet × Query → DataSet
var   v1, v2: Data
      s1, s2: DataSet
      q1,q2: Qualifier
      n1,n2: Quantifier
rew   select(vnil,Query(n1,q1))=vnil
      select(cons(d1,s1),Query(each,q1))=if(satisfies(q1,d1),
      cons(d1,select(s1,Query(each,q1))),
      select(s1,Query(each,q1)))
      select(cons(d1,s1),Query(any,q1))=if(satisfies(q1,d1),
```

cons(d1,vnil),
select(s1,Query(any,q1)))

The specification of the database model has been presented in some detail, as it also serves as an example data type specification. The following subsections are shorter on specifications; for the complete specification the reader is referred to Appendix II

9.2 The queues

A SPLICE agent communicates over the network, but it has no direct access. Instead all frames sent to and received from the network are queued in *bounded queues*, where a bounded queue is a standard queue with a maximal length, defined in such a way that additions beyond the maximum are just ignored. Of course, the only correct way to use a bounded queue is to check its length before adding it, which is why a test whether a queue is full or not is to be specified. The data types for queues and bounded queues are specified by the following.

sort frameQueue
func qnil: → frameQueue
cons:EthernetFrame × frameQueue → frameQueue
bound:frameQueue × Natural → frameQueue
map add: frameQueue × EthernetFrame → frameQueue
add: frameQueue × EthernetFrame × Natural → frameQueue
tail:frameQueue → frameQueue
head:frameQueue → EthernetFrame
empty:frameQueue → Bool
full:frameQueue → Bool
length:frameQueue → Natural
var p1,p2:EthernetFrame
q1,q2:frameQueue
m1,m2:Natural
rew add(qnil,p2)=cons(p2,qnil)
add(cons(p1,q1),p2)=cons(p1,add(q1,p2))
head(cons(p1,q1))=p1
tail(cons(p1,q1))=q1
empty(qnil)=T
empty(cons(p1,q1))=F
length(qnil)=0
length(cons(p1,q1))=s(length(q1))
add(bound(q1,m1),p2)=bound(add(q1,p2,m1),m1)
add(q1,p2,0)=q1
add(qnil,p2,s(m1))=cons(p2,qnil)
add(cons(p1,q1),p2,s(m1))=cons(p1,add(q1,p2,m1))
head(bound(q1,m1))=head(q1)
tail(bound(q1,m1))=bound(tail(q1),m1)
eq(bound(q1,m1),bound(q2,m2))=and(eq(q1,q2),eq(m1,m2))
empty(bound(q1,m1))=empty(q1)
full(bound(q1,m1))=eq(length(q1),m1)

A typical example of the usage of the queue is the processing of an *sp_application* system call, which is called by a SPLICE application to announce its existence to the rest of the system. In normal situations this results in an ethernet frame being added to the queue, but if the queue is full the agent fails. This may sound a bit drastically, but nevertheless it is a natural way of modeling: the SPLICE implementation signals a "segmentation violation: core dumped" in this case.

```
sp_application_(agentId).
        (panic.delta
          ◁ full(outQueue) ▷
        Agent(agentAddress,
          add(configuration,Present(agentId,agentAddress)),
          agentId,
          inQueue,
          add(outQueue,Frame(one,agentAddress,enil,Presence,add(nil,Present(agentId,agentAddress)))),
          pendingCategory,
          pendingPublications,
          c,
          p))
```

This fragment specifies that if the agent handles an *sp_application* call it first checks whether its output queue is full. If so, it executes a *panic* after which it just deadlocks by *delta*. If not, it continues itself by calling the Agent process with most of the parameters unchanged, except for its output queue, to which one frame is added.

Instead of handling system calls, an agent may also send over the network frames from its output queue or receive frames from the network into its input queue. All this is scheduled non-deterministically, in the sense that the agent chooses non-deterministically between the available options at any moment, with no 'intelligent' scheduling algorithm. So, if there is a choice between handling a system call and receiving a network frame while the output queue is full, the agent could choose to handle the call, leading to a *panic* due to queue overflow, while delaying this handling could have resulted in an output queue with room for frames.

To conclude this subsection, the μCRL code for the actual sending of queued messages is presented. This fragment states that for any frame, or actually any combination of frame fields, it is checked whether this frame is in front of the queue; if so, this frame is transmitted, after which it is checked whether this transmit was successful. If so, the frame is removed from the queue, or otherwise, the agent panics.

```
sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(t:EthernetType,
sum(r:Set,
    ed_transmitFrame(agentAddress,b,s,d,t,r).
        sum(e:EtherReturnCode,
            ed_return(agentAddress,e).
            (panic.delta
                ◁ not(eq(e,transmitOk)) ▷
            Agent(agentAddress,configuration,agentId,inQueue,tail(outQueue),
                    pendingCategory,pendingPublications, c,p)))
            ◁ and(not(empty(outQueue)), eq(head(outQueue),Frame(b,s,d,t,r))) ▷
        delta)))))
```

This is another diversion from the SPLICE implementation which will not just panic on a transmission error but, instead, conclude that some other client has 'died' and tries to reanimate this client on a host which is known to be up and running. The current model does not cover these aspects of SPLICE so all it can do is just panic.

9.3 A sample client

The model aims at an embedding of SPLICE within μCRL, which means that the application programs are to be written in μCRL. As an example, the following fragment specifies a client which writes a record, reads a record, and then terminates.

```
proc P0(id: AppId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sp_publish(id,Publish(id,Sort1)).
        (sp_write(id,Datum(Sort1,Record0))+sp_write(id,Datum(Sort1,Record1))).
        sum(v:Set, sp_read(id,Bind(Subscribe(id,Sort1),Query(any,qualifier1)),v ,SP_RD_MARK)).
        sp_terminate(id).
        delta
```

10. A μCRL model of ETHERNET

In μCRL, ETHERNET is specified as one process for the coaxial cable and the physical layers of all nodes attached, and one process per node for its data link layer. This section describes the most crucial parts of the specification, leaving most details to the interested reader. The full μCRL specification is included in Appendix I.

The physical layer process the parameters: the set of nodes attached, the signal that is currently being propagated over the cable, the node the signal is propagated from, and the set of nodes the signal still has to propagate to. In order to be able to refer to nodes at the physical level, a notion of *transceiver id* is introduced that is somewhat artificial in the sense that in reality, nodes are distinguished by their physically being different.

The definitions of data types include a sort *TransceiverId* which defines a number of constants, and a sort *TransceiverIdSet* which defines finite sets of these; definitions can be found in Appendix I. The process definition starts as follows.

```
proc PhysicalLayer(nodes : TransceiverIdSet,
                   toPropagate: EthernetFrame,
                   propagateFrom: TransceiverId,
                   propagateTo: TransceiverIdSet) =
```

Depending on its actual parameters, the process has a number of options. First, if there is no signal on the cable, it can execute a *transmitSignal* call from a data link layer, after which the frame is put on the cable, to be propagated from the note that issued the call to all other nodes.

```
sum(t:TransceiverId,
sum(f:EthernetFrame,
        ep_transmitSignal_(t,f)·
        PhysicalLayer(nodes,f,t,delete(nodes,t))))
          ◁ eq(propagateFrom,noTransceiverId) ▷ delta
          +
```

Second, if there is a signal being propagated over the cable and some data link layer issues a *transmitSignal* then a collision occurs and all data links that have issued a *transmitSignal* earlier, receive an *excessiveCollisionError*. After this collision, the cable is idle again.

```
sum(f:EthernetFrame,
```

```
sum(t:TransceiverId,
      ep_transmitSignal_(t,f)·
      EtherReturnLoop(difference(nodes,propagateTo),alignmentError)·
      ep_return_(t, excessiveCollisionError)))·
      PhysicalLayer(nodes, noFrame, noTransceiverId, tnil)
        ◁ not(eq(propagateTo,tnil)) ▷ delta
          +
```

Third, if a signal has been propagated successfully over the cable, leaving no nodes to propagate to, the node the signal was propagated from receives a *transmitOk* return status and all other nodes receive a *receiveOk* status.

```
EtherReturnLoop(delete(nodes,propagateFrom),receiveOk)·
ep_return_(propagateFrom, transmitOk)·
PhysicalLayer(nodes,noFrame,noTransceiverId,tnil)
      ◁ and(eq(propagateTo,tnil), not(eq(propagateFrom,noTransceiverId))) ▷ delta
      +
```

Fourth and finally, if a signal on the cable is successfully received by a node then the corresponding transceiver id is removed from the set of nodes to propagate to. Here, the function *getOne* deterministically returns one element from the set of nodes to propagate to. As a consequence, the order in which nodes are reached is fixed (this corresponds to the limitation imposed on the ETHERNET model that was described in Section 8).

```
ep_receiveSignal_(getOne(propagateTo),toPropagate)·
PhysicalLayer(nodes,toPropagate,propagateFrom,deleteOne(propagateTo))
◁ not(eq(propagateTo,tnil)) ▷ delta
```

Returning a status to a set of nodes is done by a simple utility process *EtherReturnLoop*.

```
proc EtherReturnLoop(returnTo:TransceiverIdSet, toReturn:EtherReturnCode)=

ep_return_(getOne(returnTo),toReturn)·
EtherReturnLoop(deleteOne(returnTo),toReturn)
◁ not(eq(returnTo,tnil)) ▷
nop
```

The datalink process has two parameters that remain constant throughout the process execution: the *transceiver id* and the *ethernet address*.

```
proc DatalinkLayer(transceiver: TransceiverId,
                   node : EthernetAddress)=
```

ETHERNET frames are specified as consisting of the physical/multicast bit, the source, which is an ethernet address, the destination, which is a set of addresses, a type and the data in a set. This is a slight diversion from reality, where the destination of a frame is one single 'high level' address, recognised by its physical/multicast bit being equal to 1, and interpreted as a set of addresses.

The data link layer either handles a *transmitFrame* or it handles a *receiveFrame*. In the first case, the frame is transformed into a bit stream and handed down to the physical layer, after which the status returned is translated to the client layer.

```
sum(b:Bit,
sum(d:EthernetAddressSet,
sum(t:EthernetType,
sum(v:Set,
      ed_transmitFrame_(b,node,d,t,v)·
      ep_transmitSignal(transceiver,Frame(b,node,d,t,v)))·
      sum(c:EtherReturnCode,
       ep_return(transceiver,c)·
       ed_return_(node,c))·
       DatalinkLayer(transceiver,node))))
       +
```

The handling of a *receiveFrame* resembles the handling of a *transmitFrame*, the main difference being that it is here that the destination of a frame matters. A received frame is only passed upward to the client layer if its physical/multicast bit equals 0 and the destination is a set containing the current ethernet address as its one and only element, or the bit equals 1 and the destination is empty, which is interpreted as a broadcast, or the bit equals one and the current address is contained in the destination.

