Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 203 (2009) 121–135
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Executable Contracts for Incremental Prototypes of Embedded Systems

Lionel Morel[1]

INSA Lyon, LIESP, Villeurbanne, F-69621

Louis Mandel[2]

LRI, Université Paris-Sud 11, CNRS, Orsay F-91405 INRIA Futurs, ProVal, Orsay, F-91893

Abstract

In this paper, we advocate for a seamless design-flow for embedded reactive programs. We particularly concentrate on the use of assume-guarantee contracts (as a form of non-deterministic specification) and present how these can be used for early execution of reactive specifications. We illustrate the approach on a case-study taken from an avionic application, trying to show the implications of this simulation method on the design-flow.

Keywords: Reactive systems, assume-guarantee contracts, early execution, embedded systems

1 Introduction

Reactive systems and the synchronous approach

Reactive systems, as defined in [11], are characterized by the interaction with their environment being the prominent aspect of their behavior. Software embedded in aircraft, nuclear plants or similar physical environments, is a typical example. They interact with a non-collaborative environment, which may impose its own rhythm: it does not wait, nor re-issue events. Synchronous languages [2] represent an important contribution to the programming of reactive systems. They are all based on the *synchronous hypothesis* that establishes that communications between different components of a system are instantaneous and, more importantly, that computations performed by components are seen as instantaneous from their

⋆ This work was partly carried while both authors were working at Verimag, Gières, France.

[1] Email: lionel.morel@insa-lyon.fr

[2] Email: louis.mandel@lri.fr

doi:10.1016/j.entcs.2009.03.030

environment's point of view. Among these languages, the most significant ones are Esterel [3], Lustre [8] and Signal [15]. They offer a strong formal semantics and associated validation tools, and are now commonly used in highly-critical industry for the design of control systems.

Prototyping and simulation

In software engineering in general, an answer to the increasing complexity of systems has been the definition of complete design flows. These describe the different steps in the development of a system, and associate to each of these steps some informal or formal method to help validate the system as it is developed. An interesting approach consists in following a continuous path from the early prototypes towards the final product. Each new step in such a design flow can be automated, or at least some form of consistency can be checked between successive versions. An important validation feature of such methodologies is the use of simulation techniques, by which a developer can observe the possible behaviors of the system in its current status of development, without having a full implementation yet at hand.

Seamless design flow and incremental validation

In this work, we aim at proposing a formally-based seamless design flow for reactive systems. The idea is to be able to start validating a system as soon as possible during its development life. By validating, we mean simulating, testing or verifying, although in the work presented here we are concentrating on simulation aspects. As far as simulation or test are concerned, all we can do for the moment is test the system when it is fully implemented (i.e. when it describes a fully deterministic behavior [14]). This approach is the one classically adopted for *black-box testing*. There, we assume the existence of an *executable version* of the system under test, and have a non-deterministic description of the environment. We may also provide observers for checking that the system satisfies desired properties $\mathcal{P}$ as long as the environment satisfies the system's assumptions $\mathcal{H}$. That sort of "*testing*" can only occur at the far end of the development process, by using the specification of the environment to provide the system with inputs, and observing how the system reacts against these inputs. Moreover, this is a *monolithic* approach: all we have to perform the tests on is an executable version of the whole program. Of course, one could also perform some unit tests on isolated components, even if the whole system is not implemented yet. But this gives only a partial (even very small) understanding of the system. Moreover, one would need to specify a meaningful environment to each of these components. In earlier work [19], we advocate for the use of local specifications (by mean of assume-guarantee contracts) for helping the description of systems during their development. Contracts can be used as a progressive description method: the programmer might first of all describe the global structure of the system. Then he can describe "how components behave" as far as their own environment is concerned, by writing a contract for each of them. By gradual refinement, he might make more precise the description of each component's behavior, ending the development process by giving a deterministic implementation. These contracts can also help the global understanding of the system via some verification rules that are used to delegate proof objectives to validation tools [18].

Contracts for Early Simulation

As far as simulation is concerned, the introduction of contracts in the design methodology has the following implications on: 1) *modularity.* Once the programmer has provided a component with a contract, this contract can be seen as a specification of the component's environment, and can thus be used for testing the component alone, without depending on the implementation of the whole system. Here, the standard test approach proposed in [22] can be used with the advantage of the locality of the testing (the whole system need not be implemented entirely in order to test one of its components); 2) *early simulation of non-fully implemented systems.* Since contracts are given in an early stage of development, it is possible to simulate the system without waiting for a complete implementation to be designed. Here, we aim at providing a simulation framework that follows the latter view. The method is based on generation of actual code for deterministic components and connection to a constraint solver (as used in classic test approaches) for simulating assume-guarantee contracts.

