Verifying Temporal Properties of CommUnity Designs

Nazareno Aguirre[1], Germán Regis[1], and Tom Maibaum[2]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto and CONICET, Ruta 36 Km. 601, Río Cuarto (5800), Córdoba, Argentina
{naguirre,gregis}@dc.exa.unrc.edu.ar
[2] Department of Computing & Software, McMaster University, 1280 Main St. West, Hamilton, Ontario, Canada L8S 4K1
tom@maibaum.org

Abstract. We study the use of some verification techniques for reasoning about temporal properties of CommUnity designs. We concentrate on the verification of temporal properties in the context of branching-time temporal logic using the SMV tool.

We also discuss ways of modularising the temporal reasoning, by exploiting the various kinds of morphisms between designs available in CommUnity. Moreover, we combine SMV verification with some abstract interpretation mechanisms to overcome a limitation, with respect to the use of structure for simplification of verification, of CommUnity's refinement morphisms, the lack of support for data refinement.

1 Introduction

The constant increase in the complexity of software systems demands a continuous search for more and better modularisation mechanisms in software development processes, covering not only implementation, but also earlier stages, such as analysis and design. Indeed, many new modularisation mechanisms influence not only programming language constructs, but also their associated development methodologies. Modularisation mechanisms are also of a crucial importance for formal methods, and in particular for formal specification. Appropriate modularisation mechanisms allow us to *structure* our specifications, dividing the usually large specifications (due to the degree of detail that formal models demand) into manageable parts. Also, many modern software systems have an inherent structural nature, and for these, structured specifications are better suited. Finally, and more importantly for this paper, modularisation mechanisms allow us to apply some modularity principles to analyses of properties, taking advantage of the structure of the design itself, and making some automated and semi-automated verification techniques scale up and be applicable to larger systems specifications.

There exist many formal specification languages which put an emphasis on the way systems are built out of components (e.g., those reported in [15,6,20,14]), thus aiding the modularisation of specifications and designs. CommUnity is one of these languages; it is a formal program design language which puts special emphasis on ways of composing specifications of components to form specifications

J. Davies and J. Gibbons (Eds.): IFM 2007, LNCS 4591, pp. 1–20, 2007.

of systems [4]. CommUnity is based on Unity [1] and IP [5], and its foundations lie in the categorical approach to systems design [7]. Its mechanisms for composing specifications have a formal interpretation in terms of category theory constructs [4]. Moreover, CommUnity's composition mechanisms combine nicely with a sophisticated notion of refinement, which involves separate concepts of action blocking and action progress. CommUnity also has some tool support, the CommUnity Workbench [23]. The CommUnity Workbench supports the editing, compilation, colimit generation (as explained below, colimits represent the joint behaviour of interacting components in CommUnity) and execution of CommUnity programs. However, it currently does not support the verification of logical properties of designs. For this purpose, we propose the use of well known model checking tools, in order to verify *temporal* properties of designs. More precisely, and due to some particular characteristics of CommUnity, we propose the use of CTL based model checking to analyse temporal properties of CommUnity designs. We start by defining a translation from CommUnity designs into SMV specifications in a semantics preserving way; since our goal is to verify temporal properties of designs, we have to consider a semantics for CommUnity designs that is more restrictive than (but compatible with) the semantics of open CommUnity designs described in [12]. We then attempt to modularise the verification activities via the superposition and refinement morphisms available in CommUnity, as indicated in [13]. This is very important, since it allows us to exploit the structure of CommUnity designs for verification, a task that is crucial for the successful use of model checking and other automated analysis techniques. The idea is to check properties required of a component from the specification of that component, thus exponentially reducing the search space associated with these checks, as compared to the search space associated with the much larger specification of the system. Although not all properties are necessarily preserved by including a component in a system, by means of some structuring relationship, important categories of properties are. Thus economies of scale might be achieved by using this structuring information to structure verifications. We concentrate on the information supplied by superposition relationships, used in structuring designs, but also discuss refinements. Finally, we combine model checking with predicate abstraction [8] in order to overcome a limitation (with respect to the modularisation of verification) of CommUnity refinements, namely the lack of support for data refinement [13].

The paper proceeds as follows. In section 2 we describe CommUnity and the concepts of designs and programs, including the structuring principles used to build systems from components. We also summarise the transition systems semantics of designs. Then in section 3, we discuss the verification of CommUnity designs using SMV, how the required translation is defined, and how the verification can be modularised, in some cases, by using the structure defined by the superposition morphisms used in structuring the design. We also discuss the relationship between refinement morphisms and temporal properties, and describe how we complement the CTL model checking with predicate abstraction, which is necessary due to the fact that refinement morphisms do not allow for data

refinement. We conclude with a discussion of results and future research. In order to illustrate the main ideas of the paper, we develop a case study based on a modular specification of a processor with a simple process scheduling mechanism.

2 CommUnity Designs

In this section, we introduce the reader to the CommUnity design language and its main features, by means of an example. The computational units of a system are specified in CommUnity through *designs*. Designs are abstract programs, in the sense that they describe a *class* of programs (more precisely, the class of all the programs one might obtain from the design by refinement), rather than a single program. In fact, when a design does not admit any further refinement, it is called a *program* [22].