```
sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(t:EthernetType,
sum(v:Set,
    ep_receiveSignal(transceiver,Frame(b,s,d,t,v))·
    ed_receiveFrame_(b,s,d,t,v)·
    sum(c:EtherReturnCode, ep_return(transceiver,c)· ed_return_(node,c))·
    DatalinkLayer(transceiver,node)
        ◁ and(not(eq(s,node)),
          or(and(eq(b,zero),
          eq(d,add(enil,node))),
          or(and(eq(b,one),
          eq(d,enil)),
          contains(d,node)))) ▷
    ep_receiveSignal(transceiver,Frame(b,s,d,t,v))·
    sum(c:EtherReturnCode, ep_return(transceiver,c))·
    DatalinkLayer(transceiver,node))))))
```

Typically, the ethernet process is started by putting one physicalLayer in parallel with a number of DataLinkLayer processes:

```
init encap ({ed_transmitFrame_ , ed_transmitFrame,
             ed_receiveFrame_ , ed_receiveFrame,
             ep_transmitSignal_ , ep_transmitSignal,
             ep_receiveSignal_ , ep_receiveSignal,
             ed_return_ , ed_return,
             ep_return_ , ep_return },

             PhysicalLayer(add(add(tnil,transceiver0),transceiver1),
                           noFrame,
```

```
                    noTransceiverId,
                    tnil)
  || DatalinkLayer(transceiver0,address0)
  || DatalinkLayer(transceiver1,address1)
```

11. The verification of SPLICE

The model from the previous sections has been constructed with automated verification in mind. The current section describes how this automated verification has been approached, what properties of the architecture were verified, and what the results of this verification are.

11.1 Tools and techniques

The verification of the μCRL model of SPLICE is done by *model checking*; it is done in two steps. First, a labeled transition system was generated using the μCRL tool set developed at CWI [12]. Second, this system is used as a Kripke structure in which the properties to be verified are to be interpreted, and this was done using the CÆSAR/ALDÉBARAN tool set, consisting of a number of utilities for generation, manipulation and visualisation of transition systems, among which the model checker EVALUATOR.

11.2 Some properties of SPLICE

The verification of SPLICE was not initiated with a specific property to be verified in mind. The approach taken was meant to investigate what properties can be guaranteed to hold.

Preferably, what had to be verified is the *architecture* SPLICE. That is, any constellation of agents, independent from any set of clients built onto it. In other words, the question ideally asked is what properties hold for all configurations of all clients. Although in principle it is possible to automatically verify properties independently from the clients, the size of the total transition system generated renders this type of verification unfeasible.

Another option is to focus on one specific *application* for SPLICE, such as a radar system, consisting of several radar installations. However, the complexity of any realistic example goes beyond the practical limit as imposed by the μCRL tool set, and stepping back to a simple 'toy example' was deemed unsatisfactory.

The final choice actually made was to focus on 'scenario's', i.e. typical combinations of actions found in any application, such as one process reading and one writing, or two processes writing. On the one hand, verification is feasible in these simple situations while, on the other hand, properties verified for these scenario's have some relevance for more complex applications containing these.

The set of properties to verify is to remain the same over all scenario's. So, while these properties may be refined and extended when new scenario's are developed, the final goal is to end up with one set of properties which hold for all scenario's.

The first property is a universal one: deadlock freeness. This does not hold generally, since any SPLICE system could crash due to certain exceptional circumstances as a queue overflow (see Section 9). Instead, the property to hold is *conditional deadlock freeness*: in fault-free circumstances, where no *panic* occurs and the system has not terminated, the system is deadlock free. In the EVALUATOR syntax this is specified like the following.

```
[(not ("panic" or "sp_terminate(*)"))*] <true> true
```

This should be read as follows: after all paths consisting of actions other than *panic* or *sp_terminate* it is possible to execute some arbitrary action <true> after which *true* holds.

The second property is that of soundness, which states that all that is read has also been written. The EVALUATOR code is given below.

```
[(not "csp_write(appl1,Datum(Sort1,Record1))")*.
 "csp_read(appl2,
          Bind(Subscribe(appl2,Sort1), Query(each,qualifier1)),
```

```
                unite(cons(Datum(Sort1,Record1),vnil),pnil,snil,prnil,rnil,cnil),SP_RD_MARK)"]
  false
```

This code says that in all paths containing no write actions for record *Record1* all read actions for this record result in a state where *false* holds or, equivalently, as long as *Record1* has not been written it also cannot be read.

This property reveals a shortcoming of the use of EVALUATOR as a back end for μCRL: the property is formulated for one specific record, while it is to be verified for all records. Proper quantification over data types, however, is not possible, although in this case the problem could be solved with a simple pattern-matching construct. However, a more complex property, such as "if a set of records is read then all elements have been written" is definitely *not* one of the possibilities. This is neither a deficiency of μCRL or of EVALUATOR but it arises merely from the combination of two otherwise unrelated formalisms.

The third property is that of completeness, which means that all that is written can also be read, but this does not hold generally. Due to the fact that it takes time for data to travel from the sending agent to the receiving agent it can never be guaranteed that data written by one agent can be read by another agent at some moment. It cannot even be guaranteed that data written by an agent can be read by this same agent, since it might very well be the case that this record is overwritten by a record written earlier by another agent. What remains is a very weak formulation of completeness.

The weak completeness used in this verification states that for any record written, until it is overwritten by a record with the same sort and key it holds that there are not only failing read actions. As with soundness, the formulation of this property in the EVALUATOR would benefit from a μCRL-aware model checker, so the property should be written out for all combinations of records. The following shows only one combination.

