DESIGN OF DIVERGENCE-FREE PROTOCOL CONVERTERS USING SUPERVISORY CONTROL TECHNIQUES

Hesham Hallal[1], Radu Negulescu[1], Alexandre Petrenko[2]

[1]Dept. Electrical and Computer Eng., McGill University, 3480 University St., Montreal, Canada H3A 2A7.
[2]Centre de Recherche Informatique de Montreal (CRIM), 550 Sherbrooke St., Montreal, Canada H3A 1B9.

Abstract: In this paper, we use supervisory control techniques to design a protocol converter, which avoids divergence in a communication system. A typical application is to design gateways that interface heterogeneous computer networks while minimizing the communication time with the involved parties. As an example, we design a divergence-free converter that interfaces an alternating bit sender and a non-sequenced receiver, and we implement the converter as an asynchronous circuit.

1. INTRODUCTION

The growing demand for interconnecting heterogeneous computer networks motivates the design of gateway devices that convert signaling of one communication protocol into another. Several techniques, see for instance [1, 5, 9, 10], have been devised to solve the protocol conversion problem using the supervisory control theory [11], which derives a specification for a missing part of a system from the service specification of that system and the known part of its implementation.

One of the important problems in the area of the protocol conversion is the avoidance of divergence, i.e., cycles of internal transitions in the communication system. In gateway circuitry, an example of divergence is the switching activities due to clock transitions in a circuit, while the circuit is idle. Switching activities usually increase power consumption which is often undesirable, e.g., for CMOS implementations in wireless communication applications.

In this paper, we summarize the results of our study of solving supervisory control problems [3] with the application to the protocol conversion problem. For details and proofs, the reader is referred to [3]. Our technique allows us to determine a specification for the gateway device from the specifications of the two protocols involved, and of the service that they should provide when combined. The resulting specification is in the form of a state graph, from which one can derive an implementation using automated techniques. For example, the techniques in [2, 4] can be used to produce an asynchronous circuit from a state graph.

We illustrate the technique on a case study, involving an alternating bit sender and a non-sequenced receiver.

2. PROBLEM DEFINITION

The protocol conversion problem can be formulated in the framework of supervisory control as follows. Given the specifications of a sender, a receiver, and a composite system that combines the two, find a converter that:

(a) interfaces the sender to the receiver and

(b) respects the specification of the composite system when combined with the sender and the receiver. Following [9], this formulation may be called an "asynchronous equation".

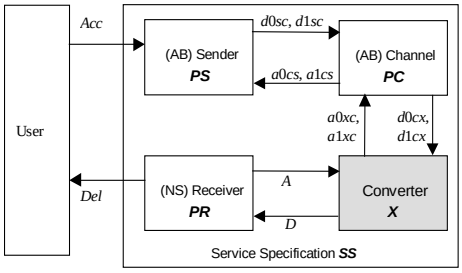

Figure 1: Block diagram of the interconnected system.

As an example, we consider the problem of designing a protocol converter to interface two heterogeneous entities: an *Alternating-Bit* (AB) sender and a *Non-Sequenced* (NS) receiver [3]. This problem is adapted from [5], with slight changes in the models involved. Figure 1 shows the block diagram of the composite system. Each entity is represented by a rectangle with incoming and outgoing labeled arrows to indicate inputs and outputs, respectively. The sender consists of an AB "protocol sender" (*PS*) and an AB "protocol channel" (*PC*). Meanwhile, the receiving part includes a NS "protocol receiver" *PR*. The converter *X* must interface the two mismatched protocols and guarantee that its composition with *PS*, *PC*, and *PR* refines, the "service specification" (*SS*) of the composite system. The events *Acc* (*Accept*) and *Del* (*Deliver*) represent the interface of the communication system with its environment (the user).

3. PRELIMINARIES

In [3] we have studied the protocol conversion problem on specifications in two different formalisms: I/O

0-7803-6542-9/00/$10.00 © 2000 IEEE

automata [6] and process spaces [7, 8]. Also, in [3] we have shown that the results can be transferred between the two formalisms by property-preserving mappings.