Before describing in some detail the refinement and composition mechanisms of CommUnity, let us describe the main constituents of a CommUnity design. Assume that we have a fixed set $\mathcal{ADT}$ of datatypes, specified as usual via a first-order specification. A CommUnity design is composed of:

- A set V of *channels*, typed with sorts in $\mathcal{ADT}$. V is partitioned into three subsets V_{in}, V_{prv} and V_{out}, corresponding to input, private and output channels, respectively. Input channels are the ones controlled, from the point of view of the component, by the environment. Private and output channels are the local channels of the component. The difference between these is that output channels can be read by the environment, whereas private channels cannot.
- A first-order sentence $Init(V)$, describing the initial states of the design[1].
- A set Γ of actions, partitioned into private actions Γ_{prv} and public actions Γ_{pub}. Each action $g \in \Gamma$ is of the form:

$$g[D(g)] : L(g), U(g) \rightarrow R(g)$$

  where $D(g) \subseteq V_{prv} \cup V_{out}$ is the (write) *frame* of g (the local channels that g modifies), $L(g)$ and $U(g)$ are two first-order sentences such that $U(g) \Rightarrow L(g)$, called the lower and upper bound guards, respectively, and $R(g)$ is a first-order sentence $\alpha(V \cup D(g)')$, indicating how the action g modifies the values of the variables in its frame ($D(g)$ is a set of channels and $D(g)'$ is the corresponding set of "primed" versions of the channels in $D(g)$, representing the new values of the channels after the execution of the action g.)

The two guards $L(g)$ and $U(g)$ associated with an action g are related to refinement, in the sense that the actual guard of an action g_r implementing the abstract action g, must lie between $L(g)$ and $U(g)$. As explained in [13], the negation of $L(g)$ establishes a blocking condition ($L(g)$ can be seen as a lower

[1] Some versions of CommUnity, such as the one presented in [13], do not include an initialisation constraint.

bound on the actual guard of an action implementing g), whereas $U(g)$ establishes a progress condition (i.e., an upper bound on the actual guard of an action implementing g).

Of course, $R(g)$ might not uniquely determine values for the variables $D(g)'$. As explained in [13], $R(g)$ is typically composed of a conjunction of implications $pre \Rightarrow post$, where *pre* is a precondition and *post* defines a multiple assignment.

To clarify the definition of CommUnity designs, let us suppose that we would like to model a processor. We will abstract away from the actual code of the processes, and represent them simply by an ordered pair of non negative integers (denoted by `nat`), where the first integer represents a label for identifying the process and the second one the number of seconds of execution remaining. Then, a processor is a simple CommUnity design composed of:

- A local channel `curr_proc:`$\langle$`nat, nat`$\rangle$, representing the current process accessing the processor. We use a dummy value $(0, 0)$ for indicating that the processor is idle.
- an input channel `in_proc:`$\langle$`nat, nat`$\rangle$, for obtaining a new process (from the environment, in an abstract sense) to be run by the processor.
- An action `load`, which loads a new process into the processor (reading the corresponding values from the input variable `in_proc`).
- An action `run`, that executes the current process for a second.
- An action `kill`, that removes the current process, replacing it by the dummy $(0, 0)$.
- An action `switch`, which, if the current process is not the dummy $(0, 0)$, replaces it by the incoming process `in_proc`.

The CommUnity design corresponding to this component is shown in Figure 1.

Design Processor
in
in_proc: <nat, nat>
out
curr_proc: <nat, nat>
init
curr_proc = (0,0)
do
load[curr_proc]: in_proc.snd > 0 ∧ in_proc.fst ≠ 0 ∧ curr_proc=(0,0)
⟶ curr_proc'=in_proc
[] ***prv*** run[curr_proc]: curr_proc.snd > 0, curr_proc.snd > 0
⟶ curr_proc'=(curr_proc.fst , curr_proc.snd−1)
[] kill [curr_proc]: curr_proc.fst ≠ 0, false ⟶ curr_proc'=(0,0)
[] switch[curr_proc]: in_proc.snd > 0 ∧ in_proc.fst ≠ 0 ∧
curr_proc.snd >0, false
⟶ curr_proc'=in_proc

Fig. 1. An abstract CommUnity design for a simple processor

In Fig. 1, one can see the different kinds of guards that an action might have. For instance, action `kill` has safety and progress guards (`curr_proc.fst` $\neq 0$ and `false`, respectively). Since the progress guard for this action is `false`, the component is not obliged to execute the action when the environment requires it to do so.

Another important point to notice in the processor design is the apparent behaviour of action `switch`. After a switch, the previous value of `curr_proc` seems to be missing, since the component does not store it anywhere else, nor "sends" it to another component. It will become clearer later on that it will be the responsibility of other components in the architecture to "extract" the current process and store it when a switch takes place. This is basically due to the fact that communication between components is achieved by means of coordination, rather than by explicit invocation.

To complete the picture, let us introduce some further designs. One is a bounded queue of processes, with the traditional enqueue (`enq`) and dequeue (`deq`) operations, implemented over an array. The other is a process generator, a design that generates new processes to feed the system. These designs are shown in Figures 2 and 3, respectively.

