Constructing "synthesizable" STG specifications
Tutorial Examples

A. Yakovlev

9th Async UK Forum, 19th December 2000, Cambridge

1 Example 1: Controller for self-timed A/D converter

The first example comes from the design of an asynchronous A/D converter described in [1]. This converter (for a 4-bit digital value) uses the well-known successive approximation method, where the digital output is determined bit-by-bit, starting from the MSB, according to the result of comparison of input voltage and the voltage produced by an D/A converter connected to the digital ouput. The idea of the conversion algorithm is captured in the Table 1.

The overall structure of the A/D converter is shown in Figure 1. It consists of a comparator, a 4-bit register and control logic. The comparator uses analogue circuitry and a bistable device at the output which has a completion detector capable of resolving metastable states (for more on the effect of metastability in A/D convereters see [1]). From the point of view of control logic, it has a special input signal, Ar (called CLAMP in [1]), which is normally at 1 to keep the dual rail output, Din, in spacer 00. When Ar goes to 0 the comparator is released to produce a valid dual-rail value on Din (the result of comparison between the adjusted and input voltage).

The 4-bit register is effectively a demux-register with one-bit, dual-rail input and parallel, four-bit output. It is built from latches that are capable of storing the value even when the enable input is withdrawn. The register is controlled by five command signals arriving from the control logic in a one-hot (non-overlapping) manner. The first command signal, R, is used to reset the register in the initial state 1.0.0.0 (with MSB=1), immediately before starting the conversion. For all other command signals, Li, the register is supposed to latch the value from the input Din in the appropriate bit y(i) and set the lower bit, y(i-1) to 1.

Ready	State	MSB (CB3)	CB2	CB1	LSB (CB0)
1	0	X	X	X	X
0	1	1	0	0	0
0	2	Comp	1	0	0
0	3	X	Comp	1	0
0	4	X	X	Comp	1
1	0	X	X	X	Comp

Table 1: Control of Converter ('X' stands for any value, 'Comp' denotes the value produced by comparator)

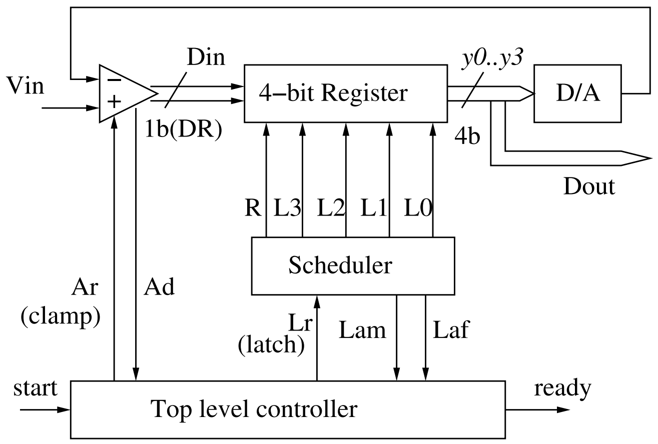

Figure 1: Block diagram of A/D converter

The control logic is structured to consist of a five-way scheduler and a top level controller. The latter is described by an STG shown in Figure 2.

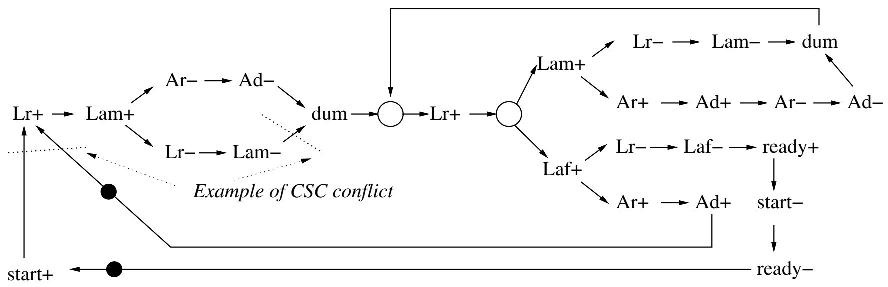

Figure 2: STG specification for top level controller

The environment issues the start+ signal when the analogue value is ready for conversion. The controller then enters a loop, in which it performs the following procedure. It first issues a request to the scheduler, Lr, and receives an acknowledgement, Lam ('m' for middle) when the appropriate latching of the next bit to the register has occured. Then it executes a handshake with the comparator to force the next bit out. When the last bit has been produced, the scheduler issues another ack signal (mutually exclusive to Lam), Laf ('f' from final), upon which the controller generates the ready+ signal to the environment, indicating that the 4-bit code is available on the outputs of the register.