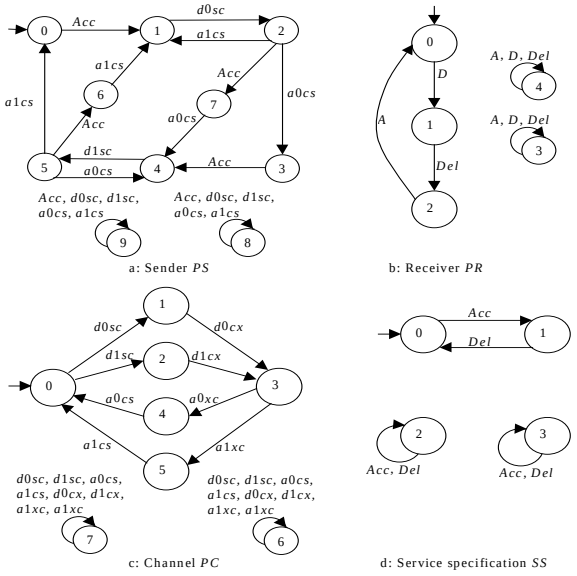

Figure 2 : The process automata *PS*, *PR*, *PC*, and *SS*. (Missing input transitions in *PS*, *PR*, *PC*, and *SS*, lead to states 8, 3, 6, and 2 respectively. Missing output transitions in *PS*, *PR*, *PC*, and *SS*, lead to states 9, 4, 7, and 3, respectively. External actions to each process automaton produce self-loops at every state of that automaton**.)**

In this paper, however, we only use specifications in the form of process automata, which convey safety properties [3, 8], i.e., they specify which events should not happen during an execution of a system (device and environment). In addition, in order to use concurrency operations, process automata distinguish between events that should be avoided by the devices they represent, and events that need to be avoided by their environments. Process automata are akin to state graphs, which can be used by CAD tools, e.g., Petrify [4], to produce circuit implementations. We focus on a particular class of process automata, called "safety–healthy". These can be obtained from state graphs by completing them with the missing input and output transitions, leading to two additional "violation" states, called reject and escape, respectively, while treating the original states as goal states [7, 8]. In our example, we use process automata to represent *PS*, *PR*, *PC*, and *SS*. The corresponding automata are shown in Figure 2 a, b, c, and d respectively. Circles represent the states, and labeled arrows represent the transitions. An initial state is distinguished by the incoming arrow that points to it. Notice the presence of two "trap" states in each automaton, which indicate violations. For example, state 2 and state 3 are trap states for the specification *SS* in Figure 2 d. The transitions that lead to these states are not shown for clarity, but the complete automaton can be directly constructed by adding the missing transitions, e.g., Figure 3.

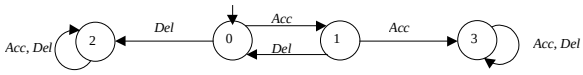

Figure 3: The complete process automaton *SS*.

4. SYNTHESIZING THE CONVERTER

In this section, we describe how to obtain a general solution for the protocol conversion problem. By general, we mean the loosest specification of the protocol converter that meets the given service specification. The algorithm is based on two main operations on concurrent systems. Product expresses the joint behavior of two or more automata in terms of both, the permissible behavior (goals) and the violations (escapes or rejects). Reflection of an automaton, on the other hand, shows the behavior of the environment of that automaton. For detailed definitions of these operations, see [3, 7, 8].

The following is a description of the algorithm applied to our working example:

1. Obtain the process automaton *Known* from the product of all the known modules in the system:
   a) Determine the product of *PS*, *PC*, and *PR*,
   b) Hide *d*0*sc*, *d*1*sc*, *a*0*cs*, and *a*1*cs*. These actions are invisible to the missing converter.
   c) Determinize and minimize the obtained process, since hiding introduces non-determinism in parts of the resulting process automaton *Known*.
2. Determine the product of *Known* with the reflection of *SS*. This yields the environment of the missing converter *X*.
3. Hide the actions, which are irrelevant to the converter. *X* needs not to observe *Acc* and *Del*.
4. Determinize and then minimize the obtained process to avoid the non-determinism that results from hiding.
5. Reflect the result to obtain the solution, *X*.

Generally speaking, applying the hide operation after each composition (Step 1 and 4) has the same effect as hiding the irrelevant actions once in Step 4. However, we chose to execute hiding twice, and determinize twice, in Step 1 and Step 4. This allows us to reduce the computation time of FIREMAPS [3, 7], the tool used to solve the problem.

The resulting process automaton *X* of the general solution is shown in Figure 4. This converter considers the actions *d*0*cx*, *d*1*cx*, *A* as its inputs and *D*, *a*0*xc*, *a*1*xc* as the outputs.

Notice that *X* is non-deterministic in issuing outputs. For example, when a message (*d*0*cx*) arrives, the converter randomly chooses between acknowledging the sender through the channel (*a*0*xc*) and delivering the message to the receiver (*D*). This non-determinism

is usually unnecessary and can be resolved to obtain a deterministic converter using various heuristics.

In addition, *X* has several states, which indicate violations. One trap state, 4, is reached by the traces including illegal inputs, e.g., (*d*0*cx*, *d*0*cx*). On the other hand, illegal outputs lead to the escape states: 11, 5, and 15. These states are reached by traces like (*d*0*cx*, *a*0*xc*, *a*0*xc*). However, the successors of these states are not all escape states. After reaching an escape state, inputs might lead to state 4, and outputs lead to other escape states. This means that the automaton is not safety-healthy because the escape set (of traces) produced is not suffix-closed [3].