Design Process_Queue
in
in_proc: <nat, nat>
out
out_proc: <nat, nat>
local
queue: array(10,<nat, nat>)
low, up, count: nat
init
out_proc = (0,0) ∧ ∀ x ∈ [1..10] :
queue[x] = (0,0) ∧ low = 1 ∧ up = 1 ∧ count = 0
do
enq[queue,out_proc,count,up]: count<10 ∧ in_proc.fst ≠ 0
⟶ queue'[up] = in_proc ∧ up' = (up mod 10)+1 ∧
out_proc' = if(count=0,in_proc,queue[low]) ∧ count'=count+1
[] deq[queue,out_proc,count,low]: count>0 , count>5
⟶ queue'[low] = (0,0) ∧ low' = (low mod 10)+1 ∧
out_proc' = queue[(low mod 10)+1] ∧count'=count−1

Fig. 2. An abstract CommUnity design for a process queue

the definition of action `enq` makes use of an if-then-else expression, in the syntax of the CommUnity Workbench. Notice that the progress guard for action `load` of the processor coincides with its blocking guard, which is too weak to guarantee a scheduling policy. Stronger progress guards for actions related to `load` will arise as a result of composing the processor with other components, to achieve

```
Design Process_Generator
out
  out_proc: <nat, nat>
local
  curr_id : nat
init
  curr_id = 1 ∧ out_proc = (0,0)
do
  prv gen[ out_proc ]: out_proc.fst ≠ curr_id
        ⟶ out_proc'. fst = curr_id ∧ out_proc'. snd > 0
[]    send[ out_proc, curr_id ]: out_proc.fst = curr_id
        ⟶ out_proc'=(0,0) ∧ curr_id ' = curr_id+1
```

Fig. 3. An abstract CommUnity design for a process generator

a proper scheduling policy. In our case, for example, we require the dequeing of processes to be ready whenever the number of processes in the queue exceeds half the queue capacity (see the progress guard of action `deq`).

2.1 Component Composition

In order to build a system out of the above components, we need a mechanism for composition. The mechanism for composing designs in Community is based on action synchronisation and the "connection" of output channels to input channels (shared memory). Since our intention is to connect both the process generator and the processor to the queue (since processes to be enqueued might be generated by the generator, or come from a processor's currently executing process being "switched out"), and the queue has a single "incoming interface", we have a kind of *architectural mismatch*. In order to overcome it, we can use a duplexer, as specified in Figure 4. The duplexer enables us to design a system in which independent use of the operations of the queue can be made by components that are clients of the queue. Using this duplexer, we can form the architecture shown in Figure 5. In Fig. 5, the architecture is shown using the CommUnity Workbench graphical notation. In this notation, boxes represent designs, with its channels and actions, and lines represent the interactions ("cables" in the sense of [13]), indicating how input channels are connected to output channels, and which actions are synchronised.

2.2 Semantics of Architectures

CommUnity designs have a semantics based on (labelled) transition systems. Architectural configurations, of the kind shown in Fig. 5, also have a precise semantics; they are interpreted as categorical diagrams, representing the architecture [13]. The category has designs as objects and the morphisms are *superposition relationships*. A superposition morphism between two designs A and B

Design Duplexer
in
in_1: <nat, nat>
in_2: <nat, nat>
out
out_proc: <nat, nat>
do
read1[out_proc]: in_1 ≠ (0,0) ∧ out_proc= (0,0) ⟶ out_proc'=in_1
[] read2[out_proc]: in_2 ≠ (0,0) ∧ out_proc= (0,0) ⟶ out_proc'=in_2
[] send: out_proc ≠ (0,0) ⟶ out_proc'=(0,0)

Fig. 4. An abstract CommUnity design for a simple duplexer

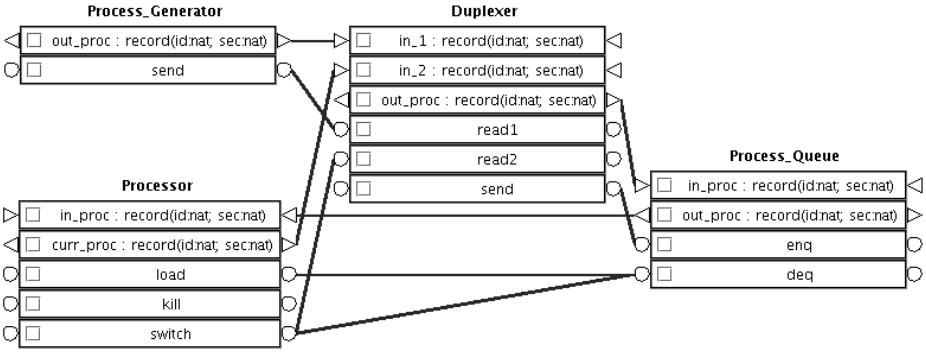

Fig. 5. A graphical view of the architecture of the system

indicates, in a formal way, that B *contains* A, and uses it while respecting the encapsulation of A (regulative superposition). The interesting fact is that the joint behaviour of the system can be obtained by taking the *colimit* of the categorical diagram corresponding to the architecture [4]. Therefore, one can obtain a single design (the colimit object), capturing the behaviour of the whole system.

