Project 22703: SYRF

Deliverable 5.1.1

Automatic verification on mixed A/S systems

---

1 Approach

Synchronous formalisms aim at the representation of reactive processes such as embedded systems, telecommunication nodes and software or hardware controlers. By nature these reactive systems are meant to dialog with an external environment, and can rarely be considered in full isolation.

When synchronous programming is used for the design of communication protocols or other similar programmed nodes embedded in a larger asynchronous setting, the environment becomes the network medium through which messages are exchanged (figure 1). Properties of correction often must deal with the global system as a whole, so that the network must be explicitly modeled and connected with the individual synchronous nodes in a formal way.

At some abstract level we shall usually model the network as some collection of bounded fifo queues –or variants of such with possible message loss or duplication. While each medium can be itself modeled as a synchronous (or asynchronous) process, the communication between synchronous protocol nodes and the network must consist of synchronizing rendez-vous (or else messages could be unduly ignored). Asynchronicity now holds between the puts and gets on messages for the same queue which supposedly take place at distinct locations (figure 2).

The finite state aspect of media buffers is important to us since we want to apply symbolic model-checking techniques based on BDDs, where a global state may be represented as a finite bit vector. The main issue concerning algorithmic efficiency is to combine the partitioning methods that allow *not to represent* the full transition relation, both at the local synchronous level *and* at the global level. In the former case one uses conjunctive partitioning along individual target registers (boolean state elements); the TiGer BDD package used here is specifically optimized for this. In the latter case, we want to apply disjunctive partitioning and apply only locally relevant communication behaviours, in an interleaved fashion (between distinct synchronous localities). This means that queue transitions should be separated in three distinct parts: message get, message deliver (at other end), message transit (internal progress), so as to be divided and combined with different local synchronous endpoints (figure 3).

In this deliverable we provide means to synthesize networks in a formal ways from succint user descriptions of buffer sizes, features and carried values (from finite types). We also provide ways to specify distant message send/receive actions in a synchronous node. We face the algorithmic issues mentioned above. In the coming years we intend to implement this (reusing parts of existing verification softwares in the pure asynchronous or synchronous cases), and conduct case studies.

2 Results

We want to provide syntax for the representation of networks, and of communication interfaces between synchronous sites and the network itself. We consider the network as a collection of individual queues, which can themselves be modeled internally either as synchronous or asynchronous processes. The only important thing is that each queue should provide put and get operations independent of one another, to synchronize at each end of the queue with synchronous modules.

Each synchronous module is connected to a number of queues, either as input or output. Thus synchronous modules are *not* directly plugged together. This allows to partition the global transition relation along synchronous sites, applying in an interleaved fashion only the *local* communications between a site and its connected queue endpoints, together with its internal synchronous reactive behaviour. The partitioning of the transition relation is an important technique for efficient state space generation in model-checking..

2.1 Connections viewed from the local synchronous modules

Synchronous reactive formalisms allow modularity through submodules (or subnodes) definitions. Therefore we shall define typical **send** and **receive** modules for synchronisation behaviours (with queues). This has practical interest since the actual body of these modules may vary from implementation to verification, with different abstraction levels (but then some equivalence maybe provably preserved).

As an example, consider the following typical behaviour for a (synchronising) **send** operation, written as an Esterel fragment:

```
weak abort [sustain is_sync_ok] when immediate sync_ok ; emit sync_value(val)
```

It keeps querying the output fifo queue, which supposedly answers as soon as not full. Then it sends the actual messages, which is readily expected. Note that the actual transmission (reception of `sync_ok` and value emission) takes place in a single atomic step, in the synchronous reactive sense.

2.2 Connections viewed from the Fifo queues

The corresponding behaviour in a fifo modeled as a synchronous process would be something dependent of its current state (full or not). As soon as not full, it performs as:

```
await immediate is_sync_ok; emit sync_ok; stored_value := ?sync_value
```

The Esterel expert will have noted why this protocol demands a `weak abort` construct in the **send** above, to avoid instantaneous causality problem. Also the actual value transmission needs no further reception notification, and is synchronous with the rest of the atomic protocol. Receive operations from fifo to synchronous sites are quite similar to model.