Content of the paper

Section 2 presents a subset of the Lustre language that we will use for describing our running example. Section 3 introduces a system example. Through this we illustrate the progressive design-flow that we introduced earlier. Starting from the general description of the architecture of the application in 3.1, we gradually refine parts of the system by giving contracts and final deterministic implementations of some components (3.2). Section 4 introduces a formalization of components and describes the approach we propose for simulating networks of deterministic and non-deterministic components. Section 5 describes a simple implementation of this framework that uses existing tools and discusses the practical problems raised. Section 6 discusses related works while section 7 concludes and introduces possible future work.

2 Mini-lustre

We first define a language for programming reactive systems. It is a light version of the Lustre language [8]. Programs manipulate lists of variables. Given two such lists $\mathcal{V}$ and $\mathcal{V}'$, we use $\mathcal{V}\&\mathcal{V}'$ as the concatenation of $\mathcal{V}$ and $\mathcal{V}'$. We will use this for describing our running example.

Programs

A mini-Lustre program is a tuple $N = (\mathcal{V}^i, \mathcal{V}^o, \mathcal{V}^l, \mathcal{F})$:

- $\mathcal{V}^i$, $\mathcal{V}^o$ and $\mathcal{V}^l$ are pairwise disjoint lists of variables for inputs, outputs and local variables. They take their values in a domain $\mathcal{D}$;
- $\mathcal{F}$ is a total function from $\mathcal{V}^o\&\mathcal{V}^l$ to $Exp(\mathcal{V}^i\&\mathcal{V}^o\&\mathcal{V}^l)$, where $Exp(\mathcal{V})$ is the set of expressions over variables in $\mathcal{V}$, defined by the following grammar: $e ::= c|x|op(e,...,e)|pre(i,x)|N(x,...,x)$. c represents constants on $\mathcal{D}$; x represents the name of a variable in $\mathcal{V}^i\&\mathcal{V}^o\&\mathcal{V}^l$; op represents for any combinatorial operator; $pre(i,x)$ stands for the preceding value of variable x (where the constant i is the value of the expression at the initial instant of the execution); $N(x,...,x)$ represents a call to the program N with variable names as effective parameters corresponding to the inputs of N. The usual concrete syntax, used for the example is the one given in fig. 1.

```
node N (<list V^i of inputs with types>)
returns (<list V^o of outputs with types>)
var <list V^l of local variables with types>
let
  - - for each x, local or output:
  x = F(x);
tel.
```

Fig. 1. Usual concrete syntax of a Lustre node.

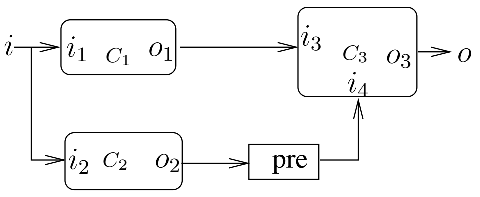

Fig. 2. An example of data-flow network.

In the example, we will also use specific Lustre operators, called *iterators*, introduced in [21], that can be seen as limited higher-order operators. The syntax $map \ll N, n \gg$ defines a mapping operation of a node N that is applied on n-elements arrays. $red \ll N, n \gg$ defines a reduction (similar to *fold* in functional languages) that iterates the node on n-elements arrays . In the current work, we only use these iterators for making the presentation of the example clearer.

Trace semantics

Each variable name v in a mini-Lustre program describes a flow of values of its type, i.e. an infinite sequence $v_0, v_1, ...$ Given a sequence of inputs, i.e. the values v_n , for each $v \in \mathcal{V}^i$, and each $n \geq 0$, we describe how to compute the sequences (or traces) of local and output flows of the program: for all instants, the value of an output or local variable v is computed according to its definition as given by $\mathcal{F}$:

$$\forall n > 0. \forall v \in \mathcal{V}^o\&\mathcal{V}^l. \forall x \in \mathcal{V}^i\&\mathcal{V}^o\&\mathcal{V}^l.$$
$$v_n = \mathcal{F}(v)[x_n/x][x_{n-1}/pre(i,x)] \quad \text{and} \quad v_0 = \mathcal{F}(v)[x_0/x][i/pre(i,x)].$$

We take the expression $\mathcal{F}(v)$, in which we replace each variable name x by its current value x_n, and each occurrence of $pre(i,x)$ by the previous value x_{n-1} (except in the initial instant of execution where we replace it by the value of initialization expression associated, i). This yields an expression in which operators are applied to constants. The set of equations we obtain for defining the values of all the flows over time must be acyclic.