```
[(not "panic")*
 . "csp_write(appl1,Datum(Sort1,Record0))"
 . (not ("csp_write(appl1,Datum(Sort1,Record1))" or "panic" or "sp_terminate(*)"))*]
<(not ("panic" or "sp_terminate(*)"))*
  . "csp_read(appl2,
              Bind(Subscribe(appl2,Sort1), Query(each,qualifier1)),
              *,
              SP_RD_MARK)">
  true
implies
<(not ("panic" or "sp_terminate(*)"))*
  . "csp_read(appl2,
              Bind(Subscribe(appl2,Sort1),Query(each,qualifier1)),
              unite(cons(Datum(Sort1,Record0),vnil),pnil,snil,prnil,rnil,cnil),
              SP_RD_MARK)">
  true
```

The set of properties verified is given once more in Appendix III.

11.3 Results

The verification has been successfully performed for a total of 6 scenario's, each consisting of two SPLICE clients. For any system consisting of three or more clients the transition system contains more than 10^7 states, which exceeds the practical time limits of the hardware[2] used. As a consequence, scenario's with context data, consisting of two clients and one client that maintains the context database, have not been successfully verified.

[2] Experiments were performed on a 300 MHz MIPS R12000 Processor

scenario	# states generated	#states reduced	# CPU time generation
1	846360	961	6h42m
2	554707	702	4h20m
3	474394	363	3h27m
4	477392	463	3h30m
5	4561900	3789	37h38m
6	4458013	2471	32h03m

Figure 14: Verification results

The results of the verification are summarised in Figure 14. Here, for each of the scenario's the size of the transition system generated is given, the size of the system after reduction modulo weak bisimulation and the time it took to generate the system. Two points are worth noting here.

First, there is a huge difference between the transition system generated and the transition system after reduction modulo weak bisimulation. This reduction means that all internal, i.e. non-observable actions like transmissions over the network, are abstracted away. The interpretation of this difference is that the complexity of SPLICE is primarily internal: for the external observer who just sees the clients of the architecture, the system behaves in a relatively simple way. However, as a consequence it is quite expensive to generate relatively simple models.

Second, the size of the transition system generated quickly increases with the complexity of the clients. This imposes a strict limit on the use of the techniques applied here, since it appears to be not feasible to define clients that even approach practical applications.

For each of these scenario's, the properties formulated in Section 11.2 have been proven. In other words, the scenario's provide evidence that these properties hold for the architecture SPLICE. With the techniques presented here it is very well possible to develop and model-check more scenario's in order to find more evidence for proof, or maybe a counter-example for which the properties do not hold.

12. EVALUATION

The current verification study revealed a number of problems of varied nature. The first problem that arose was that of modeling SPLICE, since specifications of the architecture are virtually non-existent. The documents available are [13] and [14], with the former being the most elaborate, summing up the SPLICE calls and their parameters, without describing their intended meaning in much detail. All details about the architecture were to be obtained from the SPLICE designer and programmer, in elaborate email exchanges. However, in practice, unambiguous, complete and well-written specifications are exceptional, so this first problem is not specific for SPLICE or for the technology used: it is a general problem.

The second problem is that of the formulation of the properties to be verified. The aim of the project was to model SPLICE and define and prove its properties. However, also in real-life applications, the question of what makes a system correct is not trivially answered and requires some 'requirement engineering', so it can be defended that this second problem is mirrored in practice.

The third problem is technical, in that it became imminent early that even simple SPLICE systems yield huge transition systems, with several millions of states. It has not been possible to generate transition systems for more than two SPLICE clients which implies that interesting applications fall out of reach. This is a classical problem in the field of verification and a good deal of effort has already been put into reduction techniques; the SPLICE verification once more stresses the importance of these.

The fourth problem is that of validation, i.e. checking whether the μCRL model is a correct model

of the relevant aspects of SPLICE, keeping in mind the fact that SPLICE is poorly documented and the most accurate specification is its C implementation, which is not publicly available, the only conceivable way to validate the model is to have someone with a detailed knowledge of this code study the μCRL code and compare the two. However, given the complexity of the model this approach is not realistic. So far, the only support for the validity of the model comes from the verification, which provides evidence that some properties which are intuitively known to hold for SPLICE also hold for the μCRL model, but this support does *not* apply to the management of context data, since any non-trivial SPLICE system with context consists of more than two clients, and it appeared to be impossible to generate a transition system for a SPLICE system of this complexity. Therefore, the question whether the system verified in this study is a correct model of the SPLICE implementation remains open.

13. CONCLUSIONS

The μCRL tool set is not yet ready for real-world applications like SPLICE systems. The language μCRL is largely adequate for expressing the functionality of SPLICE but the complexity of SPLICE makes the explicit enumeration of the labelled transition system problematic. Although there is no hard evidence for this, the conclusions of current verification might be extrapolated over the state-of-the-art of verification technology.

Earlier SPLICE studies [6] have not drawn this conclusion, but this is not to say that these have somehow avoided the difficulties signaled here. The focus of [6] was to gain insight in the different coordination mechanisms available in coordination languages, one of which is SPLICE. No attention was paid to the the details of the architecture, such as the communications network, so the model constructed is more abstract than the model presented in the current report. Also, these models were aimed at human theorem proving as opposed to full automatic verification.

Frustrating as it may seem, the conclusion that the technology used in this study is not ripe for systems of the complexity of SPLICE defines a good challenge for future research. There are a number of lines of research which can all contribute to pushing the boundary of state-of-the-art technology. An almost trivial approach is a more efficient implementation of well-known algorithms. For instance, the hardware used for this project features no less than 23 processors, only one of which can be used for state space generation software. A parallel implementation might stretch the reach of these tools with orders of magnitude. Another option is the research about efficient reduction strategies for transition systems that aim at symbolically transforming specifications into equivalent specifications that yield much smaller transition systems.

To sum up, the results of this project are: (1) a detailed formal model of SPLICE that in principle facilitates automatic verification; (2) a candidate theory for properties that are guaranteed to hold in SPLICE; (3) a reconnaissance of the frontier of state-of-the-art technology.

ACKNOWLEDGEMENTS

We would like to thank Erik Boasson from Hollandse Signaalapparaten B.V. for his willingness to explain and discuss the details of SPLICE, and Rashindra Manniesing and Arno Wouters for their comments on this text.

References

1. Baeten, J. C. M., and Weijland, J. P. *Process algebra*, vol. 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
2. Bergstra, J. A., Heering, J., and Klint, P., Eds. *Algebraic specification*. ACM Press frontier series. Addison-Wesley, 1989.
3. Boasson, M. Control systems software. *IEEE Transactions on Automatic Control 38*, 7 (July 1993), 1094–1106.
4. Bonsangue, M., Kok, J., Boasson, M., and de Jong, E. A Software Architecture for Distributed Control Systems and its Transition System Semantics. In *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98)* (Atlanta, Georgia, USA, February 27 – March 1 1998), J. Carroll, G. Lamont, D. Oppenheim, K. George, and B. Bryant, Eds., ACM press, pp. 159 – 168.
5. Bonsangue, M., Kok, J., and Zavattaro, G. Comparing Coordination Models based on Shared Distributed Replicated Data. In *Proc. of the 1999 ACM Symposium on Applied Computing (SAC'99)* (1999), ACM Press.
6. Bonsangue, M. M., Kok, J. N., and Zavattaro, G. Comparing software architectures for coordination languages. In *Proceedings of Coordination '99* (1999), P. Ciancarini and A. L. Wolf, Eds., vol. 1594 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 150–165.
7. CWI. *The μCRL home page*. http://www.cwi.nl/∼mcrl/mutool.html.
8. Dechering, P., Groenboom, R., de Jong, E., and Udding, J. Formalization of a Software Architecture for Embedded Systems: a Process Algebra for Splice. In *Proceedings of the Hawaiian International Conference on System Sciences (HICSS-32)* (Maui, Hawaii, January 5-8 1999), IEEE Computer Society.
9. Fernandez, J.-C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., and Sighireanu, M. Cadp (Cæsar/Aldébaran development package): A protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)* (Aug. 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *LNCS*, Springer Verlag, pp. 437–440.
10. Groote, J. F. The syntax and semantics of timed μCRL. Tech. Rep. SEN-R9709, CWI, June 1997. Available from `http://www.cwi.nl`.

11. Groote, J. F., and Dams, D. Specification and implementation of components of a μCRL toolbox. Tech. Rep. 152, Utrecht University, Department of Philosophy, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands, December 1995.

12. Groote, J. F., and Lisser, B. *Tutorial and reference guide for the μCRL toolset verson 1.0.* CWI, Feb. 1999. Available from `ftp://ftp.cwi.nl/pub/bertl/tutorial.ps`.

13. Hollandse Signaalapparaten B.V. *Splice Reference Manual*, May 1999.

14. Hollandse Signaalapparaten B.V. *Splice User Manual*, May 1999.

15. Institute of Electrical and Electronics Engineers. *802.3: Local and metropolitan area networks; Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, 1998 ed. http://standards.ieee.org, 1998.

16. Metcalfe, R. M., and Boggs, D. R. Ethernet: distributed packet switching for local computer networks. *Commun. ACM 19*, 7 (July 1976), 395–404.

Appendix I
The μCRL specification of ETHERNET

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Specification: SPLICE (Ethernet fragment)
% Version      : 1.9
% Date         : 12 October 1999
% Description  : A specification of SPLICE with distributed
%                data space based on 'Splice User Manual'
%                and 'Splice Reference Manual' with context
% Author       : Izak van Langevelde
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ethernet                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%
% Ethernet datatypes
%%%%%%%%%%%%%%%%%%%%%

% Datatype EtherReturnCode

sort EtherReturnCode
func transmitOk,
     excessiveCollisionError,
     receiveOk,
     frameCheckError,
     alignmentError:->EtherReturnCode
map  i:EtherReturnCode->Natural
     eq:EtherReturnCode#EtherReturnCode->Bool
var  e1,e2: EtherReturnCode
rew  i(transmitOk)=0
     i(receiveOk)=s(i(transmitOk))
     i(excessiveCollisionError)=s(i(receiveOk))
     i(frameCheckError)=s(i(excessiveCollisionError))
     i(alignmentError)=s(i(frameCheckError))
     eq(e1,e2)=eq(i(e1),i(e2))


% Datatype TransceiverId

sort TransceiverId
func noTransceiverId, transceiver0, transceiver1, transceiver2,
     transceiver3, transceiver4, transceiver5, transceiver6:->TransceiverId
map  i:TransceiverId->Natural
     eq: TransceiverId#TransceiverId->Bool
     lt: TransceiverId#TransceiverId->Bool
     gt: TransceiverId#TransceiverId->Bool
var  id1, id2: TransceiverId
rew  i(noTransceiverId)=0
     i(transceiver0)=s(i(noTransceiverId))
     i(transceiver1)=s(i(transceiver0))
     i(transceiver2)=s(i(transceiver1))
     i(transceiver3)=s(i(transceiver2))
     i(transceiver4)=s(i(transceiver3))
     i(transceiver5)=s(i(transceiver4))
     i(transceiver6)=s(i(transceiver5))
     eq(id1,id2)=eq(i(id1),i(id2))
     lt(id1,id2)=lt(i(id1),i(id2))
     gt(id1,id2)=gt(i(id1),i(id2))


% Datatype TransceiverIdSet

sort TransceiverIdSet
func tnil:->TransceiverIdSet
     cons:TransceiverId#TransceiverIdSet->TransceiverIdSet
map  add: TransceiverIdSet#TransceiverId->TransceiverIdSet
     union: TransceiverIdSet#TransceiverIdSet->TransceiverIdSet
     difference: TransceiverIdSet#TransceiverIdSet->TransceiverIdSet
     eq:TransceiverIdSet#TransceiverIdSet->Bool
     if:Bool#TransceiverIdSet#TransceiverIdSet->TransceiverIdSet
     delete:TransceiverIdSet#TransceiverId->TransceiverIdSet
     contains:TransceiverIdSet#TransceiverId->Bool
     getOne:TransceiverIdSet->TransceiverId
     deleteOne:TransceiverIdSet->TransceiverIdSet
var  v1, v2: TransceiverId
     s1, s2: TransceiverIdSet
rew  eq(tnil,tnil)=T
     eq(tnil,cons(v2,s2))=F
     eq(cons(v1,s1),tnil)=F
     eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
     add(tnil,v1)=cons(v1,tnil)
     add(cons(v1,s1),v2)=if(lt(v1,v2),
                            cons(v1,add(s1,v2)),
                            if(lt(v2,v1),cons(v2,cons(v1,s1)),cons(v1,s1)))
     if(T,s1,s2)=s1
     if(F,s1,s2)=s2
     delete(tnil,v2)=tnil
     delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
     contains(tnil,v2)=F
     contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
     union(tnil,s2)=s2
     union(cons(v1,s1),s2)=union(s1,add(s2,v1))
     difference(s1,tnil)=s1
     difference(s1,cons(v2,s2))=delete(difference(s1,s2),v2)
     getOne(cons(v1,s1))=v1
     deleteOne(cons(v1,s1))=s1


% Datatype EthernetType

sort EthernetType
func noType, PublishSubscribe, Data, Presence, Request:->EthernetType
map  eq:EthernetType#EthernetType->Bool
     i:EthernetType->Natural
var  t1, t2: EthernetType
rew  i(noType)=0
     i(PublishSubscribe)=s(i(noType))
     i(Data)=s(i(PublishSubscribe))
     i(Presence)=s(i(Data))
     i(Request)=s(i(Presence))
     eq(t1,t2)=eq(i(t1),i(t2))


% Datatype Ethernet Address

sort EthernetAddress
func noAddress, address0, address1, address2,
     address3, address4, address5, address6:->EthernetAddress
map  i:EthernetAddress->Natural
     eq: EthernetAddress#EthernetAddress->Bool
     lt: EthernetAddress#EthernetAddress->Bool
     gt: EthernetAddress#EthernetAddress->Bool
     if:Bool#EthernetAddress#EthernetAddress->EthernetAddress
var  id1, id2: EthernetAddress
rew  i(noAddress)=0
     i(address0)=s(i(noAddress))
     i(address1)=s(i(address0))
     i(address2)=s(i(address1))
     i(address3)=s(i(address2))
     i(address4)=s(i(address3))
     i(address5)=s(i(address4))
     i(address6)=s(i(address5))
     eq(id1,id2)=eq(i(id1),i(id2))
     lt(id1,id2)=lt(i(id1),i(id2))
     gt(id1,id2)=gt(i(id1),i(id2))
     if(T,id1,id2)=id1
     if(F,id1,id2)=id2
```

```
% Datatype EthernetAddressSet

sort EthernetAddressSet
func enil:->EthernetAddressSet
     cons:EthernetAddress#EthernetAddressSet->EthernetAddressSet
map  add: EthernetAddressSet#EthernetAddress->EthernetAddressSet
     union: EthernetAddressSet#EthernetAddressSet->EthernetAddressSet
     difference: EthernetAddressSet#EthernetAddressSet->EthernetAddressSet
     eq:EthernetAddressSet#EthernetAddressSet->Bool
     if:Bool#EthernetAddressSet#EthernetAddressSet->EthernetAddressSet
     delete:EthernetAddressSet#EthernetAddress->EthernetAddressSet
     contains:EthernetAddressSet#EthernetAddress->Bool
     getOne:EthernetAddressSet->EthernetAddress
     deleteOne:EthernetAddressSet->EthernetAddressSet
var  v1, v2: EthernetAddress
     s1, s2: EthernetAddressSet
rew  eq(enil,enil)=T
     eq(enil,cons(v2,s2))=F
     eq(cons(v1,s1),enil)=F
     eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
     add(enil,v1)=cons(v1,enil)
     add(cons(v1,s1),v2)=if(lt(v1,v2),
                            cons(v1,add(s1,v2)),
                            if(lt(v2,v1),cons(v2,cons(v1,s1)),cons(v1,s1)))
     if(T,s1,s2)=s1
     if(F,s1,s2)=s2
     delete(enil,v2)=enil
     delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
     contains(enil,v2)=F
     contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
     union(enil,s2)=s2
     union(cons(v1,s1),s2)=union(s1,add(s2,v1))
     difference(s1,enil)=s1
     difference(cons(v1,s1),s2)=difference(s1,delete(s2,v1))
     getOne(cons(v1,s1))=v1
     deleteOne(cons(v1,s1))=s1


% Datatype Ethernet Frame

sort EthernetFrame
func noFrame:->EthernetFrame
     Frame:Bit#EthernetAddress#EthernetAddressSet#EthernetType#Set->EthernetFrame
map  pmbit:EthernetFrame->Bit
     source:EthernetFrame->EthernetAddress
     destination:EthernetFrame->EthernetAddressSet
     data:EthernetFrame->Set
     eq:EthernetFrame#EthernetFrame->Bool
var  b1,b2:Bit
     s1,s2:EthernetAddress
     d1,d2:EthernetAddressSet
     t1,t2:EthernetType
     v1,v2:Set
rew  pmbit(Frame(b1,s1,d1,t1,v1))=b1
     source(Frame(b1,s1,d1,t1,v1))=s1
     destination(Frame(b1,s1,d1,t1,v1))=d1
     data(Frame(b1,s1,d1,t1,v1))=v1
     eq(noFrame,noFrame)=T
     eq(noFrame,Frame(b2,s2,d2,t2,v2))=F
     eq(Frame(b1,s1,d1,t1,v1),noFrame)=F
     eq(Frame(b1,s1,d1,t1,v1),Frame(b2,s2,d2,t2,v2))=and(eq(b1,b2),
                                                         and(eq(s1,s2),
                                                         and(eq(d1,d2),
                                                         and(eq(t1,t2),
                                                             eq(v1,v2)))))


%%%%%%%%%%%%%%%%%%%%%%%%
% Ethernet processes
%%%%%%%%%%%%%%%%%%%%%%%%

% The Ethernet coaxial cable and physical layer:
% nodes:         the nodes attached to the ether
% toPropagate:   the current signal
% propagateFrom: the node that sent the current signal
% propagateTo:   the nodes that still have to be reached by the current signal

proc PhysicalLayer(nodes      :   TransceiverIdSet,
          toPropagate:   EthernetFrame,
          propagateFrom: TransceiverId,
          propagateTo:   TransceiverIdSet) =

  % A signal is transmitted by a transceiver over an idle ether:
  % the signal is propagated over the ether

  sum(t:TransceiverId,
  sum(f:EthernetFrame,
     ep_transmitSignal_(t,f).
     PhysicalLayer(nodes,f,t,delete(nodes,t))))
        <| eq(propagateFrom,noTransceiverId) |> delta
 +
  % A signal is transmitted by a transceiver over a busy ether:
  % the hanging transmit and all hanging receives return an error code

  sum(f:EthernetFrame,
  sum(t:TransceiverId,
     ep_transmitSignal_(t,f).
     EtherReturnLoop(difference(nodes,propagateTo),alignmentError).
     ep_return_(t, excessiveCollisionError))).
     PhysicalLayer(nodes, noFrame, noTransceiverId, tnil)
        <| not(eq(propagateTo,tnil)) |> delta
 +
  % A signal has been transmitted successfully:
  % sender receives a transmitOk return code and all others receive a
  % receiveOk return code

  EtherReturnLoop(delete(nodes,propagateFrom),receiveOk).
  ep_return_(propagateFrom, transmitOk).
  PhysicalLayer(nodes,noFrame,noTransceiverId,tnil)
     <| and(eq(propagateTo,tnil),
            not(eq(propagateFrom,noTransceiverId))) |> delta
 +
  % A signal is received

  ep_receiveSignal_(getOne(propagateTo),toPropagate).
  PhysicalLayer(nodes,toPropagate,propagateFrom,deleteOne(propagateTo))
     <| not(eq(propagateTo,tnil)) |> delta



% Send return code 'toReturn' to all nodes in 'returnTo'

proc EtherReturnLoop(returnTo:TransceiverIdSet, toReturn:EtherReturnCode)=

  ep_return_(getOne(returnTo),toReturn).
  EtherReturnLoop(deleteOne(returnTo),toReturn)
    <| not(eq(returnTo,tnil)) |>
  nop



% The Ethernet datalink layer

proc DatalinkLayer(transceiver: TransceiverId,
                   node       : EthernetAddress)=

        % Pass frame from client layer to coaxial cable

        sum(b:Bit,
        sum(d:EthernetAddressSet,
        sum(t:EthernetType,
        sum(v:Set,
          ed_transmitFrame_(node,b,node,d,t,v).
```

```
            ep_transmitSignal(transceiver,Frame(b,node,d,t,v))).
            sum(c:EtherReturnCode,
               ep_return(transceiver,c).
               ed_return_(node,c)).
            DatalinkLayer(transceiver,node))))
         +
          % Pass/ignore frame from coaxial cable to client layer

          sum(b:Bit,
          sum(s:EthernetAddress,
          sum(d:EthernetAddressSet,
          sum(t:EthernetType,
          sum(v:Set,
            ep_receiveSignal(transceiver,Frame(b,s,d,t,v)).
            ed_receiveFrame_(node,b,s,d,t,v).
            sum(c:EtherReturnCode,
               ep_return(transceiver,c).
               ed_return_(node,c)).
            DatalinkLayer(transceiver,node)
               <| and(not(eq(s,node)),
                      or(and(eq(b,zero),
                             eq(d,add(enil,node))),
                      or(and(eq(b,one),
                             eq(d,enil)),
                         contains(d,node)))) |>
            ep_receiveSignal(transceiver,Frame(b,s,d,t,v)).
            sum(c:EtherReturnCode, ep_return(transceiver,c)).
            DatalinkLayer(transceiver,node))))))



%%%%%%%%%%%%%%%%%%%%%%%
% Ethernet actions
%%%%%%%%%%%%%%%%%%%%%%%

act  ed_transmitFrame_:   EthernetAddress#Bit#EthernetAddress#EthernetAddressSet#EthernetType#Set
     ed_receiveFrame_:    EthernetAddress#Bit#EthernetAddress#EthernetAddressSet#EthernetType#Set
     ed_return_:          EthernetAddress#EtherReturnCode
     ep_transmitSignal_:  TransceiverId#EthernetFrame
     ep_receiveSignal_:   TransceiverId#EthernetFrame
     ep_return_:          TransceiverId#EtherReturnCode

     ed_transmitFrame:    EthernetAddress#Bit#EthernetAddress#EthernetAddressSet#EthernetType#Set
     ed_receiveFrame:     EthernetAddress#Bit#EthernetAddress#EthernetAddressSet#EthernetType#Set
     ed_return:           EthernetAddress#EtherReturnCode
     ep_transmitSignal:   TransceiverId#EthernetFrame
     ep_receiveSignal:    TransceiverId#EthernetFrame
     ep_return:           TransceiverId#EtherReturnCode

     ced_transmitFrame:   EthernetAddress#Bit#EthernetAddress#EthernetAddressSet#EthernetType#Set
     ced_receiveFrame:    EthernetAddress#Bit#EthernetAddress#EthernetAddressSet#EthernetType#Set
     cep_receiveSignal:   TransceiverId#EthernetFrame
     cep_transmitSignal:  TransceiverId#EthernetFrame
     ced_return:          EthernetAddress#EtherReturnCode
     cep_return:          TransceiverId#EtherReturnCode


%%%%%%%%%%%%%%%%%%%%%%%
% General datatypes
%%%%%%%%%%%%%%%%%%%%%%%

% Datatype Bool

sort Bool
func T,F:->Bool
map  or,and:Bool#Bool->Bool
     not:Bool->Bool
var b:Bool
rew or(T,b)=T
    or(b,T)=T
    or(b,F)=b
    or(F,b)=b
    and(T,b)=b
    and(b,T)=b
    and(b,F)=F
    and(F,b)=F
    not(T)=F
    not(F)=T
    not(not(b))=b

% Datatype Bit

sort Bit
func zero,one:->Bit
map  eq:Bit#Bit->Bool
rew  eq(zero,zero)=T
     eq(zero,one)=F
     eq(one,zero)=F
     eq(one,one)=T

% Datatype for the natural numbers

sort Natural
func 0:->Natural
     s: Natural -> Natural
map  eq: Natural # Natural -> Bool
     lt: Natural # Natural -> Bool
     gt: Natural # Natural -> Bool
     add: Natural # Natural -> Natural
var x,y:Natural
rew eq(0,0)=T
    eq(s(x),0)=F
    eq(0,s(y))=F
    eq(s(x),s(y))=eq(x,y)
    lt(0,0)=F
    lt(0,s(y))=T
    lt(s(x),0)=F
    lt(s(x),s(y))=lt(x,y)
    gt(0,0)=F
    gt(0,s(y))=F
    gt(s(x),0)=T
    gt(s(x),s(y))=gt(x,y)
    add(0,y)=y
    add(s(x),y)=s(add(x,y))
```