The behavioural rules of each fifo queue is in fact divided in three: *put into queue* at one end, *get from queue* at other end, and *transit* operations. The third kind of behaviour is internal to the queue, it pushes values to contiguous cells closest to the exit. This allows the two other types of synchronisation to be concerned only with well-defined parts of the queue only, in a perspective of partition of the transition relation. The case of queues with capacity 1 is properly handled, since the put and get operations will be interleaved.

The (bounded) queues are usually modeled as an array of cells properly linked. Various properties can be modeled (message loss, duplication, desequencing). The actual representation is not imposed, as long as the behaviours can be divided along the three categories mentioned above. In fact the model can be synthesized from simple typed description (for instance, `MayLooseFifo(`*`size, mess_type`*`)` where size is a constant, can be expanded automatically, by generating the proper register space and transition relations. The values transmitted can be of discrete type, and then possibly represented in actual model-checking); for other types some abstract abstraction is required then (see deliverable on *verification with numerical values* for instance).

2.3 The network description

We adopt a descriptive notation in a process algebraic style: queues and synchronous processes are put in parallel, and renamings allow proper connections. We can impose technical conditions for simplicity: a queue should not be used as buffer from a synchronous site to itself (so that locally to this site one would possibly put and get to and from this file in a single reaction); sharing queues in input or output is also a matter of convenience (but can be modeled alright).

It is important in our modeling framework that the actual synchronisation be atomic (as soon as the queue agrees to a put or get, when it is not full or empty respectively). As synchronous processes may see parts of their behaviours be aborted anytime because of locl preemption on their local clock, it is virtually impossible to make sure that synchronisations were successful unless they are atomic (in local clock time) or drastic syntactic restrictions are taken (absence of preemption on **send/receive** operations. In particular it is provably impossible to implement trustful synchronous rendez-vous on top of our asynchronous communications (through fifo queues) if synchronous sites may see their protocols aborted at any reaction.

We are currently implementing a prototype tool to validate these ideas through experiments, using BDD symbolic techniques.

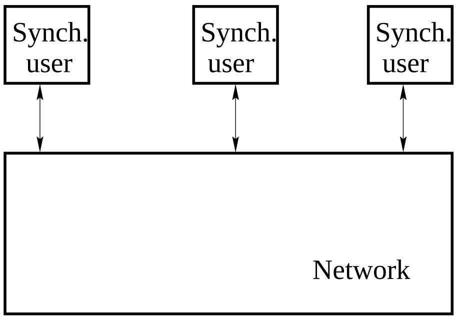

Figure 1: Synchronous sites and network

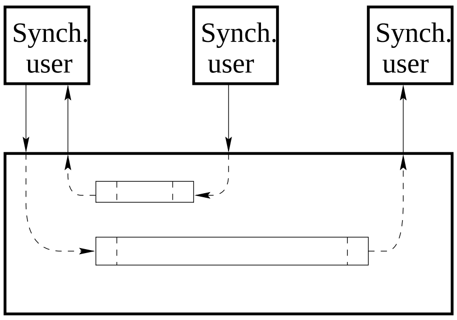

Figure 2: Fifo queues with (asynchronous) local ends

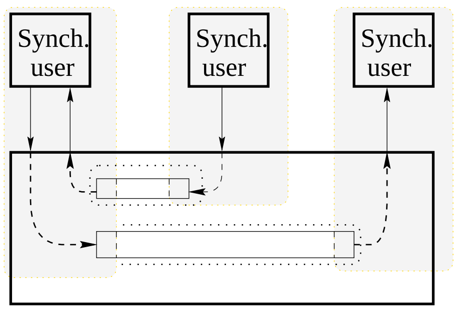

Figure 3: Local parts of the transition relation (here 5)