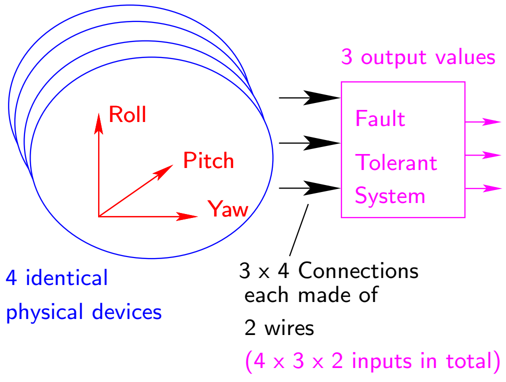

Fig. 3. The gyroscopic system with its environment.

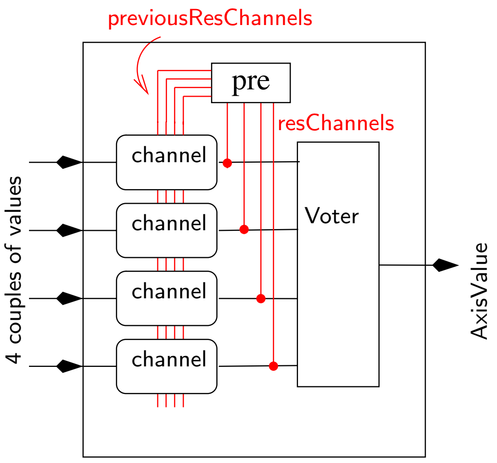

Fig. 4. Structure of one axis.

Synchronous composition

Components are arranged in a data-flow network such as the one of fig. 2. There exist two kinds of connections between components: *combinatorial* (e.g. between i_3 and o_1) dependencies, made explicit by the "*wire*" connections and *non-combinatorial* dependencies (e.g. between i_4 and o_2) using the pre operator provided in the language. We require that any circular dependency between the inputs and the outputs of a same component is broken by a pre operator. This constraint allows for simpler analysis techniques (e.g. used as a standard in the SCADE tool) and separate compilation of Lustre nodes.

3 Example

3.1 General presentation

Throughout the rest of the paper, we will use a common example to serve our purpose. This application is taken from the avionics industry and deals with the treatment of position variations of an airplane. Figure 3 describes the system and its physical environment. The system is connected to four gyroscopes, each of them measuring the angle variations along the three axes named *roll*, *pitch* and *yaw*. The values obtained for one axis by each physical device are transmitted to the computing system along two wires vA and vB. Hence the system receives $4 \times 3 \times 2$ values. From these 24 values, it computes three *secure values* that are intended for the rest of the command system of the airplane as *references* of the position variation of the airplane.

The behavior of the three axes is the same. Let us concentrate on one axis only, say roll. The internal structure of the system is as follows (see fig. 4): it is made of four channel components each of them being in charge of the two wires vA and vB (representing the same physical value) that come from one of the four gyroscopes. Each channel delivers one output value defined by the roll value and the local failure status. A voter computes one signal value (AxisValue) out of four (resChannel), depending on current fault conditions: the value is the average of the values transmitted by the channels that do not declare themselves "failed". Three or more channels having failed at the same time is supposed to have a very low probability. This is determined by traditional fault-tolerance analysis that are not the subject of this paper.

The system handles two kinds of faults: *link faults*, that are due to some bad behavior of a physical link between the measurement devices and the computer; and *sensor faults*, that are due to the measurement devices (sensors) themselves being broken or not working properly for some time. Each channel compares the values it receives on the two wires (vA and vB), and is able to detect local discrepancies. This dual transmission of values from one gyroscope to the computer system is there to detect link faults. For an error to be reported, the two values have to differ of more than Δ_v, during more than Δ_t units of time. Moreover, in order to support sensor faults, channels talk to each other and exchange values, so that each of them can compare its own value to the three other ones. If one of the gyroscopes is not working, the value it delivers will probably differ from the values given by the three other devices. Channels also have to exchange their failure status, because each one should compare its value to the values of the other channels, but only those *that do not declare themselves* failed (recall a channel may declare itself failed, due to the *local* comparisons it performs).

Of course, recovery after detection of faults should be taken care of in the description of the system. But this is not the topic of the presented work and we will not go into the details of this.

3.2 The Design-flow in practice

We propose a design-flow that we roughly decompose into three steps: architectural design, description of local specifications and implementation of components.

Architectural design

During the first step, the user defines the global architecture of the application, i.e. the components input/output interfaces and their connection. This is exactly what we have done in 3.1. For example, we gave the following LUSTRE interface to the component channel:

```
node channel(previousResChannel: Valid_ChannelT^4;
             nbInChannel: int; inChannel: Faulty_ChannelT)
returns (nextResChannel: Valid_ChannelT^4; outChannel: Valid_ChannelT);
```

Local specifications

The second design step, is achieved by designing assume-guarantee contracts [19] for each component.

A contract is a couple (A, G) that can be used to describe a partial behavior of a component. A is used to encode an *assumption* that the component has on its environment. G is used to encode a property that the component *guarantees* about its behavior. In practice, both the assumption and the guarantee can be expressed by means of synchronous observers [9]. A relates to inputs of the component, and G relates to both inputs and outputs of the component.