Note the places where this specification contains CSC conflicts.

The STG model of the scheduler is shown in Figure 3

It consists of five identical fragments. Note the similarity of this description with an STG model of a modulo-N counter. Again, this STG has a large number of CSC conflicts. It requires internal memory to record the position of the "token" in this "counter".

Giving this STG directly to Petrify is not the best solution, because Petrify solves CSC conflicts by using bi-partition, which indirectly implies the use of log-type encoding. Furthermore, the csc signals added by Petrify may switch in sequences. This way, the final circuit may consist of relatively deep logic consisting complex gates, which will lead to large cycle time.

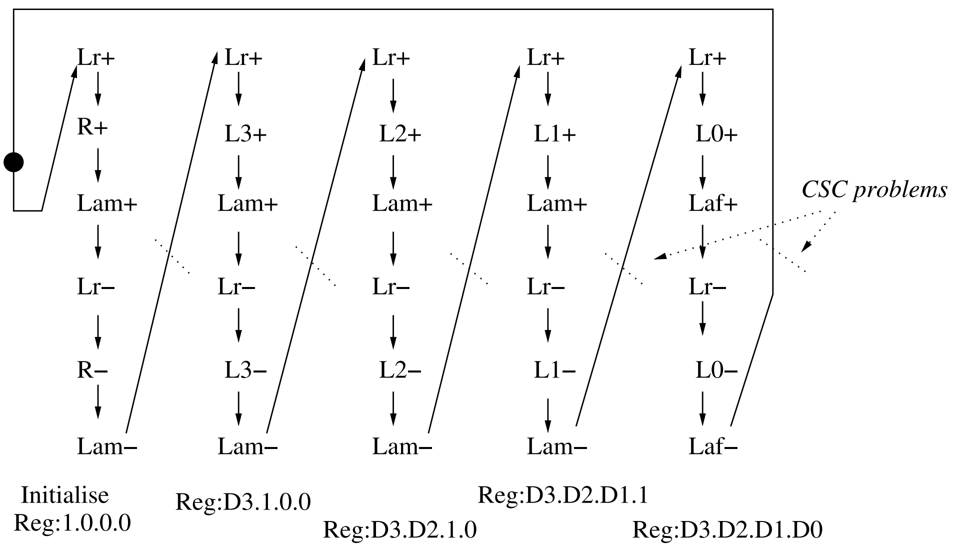

Figure 3: STG specification of scheduler

Unfortunately, for such regular STG models, the 'best' way (at the state of the art of synthesis tools) would be to insert internal memory signals by hand, in a 'one-hot' (one per fragment) manner.

An alternative way would be to follow the idea of regularity in the scheduler and further decompose it into cells, each responsible for one command signal.

The STG model of a cell in the scheduler is shown in Figure 4. This model is based on the idea of an 'activity token' moving cyclically between the cells.