2.3 Semantics for Abstract CommUnity Designs

In [13], the authors state that designs have an operational semantics when they are *closed* (i.e., they do not have input channels), the safety and progress guards for each action coincide, and the assignment for each action fully determines the value for each v', where v is in the frame of the action. For abstract CommUnity designs (i.e., not programs), it is not difficult to define a transition system semantics, by assuming that input channels can change arbitrarily and that, when no action occurs, the values of the local variables are preserved. This is exactly the idea followed in the definition of a denotational semantics for abstract CommUnity designs given in [12]. The semantics defined therein is, however, not

completely adequate for our purposes, since many labelled transition systems might correspond to an open design. Since we want to verify temporal properties of designs, we are forced to interpret these, when they are opened, in a particular way; we have been careful to do so in a way that is compatible with the semantics of open CommUnity designs given in [12] (i.e., we interpret designs as particular transition systems within the possible interpretations as described in [12]). Moreover, when a design is a *program*, the interpretation coincides with the operational semantics of these, as described in [13]. The semantics described below, which is a specialisation of that defined in [12], will allow us to establish a direct connection between arbitrary CommUnity designs (including programs) and temporal logic, with the aim of verifying temporal properties of designs.

Let $\langle L_{\mathcal{ADT}}, \Phi \rangle$ be a first-order specification of datatypes, $\mathcal{U}_{\mathcal{ADT}}$ a model of $\langle L_{\mathcal{ADT}}, \Phi \rangle$ and $P = \langle V_f, G \rangle$ a CommUnity design. Then, P defines a transition system $T_P = \langle V_f, \theta, \mathcal{T} \rangle$ over $L_{\mathcal{ADT}}$ and $\mathcal{U}_{\mathcal{ADT}}$, where:

- the set of flexible variables is the set V_f of channels of P,
- the initialisation condition θ is the initialisation *Init* of P,
- for each action $g \in G$, we include a transition t_g in $\mathcal{T}$, whose transition relation is the following:

$$\rho_{t_g} : L(g) \wedge R(g) \wedge ST(\overline{D(g)})$$

  where $ST(\overline{D(g)})$ is the formula $\bigwedge_{v \in (Loc(V_f - D(g)))}(v = v')$ (stuttering of the local variables not in the frame of g),
- $\mathcal{T}$ includes a stuttering transition t_I,
- $\mathcal{T}$ also includes a *local* stuttering transition id, whose transition relation is the following:

$$\rho_{id} : \bigwedge_{v \in Loc(V_f)} v = v'$$

The first two points in the above construction of the transition system T_P are easy to understand. The third point indicates that the actions of P correspond to transitions of T_P, as one might have expected. Notice that both the safety guard and the precondition for an action g (the first captured by the conjunct $L(g)$ and the second is embedded in $R(g)$) are considered in the transition; moreover, the corresponding assignment has to take place and the values of those local variables not in the frame of g are required to be preserved. The fourth and fifth points characterise the steps in which the design P is not actively involved (computation steps of the environment); note that input channels are allowed to change in a stuttering step of the design P.

The reader might notice that several constructs of CommUnity designs are ignored in the above described construction of transition systems. The most notable case is that of progress guards. Progress guards are not taken into account in the construction of transition systems for designs, because they represent "readiness" constraints which are not part of the transition system definition, but restrictions on the allowed models. For the particular models that we have chosen as the interpretations for CommUnity designs, these trivially hold, as

long as the progress guards of actions are stronger than the corresponding safety ones. More precisely, when the progress guard $U(g)$ of an action g holds, g must be "available" to be executed (more formally, any state s in a computation of a design P in which $U(g)$ holds must have a t_g-successor state s'); since the enabling condition for actions, according to our interpretation, is the safety guard, whenever $L(g)$ is true the action is available, thus guaranteeing that $U(g)$ implies the availability of g. Clearly, the logical characterisation of progress constraints requires the use of path quantifiers. The reason for adopting a branching time temporal logic is to be able to express such constraints. These are useful, since the user might want to manually strengthen the enabling guards of actions, which is a sound activity (with respect to [12]) as long as they are not strengthened "beyond" the corresponding progress guards. Finally, according to [12], one must restrict runs of a transition system T_P for a design P to strongly fair runs with respect to private actions, taking as their enabling conditions their corresponding safety guards.

Notice also that the difference between private and shared actions does not have an impact in the construction of transition systems for designs. This is due to the fact that, as explained in [13], the difference between private and shared actions only has to do with the allowed forms of interaction between designs.

3 Verifying Temporal Properties of Designs

3.1 The SMV System

SMV (Symbolic Model Verifier) is one of the most widely used model checking tools. Originally developed at Carnegie Mellon [18], SMV was the first model checking tool that used a symbolic representation of transition systems based on binary decision diagrams, which allowed for the application of model checking techniques to larger finite state systems. SMV comprises a modular notation for describing transition systems, as well as a notation for describing properties of these, in the CTL temporal logic. We will not give a full description of SMV, but just a brief overview of the notation, so that the reader not familiar with it can straightforwardly follow our descriptions.

The SMV description of a system is organised in *modules*. Each module describes a portion of a finite state system, and its specification is given in terms of typed variables, initialisation constraints and a transition relation. More precisely, a module description starts with *declarations*, which are essentially given as a list of typed variables. These types for variables must be bounded. The variables in declarations can be accompanied by a declaration of new types or aliases of types, for variable typing. The state space associated with a module will then be given by all the combinations of values of the corresponding types for the declared variables. The *transition system* associated with the system corresponding to a module is defined in terms of:

– a definition of the initial state, declared as initial values for each of the declared variables, and

– a definition of the transition relation, typically given as a "case" expression for the next value to be assumed for each of the declared variables.