Our Channel component has the contract defined by the nodes assumeChannel and guaranteeChannel (see fig. 5). Each channel assumes that at most 1 other channel reports local discrepancies and that the channels that work properly provide a value that is not far from some *ideal* value. When put in an environment satisfying this assertion, the channel guarantees some temporal property stating that: "*either a local discrepancy is detected (which means the two values it receives as inputs are too different for a period of time too long), or there is no local discrepancy detected and the current output value is not too far from the two values it receives as inputs.*" In the program of fig. 5 , this behavior is represented by the boolean variable normalProp.

Note that we also need an initialization property, because the guarantee depends on some past values that are not available during the first instants of the execution. The global guarantee of the channel is encoded by a call to node _during_then_(initProp, TIME, normalProp) which states that initProp holds during the first TIME instants (TIME is a constant) and then normalProp holds for the rest of the execution. A contract for the Voter can also be designed, but we will not give it here, for obvious space reasons.

Environment

We also give a description of the environment of the whole application to be able to do simulations. The simplest environment env takes as inputs the AxisValue computed by the Voter and gives some channel inputs back to the channels. Describing such an environment is a complex part of the application design, but we will not concentrate on it here. We will assume that such a description is given in the form e.g. of a LUCKY automaton [13] that randomly chooses channel inputs that satisfy the desired specification for the environment.

Implementation

This last step is done by writing actual LUSTRE components that deterministically define their outputs (in contrast with the non-determinacy implied by contracts). The Voter (fig. 5) actually computes an average of the values provided by those channels that do not declare local discrepancies.

4 Simulation methodology

We now give a language-independent formal definition of reactive components. We start by presenting a formal semantics of components encompassing both deterministic components and assume-guarantee contracts. We then give the general idea of our simulation method.

```
node assumeChannel(-- inputs of the channel component
                   previousResChannel: Valid_ChannelT^4; nbInChannel: int;
                   inChannel: Faulty_ChannelT)
returns (assumeOK: bool);
var unfailedChannels_notFarFromIdeal: bool;
let
  assumeOK = NbOtherFailures(previousResChannel, nbInChannel)<3
             and unfailedChannels_notFarFromIdeal;
  unfailedChannels_notFarFromIdeal = NotFarFromIdeal(previousResChannel,
                                     nbInChannel, ideal, delta_to_ideal);
tel

node guaranteeChannel(-- inputs of the channel component
                      previousResChannel: Valid_ChannelT^4; nbInChannel: int;
                      inChannel: Faulty_ChannelT;
                      -- outputs of the channel component
                      nextResChannel: Valid_ChannelT^4; outChannel: Valid_ChannelT)
returns (guaranteeOK: bool);
var lastValidChannel: Faulty_ChannelT;initProp, normalProp: bool;
let
  normalProp = outChannel.local_failure
               or ((abs((lastValidChannel.valuea - outChannel.local_value)) <= delta)
               and (abs((lastValidChannel.valueb - outChannel.local_value)) <= delta));
  initProp = outChannel.local_failure or ((abs((inChannel.valuea - outChannel.local_value)) <= delta)
             and (abs((inChannel.valueb - outChannel.local_value)) <= delta));
  lastValidChannel = lastValid(inChannel,delta);
  guaranteeOK = _during_then_(initProp, TIME, normalProp);
tel

node Voter(resChannels: Valid_ChannelT^4)
returns (vote: int); var sum, nbValid: int;
let
  nbValid = countValidChannels(resChannels);
  sum = red<<addIfValid;4>>(0.0,resChannels);
  vote = (globalSum / nbValid);
tel
```

Fig. 5. Definition of the contract of the channel component and of the implementation of the voter. Note that delta, ideal and delta_to_ideal are global constants.

4.1 Variables, valuations, traces, components

We start with a set of variables $\mathcal{V}$, taking their values in a domain $\mathcal{D}$, intended to represent the state of a dynamical system. The semantics of a reactive system is given in terms of step-relations. Intuitively, a step-relation relates two valuations of $\mathcal{V}$, that are intended to represent the valuations at two successive instants in time. Then we define components on top of step-relations. Variables may be typed. However, we forget about the typing mechanism, which is necessarily particular to the language in which step-relations are described.

Definition 4.1 A *valuation* σ of $\mathcal{V}$ is a total function from $\mathcal{V}$ to $\mathcal{D}$; Let us denote by $\mathsf{Vals}(\mathcal{V},\mathcal{D})$ the set of all such valuations. Then a *step-relation* on $\mathcal{V}$ relates valuations, hence it is a subset of $\mathsf{Vals}(\mathcal{V},\mathcal{D}) \times \mathsf{Vals}(\mathcal{V},\mathcal{D})$. The set of all step-relations on $\mathcal{V}$ is $\mathsf{Step\text{-}Rels}(\mathcal{V},\mathcal{D}) = \mathcal{P}(\mathsf{Vals}(\mathcal{V},\mathcal{D}) \times \mathsf{Vals}(\mathcal{V},\mathcal{D}))$.