Appendix II
The μCRL specification of SPLICE

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Specification: SPLICE (Splice fragment)
% Version      : 1.9
% Date         : 12 October 1999
% Description  : A specification of SPLICE with distributed
%                data space based on 'Splice User Manual'
%                and 'Splice Reference Manual' with context
% Author       : Izak van Langevelde
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Splice                                                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%
% Database datatypes
%%%%%%%%%%%%%%%%%%%%%%%%

% Datatype Sort (sorts of data in the database)

sort Sort
func Sort0, Sort1, Sort2:->Sort
map  i:Sort->Natural
     eq: Sort#Sort->Bool
     lt: Sort#Sort->Bool
     gt: Sort#Sort->Bool
var  s1, s2: Sort
rew  i(Sort0)=0
     i(Sort1)=s(0)
     i(Sort2)=s(s(0))
     eq(s1,s2)=eq(i(s1),i(s2))
     lt(s1,s2)=lt(i(s1),i(s2))
     gt(s1,s2)=gt(i(s1),i(s2))



% Datatype Category

sort Category
func noCategory, Context:->Category
map  i:Category->Natural
     eq: Category#Category->Bool
     lt: Category#Category->Bool
     gt: Category#Category->Bool
     if: Bool#Category#Category->Category
     category:Category#Sort->Bool
var  s1, s2: Category
rew  i(noCategory)=0
     i(Context)=s(i(noCategory))
     eq(s1,s2)=eq(i(s1),i(s2))
     lt(s1,s2)=lt(i(s1),i(s2))
     gt(s1,s2)=gt(i(s1),i(s2))
     category(Context,Sort0)=T
     category(Context,Sort1)=T
     category(Context,Sort2)=T
     category(noCategory,Sort0)=F
     category(noCategory,Sort1)=F
     category(noCategory,Sort2)=F
     if(T,s1,s2)=s1
     if(F,s1,s2)=s2



% Datatype Value

sort Value
func Value0, Value1, Value2, Value3, Value4:->Value
map  i:Value->Natural
     eq: Value#Value->Bool
     lt: Value#Value->Bool
     gt: Value#Value->Bool
var  v1, v2: Value
rew  i(Value0)=0
     i(Value1)=s(i(Value0))
     i(Value2)=s(i(Value1))
     i(Value3)=s(i(Value2))
     i(Value4)=s(i(Value3))
     eq(v1,v2)=eq(i(v1),i(v2))
     lt(v1,v2)=lt(i(v1),i(v2))
     gt(v1,v2)=gt(i(v1),i(v2))


% Datatype Record

sort Record
func Record0, Record1, Record2, Record3, Record4:->Record
map  i:Record->Natural
     eq: Record#Record->Bool
     lt: Record#Record->Bool
     gt: Record#Record->Bool
     key:Record->Value
var  v1, v2: Record
rew  i(Record0)=0
     i(Record1)=s(i(Record0))
     i(Record2)=s(i(Record1))
     i(Record3)=s(i(Record2))
     i(Record4)=s(i(Record3))
     eq(v1,v2)=eq(i(v1),i(v2))
     lt(v1,v2)=lt(key(v1),key(v2))
     gt(v1,v2)=gt(key(v1),key(v2))
     key(Record0)=Value0
     key(Record1)=Value0
     key(Record2)=Value0
     key(Record3)=Value1
     key(Record4)=Value1


% Datatype Data

sort Data
func Datum:Sort#Record->Data
map  eq:Data#Data->Bool
     lt:Data#Data->Bool
     gt:Data#Data->Bool
var  d1, d2: Record
     s1, s2: Sort
rew  eq(Datum(s1,d1),Datum(s2,d2)) = and(eq(s1,s2),eq(d1,d2))
     lt(Datum(s1,d1),Datum(s2,d2)) = or(lt(s1,s2),and(eq(s1,s2),lt(d1,d2)))
     gt(Datum(s1,d1),Datum(s2,d2)) = or(gt(s1,s2),and(eq(s1,s2),gt(d1,d2)))


% Datatype DataSet

sort DataSet
func vnil:->DataSet
     cons:Data#DataSet->DataSet
map  add: DataSet#Data->DataSet
     union: DataSet#DataSet->DataSet
     difference: DataSet#DataSet->DataSet
     eq:DataSet#DataSet->Bool
     if:Bool#DataSet#DataSet->DataSet
     delete:DataSet#Data->DataSet
     contains:DataSet#Data->Bool
     select:DataSet#Query->DataSet
     select:DataSet#Sort->DataSet
     size:DataSet->Natural
var  v1, v2: Data
     s1, s2: DataSet
rew  eq(vnil,vnil)=T
     eq(vnil,cons(v2,s2))=F
     eq(cons(v1,s1),vnil)=F
```

```
      eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
      add(vnil,v1)=cons(v1,vnil)
      add(cons(v1,s1),v2)=if(lt(v1,v2),
                             cons(v1,add(s1,v2)),
                             if(lt(v2,v1),
                                cons(v2,cons(v1,s1)),
                                cons(v2,s1)))
      if(T,s1,s2)=s1
      if(F,s1,s2)=s2
      delete(vnil,v2)=vnil
      delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
      contains(vnil,v2)=F
      contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
      union(vnil,s2)=s2
      union(cons(v1,s1),s2)=union(s1,add(s2,v1))
      difference(s1,vnil)=s1
      difference(cons(v1,s1),s2)=difference(s1,delete(s2,v1))
      size(vnil)=0
      size(cons(v1,s1))=s(size(s1))
var   s1,s2: DataSet
      d1,d2: Data
      q1,q2: Qualifier
      n1,n2:Quantifier
      t1, t2: Sort
      r1, r2: Record
rew   select(vnil,Query(n1,q1))=vnil
      select(cons(Datum(t1,r1),s1),t2)=if(eq(t1,t2),
                             cons(Datum(t1,r1),select(s1,t1)),
                             select(s1,t1))
      select(cons(d1,s1),Query(each,q1))=if(satisfies(q1,d1),
                                            cons(d1,select(s1,Query(each,q1))),
                                            select(s1,Query(each,q1)))
      select(cons(d1,s1),Query(any,q1))=if(satisfies(q1,d1),
                                            cons(d1,vnil),
                                            select(s1,Query(any,q1)))

% Datatype qualifier

sort Qualifier
func noQuery,qualifier1, qualifier2:-> Qualifier
map   satisfies:Qualifier#Data->Bool
      eq:Qualifier#Qualifier->Bool
      i:Qualifier->Natural
var   v: Record
      s: Sort
      q1,q2: Qualifier
rew   satisfies(qualifier1,Datum(s,v))=eq(s,Sort1)
      satisfies(qualifier2,Datum(s,v))=eq(s,Sort2)
      satisfies(noQuery,Datum(s,v))    =T
      i(noQuery)=0
      i(qualifier1)=s(0)
      i(qualifier2)=s(s(0))
      eq(q1,q2)=eq(i(q1),i(q2))

% Datatype quantifier

sort Quantifier
func each,any:-> Quantifier
map   eq:Quantifier#Quantifier->Bool
      i:Quantifier->Natural
var   q1,q2: Quantifier
rew   i(each)=0
      i(any)=s(0)
      eq(q1,q2)=eq(i(q1),i(q2))

% Datatype query

sort Query
func Query: Quantifier#Qualifier->Query
map   eq: Query#Query->Bool
var   n1, n2: Quantifier
      q1, q2: Qualifier
rew   eq(Query(n1,q1),Query(n2,q2))=and(eq(n1,n2),eq(q1,q2))