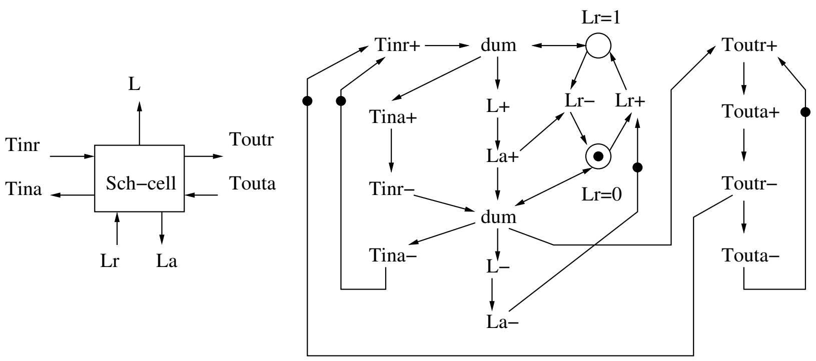

Figure 4: Building scheduler from cells

The fact that the cell model is functionally correct (corresponds to the overall behaviour of scheduler defined in Figure 3) may be verified by composing such STGs together and exploring the behaviour of the composition with the internal signals (corresponding to the token handshakes between cells) considered as silent events and eventually projected away (there are appropriate commands in Petrify). Such a composition is shown in Figure 5.

Note in this diagram how the actions on the three-wire (Lr,Lam/Laf) handshake are all merged into a single fragment ("subroutine" style) compared to their unfolded instances in Figure 3.

Finally, for completeness, we can also apply STGs to synthesise some data-path logic, namely

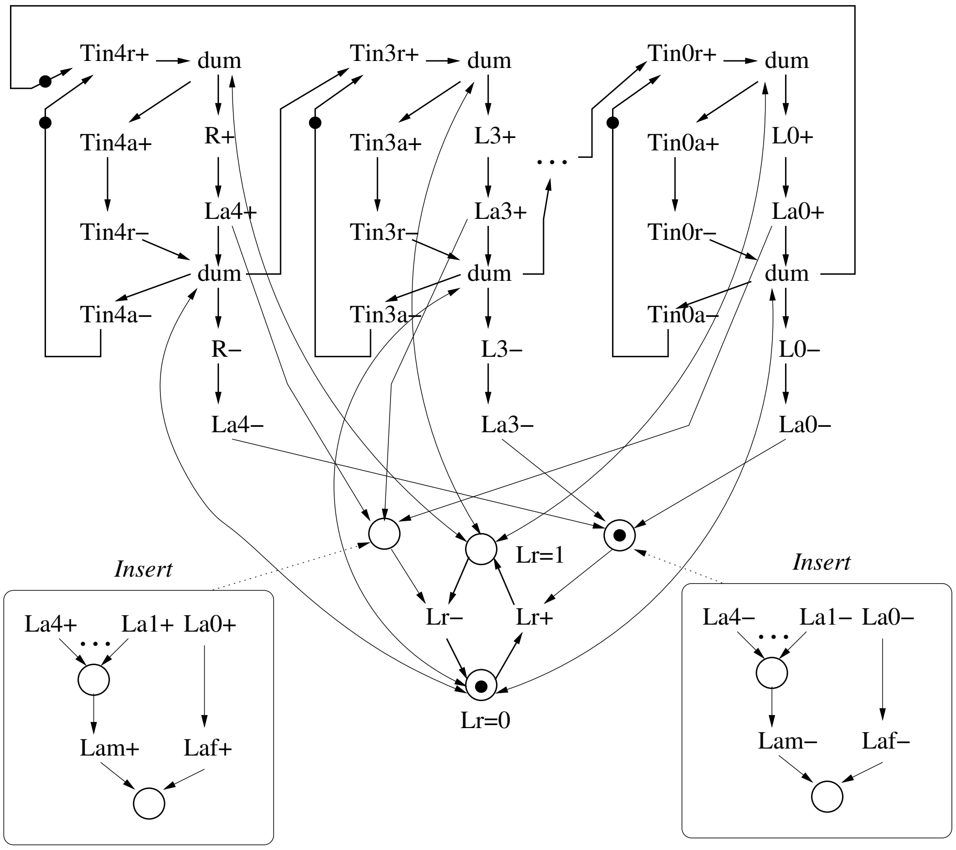

Figure 5: Verifying the composition of cells

one bit latch in the register (see Figures 6 and 7). Note the use of the assignment-type notation, e.g. 'y0ˆ1', for the transitions in the STG. This actually stands for a fragment of a Petri net, inserted by Petrify automatically, which sets the value of signal y0 to 1 if y0=0 initially (corresponds to normal transition 'y0+') or keeps it the same if it was already y0=1 (corresponds to a dummy event).