Definition 4.2 A step-relation $R \in \mathsf{Step\text{-}Rels}(\mathcal{V},\mathcal{D})$ is said to be *deterministic* (resp. *reactive*) iff:

$$\forall\sigma \in \mathsf{Vals}(\mathcal{V},\mathcal{D}).\, |\{\sigma' \text{ such that } (\sigma,\sigma') \in R\}| \leq 1 \text{ (resp. } \geq 1)$$

Definition 4.3 $\mathsf{Traces}(\mathcal{V})$ is the set of all possible *traces* on variables of $\mathcal{V}$. If $t \in \mathsf{Traces}(\mathcal{V})$, we note t_i the i-th element of t, i.e. the value of the variables of $\mathcal{V}$ at the i-th instant of execution, according to t.

If $t \in \mathsf{Traces}(\mathcal{V})$ and $\mathcal{V}' \subset \mathcal{V}$, then the trace $t' \in \mathsf{Traces}(\mathcal{V}')$ is the projection of t over $\mathcal{V}'$ ($t' = t \downarrow_{\mathcal{V}'}$) if and only if $|t| = |t'|$ and $\forall x \in \mathcal{V}'$, $\forall n$, $t_n(x) = t'_n(x)$. This operation can be lifted to the set of traces.

Definition 4.4 A *behavior* $B \in \mathsf{Behaviors}(\mathcal{V})\mathcal{D}$ is a tuple $(\mathcal{V}, \mathcal{V}^l, \sigma, R)$ where $\mathcal{V}$ is a set of variables which represents the interface of the behavior. $\mathcal{V}^l$ is the set of local variables such that $\mathcal{V}$ and $\mathcal{V}^l$ are pairwise disjoint. $\sigma \in \mathsf{Vals}(\mathcal{V}\&\mathcal{V}^l, \mathcal{D})$ is the initial valuation of the variables and $R \in \mathsf{Step\text{-}Rels}(\mathcal{V}\&\mathcal{V}^l, \mathcal{D})$ is a step-relation describing the behavior after its initialization.

Definition 4.5 The *semantics of a behavior* $B = (\mathcal{V},\ \mathcal{V}^l,\ \sigma, R)$ in terms of set of traces over the variables of V is defined by:

$$\mathcal{T}(B) = \{t \in \mathsf{Traces}(\mathcal{V}\&\mathcal{V}^l) \mid t_0 = \sigma \ \wedge\ \forall n.\ 0 < n < |t| \ \wedge\ (t_{(n-1)}, t_n) \ \in\ R\} \downarrow_{\mathcal{V}}$$

Now we can define a notion of component, that will encompass assume-guarantee contracts.

Definition 4.6 A *component* C is a tuple $(\mathcal{V}^i, \mathcal{V}^o,\ B_a,\ B_g)$ where:

- $\mathcal{V}^i$ and $\mathcal{V}^o$ are the sets (pairwise disjoint) of inputs and outputs of the component;
- $B_a \in \mathsf{Behaviors}(\mathcal{V}^i)\mathcal{D}$ (resp. $B_g \in \mathsf{Behaviors}(\mathcal{V}^i\&\mathcal{V}^o)\mathcal{D}$) is a description of the assertion (resp. guarantee) of the component.

Definition 4.7 The *semantics of a component* described by its contract is given as the set of traces $\mathcal{T}(C)$, defined as:

$$\mathcal{T}(C) = \{t \in \mathsf{Traces}(\mathcal{V}^i\&\mathcal{V}^o) \mid t \downarrow_{\mathcal{V}^i} \in \mathcal{T}(B_a) \Rightarrow t \in \mathcal{T}(B_g)\}$$

A behavior $B \in \mathsf{Behaviors}(\mathcal{V}^i\&\mathcal{V}^o)\mathcal{D}$ is an implementation of a component $C = (\mathcal{V}^i, \mathcal{V}^o, B_a, B_g)$ if
$\mathcal{T}(B) \subseteq \mathcal{T}(C)$.

4.2 Executing components

The previous semantics gives us a denotational characterization of our deterministic components, as well as of assume-guarantee contracts. From a simulation point-of-view, we need a way to describe operationally the generation of traces of values that satisfy these specifications. Practically, we need to be able to describe a component C by some functions $init_C()$, $computeOut_C()$ and $updateState_C()$ that can be used to build those traces. The exact implementation of these functions depends on the form of each component.