%%%%%%%%%%%%%%%%%%%%%%%
% SPLICE datatypes
%%%%%%%%%%%%%%%%%%%%%%%

% Datatype SpliceReturnCode

sort SpliceReturnCode
func sp_ok,
     sp_err_invalid_handle,
     sp_err_permission_denied,
     sp_err_invalid_parameter,
     sp_err_out_of_memory:->SpliceReturnCode
map   i: SpliceReturnCode->Natural
      eq:SpliceReturnCode#SpliceReturnCode->Bool
var   r1, r2:SpliceReturnCode
rew   i(sp_ok)=0
      i(sp_err_invalid_handle)=s(i(sp_ok))
      i(sp_err_permission_denied)=s(i(sp_err_invalid_handle))
      i(sp_err_invalid_parameter)=s(i(sp_err_permission_denied))
      i(sp_err_out_of_memory)=s(i(sp_err_invalid_parameter))
      eq(r1,r2)=eq(i(r1),i(r2))


% Sort Flag

sort Flag
func SP_RD_WIPE, SP_RD_MARK:-> Flag
map   eq:Flag#Flag->Bool
rew   eq(SP_RD_WIPE,SP_RD_WIPE)=T
      eq(SP_RD_WIPE,SP_RD_MARK)=F
      eq(SP_RD_MARK,SP_RD_WIPE)=F
      eq(SP_RD_MARK,SP_RD_MARK)=T

% Datatype ApplId

sort ApplId
func noAppl, appl0, appl1, appl2, appl3, appl4, appl5, appl6:->ApplId
map   i:ApplId->Natural
      eq: ApplId#ApplId->Bool
      lt: ApplId#ApplId->Bool
      gt: ApplId#ApplId->Bool
      if: Bool#ApplId#ApplId->ApplId
var   id1, id2: ApplId
rew   i(noAppl)=0
      i(appl0)=s(i(noAppl))
      i(appl1)=s(i(appl0))
      i(appl2)=s(i(appl1))
      i(appl3)=s(i(appl2))
      i(appl4)=s(i(appl3))
      i(appl5)=s(i(appl4))
      i(appl6)=s(i(appl5))
      eq(id1,id2)=eq(i(id1),i(id2))
      lt(id1,id2)=lt(i(id1),i(id2))
      gt(id1,id2)=gt(i(id1),i(id2))
      if(T,id1,id2)=id1
      if(F,id1,id2)=id2


% Datatype ApplIdSet

sort ApplIdSet
func anil:->ApplIdSet
     cons:ApplId#ApplIdSet->ApplIdSet
map   add: ApplIdSet#ApplId->ApplIdSet
      union: ApplIdSet#ApplIdSet->ApplIdSet
      difference: ApplIdSet#ApplIdSet->ApplIdSet
      eq:ApplIdSet#ApplIdSet->Bool
      if:Bool#ApplIdSet#ApplIdSet->ApplIdSet
```

```
      delete:ApplIdSet#ApplId->ApplIdSet
      contains:ApplIdSet#ApplId->Bool
      size:ApplIdSet->Natural
  var v1, v2: ApplId
      s1, s2: ApplIdSet
  rew eq(anil,anil)=T
      eq(anil,cons(v2,s2))=F
      eq(cons(v1,s1),anil)=F
      eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
      add(anil,v1)=cons(v1,anil)
      add(cons(v1,s1),v2)=if(lt(v1,v2),
                             cons(v1,add(s1,v2)),
                             if(lt(v2,v1),cons(v2,cons(v1,s1)),cons(v1,s1)))
      if(T,s1,s2)=s1
      if(F,s1,s2)=s2
      delete(anil,v2)=anil
      delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
      contains(anil,v2)=F
      contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
      union(anil,s2)=s2
      union(cons(v1,s1),s2)=union(s1,add(s2,v1))
      difference(s1,anil)=s1
      difference(s1,cons(v2,s2))=delete(difference(s1,s2),v2)
      size(anil)=0
      size(cons(v1,s1))=s(size(s1))


  % Datatype Subscription

  sort Subscription
  func Subscribe:ApplId#Sort->Subscription
  map  eq:Subscription#Subscription->Bool
       lt:Subscription#Subscription->Bool
       gt:Subscription#Subscription->Bool
  var  i1,i2:ApplId
       s1,s2:Sort
  rew  eq(Subscribe(i1,s1),Subscribe(i2,s2))=and(eq(s1,s2),eq(i1,i2))
       lt(Subscribe(i1,s1),Subscribe(i2,s2))=or(lt(i1,i2), and(eq(i1,i2),lt(s1,s2)))
       gt(Subscribe(i1,s1),Subscribe(i2,s2))=or(gt(i1,i2), and(eq(i1,i2),gt(s1,s2)))

  % Datatype SubscriptionSet

  sort SubscriptionSet
  func snil:->SubscriptionSet
       cons:Subscription#SubscriptionSet->SubscriptionSet
  map  add: SubscriptionSet#Subscription->SubscriptionSet
       union: SubscriptionSet#SubscriptionSet->SubscriptionSet
       difference: SubscriptionSet#SubscriptionSet->SubscriptionSet
       eq:SubscriptionSet#SubscriptionSet->Bool
       if:Bool#SubscriptionSet#SubscriptionSet->SubscriptionSet
       delete:SubscriptionSet#Subscription->SubscriptionSet
       contains:SubscriptionSet#Subscription->Bool
       subscribers:SubscriptionSet#Sort->ApplIdSet
       select:SubscriptionSet#Category->SubscriptionSet
       getOne:SubscriptionSet->Subscription
       deleteOne:SubscriptionSet->SubscriptionSet
       size:SubscriptionSet->Natural
  var  v1, v2: Subscription
       s1, s2: SubscriptionSet
       t1, t2: Sort
       i1, i2: ApplId
       c: Category
  rew  eq(snil,snil)=T
       eq(snil,cons(v2,s2))=F
       eq(cons(v1,s1),snil)=F
       eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
       add(snil,v1)=cons(v1,snil)
       add(cons(v1,s1),v2)=if(lt(v1,v2),
                              cons(v1,add(s1,v2)),
                              if(lt(v2,v1),cons(v2,cons(v1,s1)),cons(v1,s1)))
       if(T,s1,s2)=s1
       if(F,s1,s2)=s2
       delete(snil,v2)=snil
       delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
       contains(snil,v2)=F
       contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
       union(snil,s2)=s2
       union(cons(v1,s1),s2)=union(s1,add(s2,v1))
       subscribers(snil, t2)=anil
       subscribers(cons(Subscribe(i1,t1),s1),t2)=if(eq(t1,t2),
                                                    cons(i1,subscribers(s1,t2)),
                                                    subscribers(s1,t2))
       select(snil,c)=snil
       select(cons(Subscribe(i1,t1),s1),c)=if(category(c,t1),cons(Subscribe(i1,t1),select(s1,c)),select(s1,c))
       getOne(cons(v1,s1))=v1
       deleteOne(cons(v1,s1))=s1
       difference(s1,snil)=s1
       difference(s1,cons(v2,s2))=delete(difference(s1,s2),v2)
       size(snil)=0
       size(cons(v1,s1))=s(size(s1))

  sort CSubscription
  func Subscribe:ApplId#Category->CSubscription
  map  eq:CSubscription#CSubscription->Bool
       lt:CSubscription#CSubscription->Bool
       gt:CSubscription#CSubscription->Bool
  var  i1,i2:ApplId
       s1,s2:Category
  rew  eq(Subscribe(i1,s1),Subscribe(i2,s2))=and(eq(s1,s2),eq(i1,i2))
       lt(Subscribe(i1,s1),Subscribe(i2,s2))=or(lt(i1,i2), and(eq(i1,i2),lt(s1,s2)))
       gt(Subscribe(i1,s1),Subscribe(i2,s2))=or(gt(i1,i2), and(eq(i1,i2),gt(s1,s2)))

  % Datatype CSubscriptionSet

  sort CSubscriptionSet
  func cnil:->CSubscriptionSet
       cons:CSubscription#CSubscriptionSet->CSubscriptionSet
  map  add: CSubscriptionSet#CSubscription->CSubscriptionSet
       union: CSubscriptionSet#CSubscriptionSet->CSubscriptionSet
       difference: CSubscriptionSet#CSubscriptionSet->CSubscriptionSet
       eq:CSubscriptionSet#CSubscriptionSet->Bool
       if:Bool#CSubscriptionSet#CSubscriptionSet->CSubscriptionSet
       delete:CSubscriptionSet#CSubscription->CSubscriptionSet
       contains:CSubscriptionSet#CSubscription->Bool
       size:CSubscriptionSet->Natural
  var  v1, v2: CSubscription
       s1, s2: CSubscriptionSet
       t1, t2: Category
       i1, i2: ApplId
  rew  eq(cnil,cnil)=T
       eq(cnil,cons(v2,s2))=F
       eq(cons(v1,s1),cnil)=F
       eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
       add(cnil,v1)=cons(v1,cnil)
       add(cons(v1,s1),v2)=if(lt(v1,v2),
                              cons(v1,add(s1,v2)),
                              if(lt(v2,v1),cons(v2,cons(v1,s1)),cons(v1,s1)))
       if(T,s1,s2)=s1
       if(F,s1,s2)=s2
       delete(cnil,v2)=cnil
       delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
       contains(cnil,v2)=F
       contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
       union(cnil,s2)=s2
       union(cons(v1,s1),s2)=union(s1,add(s2,v1))
       difference(s1,cnil)=s1
       difference(s1,cons(v2,s2))=delete(difference(s1,s2),v2)
       size(cnil)=0
       size(cons(v1,s1))=s(size(s1))

  % Datatype Publication

  sort Publication
```

```
     func Publish:ApplId#Sort->Publication
     map  eq:Publication#Publication->Bool
435       lt:Publication#Publication->Bool
          gt:Publication#Publication->Bool
     var  i1,i2:ApplId
          s1,s2:Sort
     rew  eq(Publish(i1,s1),Publish(i2,s2))=and(eq(s1,s2),eq(i1,i2))
440       lt(Publish(i1,s1),Publish(i2,s2))=or(lt(i1,i2), and(eq(i1,i2),lt(s1,s2)))
          gt(Publish(i1,s1),Publish(i2,s2))=or(gt(i1,i2), and(eq(i1,i2),gt(s1,s2)))

     % Datatype PublicationSet

445  sort PublicationSet
     func pnil:->PublicationSet
          cons:Publication#PublicationSet->PublicationSet
     map  add: PublicationSet#Publication->PublicationSet
          union: PublicationSet#PublicationSet->PublicationSet
450       difference: PublicationSet#PublicationSet->PublicationSet
          eq:PublicationSet#PublicationSet->Bool
          if:Bool#PublicationSet#PublicationSet->PublicationSet
          delete:PublicationSet#Publication->PublicationSet
          contains:PublicationSet#Publication->Bool
455       select:PublicationSet#Category->PublicationSet
          getOne:PublicationSet->Publication
          deleteOne:PublicationSet->PublicationSet
          makeRequests: ApplId#PublicationSet#Category->RequestSet
          makePublications: ApplId#DataSet#Category->PublicationSet
460       size:PublicationSet->Natural
     var  v1, v2: Publication
          s1, s2: PublicationSet
          ds1   : DataSet
          c: Category
465       a1, a2: ApplId
          s: Sort
          r: Record
     rew  eq(pnil,pnil)=T
          eq(pnil,cons(v2,s2))=F
470       eq(cons(v1,s1),pnil)=F
          eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
          add(pnil,v1)=cons(v1,pnil)
          add(cons(v1,s1),v2)=if(lt(v1,v2),
                                 cons(v1,add(s1,v2)),
475                              if(lt(v2,v1),cons(v2,cons(v1,s1)),cons(v1,s1)))
          if(T,s1,s2)=s1
          if(F,s1,s2)=s2
          delete(pnil,v2)=pnil
          delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
480       contains(pnil,v2)=F
          contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
          union(pnil,s2)=s2
          union(cons(v1,s1),s2)=union(s1,add(s2,v1))
          difference(s1,pnil)=s1
485       difference(s1,cons(v2,s2))=delete(difference(s1,s2),v2)
          select(pnil,c)=pnil
          select(cons(Publish(a1,s),s1),c)=if(category(c,s),cons(Publish(a1,s),select(s1,c)),select(s1,c))
          makeRequests(a1, pnil,c)=rnil
          makeRequests(a1, cons(Publish(a2,s),s1),c)=if(and(eq(a1,a2),category(c,s)),
490                                                       cons(Request(a1,s),makeRequests(a1, s1,c)),
                                                          makeRequests(a1, s1,c))
          makePublications(a1, vnil, c)=pnil
          makePublications(a1, cons(Datum(s,r),ds1), c)=
             if(category(c,s),
495             cons(Publish(a1,s), makePublications(a1,ds1,c)),
                makePublications(a1,ds1,c))
          getOne(cons(v1,s1))=v1
          deleteOne(cons(v1,s1))=s1
          size(pnil)=0
500       size(cons(v1,s1))=s(size(s1))