Let us provide, as a simple example, the following module definition, which corrresponds to a manual translation of the simplest CommUnity design of our example, the process generator: In our SMV models, `MAXINT` is a user provided

```
MODULE Process_Generator() {

  out_proc: array 0..1 of 0.. MAXINT;
  curr_id : array 0..1 of 0.. MAXINT;

  init(out_proc):= [0,0];
  init(curr_id ):= 1;

  next(curr_id):= case {
    out_proc[0] = curr_id: curr_id +1; -- action send
    out_proc[0] ~= curr_id: curr_id ;  -- action gen
  };

  next(out_proc):= case {
    out_proc[0] = curr_id: [0,0];                        -- action send
    out_proc[0] ~= curr_id: [curr_id ,1.. MAXINT]; -- action gen
  };
}
```

positive constant, representing the maximum positive integer we consider. Notice also that it is possible to represent nondeterministic assignment: in the above example, the second component of the `out_proc` variable is nondeterministically assigned a positive value, in the definition of its next value associated with action `gen`.

3.2 Translating CommUnity Designs into SMV

We now describe our general characterisation of CommUnity designs in the language of the SMV tool. We will illustrate the translation from CommUnity into SMV by means of a detailed example. It is worth mentioning that we have chosen Cadence SMV [19] because of its richer language, which allows us to describe transitions involving structured-typed variables, such as arrays, in a more concise way.

The translation we describe only involves designs and not architectural configurations. As explained before, any valid configuration is a representation of a single design (the colimit of the categorical diagram corresponding to the architecture), so we do not lose generality.

The simplest part is the characterisation of channels. These are simply translated as variables in SMV, and for obvious reasons we limit ourselves to the types supported by Cadence SMV. For our simple Processor design described before, the channels are represented as follows:

```
in_proc : array 0..1 of 0.. MAXINT; -- Input variable
curr_proc : array 0..1 of 0.. MAXINT;
```

The initialisation of channels is translated into "init" specifications for the corresponding variables in SMV, as one might expect:

```
-- Initialisation of variables
  init(in_proc) := [0.. MAXINT,0..MAXINT]; --Input variable
  init(curr_proc):= [0,0];
```

The slightly more complicated part is the characterisation of actions. These need to be encoded into the "next" relationships for the variables. Since we need to simulate a scheduler for actions, which chooses nondeterministically one of the available actions, we introduce a "random variable". This variable randomly takes a numeric value corresponding to an action (including *skip*) to be executed in the next step as long as its safety guard is satisfied; if the safety guard of the chosen action is not true, then the action executed will be *skip*. For the Processor design, the scheduling of the actions is represented in the following way:

```
-- Definition of Scheduler
  -- Generation of random values used to schedule actions
  init(rnd) := 0;
  next(rnd) := 0..4;

  init(curr_action ) := skip;
  next(curr_action) := case{
    rnd = 0 : skip;
    rnd = 1 & (next(in_proc[1]) > 0 & next(in_proc[0]) ~= 0 &
                                                  next(curr_proc) = [0,0] ) : load;
    rnd = 2 & (next(curr_proc[1]) > 0): run;
    rnd = 3 & true : kill ;
    rnd = 4 & (next(in_proc[1]) > 0 & next(in_proc[0]) ~= 0 &
                                                  next(curr_proc[1]) > 0) : switch;
    1: skip;
  };
```

A point worth noticing is that the execution of the system in the SMV representation of a design P starts with a *skip*. This simplifies the specification of the initialisation statement in the translation, since otherwise we would need to take into account the initialisation constraints in P for the scheduling of the first action to be executed. Our alternative does not restrict the executions of the system, which from the second instant onwards will evolve by randomly chosen (available) actions. Notice that safety guards are part of the scheduling. The assignments of the actions, on the other hand, appear on the "next" definitions for the channels, which are formed by a "case" expression which depends on the action executed:

```
-- Definition of next value of variables
  next(in_proc) := [0.. MAXINT,0..MAXINT]; --Input variable

  next(curr):= case{
    curr_action  = skip :  curr_proc;
    curr_action  = load :  in_proc;
    curr_action  = run : [curr_proc[0], curr_proc[1]  - 1];
    curr_action  = kill  & curr_proc[0]  = 0 :  curr_proc;
    curr_action  = kill  & curr_proc[0]  ~= 0 :  [0,0];
    curr_action  = switch :  in_proc;
  };
```

Notice that, since `in_proc` is an input variable, it can change arbitrarily in each step.

Finally, we need to represent the constraints corresponding to progress guards and strong fairness for private actions. These are easily characterised in CTL, using an ASSUME clause for progress guards constraints and a FAIRNESS clause for strong fairness on private actions:

```
-- Fairness for private actions
FAIRNESS
curr_action  = {run};

-- Specification of progress guards as CTL formulae
ASSUME progress_switch;
 progress_switch :  SPEC AG ((curr_proc[1] > 4 & in_proc ~= [0,0])
                                                  → EX( curr_action = switch ));
```

Notice that progress guards are interpreted as (redundant) ASSUME clauses. If the user decides to strengthen some guards of actions in order to obtain more restrictive interpretations of a design, these must not go beyond the corresponding progress guards, in order not to make the SMV specification inconsistent.