This hierarchical controller can be used as a ballpark design, but it involves some overhead, such as for example that the signalling between the register and comparator is done via the control logic. Additionall signalling is also introduced in the controller between the environment and the scheduler.

The design published in [1] used a number of optimisations. For example, it was eventually decided that a faster control logic would be obtained by allowing some direct pipelining between the register and the comparator, as well as imposing the responsibility to interact with the environment on the scheduler. All such improvements were however the result of better understanding the controller when it was first considered as a decomposition.

Lessons to be learnt from the first example:

1. It showed how to *decompose* a controller into parts and construct a manageable STG model for each part. Some optimisation at a later stage may be possible.
2. These STGs involved *concurrency, choice, multiple labels, and use of short-hand assignment-*

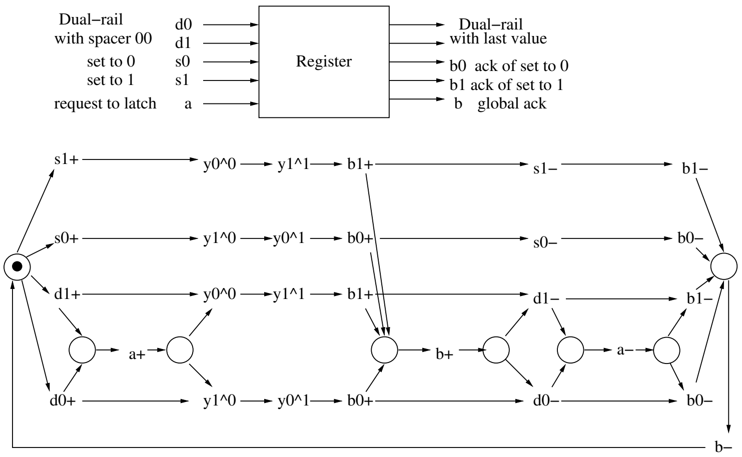

Figure 6: STG model of register bit

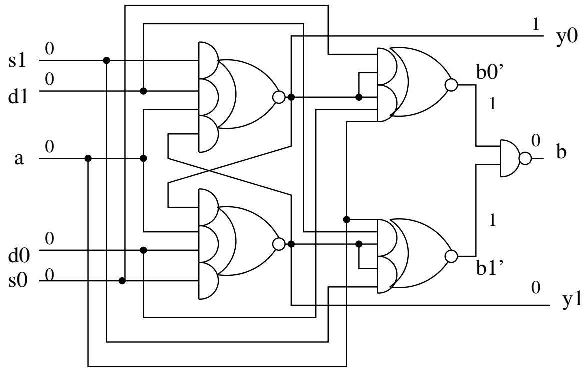

Figure 7: Circuit implementation of register bit

type transitions. Various ways of reshuffling handshakes could be applied in order to improve performance. Relative timing assumptions could also be used.
3. The STGs involved *regular patterns*, which led to many CSC conflicts. Those were better dealt with either by inserting internal memory signals by hand or by further decomposition of the controller into cells. Currently, Petrify is unable to deal with regular fragments and it uses a log-like state encoding policy, which may not be good for performance.
4. There was no arbitration in this example. Any sort of choice in those STGs was purely *non-deterministic choice made in the environment*.

2 Example 2: "Lazy" Token Ring adapter

Our second example will require building a circuit with arbitration.

Figure 8 shows a distributed arbitration system with a ring architecture. Each client, who may want to access the shared resource (e.g. a communication channel), interacts with the system via its adapter. The protocol between the client and its adapter is a simple four-phase two-wire (request-grant) handshake.

The basic idea of the distribution of the so-called "privelege token" is depicted in the Petri net of Figure 8, which describes a "regular section" in the aribtration process. Note that at any time only one adapter holds the token. Thus only one client can obtain the grant at a time.