     % Datatype Presence

505

     sort Presence
     func Present:ApplId#EthernetAddress->Presence
     map  eq:Presence#Presence->Bool
          lt:Presence#Presence->Bool
510       gt:Presence#Presence->Bool
          if:Bool#Presence#Presence->Presence
     var  i1,i2:ApplId
          ad1, ad2: EthernetAddress
          p1, p2: Presence
515  rew  eq(Present(i1,ad1),Present(i2,ad2))=and(eq(i1,i2),eq(ad1,ad2))
          lt(Present(i1,ad1),Present(i2,ad2))=or(lt(i1,i2),and(eq(i1,i2),lt(ad1,ad2)))
          gt(Present(i1,ad1),Present(i2,ad2))=or(gt(i1,i2),and(eq(i1,i2),gt(ad1,ad2)))
          if(T,p1,p2)=p1
          if(F,p1,p2)=p2
520
     % Datatype PresenceSet

     sort PresenceSet
     func prnil:->PresenceSet
525       cons:Presence#PresenceSet->PresenceSet
     map  add: PresenceSet#Presence->PresenceSet
          union: PresenceSet#PresenceSet->PresenceSet
          difference: PresenceSet#PresenceSet->PresenceSet
          eq:PresenceSet#PresenceSet->Bool
530       if:Bool#PresenceSet#PresenceSet->PresenceSet
          delete:PresenceSet#Presence->PresenceSet
          contains:PresenceSet#Presence->Bool
          lookup:PresenceSet#ApplId->EthernetAddress
          lookup:PresenceSet#EthernetAddress->ApplId
535       lookup:PresenceSet#ApplIdSet->EthernetAddressSet
          size:PresenceSet->Natural
     var  v1, v2: Presence
          s1, s2: PresenceSet
          i1, i2: ApplId
540       a1, a2: EthernetAddress
          is1: ApplIdSet
     rew  eq(prnil,prnil)=T
          eq(prnil,cons(v2,s2))=F
          eq(cons(v1,s1),prnil)=F
545       eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
          add(prnil,v1)=cons(v1,prnil)
          add(cons(v1,s1),v2)=if(lt(v1,v2),
                                 cons(v1,add(s1,v2)),
                                 if(lt(v2,v1),cons(v2,cons(v1,s1)),cons(v1,s1)))
550       if(T,s1,s2)=s1
          if(F,s1,s2)=s2
          delete(prnil,v2)=prnil
          delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
          contains(prnil,v2)=F
555       contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
          lookup(prnil,a2)=noAppl
          lookup(cons(Present(i1,a1),s1), a2)=if(eq(a1,a2),i1,lookup(s1,a1))
          lookup(prnil,i1)=noAddress
          lookup(cons(Present(i1,a1),s1), i2)=if(eq(i1,i2),a1,lookup(s1,i1))
560       lookup(s1,anil)=enil
          lookup(s1,cons(i1,is1))=add(lookup(s1,is1),lookup(s1,i1))
          union(prnil,s2)=s2
          union(cons(v1,s1),s2)=union(s1,add(s2,v1))
          difference(s1,prnil)=s1
565       difference(s1,cons(v2,s2))=delete(difference(s1,s2),v2)
          size(prnil)=0
          size(cons(v1,s1))=s(size(s1))

     % Datatype Request

570
     sort Request
     func Request:ApplId#Sort->Request
     map  eq:Request#Request->Bool
          lt:Request#Request->Bool
575       gt:Request#Request->Bool
          if:Bool#Request#Request->Request
```

```
     var  i1,i2:Sort
          a1,a2:ApplId
          p1, p2: Request
580  rew  eq(Request(a1,i1),Request(a2,i2))=and(eq(a1,a2),eq(i1,i2))
          lt(Request(a1,i1),Request(a2,i2))=or(lt(a1,a2),and(eq(a1,a2),lt(i1,i2)))
          gt(Request(a1,i1),Request(a2,i2))=or(gt(a1,a2),and(eq(a1,a2),gt(i1,i2)))
          if(T,p1,p2)=p1
          if(F,p1,p2)=p2
585
     % Datatype RequestSet

     sort RequestSet
     func rnil:->RequestSet
590       cons:Request#RequestSet->RequestSet
     map  add: RequestSet#Request->RequestSet
          union: RequestSet#RequestSet->RequestSet
          difference: RequestSet#RequestSet->RequestSet
          eq:RequestSet#RequestSet->Bool
595       if:Bool#RequestSet#RequestSet->RequestSet
          delete:RequestSet#Request->RequestSet
          contains:RequestSet#Request->Bool
          lookup:RequestSet#ApplId->EthernetAddress
          lookup:RequestSet#ApplIdSet->EthernetAddressSet
600       size:RequestSet->Natural
     var  v1, v2: Request
          s1, s2: RequestSet
     rew  eq(rnil,rnil)=T
          eq(rnil,cons(v2,s2))=F
605       eq(cons(v1,s1),rnil)=F
          eq(cons(v1,s1),cons(v2,s2))=and(eq(v1,v2),eq(s1,s2))
          add(rnil,v1)=cons(v1,rnil)
          add(cons(v1,s1),v2)=if(lt(v1,v2),
                                 cons(v1,add(s1,v2)),
610                              if(lt(v2,v1),cons(v2,cons(v1,s1)),cons(v1,s1)))
          if(T,s1,s2)=s1
          if(F,s1,s2)=s2
          delete(rnil,v2)=rnil
          delete(cons(v1,s1),v2)=if(eq(v1,v2),s1,cons(v1,delete(s1,v2)))
615       contains(rnil,v2)=F
          contains(cons(v1,s1),v2)=or(eq(v1,v2),contains(s1,v2))
          union(rnil,s2)=s2
          union(cons(v1,s1),s2)=union(s1,add(s2,v1))
          difference(s1,rnil)=s1
620       difference(s1,cons(v2,s2))=delete(difference(s1,s2),v2)
          size(rnil)=0
          size(cons(v1,s1))=s(size(s1))




625
     % Datatype Set

     sort Set
     func unite:DataSet#PublicationSet#SubscriptionSet#PresenceSet#RequestSet#CSubscriptionSet->Set
630  map  nil:->Set
          add: Set#Data->Set
          add: Set#Publication->Set
          add: Set#Subscription->Set
          add: Set#Presence->Set
635       add: Set#Request->Set
          add: Set#CSubscription->Set
          union: Set#Set->Set
          difference: Set#Set->Set
          eq:Set#Set->Bool
640       if:Bool#Set#Set->Set
          delete:Set#Publication->Set
          delete:Set#Subscription->Set
          delete:Set#Data->Set
          delete:Set#Presence->Set
645       delete:Set#Request->Set
          delete:Set#CSubscription->Set
          contains:Set#Publication->Bool
          contains:Set#Subscription->Bool
          contains:Set#Data->Bool
650       contains:Set#Presence->Bool
          contains:Set#Request->Bool
          contains:Set#CSubscription->Bool
          select:Set#Query->Set
          select:Set#Category->Set
655       makeRequests:ApplId#Set#Category->Set
          makePublications: ApplId#Set#Category->Set
          select:Set#Sort->Set
          subscribers:Set#Sort->ApplIdSet
          getOne:Set->Publication
660       deleteOne:Set->Set
          size:Set->Natural
     var  vs1, vs2: DataSet
          ps1, ps2: PublicationSet
          ss1, ss2: SubscriptionSet
665       prs1, prs2: PresenceSet
          rs1, rs2: RequestSet
          cs1, cs2: CSubscriptionSet
          s1, s2: Set
          v: Data
670       p: Publication
          s: Subscription
          cs: CSubscription
          pr: Presence
          r: Request
675       q: Query
          t: Sort
          c: Category
          a: ApplId
     rew  nil=unite(vnil,pnil,snil,prnil,rnil,cnil)
680       eq(unite(vs1,ps1,ss1,prs1,rs1,cs1),unite(vs2,ps2,ss2,prs2,rs2,cs2))=and(eq(vs1,vs2),
                                                                                 and(eq(ps1,ps2),
                                                                                 and(eq(ss1,ss2),
                                                                                 and(eq(prs1,prs2),
                                                                                 and(eq(rs1,rs2),
685                                                                                  eq(cs1,cs2))))))
          add(unite(vs1,ps1,ss1,prs1,rs1,cs1),v)=unite(add(vs1,v),ps1,ss1,prs1,rs1,cs1)
          add(unite(vs1,ps1,ss1,prs1,rs1,cs1),p)=unite(vs1,add(ps1,p),ss1,prs1,rs1,cs1)
          add(unite(vs1,ps1,ss1,prs1,rs1,cs1),s)=unite(vs1,ps1,add(ss1,s),prs1,rs1,cs1)
          add(unite(vs1,ps1,ss1,prs1,rs1,cs1),pr)=unite(vs1,ps1,ss1,add(prs1,pr),rs1,cs1)
690       add(unite(vs1,ps1,ss1,prs1,rs1,cs1),r)=unite(vs1,ps1,ss1,prs1,add(rs1,r),cs1)
          add(unite(vs1,ps1,ss1,prs1,rs1,cs1),cs)=unite(vs1,ps1,ss1,prs1,rs1,add(cs1,cs))
          delete(unite(vs1,ps1,ss1,prs1,rs1,cs1),v)=unite(delete(vs1,v),ps1,ss1,prs1,rs1,cs1)
          delete(unite(vs1,ps1,ss1,prs1,rs1,cs1),p)=unite(vs1,delete(ps1,p),ss1,prs1,rs1,cs1)
          delete(unite(vs1,ps1,ss1,prs1,rs1,cs1),s)=unite(vs1,ps1,delete(ss1,s),prs1,rs1,cs1)
695       delete(unite(vs1,ps1,ss1,prs1,rs1,cs1),pr)=unite(vs1,ps1,ss1,delete(prs1,pr),rs1,cs1)
          delete(unite(vs1,ps1,ss1,prs1,rs1,cs1),r)=unite(vs1,ps1,ss1,prs1,delete(rs1,r),cs1)
          delete(unite(vs1,ps1,ss1,prs1,rs1,cs1),cs)=unite(vs1,ps1,ss1,prs1,rs1,add(cs1,cs))
          contains(unite(vs1,ps1,ss1,prs1,rs1,cs1),v)=contains(vs1,v)
          contains(unite(vs1,ps1,ss1,prs1,rs1,cs1),p)=contains(ps1,p)
700       contains(unite(vs1,ps1,ss1,prs1,rs1,cs1),s)=contains(ss1,s)
          contains(unite(vs1,ps1,ss1,prs1,rs1,cs1),pr)=contains(prs1,pr)
          contains(unite(vs1,ps1,ss1,prs1,rs1,cs1),r)=contains(rs1,r)
          contains(unite(vs1,ps1,ss1,prs1,rs1,cs1),cs)=contains(cs1,cs)
          union(unite(vs1,ps1,ss1,prs1,rs1,cs1),unite(vs2,ps2,ss2,prs2,rs2,cs2))=
705          unite(union(vs1,vs2),union(ps1,ps2),union(ss1,ss2),union(prs1,prs2),union(rs1,rs2),union(cs1,cs2))
          difference(unite(vs1,ps1,ss1,prs1,rs1,cs1),unite(vs2,ps2,ss2,prs2,rs2,cs2))=
             unite(difference(vs1,vs2),difference(ps1,ps2),difference(ss1,ss2),
                   difference(prs1,prs2),difference(rs1,rs2),difference(cs1,cs2))
          select(unite(vs1,ps1,ss1,prs1,rs1,cs1),q)=unite(select(vs1,q),pnil,snil,prnil,rnil,cnil)
710       select(unite(vs1,ps1,ss1,prs1,rs1,cs1),c)=unite(vnil,select(ps1,c),ss1,prnil,rnil,cnil)
          select(unite(vs1,ps1,ss1,prs1,rs1,cs1),t)=unite(select(vs1,t),ps1,ss1,prs1,rs1,cs1)
          subscribers(unite(vs1,ps1,ss1,prs1,rs1,cs1),t)=subscribers(ss1,t)
          getOne(unite(vs1,ps1,ss1,prs1,rs1,cs1))=getOne(ps1)
          deleteOne(unite(vs1,ps1,ss1,prs1,rs1,cs1))=unite(vs1,deleteOne(ps1),ss1,prs1,rs1,cs1)
715       makeRequests(a,unite(vs1,ps1,ss1,prs1,rs1,cs1),c)=unite(vnil,pnil,snil,prnil,makeRequests(a,ps1,c),cnil)
          makePublications(a,unite(vs1,ps1,ss1,prs1,rs1,cs1),c)=unite(vnil,makePublications(a,vs1,c),snil,prnil,rnil,cnil)
          if(T,s1,s2)=s1
          if(F,s1,s2)=s2