Now we only need to provide the CTL formulae to be verified. For instance, we might want to check that if the id of the current process is 0 then it is the dummy process (i.e., the number of seconds remaining is also 0):

```
-- Properties to be verified
  NoInvalidProccess :SPEC AG (curr_proc[1] >0 →curr_proc[0]>0);
```

3.3 Modularising the Verification Through Morphisms

As put forward in [4] and later work, different notions of component relationships can be captured by morphisms, in the sense of category theory. We now exploit these morphisms in order to modularise the SMV-based verification, in the way indicated in [13].

Superposition Morphisms. We start by describing how superposition morphisms, which are used in the composition of CommUnity designs, are exploited. Let us first recall the formal notion of superposition morphism. A superposition morphism $\sigma : A \to B$ is a pair of mappings $\langle \sigma_{ch}, \sigma_{act} \rangle$ such that: *(i)* σ_{ch} is a total mapping from channels in A to channels in B, respecting the type and kind[2] of channels (except that input channels can be mapped to input or output channels), *(ii)* σ_{act} is a partial mapping from actions of B to actions of A, which preserves the kind (shared or private) of actions, does not reduce the write frame of actions of A, and the lower bound, upper bound and assignment for each action of A is strengthened in the corresponding actions of B; moreover, the encapsulation of A must be preserved, meaning that every action of B that modifies a channel v of $\mathrm{ran}(\sigma_{ch})$ must "invoke" an action of A that includes $\sigma_{ch}^{-1}(v)$ in its write frame.

Basically, σ_{ch} indicates how the channels of A are embedded as channels of B. The mapping σ_{act}, on the other hand, indicates, for each action a of A, all the actions that use it in B (through $\sigma_{act}^{-1}(a)$).

The main result that enables us to modularise the verification via superposition morphisms is reported in [12]. Therein, the authors indicate that superposition morphisms preserve invariants, the effect of actions on channels and the restrictions to the occurrence of actions. More generally, we can affirm that superposition morphisms preserve safety properties, which is a direct consequence of the following theorem:

Theorem 1. *Let A and B be CommUnity designs, and $\langle \sigma_{ch}, \sigma_{act} \rangle : A \to B$ a superposition morphism. Let s be a computation of B, according to the above defined semantics of designs, and defined over an interpretation $\mathcal{U}$ for datatypes. The computation s^A, defined as the restriction of states in s to channels in $\sigma_{ch}(V_A)$, is a computation of A.*

Applied to our example, this means that we can reason locally about safety properties of the components of a system. We have some examples below in which we show the improvement that local verification of safety properties for our case study constitutes. Of course, as is well known, this does not hold for liveness properties, which are not necessarily preserved by superposition (it is well known that, when a component is put to interact with others, some of its liveness properties might be lost).

Notice also that, among all possible interpretations of an open CommUnity design, we choose the less restrictive one, i.e., that in which the actions are enabled under the weakest possible conditions. This has as a consequence that the safety properties of the design that are verified using our SMV translation are indeed properties of *all* the valid transition system interpretations (according to [12]) of the design.

[2] By the type of the channel we mean the sort with which it is associated; by the kind of a channel we mean its "input", "output" or "private" constraint.

Refinement Morphisms. An important relationship between designs is refinement. Refinement, besides relating abstract designs with more concrete "implementations", is also useful for characterising parametrisation and parameter instantiation. In [13], the authors present a characterisation of refinement in terms of category theory constructions. Essentially, they demonstrate that CommUnity designs and morphisms capturing the notion of refinement constitute a category. As defined in [13], a refinement σ between designs A and B is a pair of mappings $\langle \sigma_{ch}, \sigma_{act} \rangle$, such that *(i)* σ_{ch} is a total mapping from channels in A to channels in B, respecting the type and kind of channels, and injectively mapping different output and input channels of A to different output and input channels of B; *(ii)* σ_{act} is a partial mapping from actions of B to actions of A, which preserves the kind of actions, does not reduce the frame of actions of A, the lower bound and assignment for each action of A is strengthened in the corresponding actions of B; moreover, the upper bound of each action a of A must be weakened by the disjunction of the upper bounds of all actions in B refining a, meaning that every action of B that modifies a channel v of $\mathrm{ran}(\sigma_{ch})$ must "invoke" an action of A that includes $\sigma_{ch}^{-1}(v)$ in its frame. Also, shared actions of A must have at least one corresponding action in B, and all new actions of B do not modify the local channels of A.