If an adapter receives a request from its client this request must be forwarded to the right, in order to fetch the token. This request will propagate clockwise through the ring until it reaches the adapter which currently holds the token. The token then will move ant-clockwise until it reaches the adapter whose client has generated the request. This adapter then issues the grant to the client. When the service of the client is finished, the token remains in the adapter until some other adapter, on its left, produces a request. We call this arbitration mechanism "Lazy Token Ring" due to the fact that token remains static until it is requested. By contrast, an alternative mechanism, where the token is in constant polling, is called "Busy Token Ring". Various designs for such arbitration systems have been considered in literature (cf. Alain Martin's papers). The technical report in [2] describes a number of designs using Petri nets, STGs and synthesis tools.

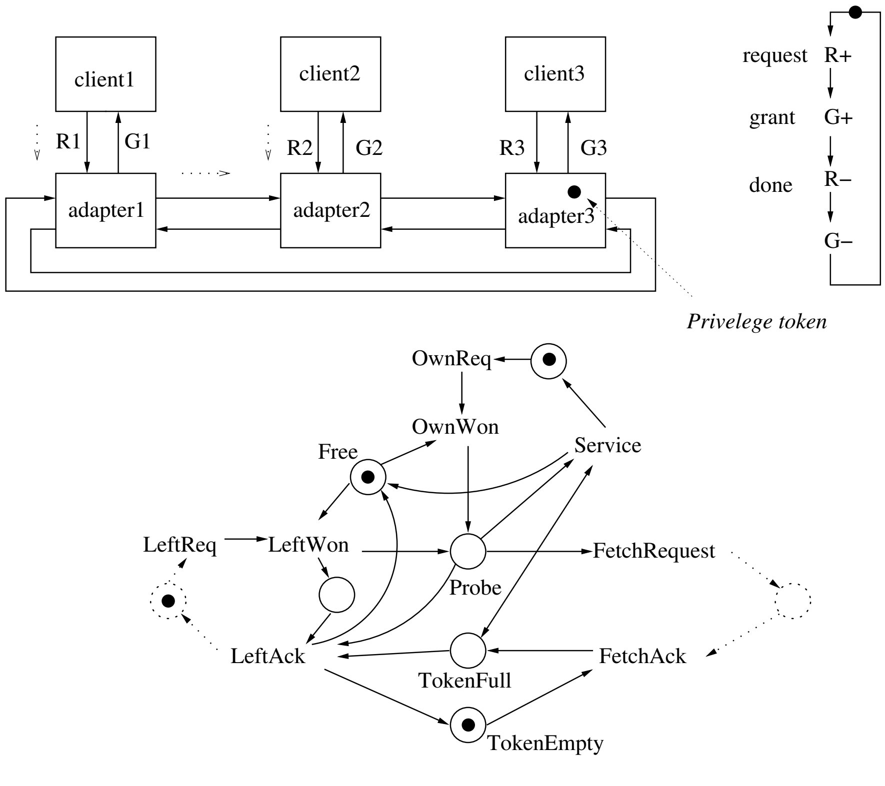

Figure 8: High-level description of the "lazy ring" protocol

An STG specification for an adapter is shown in Figure 9. The key element in this specification is arbitration between the request from the client (R) and the request from the left neighbour in the ring (Lr). The Petri net place labelled 'me', with a single token in it, designates the presence of dynamic (output) choice in the circuit.

The specification consists of four main operation modes. These are determined, firstly, by whether the local arbitration is won by the neighbour's or client's request. Within each of those halves, there is also choice on whether the privelege token is already in the adapter or must be fetched from the right neighbour.

Note that this model has CSC problems.