For deterministic components, they are obtained by classical compilation of the source code. From a global point of view, however, we need to ensure separate compilations of components. For that, we require, as explained before, that all dependency cycles in the source code be broken by pre.

From assume-guarantee contracts, this correspondence is less trivial. The algorithm of the $computeOut_C()$ function is given figure 6. The assertion can be seen as a constraint that should be checked at each instant on input values. We thus define a boolean function $check_{\mathsf{Assert}}()$ that takes current values of the input variables of the component and checks whether those values satisfy the constraint it represents. This test function can be compiled out and used whenever the component is to be activated. The non-deterministic aspect of the assertion is left aside here since it is not used to actually generate values.

If the assertion is violated a warning message should be addressed to the user. Otherwise, if the assertion is satisfied, one can then use the guarantee to generate valid outputs. Now the guarantee part of the contract basically represents a set of constraints φ on the variables of $\mathcal{V}^i \& \mathcal{V}^o$. Realizing the corresponding *computOut*() function means providing a function *choose*() that, given a value for $\mathcal{V}^i$ will provide a value for $\mathcal{V}^o$ by solving the constraint φ. It is of course not possible to *compile* this *choose* function out of every contract. Rather, it is implemented by a constraint solver that is asked to solve φ every time a step is asked from the corresponding contract.

Solving such constraints is not decidable in general. In practice, the *choose* function can be delegated to a test case generation tool that is able to solve constraints mixing boolean and linear relations on numerical variables. Since we require this method to be fully automatic, it naturally requires restrictions on the nature of the numerical constraints to be solved. For example, the Lurette tool we use in our practical experiments (see sec. 5) requires those constraints to be linear: it uses polyhedra to solve these constraints.

The system's environment

The description of the environment is given in the form of a non-deterministic automaton that is interpretable by the test case generation tool mentioned earlier. We can also represent it as a reactive component that provides a *computeOut*() function which realization is implemented as a call to the above-mentioned function *choose*. The environment is thus to be seen as *just another component* from a simulation perspective.

4.3 Executing networks of components

Now, we are able to associate to each component C_j a function $computeOut_{Cj}()$ and a function $updateMem_{Cj}()$. Whether the component is deterministic or not, there is no difference in how we consider it in the writing of the simulation algorithm. The only difference, as noted earlier, is in the way the *computeOut* will be implemented.

Our goal here is to be able to simulate networks of components such as the one of fig. 2. The problem to solve is determining the order of execution of components in such a network. This is done by following the method proposed in [10]. For the sake of presentation, we assume that this order is the natural order on the indexes of the components: C_1 gets executed first, then C_2, etc. Then the corresponding simple simulation algorithm is shown in fig. 7.

Applying this simulation technique to our running example will yield traces like

```
if $check_{\mathsf{Assert}}()$
then $choose()$;
else $EmitWarning()$;
```

Fig. 6. *ComputeOut* function of a (A, G) component.

```
for $j$ in $1..nbComponents$
  $init_{C_j}()$;
while(true)
  for $j$ in $1..nbComponents$
    $computeOut_{C_j}()$;
  for $j$ in $1..nbComponents$
    $UpdateState_{C_j}()$;
```

Fig. 7. Simulation algorithm.

inChannel[1].vA	...	5.38	5.25	4.77	...
inChannel[1].vB	...	5.47	4.53	4.71	...
⋮					
inChannel[4].vA	...	5.24	5.28	4.94	...
inChannel[4].vB	...	4.65	4.81	5.43	...
resChannel[1]	...	<ff,5.41>	<ff,5.20>	<ff,4.73>	...
⋮					
resChannel[4]	...	<ff,5.21>	<ff,5.20>	<ff,4.98>	...
vote	...	5.16	5.14	4.95	...

Fig. 8. Possible traces for the gyroscope application.

the one of figure 8.

5 Implementation

To illustrate these propositions, we have implemented the example of section 3 using the languages ReactiveML (RML) [16] and Lucky [13]. The simulation of the system is performed thanks to the connection existing between the ReactiveML compiler and the Lurette tool [12].

The Lucky language and the Lurette tool

Lucky is a language dedicated to the description of environments of reactive synchronous programs. It is designed to express sequences of testing scenarios in a compact manner. Its main characteristics are that it allows non-determinism and probabilistic descriptions. It is the source language for the Lurette tool which performs automatic generation of test sequences. It takes as input a Lucky description of the environment, an executable version of the program to test and a description of the properties to be checked for on the program. Lurette handles boolean but also numerical aspects of the environment description.