Notice that, with respect to the assignment and lower bounds of actions, the refinement morphisms make them *stronger* when refining a design. Therefore, we again can affirm, as for superposition morphisms, that, if σ is a refinement morphism between designs A and B, then every execution trace of B, restricted to the channels originating in A, is an execution of A, and therefore safety properties are preserved along refinement morphisms. Moreover, as shown in [13], refinement morphisms also preserve properties expressing the readiness of actions (called *co-properties* in [13]). This does not mean, however, that refinement morphisms are theorem preserving morphisms, with respect to the logic CTL. Many liveness properties expressible in CTL, for example, are not necessarily preserved along refinement morphisms. Consider, as a trivial example, a design containing, among other things, a private action `a`:

```
Design P
...
out
    x : int
    ...
init
    x = 0 ∧ ...
do
      prv a[ x ] : true, false ⟶ x' = x + 1
      ...
```

where the variable `x` can only be modified by action `a`. Consider a refinement of this design, in which all actions and channels are maintained without modifications, except for `a`, which is refined as follows:

```
Design P'
...
out
    x : int
    ...
init
    x = 0 ∧ ...
do
     prv a[ x ] : false , false ⟶ x' = x + 1
     ...
```

It is easy to see that, due to the strong fairness constraints imposed on private actions, the CTL liveness property $AF(x = 1)$ holds for the original design, but it does not hold for its described refinement.

One might be interested in exploiting refinement morphisms for simplifying the verification of properties of designs, since some safety and readiness properties might be easier to verify in more abstract designs, i.e., designs with fewer and simpler actions. However, the simplification one might obtain by moving from a design to more abstract (i.e., less refined) ones is limited, since refinement morphisms do not allow for data refinement (the types of channels must be preserved by refinement). This means, basically, that the state space of designs does not change through refinement morphisms. Thus, refinement morphisms are quite restricted for the simplification of verification, especially in the context of automated verification, where data abstraction is known to have a big impact on the verification times. For this reason, we complement below CommUnity's morphisms with abstraction mechanisms.

Abstraction. As we mentioned, abstraction is known to have a big impact in automated verification, especially for model checking [2]. Since refinement morphisms do not support data refinement, we considered the use of *predicate abstraction* [8], as a way of improving the SMV-based verification of CommUnity designs. Essentially, predicate abstraction consists of, given a (possibly infinite state) transition system, constructing an abstract version of it, whose abstract state space is determined by a number of predicates on the original state space. Basically, the state space of the abstract transition system is composed of equivalence classes of the original states, according to the provided (abstraction) predicates [8]. The more complex part is the construction of abstract transitions corresponding to the concrete ones, which requires checking to which of the equivalence class(es) the source and target states of each transition correspond. This can be computed automatically in many cases, and its complexity (not from a computational point of view) greatly depends on the provided predicates.

We used predicate abstraction in order to improve the verification for our example. For instance, we can concentrate on the processor design, and consider the following predicates to do the abstraction:

- the number of seconds remaining for `curr_proc` is 0,
- the process id for `curr_proc` is 0.

This leads us to the following four possibilities for `curr_proc`:

- *dummy*, if the number of seconds remaining is 0 and the process id is 0,
- *finished*, if the number of seconds remaining is 0 and the process id is not 0,
- *unfinished*, if the number of seconds remaining is not 0 and the process id is not 0,
- *invalid*, otherwise.

We can reproduce this abstraction for the `in_proc` variable, which leads us to a version of the SMV specification for the processor in which we do not distinguish the actual values of `curr_proc` and `in_proc`, but only whether their ids and remaining seconds are nil or not, which obviously makes the transition system for the design much smaller.

The corresponding abstract version of the SMV Processor module is the following:

```
typedef PROCCESS {dummy,finished,unfinished,invalid};
MODULE main (){
  rnd : 0..4;  -- used to schedule actions randomly
  curr_action : {skip, load, run, kill , switch1 };
-- Definition of the variables
    in_proc : PROCCESS; --Input variable
    curr_proc : PROCCESS;
-- Definition of Scheduler
  init(rnd) := 0;
  next(rnd) := 0..4;
  init(curr_action ) := skip;
  next(curr_action) := case{
    rnd = 0 : skip;
    rnd = 1 & (next(in_proc) = unfinished & next(curr_proc) = dummy ) : load;
    rnd = 2 & (next(curr_proc) = unfinished) : run;
    rnd = 3 & true : kill ;
    rnd = 4 & (next(in_proc) = unfinished & next(curr_proc) = unfinished ) :
                                                                    switch1;
    1: skip;  };
-- Initialisation of variables
  init(in_proc) := { dummy,finished,unfinished,invalid};
  init(curr_proc):= dummy;
-- Definition of next value of variables
  next(in_proc) := { dummy,finished,unfinished,invalid}; --Input variable
  next(curr_proc):= case{
    curr_action = skip : curr_proc;
    curr_action = load : in_proc;
    curr_action = run : {unfinished , finished } ;
    curr_action = kill & (curr_proc = dummy | curr_proc = invalid) : curr_proc;
    curr_action = kill & (curr_proc = unfinished | curr_proc = finished ) : dummy;
    curr_action = switch1 : in_proc;  };
-- Fairness for private actions
  FAIRNESS curr_action = {run};
```

```
-- Specification of progress guards as CTL formulae
  ASSUME progress_switch1;
  progress_switch1 : SPEC AG ((curr_p = unfinished & in_p ~= dummy)
                                          → EX( curr_action = switch1 ));
}
```

We can verify the property that if the id of the current process is 0 then it is the dummy process, whose concrete and abstract versions are the following:

```
NoInvalidProccess :SPEC AG (curr_proc[1] >0 →curr_proc[0]>0);
```

```
NoInvalidProccess :SPEC AG (curr_proc ~= invalid);
```

The validity of the abstract version of this property implies the validity of its concrete version [2]. Since this is a safety property, it is guaranteed that it will also hold for the complete system (since superposition morphisms preserve safety properties).