Because one cannot synthesise a logic circuit with dynamic choice (conflicts) between output

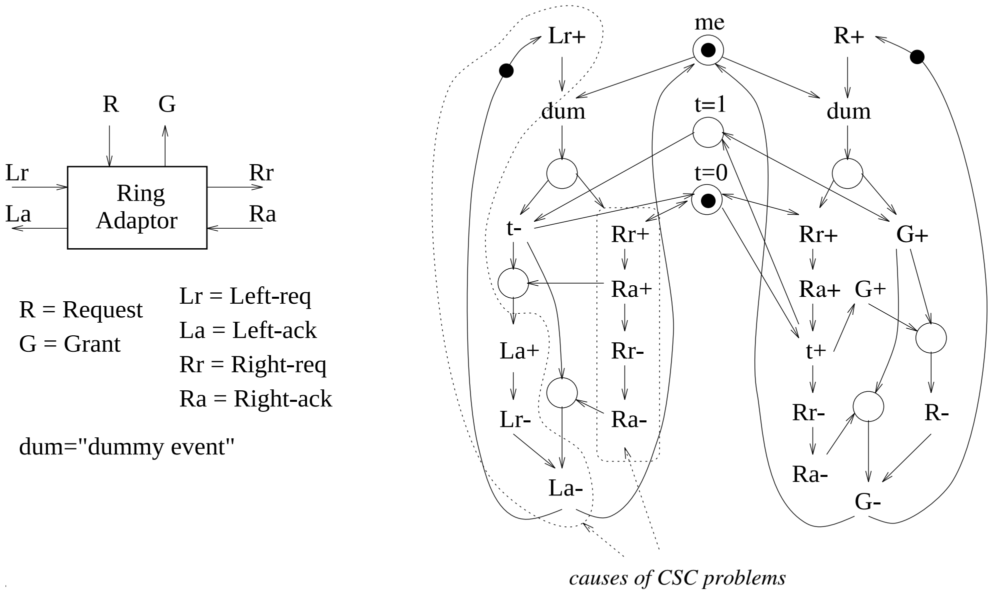

Figure 9: Adapter specification

transitions (cf. the requirement of *output persistency* in Petrify), we should first "factor out" the conflict from the synthesized logic circuit. This is done by inserting mutual exclusion signal transitions into the STG, and declaring the grant signals of the mutex element as *inputs* to the synthesized logic circuit (Figure 10). Note that the insertion of mutex events must keep the overall specification equivalent to the original STG with respect to the order between the target signal transitions (this can sometimes be a problem, and requires making various assumptions, such as relative timing assumptions - more on this is in [3]).

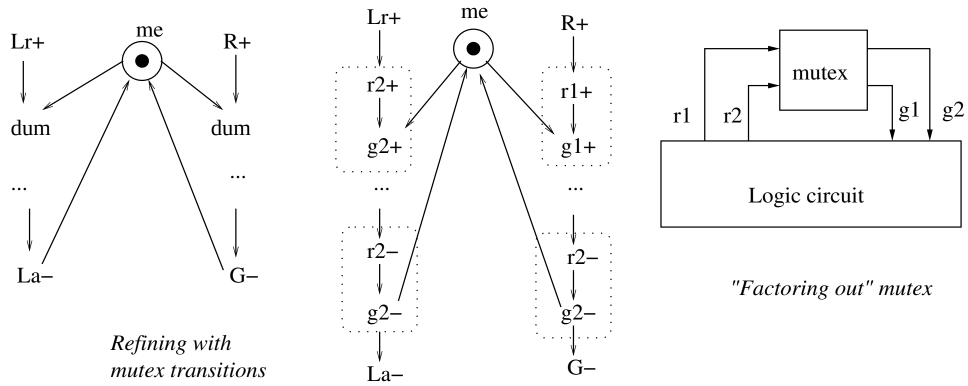

Figure 10: Inserting mutex events

The result of refinement and the circuit synthesized from the STG, which also has an additional internal signal for CSC resolution, is shown in Figure 11.

Finally, Figure 12 shows the advantages of using Petri nets in analysis of performance. The figure depicts the unfolding of the composition of Petri net models of adapters into a partial order graph, from which one can determine the max delay path between the pairs of events of interest (cf. the dotted paths).