ReactiveML

RML is a language designed for the implementation of reactive systems. It combines the synchronous paradigm found in the Esterel with the classical features found in asynchronous settings (such as dynamic creation of processes). It is built on top of OCaml, and thus benefits from its power of expression. An extension of RML has been recently developed in which one can describe the behavior of a particular component in Lucky. This extension was first designed to allow the description of non-deterministic environments for RML programs. The mechanism used is as follows. At each time tick, the RML code is executed. When it comes to executing the Lucky environment, the RML program calls the Lurette tool so as to produce the environment new outputs. The solving of the non-deterministic aspect of the environment is totally delegated to Lurette. In the paper, we have kept a Lustre representation of the programs, in order not to introduce too many notations, but the translation from Lustre to RML should be automated.

Building our example with RML and Lucky

We have used Lucky to describe non-deterministic behaviors of components, and not only the environment. Each couple (A, G) of Lustre nodes forming a contract has been translated *by hand* into two equivalent Lucky programs. The first is then used to detect input values that violate the component's assertion. The second one is used as mentioned before to generate valid outputs. RML was then used to describe both deterministic components (the translation from the Lustre version was, there again, done by hand) and the global architecture of the application. From this program, the RML compiler generates an executable that basically implements the whole algorithm of fig. 7. The executable also implements each deterministic components (each deterministic $computeOut_{C_j}$ is compiled out to actual executable code). For each non-deterministic component (described by contracts) this code contains system calls to the Lurette tool implementing the calls of the forms $check_{\mathsf{Assert}}$ and *choose*.

Comments

The whole system could be described in Lustre. Here we benefit of the existing connections between the tools. The translation of Lustre contracts into Lucky automata is not an obvious process, and was performed in an *ad hoc* manner here. Full automation of this translation should be studied. Limitations would certainly appear in the general case.

6 Related Work

In the context of real-time reactive systems, there has been a lot of work around simulation. The most important ones are maybe those implemented in Ptolemy [5], which is a tool set for the design of heterogeneous systems. Simulation of this type of system is particularly described in [6]. In [24], a simulation method is proposed for the specific case of synchronous reactive systems. One important drawback of this approach is that it proposes a simulation method of deterministic systems, and thus can not be used in the early stages of development, as we propose to do. In [20] Metzger and Queins propose a methodology for generating deterministic prototypes from informal requirements of the application being designed. The requirements are first translated into SDL [1] and non-determinism is compiled out from the SDL program (thanks to a set of predefined SDL libraries). As pointed by the authors themselves, the main difficulty of this approach unsurprisingly consists in the semantic gap between natural language requirements and SDL formal specifications. Here, we do not claim to fill this gap. Rather, our proposal stand on a quite different status, since we do not compile an executable from the non-deterministic specifications, but we truly *simulate* them. Closer to our approach, but aimed at a different type of applications is the Co-Java approach [4]. Co-Java is an extension to Java in which certain part of the behavior of the system can be expressed using constraints on program variable. Upon simulation, these constraints are interpreted in a way similar to what LUCKY does in our case. Finally, this work can be seen as an attempt to propose a methodology impact of the work presented in [23]. There, the author proposes a simulation framework for arbitrary synchronous data-flow networks where non-deterministic components are expressed directly in the LUTIN language. Our approach is only different with respect to the language used (we use assume-guarantee contracts which are a strict subset of LUTIN). The exact same simulation algorithm can used (the one implemented in Lurette). We are interested in defining a particular usage of these simulation techniques in a contract-based design flow. Probably, an interesting extension of this work would be to allow description of contracts by couples of LUTIN components.

7 Conclusions and Perspectives

We have proposed a simulation methodology for reactive embedded systems where non-deterministic components can be described alongside with fully implemented ones. The deterministic components are compiled to standard executables while the non-deterministic ones, described by their contracts are interpreted by a standard test tool. The main advantage of this approach is that it can be used in the earliest stages of development, which can save enormous efforts usually spent on bug retrieval at the implementation level. The program itself, with all its components, can be described with varying levels of detail, depending on the stage of development.

It was not shown in this example, but it is trivially possible to plug in additional input generators or observers that are not directly part of the system itself but that can significantly help in its development. These can be described as LUSTRE nodes or observers and can then be simulated in the same manner. In that sense, this approach seems quite promising.

The whole methodology has been applied to the example described in this paper with *ad-hoc* transformation techniques. It has been entirely re-written in RML by hand and the translation of contracts into their Lucky counterpart was also done by hand. Still, this has shown to be quite promising in the end: we were able to simulate a partially defined version of the system and this allowed the discovery of several features early in the design process, that could probably have saved development effort afterward. However, the impact of the simulation on the global validation of this particular example was somewhat limited by the fact that we had previous running versions of the application that had already reached very stable statuses. Applying this methodology to a brand new case-study would be very beneficial in that sense.