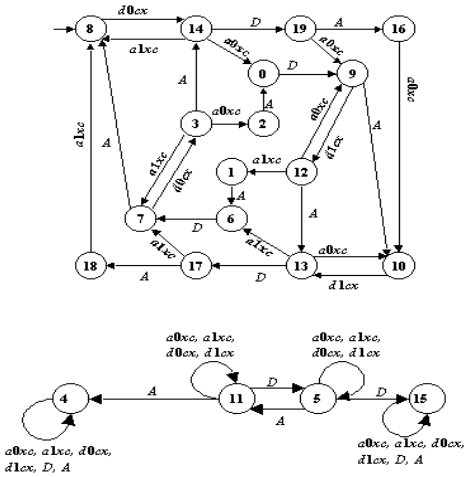

Figure 4: The process automaton *X*.

The solutions obtained by our method are always process automata that convey safety information. However, in some cases as is for our converter, the automata obtained can not be directly used to implement the corresponding circuits. In other cases, though not present in this example, illegal inputs may lead to escapes (escape-inputs) and illegal outputs to rejects (reject-outputs) [3].

In the following, we describe an algorithm that refines process automata to safety-healthy process automata; i.e., it yields a process automaton, which refines the original automaton. This is a general algorithm that can be applied independently from asynchronous equations. We call it the "upgrade algorithm", and it includes the following three steps [3]:

1. Trace back all the executions that end with an escape-input until an output is reached. Then, mark as escapes all states that are reachable by issuing the latter output.
2. Mark as escapes all states that are reachable by issuing a reject-output.
3. Mark as escapes all the successors of any escape state.

In the case of the converter *X* in Figure 4, we only need to apply step 3 to close the escape set. This will cause the three escape states (11, 5, and 15) to become equivalent. Hence they are merged into one escape state. On the other hand, the upgrade algorithm allows us to detect the absence of a solution to the supervisory control problem. This is indicated by yielding an empty solution, which consists of a single escape state.

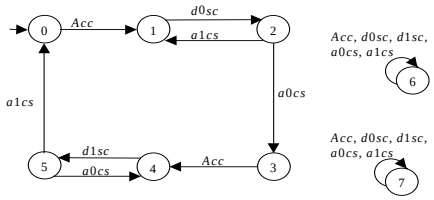

Figure 5: The modified specification of *PS*.

To illustrate, we use the automaton of Figure 5 to replace the specification of the sender *PS* in Figure 2 a. With the new specification, the sender rejects two consecutive *Acc* signals even if they are interleaved with *DEL*, which is an acceptable behavior for the user. The solution obtained is not safety-healthy, and it is reduced to a single escape state by the upgrade algorithm.

5. DIVERGENCE FREEDOM

In the previous section, we described how to obtain a solution for the protocol conversion problem constrained only by the service specification. This solution, however, does not restrict internal communications, which might lead to divergence. For example, sequences like *d*0*cx a*1*xc*, (see Figure 4) might be executed infinitely many times before the communication system can respond to its environment.

In this section, we describe how to design a divergence-free protocol converter. Such a converter avoids unbounded communications with other parts in the system while satisfying the service specification. We aim to avoid both "catastrophic" divergence which leads to unfair behavior and "non catastrophic" divergence which leads to undue power consumption. We use the same procedure from the previous section. However, we augment the service specification by combining *SS* with an auxiliary process automaton, $limit_n$, where *n* is a positive integer [3].

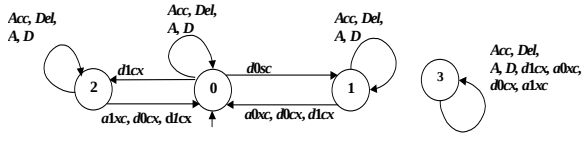

Figure 6: The process automaton $limit_1$.

The automaton $limit_n$ monitors the actions on the boundaries of the converter, and prevents them from causing divergence. For example, when $limit_n$ observes the sequence *d0cx a1xc*, it signals a violation and enters

a trap state. Figure 6 shows the process automaton $limit_1$, which forbids the converter from issuing negative acknowledgments to the messages it receives because they might lead to cycles. Notice that state 3 is an escape state from which the process cannot resume normal behavior.

As a result only Step 2 of the original algorithm needs to be modified. It becomes as follows:

2. Determine the product of *Known* with the reflection of the modified specification SS_n obtained as the product of *SS* with $limit_n$.