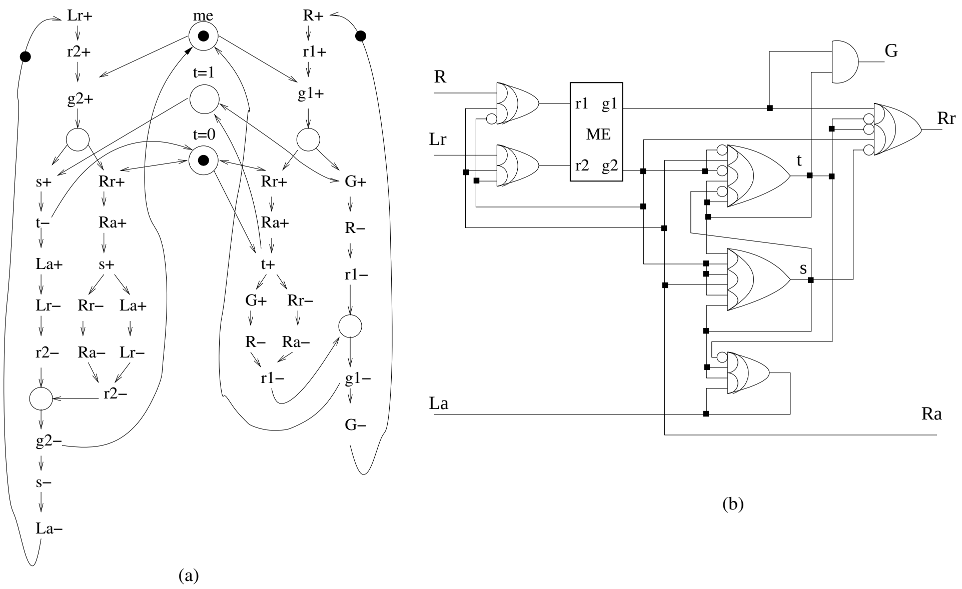

Figure 11: Refined STG and circuit implementation

Lessons to be learnt from the second example:

1. STG models of controllers with arbitration use dynamic *arbitrating choice* (as opposed to free-choice for nondeterministic selection made by the environment) and *mutex places*.
2. Designing controllers with arbitration requires handling *nonpersistent* output signals. Mutex elements must be used for that, which requires inserting mutex events in the STG, and declaring the grants as inputs. Sometimes, for multiple arbitration points, one may decide either to use many mutex elements or to multiplex requests on one mutex.

There are examples where it can be difficult to insert a mutex for signals which may in some places of the algorithm be sampled with arbitration involved while at other times be either sampled without arbitration (stable) or simply ignored. In such cases, the arbitration element (e.g. mutex or synchroniser) should be considered as part of the controlled environment and the STG model constructed as the one with free (nondeterministic, environment) choice, as shown in Figure 13.

Concluding remarks

Other examples of constructing Petri nets and STG for subsequent synthesis using Petrify may be found in the attached references:

- Examples of Petri net and STG models of arbiters [4] and interrupt handlers [6]. They show various ways of inserting mutex elements.
- Deriving a Petri net model from a State Graph specification of a counterflow pipeline stage controller [5]. This example shows many 'tricks' in constructing a synthesizable STG, including refinement of the State Graph to a form which can produce a Petri net (using theory of regions), refinement of the Petri net into an STG with handshake signals, insertion of mutex events, CSC signal insertion.
- Construction of Petri net models for a loadable modulo-N counter [6]. This example shows decomposition of a controller (counter) into small parts which can be managed by synthesis