720  size(unite(vs1,ps1,ss1,prs1,rs1,cs1))=add(size(vs1),add(size(ps1),add(size(ss1),add(size(prs1),add(size(rs1),size(cs1))))))
```

```
% Datatype Bond

sort Bond
func Bind:Subscription#Query->Bond
map  eq:Bond#Bond->Bool
var  s1,s2:Subscription
     q1,q2:Query
rew  eq(Bind(s1,q1),Bind(s2,q2))=and(eq(s1,s2),eq(q1,q2))

% Datatype frameQueue

sort frameQueue
func qnil:->frameQueue
     cons:EthernetFrame#frameQueue->frameQueue
     bound:frameQueue#Natural->frameQueue
map  add: frameQueue#EthernetFrame->frameQueue
     add: frameQueue#EthernetFrame#Natural->frameQueue
     tail:frameQueue->frameQueue
     head:frameQueue->EthernetFrame
     eq:frameQueue#frameQueue->Bool
     if:Bool#frameQueue#frameQueue->frameQueue
     empty:frameQueue->Bool
     full:frameQueue->Bool
     length:frameQueue->Natural
var  p1,p2:EthernetFrame
     q1,q2:frameQueue
     m1,m2:Natural
rew  add(qnil,p2)=cons(p2,qnil)
     add(cons(p1,q1),p2)=cons(p1,add(q1,p2))
     head(cons(p1,q1))=p1
     tail(cons(p1,q1))=q1
     eq(qnil,qnil)=T
     eq(qnil,cons(p2,q2))=F
     eq(cons(p1,q1),qnil)=F
     eq(cons(p1,q1),cons(p2,q2))=and(eq(p1,p2),eq(q1,q2))
     empty(qnil)=T
     empty(cons(p1,q1))=F
     length(qnil)=0
     length(cons(p1,q1))=s(length(q1))

     add(bound(q1,m1),p2)=bound(add(q1,p2,m1),m1)
     add(q1,p2,0)=q1
     add(qnil,p2,s(m1))=cons(p2,qnil)
     add(cons(p1,q1),p2,s(m1))=cons(p1,add(q1,p2,m1))
     head(bound(q1,m1))=head(q1)
     tail(bound(q1,m1))=bound(tail(q1),m1)
     eq(bound(q1,m1),bound(q2,m2))=and(eq(q1,q2),eq(m1,m2))
     empty(bound(q1,m1))=empty(q1)
     full(bound(q1,m1))=eq(length(q1),m1)
     if(T,q1,q2)=q1
     if(F,q1,q2)=q2



%%%%%%%%%%%%%%%%%%%
% Splice actions
%%%%%%%%%%%%%%%%%%%

act  sp_init_superapi_:   ApplId
     sp_application_:     ApplId
     sp_terminate_:       ApplId
     sp_use_sort_:        ApplId#Sort
     sp_publish_:         ApplId#Publication
     sp_write_:           ApplId#Data
     sp_subscribe_simple_:ApplId#Subscription
     sp_subscribe_:       ApplId#Subscription
     sp_subscribe_to_category_: ApplId#CSubscription
     sp_bind_query_:      ApplId#Bond
     sp_read_:            ApplId#Bond#Set#Flag
     sp_init_superapi:    ApplId
     sp_application:      ApplId
     sp_terminate:        ApplId
     sp_use_sort:         ApplId#Sort
     sp_publish:          ApplId#Publication
     sp_write:            ApplId#Data
     sp_subscribe_simple: ApplId#Subscription
     sp_subscribe:        ApplId#Subscription
     sp_subscribe_to_category: ApplId#CSubscription
     sp_read:             ApplId#Bond#Set#Flag

     csp_init_superapi:   ApplId
     csp_application:     ApplId
     csp_terminate:       ApplId
     csp_use_sort:        ApplId#Sort
     csp_publish:         ApplId#Publication
     csp_write:           ApplId#Data
     csp_subscribe:       ApplId#Subscription
     csp_subscribe_to_category: ApplId#CSubscription
     csp_read:            ApplId#Bond#Set#Flag
     panic

     nop

comm sp_init_superapi_   | sp_init_superapi    = csp_init_superapi
     sp_application_     | sp_application      = csp_application
     sp_terminate_       | sp_terminate        = csp_terminate
     sp_use_sort_        | sp_use_sort         = csp_use_sort
     sp_publish_         | sp_publish          = csp_publish
     sp_write_           | sp_write            = csp_write
     sp_subscribe_       | sp_subscribe        = csp_subscribe
     sp_subscribe_to_category_ | sp_subscribe_to_category       = csp_subscribe_to_category
     sp_read_            | sp_read             = csp_read

     ep_receiveSignal_   | ep_receiveSignal    = cep_receiveSignal
     ep_transmitSignal_  | ep_transmitSignal   = cep_transmitSignal
     ed_receiveFrame_    | ed_receiveFrame     = ced_receiveFrame
     ed_transmitFrame_   | ed_transmitFrame    = ced_transmitFrame
     ep_return_          | ep_return           = cep_return
     ed_return_          | ed_return           = ced_return



% The SPLICE agent:

proc Agent(agentAddress: EthernetAddress,  % Ethernet address of this agent
           configuration:PresenceSet,      % Table of application ids and their addresses
           agentId:      ApplId,           % Application id of this agent
           inQueue:      frameQueue,       % Queue of incoming ethernet frames
           outQueue:     frameQueue,       % Queue of outgoing ethernet frames
           pendingCategory: Category,      % Category of category subscription in process
           pendingPublications: Set,      % Subscriptions of category subscription in process
           c:            Set,              % Records consumed
           p:            Set)=             % Records produced

  nop.
  Agent(agentAddress,configuration,agentId,inQueue,
        add(outQueue,Frame(one,agentAddress,enil,Presence,add(nil,Present(agentId,agentAddress)))),
        pendingCategory,pendingPublications,c,p)

 +

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Handle the SPLICE api calls
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % Handle sp_init_superapi
```

```
sp_init_superapi_(agentId).
   Agent(agentAddress,configuration,agentId,inQueue,
         outQueue,pendingCategory,pendingPublications,c,p)
+

% Handle sp_terminate

sp_terminate_(agentId).
   delta
+

% Handle sp_application

sp_application_(agentId).
   Agent(agentAddress,add(configuration,Present(agentId,agentAddress)),agentId,inQueue,
         add(outQueue,Frame(one,agentAddress,enil,Presence,add(nil,Present(agentId,agentAddress)))),
         pendingCategory,pendingPublications,c,p)
+

% Handle sp_use_sort

sum(s:Sort,
   sp_use_sort_(agentId,s)).
   Agent(agentAddress,configuration,agentId,inQueue,
         outQueue,pendingCategory,pendingPublications,c,p)
+

% Handle sp_publish

sum(s:Sort,
   sp_publish_(agentId, Publish(agentId,s)).
  (panic.delta
      <| full(outQueue) |>
   Agent(agentAddress,configuration, agentId, inQueue,
         add(outQueue,Frame(one,agentAddress,enil,PublishSubscribe,add(nil,Publish(agentId,s)))),
         pendingCategory,pendingPublications, c, p)))
+

% Handle sp_write

sum(s:Sort, sum(v:Record,
   sp_write_(agentId, Datum(s,v)).
  (panic.delta
      <| full(outQueue) |>
   Agent(agentAddress,configuration, agentId, inQueue,
         add(outQueue, Frame(one,agentAddress,lookup(configuration,subscribers(c,s)),Data,add(nil,Datum(s,v)))),
         pendingCategory,pendingPublications, c, add(p,Datum(s,v))))))
+

% Handle sp_subscribe

sum(s:Sort,
   sp_subscribe_(agentId, Subscribe(agentId,s)).
  (panic.delta
      <| full(outQueue) |>
   Agent(agentAddress,configuration, agentId, inQueue,
         add(outQueue,Frame(one,agentAddress,enil,PublishSubscribe,add(nil,Subscribe(agentId,s)))),
         pendingCategory,pendingPublications, c, p)))
+

% Handle sp_subscribe_to_category

sum(s:Category,
   sp_subscribe_to_category_(agentId, Subscribe(agentId,s)).
  (panic.delta
      <| full(outQueue) |>
   Agent(agentAddress,configuration, agentId, inQueue,
         add(outQueue,Frame(one,agentAddress,enil,PublishSubscribe,add(nil,Subscribe(agentId,s)))),
         s, pendingPublications, c, p)))
+

% Handle sp_bind_query

sum(b:Bond,
   sp_bind_query_(agentId,b)).
   Agent(agentAddress,configuration,agentId,inQueue,
         outQueue,pendingCategory,pendingPublications,c,p)
+

% Handle sp_read

sum(s:Sort, sum(q:Qualifier, sum(n: Quantifier,
   sp_read_(agentId,Bind(Subscribe(agentId,s),Query(n,q)),select(c,Query(n,q)),SP_RD_WIPE).
   Agent(agentAddress,configuration,agentId,inQueue,
         outQueue,pendingCategory,pendingPublications,difference(c,select(c,Query(n,q))),p)
      <| or(eq(n,each),eq(size(select(c,Query(n,q))),s(0))) |> delta)))
+

% Handle sp_read

sum(s:Sort, sum(q:Qualifier, sum(n: Quantifier,
   sp_read_(agentId,Bind(Subscribe(agentId,s),Query(n,q)),select(c,Query(n,q)),SP_RD_MARK).
   Agent(agentAddress,configuration,agentId,inQueue,
         outQueue,pendingCategory,pendingPublications,c,p)
      <| or(eq(n,each),eq(size(select(c,Query(n,q))),s(0))) |> delta)))
+

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Queue incoming Ethernet frames
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(t:EthernetType,
sum(r:Set,
   ed_receiveFrame(agentAddress,b,s,d,t,r).
   sum(e:EtherReturnCode,
      ed_return(agentAddress,e).
        (panic.delta
           <| or(full(inQueue),not(eq(e,receiveOk))) |>
         Agent(agentAddress,configuration,agentId,add(inQueue,Frame(b,s,d,t,r)),
               outQueue,pendingCategory,pendingPublications, c,p))))))))

+

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Send queued outgoing frames
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(t:EthernetType,
sum(r:Set,
   ed_transmitFrame(agentAddress,b,s,d,t,r).
   sum(e:EtherReturnCode,
      ed_return(agentAddress,e).
      (panic.delta
         <| not(eq(e,transmitOk)) |>
       Agent(agentAddress,configuration,agentId,inQueue,
             tail(outQueue),pendingCategory,pendingPublications, c,p)))
         <| and(not(empty(outQueue)),
                eq(head(outQueue),Frame(b,s,d,t,r))) |> delta)))))
+