The divergence-free converter X_1 is shown, after upgrade, in Figure 7. The execution (*d0cx, a1xc*), for example, does not cause a cycle anymore. It leads to the escape state 33.

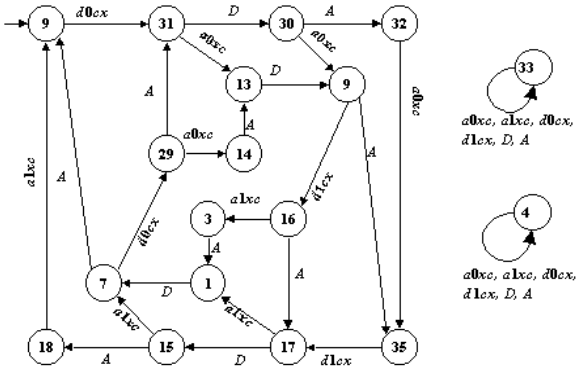

Figure 7: The automaton of the divergence-free solution.

Finally, we derive an asynchronous implementation of our converter using Petrify [4]. For this, we use a deterministic refinement of the automaton shown in Figure 7. In fact, it executes only the sequence *d*0*cx*, *a*0*xc*, *D*, *A*, *d*1*cx*, *a*1*xc*, *D*, *A*. The resulting implementation consists of 2 wires and one XOR gate, and can be represented by the following logic functions:

[*D*] = *d*0*cx d*1*cx*' + *d*0*cx*' *d*1*cx*.
[*a*0*xc*] = *A*' *d*1*cx* + *A d*0*cx*.
[*a*1*xc*] = *A*' *a*0*xc* + *A a*1*xc*.

6. CONCLUSION

In this paper, we have described how to solve asynchronous equations to obtain specifications for protocol converters that can then be synthesized using automated methods. We illustrated our technique using an example from [5]. A typical application is the synthesis of low power asynchronous converters from state graphs, using techniques from [2, 4]. A state graph can be obtained by producing a safety-healthy process automaton. In contrast to [5], our approach deals with divergence freedom and can indicate the absence of a solution for some specifications.

In contrast to previous attempts to compute divergence-free solutions to the asynchronous equation problem [9, 10], we insert a limiting process in the specification itself. In other words, we apply a general technique for deriving supervisory control on an altered specification that includes a divergence-freedom requirement in the form of a distinct process. A benefit entailed is uniformity, which means that both the theoretical underpinnings and the tools can be reused for other problems. The algorithms from Section 4 and 5 have already been implemented, with the exception of the upgrade algorithm of Section 4, which is left for future work.

7. ACKNOWLEDGMENTS

We are grateful for the financial support from CRIM and NSERC grants OGP0194381 and RGPIN217338.

8. REFERENCES

[1] J. Drissi and G. Bochmann. *Submodule Construction for Systems of I/O Automata*. Technical Report no. 1133, Department d'Informatique et de Recherche Operationelle, University of Montreal, 1999.

[2] R-D. Chen, J-M. Jou and Y-H. Shiau. "An Efficient Method for the Decomposition and Resynthesis of Speed-Independence Circuits". *In proc. International Conference on Electronics, Circuits, and Systems*, Cyprus, 1999, pp.339-342.

[3] H. Hallal, R Negulescu, and A. Petrenko. *Supervisory Control with Divergence Freedom in Two Safety Models*. Internal Report CRIM-00-01-02 Centre de Recherche Informatique de Montreal. Montreal, Canada.

[4] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. "Technology Mapping for speed-independent circuits: Decomposition and Resynthesis". *In proc. Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997, pp. 240-253.

[5] R. Kumar, S. Nelvagal, and S. Marcus. "Protocol Conversion Using Supervisory Control Techniques". *In Proc. of the 1996 IEEE International Symposium on Computer-Aided Control System Design*, 1996, pp. 32-37

[6] N. Lynch and M. Tuttle. *An Introduction to Input/Output Automata.* Laboratory for Computer Science. Massachusetts Institute of Technology, November 1988.

[7] R. Negulescu. "Process Spaces". *In Proc. of the 11th International Conference on Concurrency Theory (CONCUR 2000).* Pennsylvania, USA. Aug, 2000.

[8] R. Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. Ph.D. Thesis, Department of Computer Science, University of Waterloo, August 1998.

[9] A. Overkamp. "Supervisory Control Using Failure Semantics and Partial Specifications". *IEEE transactions on Automatic Control*, Vol. 42, No. 4, April 1997.

[10] A. Petrenko and N. Yevtushenko. "Solving Asynchronous Equations". *In Proc. of FORTE-PSTV'98.* Paris, France.

[11] P. J. Ramadge and W. M. Wonham. "The Control of Discrete Event Systems". In Proc. IEEE, vol. 77, pp. 81-98, Jan. 1989.