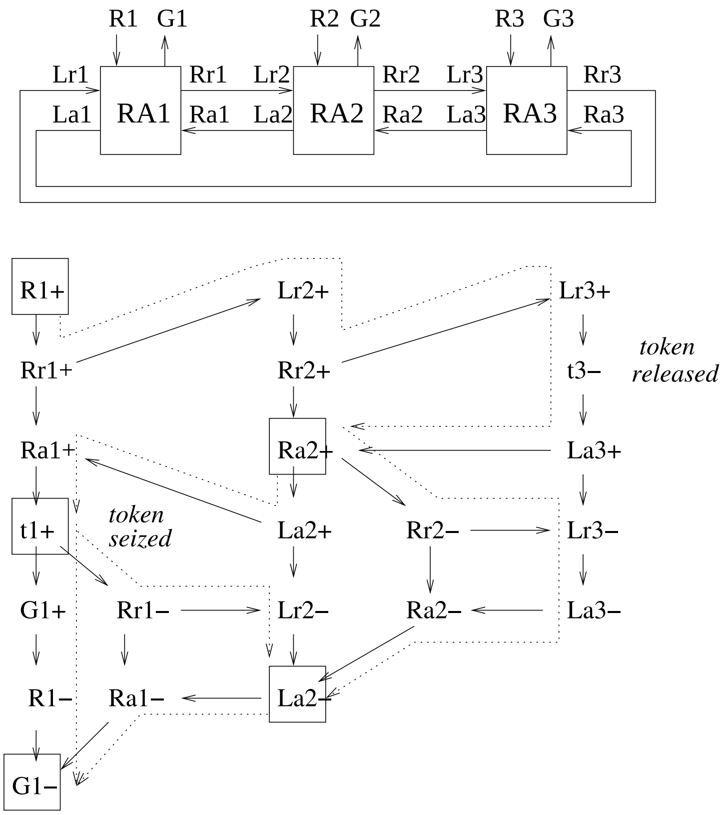

Figure 12: Using partial order (unfolding) for performance analysis

tool. It also shows Petri nets with free-choice and concurrency. It also demonstrates the effect of Petri net unfolding on understanding the behaviour of the counter and analysis of its performance (e.g. response time is constant and independent on the size).

Finally, it is clear that a large class of controllers cannot be efficiently handled at the low (STG) level and with Petrify. These controllers require a different strategy in deriving the implementation (including the problem of solving the Complete State Coding) – namely, the use of direct translation techniques from Petri nets to structures like those based on David cells. We are currently working on these methods and tools. A good benchmark example is a controller for Steve Furber's duplex protocol, for which direct translation techniques prove to be superior.

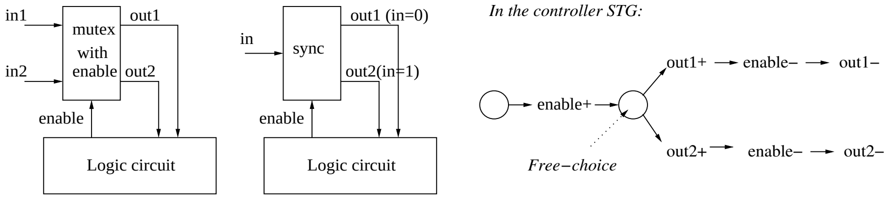

Figure 13: Other ways of factoring out arbitration

References

[1] D.J. Kinniment, B. Gao, A.V. Yakovlev and F. Xia, Towards asynchronous A-D conversion, Proc. 4th Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, March-April 1998, San Diego, CA, IEEE Computer Society Press, pp. 206-215.

[2] K.S.Low and A. Yakovlev, Token Ring Arbiters: an exercise in asynchronous logic design with Petri nets,TR. no. 537, Department of Computing Science, University of Newcastle upon Tyne, November 1995.

[3] J. Cortadella, L. Lavagno, P. Vanbekbergen, and A. Yakovlev. Designing Asynchronous Circuits from Behavioural Specifications with Internal Conflicts, Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, SaltLake City, Nov. 1994, IEEE Comp. Society Press, N.Y., pp. 106-115.

[4] A. Yakovlev, Designing arbiters using Petri nets. Proceedings of the 1995 Israel Workshop on Asynchronous VLSI, Nof Genossar, Israel, March 1995, VLSI Systems Research Center, Technion, Haifa, Israel, pp. 178-201.

[5] A. Yakovlev. Designing Control Logic for Counterflow Pipeline Processor Using Petri nets, Formal Methods in Systems Design (Kluwer), Vol. 12, No.1 (January 1998), pp. 39-71.

[6] A. Yakovlev, Solving ACiD-WG design problems with Petri net based methods. Proc. ESPRIT ACiD-WG Workshop on Asynchronous Circuit Design, Groningen, Sept. 9-10, 1996, TR CSN9602, Computer Science Notes Series, University of Groningen.