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Handle queued incoming frames
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Handle incoming publication and/or subscription if no category subscription is pending
```

```
sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(r:Set,
  (panic.delta
      <| full(outQueue) |>
   nop.
   Agent(agentAddress,configuration,agentId,tail(inQueue),
         add(outQueue,Frame(zero,agentAddress,add(enil,s),Data,select(p, Query(each,noQuery)))),
         pendingCategory,pendingPublications,union(c,r),p))
      <| and(not(empty(inQueue)),
         and(eq(head(inQueue),Frame(b,s,d,PublishSubscribe,r)),
             eq(pendingCategory,noCategory))) |> delta))))
+

% Handle incoming publication and/or subscription if category subscription is pending:
% request the data for the sort and add the subscription to the pending subscriptions

sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(r:Set,
   (panic.delta
      <| full(outQueue) |>
    nop.
    Agent(agentAddress, configuration, agentId, tail(inQueue),
         if (eq(makeRequests(agentId,r,pendingCategory),nil),
             outQueue,
             add(outQueue, Frame(zero, agentAddress,add(enil,s),Request, makeRequests(agentId,r,pendingCategory))),
         if (eq(union(pendingPublications, select(r,pendingCategory)),nil), noCategory, pendingCategory),
         union(pendingPublications, select(r,pendingCategory)),
         union(c,difference(r,select(r,pendingCategory))),
         p))
      <| and(not(empty(inQueue)),
         and(eq(head(inQueue),Frame(b,s,d,PublishSubscribe,r)),
             not(eq(pendingCategory,noCategory)))) |> delta))))
+

% Handle incoming category subscription

sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(ca:Category,
sum(a:ApplId,
   (panic.delta
      <| full(outQueue) |>
    nop.
       Agent(agentAddress,configuration,agentId,tail(inQueue),
             add(outQueue,Frame(zero,agentAddress,add(enil,s),PublishSubscribe,
                                union(select(p,ca),select(c,ca)))),
             pendingCategory,pendingPublications,add(c,Subscribe(a,ca)),p))
      <| and(not(empty(inQueue)),
             eq(head(inQueue),Frame(b,s,d,PublishSubscribe,add(nil,Subscribe(a,ca))))) |> delta)))))
+

% Handle incoming data if no publication is pending

sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(r:Set,
   nop.
   Agent(agentAddress,configuration,agentId, tail(inQueue),
         outQueue, pendingCategory,
         difference(pendingPublications,
                    makePublications(lookup(configuration,s),r,pendingCategory)),
         union(c,r),
         p)
      <| and(eq(pendingPublications,nil),
         and(not(empty(inQueue)),
             eq(head(inQueue),Frame(b,s,d,Data,r)))) |> delta))))
+

% Handle incoming request

sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(a:ApplId,
sum(t:Sort,
   (panic.delta
      <| full(outQueue) |>
    nop.
      Agent(agentAddress,configuration,agentId,tail(inQueue),
            add(outQueue,Frame(zero,agentAddress,add(enil,lookup(configuration,a)),Data,select(p,t))),
            pendingCategory,pendingPublications,c,p))

      <| and(not(empty(inQueue)),
            eq(head(inQueue),Frame(b,s,d,Request,add(nil,Request(a,t))))) |> delta)))))
+

% Handle incoming presence

sum(b:Bit,
sum(s:EthernetAddress,
sum(d:EthernetAddressSet,
sum(r:Presence,
   nop.
   Agent(agentAddress,add(configuration,r),agentId,tail(inQueue),
        outQueue,pendingCategory,pendingPublications,c,p
)
      <| and(not(empty(inQueue)),
            eq(head(inQueue),Frame(b,s,d,Presence,add(nil,r)))) |> delta))))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SPLICE context process
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc ContextProcess=
       sp_init_superapi(appl2).
       sp_application(appl2).
       sp_subscribe_to_category(appl2,Subscribe(appl2,Context)).
       delta



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SPLICE client process 0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc P0(id: ApplId)=
       sp_init_superapi(id).
       sp_application(id).
       sp_use_sort(id,Sort1).
       sp_subscribe(id,Subscribe(id,Sort1)).
       sp_publish(id,Publish(id,Sort1)).
       (sp_write(id,Datum(Sort1,Record0))+sp_write(id,Datum(Sort1,Record1))).
       sum(v:Set, sp_read(id,Bind(Subscribe(id,Sort1),Query(each,qualifier1)),v,SP_RD_MARK)).
       sp_terminate(id).
       delta

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SPLICE client process 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc P1(id: ApplId)=
       sp_init_superapi(id).
       sp_application(id).
       sp_use_sort(id,Sort1).
       sp_subscribe(id,Subscribe(id,Sort1)).
```

```
            sp_publish(id,Publish(id,Sort1)).
            sum(v:Set, sp_read(id,Bind(Subscribe(id,Sort1),Query(each,qualifier1)),v,SP_RD_MARK)).
            (sp_write(id,Datum(Sort1,Record0))+sp_write(id,Datum(Sort1,Record1))).
            sp_terminate(id).
            delta



%%%%%%%%%%%%%%%%%%%%
% Initial process
%%%%%%%%%%%%%%%%%%%%

init
          hide ({ced_transmitFrame,
                 ced_receiveFrame,
                 cep_transmitSignal,
                 ced_receiveFrame,
                 ced_return,
                 cep_return},

          encap ({sp_init_superapi_   , sp_init_superapi   ,
                  sp_terminate_       , sp_terminate       ,
                  sp_application_     , sp_application     ,
                  sp_use_sort_        , sp_use_sort        ,
                  sp_publish_         , sp_publish         ,
                  sp_write_           , sp_write           ,
                  sp_subscribe_       , sp_subscribe       ,
                  sp_subscribe_to_category_,  sp_subscribe_to_category,
                  sp_read_            , sp_read            ,
                  ed_transmitFrame_   , ed_transmitFrame,
                  ed_receiveFrame_    , ed_receiveFrame,
                  ep_transmitSignal_  , ep_transmitSignal,
                  ep_receiveSignal_   , ep_receiveSignal,
                  ed_return_          , ed_return,
                  ep_return_          , ep_return},

      PhysicalLayer(add(add(tnil,transceiver1),transceiver2),
                    noFrame,
                    noTransceiverId,
                    tnil)                                                    ||
      DatalinkLayer(transceiver1,address1)                                   ||
      DatalinkLayer(transceiver2,address2)                                   ||
      P0(appl1)                                                              ||
      P1(appl2)                                                              ||
      Agent(address1,prnil,appl1,bound(qnil,s(0)),bound(qnil,s(0)),noCategory,nil,nil,nil) ||
      Agent(address2,prnil,appl2,bound(qnil,s(0)),bound(qnil,s(0)),noCategory,nil,nil,nil)))
```

Appendix III
The EVALUATOR properties

```
   (* All deadlocks are preceded by a "panic" or a "terminate" *)

   [(not ("panic" or "sp_terminate(*)"))*] <true> true

and

   (* Correctness: each record read has been written *)

   [(not "csp_write(appl1,Datum(Sort1,Record0))")*.
    "csp_read(appl2,Bind(Subscribe(appl2,Sort1),Query(each,qualifier1)),
              unite(cons(Datum(Sort1,Record0),vnil),pnil,snil,prnil,rnil,cnil),
              SP_RD_MARK)"]
   false

and

   (* Weak completeness: as long as a record written is not overwritten, not all reads fail *)

   [(not "panic")*
    . "csp_write(appl1,Datum(Sort1,Record0))"
    . (not ("csp_write(appl1,Datum(Sort1,Record1))" or "panic" or "sp_terminate(*)"))*]
        <(not ("panic" or "sp_terminate(*)"))*
        . "csp_read(appl2,Bind(Subscribe(appl2,Sort1),Query(each,qualifier1)),
                    *,
                    SP_RD_MARK)">
        true

      implies

        <(not ("panic" or "sp_terminate(*)"))*
        . "csp_read(appl2,Bind(Subscribe(appl2,Sort1),Query(each,qualifier1)),
                    unite(cons(Datum(Sort1,Record0),vnil),pnil,snil,prnil,rnil,cnil),
                    SP_RD_MARK)">
```

```
true
```

Appendix IV
The verified scenarios

1. SCENARIO 1

```
P0(id: AppId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sp_publish(id,Publish(id,Sort1)).
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))).
        sum(v:Set, sp_read(id,
                           Bind(Subscribe(id,Sort1),
                                Query(any,qualifier1)),
                           v,
                           SP_RD_MARK)).
        sp_terminate(id).
        delta
```

```
P1(id: AppId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_publish(id,Publish(id,Sort1)).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sum(v:Set,
        sp_read(id,
                Bind(Subscribe(id,Sort1),Query(any,qualifier1)),
                v,
                SP_RD_MARK)).
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))).
        sp_terminate(id).
        delta
```

2. SCENARIO 2

```
P0(id: AppId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sp_publish(id,Publish(id,Sort1)).
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))).
        sum(v:Set, sp_read(id,
                        Bind(Subscribe(id,Sort1),
                                Query(each,qualifier1)),
                        v,
                        SP_RD_MARK)).
        sp_terminate(id).
        delta
```

```
P1(id: AppId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_publish(id,Publish(id,Sort1)).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sum(v:Set,
        sp_read(id,
                Bind(Subscribe(id,Sort1),Query(each,qualifier1)),
                v,
                SP_RD_MARK)).
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))).
        sp_terminate(id).
        delta
```

3. SCENARIO 3

```
P0(id: AppId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_publish(id,Publish(id,Sort1)).
        P0_loops(id)

P0_loops(id: AppId)=
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))).
        P0_loops(id)
```

```
P1(id: AppId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        P1_loops(id)

P1_loops(id: AppId)=
        sum(v:Set,
        sp_read(id,
                Bind(Subscribe(id,Sort1),Query(each,qualifier1)),
                v,
                SP_RD_MARK)).
        P1_loops(id)
```

4. SCENARIO 4

```
P0(id: ApplId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_publish(id,Publish(id,Sort1)).
        P0_loops(id)

P0_loops(id: ApplId)=
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))).
        P0_loops(id)
```

```
P1(id: ApplId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        P1_loops(id)

P1_loops(id: ApplId)=
        sum(v:Set,
        sp_read(id,
                Bind(Subscribe(id,Sort1),Query(any,qualifier1)),
                v,
                SP_RD_MARK)).
        P1_loops(id)
```

5. SCENARIO 5

```
P0(id: ApplId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sp_publish(id,Publish(id,Sort1)).
        P0_loops(id)

P0_loops(id: ApplId)=
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))+
        sum(v:Set,
            sp_read(id,
                    Bind(Subscribe(id,Sort1),
                         Query(any,qualifier1)),
                    v,
                    SP_RD_MARK))).
        P0_loops(id)
```

```
P1(id: ApplId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sp_publish(id,Publish(id,Sort1)).
        P1_loops(id)

P1_loops(id: ApplId)=
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))+
        sum(v:Set,
            sp_read(id,
                    Bind(Subscribe(id,Sort1),
                         Query(any,qualifier1)),
                    v,
                    SP_RD_MARK))).
        P1_loops(id)
```

6. SCENARIO 6

```
P0(id: ApplId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sp_publish(id,Publish(id,Sort1)).
        P0_loops(id)

P0_loops(id: ApplId)=
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))+
        sum(v:Set,
            sp_read(id,
                    Bind(Subscribe(id,Sort1),
                            Query(each,qualifier1)),
                    v,
                    SP_RD_MARK))).
                    P0_loops(id)
```

```
P1(id: ApplId)=
        sp_init_superapi(id).
        sp_application(id).
        sp_use_sort(id,Sort1).
        sp_subscribe(id,Subscribe(id,Sort1)).
        sp_publish(id,Publish(id,Sort1)).
        P1_loops(id)

P1_loops(id: ApplId)=
        (sp_write(id,Datum(Sort1,Record0))+
        sp_write(id,Datum(Sort1,Record1))+
        sum(v:Set,
            sp_read(id,
                    Bind(Subscribe(id,Sort1),
                            Query(each,qualifier1)),
                    v,
                    SP_RD_MARK))).
                    P1_loops(id)
```