We still would like to push it further in several directions. The first one concerns tool support. As we mentioned earlier, the translation from LUSTRE to RML is performed manually. We would like to either automate this translation or preferably work directly in LUSTRE. One idea we would prefer would be to integrate the use of the LURETTEtool in a debugger like LUDIC [17]. More important issues concern the expressiveness of the languages used. First we would like to extend this approach to consider the probabilistic part of LUCKY. This is a feature which has proved very useful in testing [23], and we believe it should be incorporated in our framework. It would then mean finding a good way of expressing these probabilistic behaviors as contracts. Second is the extension of this approach from purely synchronous systems to GALS (Globally Asynchronous, Locally Synchronous) systems. From a language point of view, it would be interesting to see how our contracts can be used for specifying GALS systems (using the work of [7] as a basis). We will study the application of our simulation method to these systems.

References

[1] *Specification and description language (SDL)*, ITU-T Recommendation Z.100, International Telecommunications Union (1999).

[2] Benveniste, A., P. Caspi, S. Edwards, N. Halbwachs, P. LeGuernic and R. de Simone, *Synchronous languages, 12 years later*, Proceedings of the IEEE (2003).

[3] Berry, G. and G. Gonthier, *The* ESTEREL *synchronous programming language: Design, semantics, implementation*, Science of Computer Programming **19** (1992), pp. 87–152.

[4] Brodsky, A. and H. Nash, *Cojava: a unified language for simulation and optimization*, in: *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), pp. 194–195.

[5] Brooks, C., E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)," EECS, University of California, Brekeley, CA USA 94720 (2005), memorandum UCB/ERL M05/21.

[6] Evans, B., A. Kamas and E. Lee, *Design and simulation of heterogeneous systems using ptolemy* (1994).

[7] Halbwachs, N. and S. Baghdadi, *Synchronous modeling of asynchronous systems*, in: *EMSOFT'02* (2002).

[8] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, *The synchronous dataflow programming language* LUSTRE, Proceedings of the IEEE **79** (1991), pp. 1305–1320.

[9] Halbwachs, N., F. Lagnier and P. Raymond, *Synchronous observers and the verification of reactive systems*, in: M. Nivat, C. Rattray, T. Rus and G. Scollo, editors, *Third International Conference on Algebraic Methodology and Software Technology, AMAST'93* (1993).

[10] Halbwachs, N., P. Raymond and C. Ratel, *Generating efficient code from data-flow programs*, in: *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), 1991.

[11] Harel, D. and A. Pnueli, "On the Development of Reactive Systems," Springer-Verlag New York, Inc., 1985 pp. 477–498.

[12] Jahier, E., *The lurette v2 user guide*, Technical report, Verimag, Centre Équation, 38610 Gières (2004).

[13] Jahier, E. and P. Raymond, *The lucky language reference manual*, Technical report, Verimag, Centre Équation, 38610 Gières (2004).

[14] Jahier, E., P. Raymond and P. Baufreton, *Case studies with lurette v2*, in: *ISoLa 2004, First International Symposium on Leveraging Applications of Formal Method*, Paphos, Cyprus, 2004.

[15] Le Guernic, P. and A. Benveniste, *The synchronous language* SIGNAL, in: M. R. Barbacci, editor, *Proceedings from the Second Workshop on Large-Grained Parallelism* (1987), pp. 56–57.

[16] Mandel, L. and M. Pouzet, *Reactiveml, a reactive extension to ml*, in: *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, 2005.

[17] Maraninchi, F. and F. Gaucher, *Step-wise + algorithmic debugging for reactive programs: Ludic, a debugger for lustre*, in: *AADEBUG'2000 – Fourth International Workshop on Automated Debugging*, Munich, 2000.

[18] Maraninchi, F. and L. Morel, *Arrays and contracts for the specification and analysis of regular systems*, in: *Fourth International Conference on Application of Concurrency to System Design (ACSD)*, Hamilton, Ontario, Canada, 2004.

[19] Maraninchi, F. and L. Morel, *Logical-time contracts for reactive embedded components*, in: *30th EUROMICRO Conference on Component-Based Software Engineering T rack, ECBSE'04*, Rennes, France, 2004.

[20] Metzger, A. and S. Queins, *Early prototyping of reactive systems through the generation of sdl specifications from semi-formal development documents* (2002).

[21] Morel, L., *Efficient compilation of array iterators for lustre*, in: F. Maraninchi, A. Girault and E. Rutten, editors, *Workshop on Synchronous Languages, Programming and Applications, SLAP'02* (2002).

[22] Raymond, P., D. Weber, X. Nicollin and N. Halbwachs, *Automatic testing of reactive systems*, in: *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[23] Roux, Y., *Description et simulation de systèmes réactifs non-déterministes*, Phd thesis, Institut National Polytechnique de Grenoble (2004).

[24] Whitaker, P., "The Simulation of Synchronous Reactive Systems in Ptolemy II," Master's thesis, Department of Electrical Engineering and Computre Science, University of California at Berkley (2001).