A point regarding abstraction and readiness is worth noticing. As indicated in [2], all CTL formulae not involving existential path quantifiers are preserved through abstraction. Readiness assertions require existential path quantifiers to be expressed in CTL, and therefore these (expressing required non determinism of components) might not be preserved through abstractions.

3.4 Some Sample Properties

To end this section, we provide some sample properties we have been able to verify using our translation into SMV:

"Variables `up` and `low` are always valid positions of `queue`"

```
Bounds:SPEC AG(low_2 >= 1 & low_2 <= SIZE & up_2 >= 1 & up_2 <= SIZE);
```

"Variable `count` ranges from 0 (empty queue) to `SIZE-1` (full queue)"

```
Count:SPEC AG(count_2 >= 0 & count_2 <= SIZE−1);
```

"Variable `out_proc` of the duplexer always holds a dummy process or a valid process (a positive process id and a positive number of seconds)"

```
NoInvalidDuplexerOut:SPEC AG(out_p_0 = [0,0] | (out_p_0[0] >0 & out_p_0[1] >0))
```

"All processes held in `queue` have a positive number of seconds remaining to be run"

```
for (i =1;i<=SIZE;i=i+1){NoFinished[i]:SPEC AG(q_2[i][0] >0 →q_2[i][1]>0);}
```

"All live processes (in the processor) eventually finish"

```
for (i =1;i<=MAXINT;i=i+1){Proccessed[i]:
  SPEC AG(curr_p_3[0]=i & curr_p_3[1] >0 →AF (curr_p_3[0]=i & curr_p_3[1]=0));}
```

Some of these properties can be verified locally within one component's design (for instance, the first two properties above are properties of the queue). This is so thanks to the fact that safety properties of designs are preserved through superposition morphisms. Other properties, such as the third one above, are *emergent properties*, in the sense of [3], i.e., they are properties of a design that emerge due to the interaction of it with other designs of the system.

4 Conclusions

We have presented a characterisation of CommUnity designs in SMV, defined with the aim of verifying temporal properties of these designs. We have experimented with the modularisation of the verification activities in SMV by exploiting Community's superposition morphisms, in the way indicated in [13]. We also observed that refinement morphisms, related to abstraction, are not powerful enough with respect to the improvement of automated verification, since they do not allow for data refinement. In order to overcome this limitation, we used an abstract interpretation mechanism known as predicate abstraction [8]. We also observed that, although predicate abstraction preserves a wide range of properties of designs, it does not necessarily preserve readiness properties of actions, related to the required non determinism of components. We developed a case study based on a modular specification of a processor with a simple process scheduling mechanism, and verified several temporal properties, including safety and liveness properties. Some of these were verified modularly, using abstraction and superposition morphisms.

We believe that CommUnity is an interesting language that deserves more attention. As we mentioned, it is a language that puts special emphasis on ways of composing specifications of components via their coordination, and clearly distinguishes action availability from action readiness. Moreover, there have been recently some extensions of it in order to capture complex notions such as mobility and dynamic reconfiguration. Of course, having appropriate tool support would improve the use of the language, and the work we report here is an initial attempt in this direction.

We are implementing a tool for verifying temporal properties of CommUnity designs. This tool is, at the moment, just a compiler that implements the translation of CommUnity designs into SMV. We are using the colimit generation procedure available in the CommUnity Workbench, and using a SAT solver to check the proof obligations associated to the construction of the abstract

transition systems related to predicate abstraction. This is done manually, at the moment, but we plan to incorporate the generation and verification of these proof obligations to the tool. We also plan to exploit the hierarchical structure of SMV specifications in our translation (at the moment, our translations generate unstructured SMV specifications).

As work in progress, we are trying to characterise the abstraction associated with predicate abstraction in a categorical way (via an appropriate morphism capturing data refinement). We are also studying the verification of temporal properties of CommUnity's *dynamic software architectures* [22], i.e., architectural configuration that might change at run time (e.g., via the deletion or creation of components, and the deletion and creation of connectors between components). Reasoning about temporal properties of dynamic architectures is notably more complex, since it is necessary to characterise the architectural configuration of the system as part of the state of the system. We are also looking at how aspects, in the sense of [10], can be applied to CommUnity designs. Aspects, as already observed in [9] and later work, can be implemented via combinations of superimpositions. In [11], the authors show how several aspects can be successfully characterised and combined in an organised way in CommUnity, via the use of higher-order architectural connectors (aspect weaving would correspond to colimit construction). In fact, we believe that most aspects can be characterised as architectural transformation patterns, replacing some part of a system design, defined by a pattern of components and connectors, by another pattern of components and connectors. However, for this approach to be powerful enough, we believe it is necessary to use an additional kind of superposition, so called invasive superpositions, that can break encapsulation and weakens lower and upper guards, and generalise CommUnity's designs to allow the design of *hierarchical*, reconfigurable systems. Aspect "weaving" would still be realised by the colimit construction.

Acknowledgements

The first author was partially supported by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT) and the Agencia Córdoba Ciencia, and his visits to McMaster University that contributed to this work were supported by McMaster University and the Canada Research Chair programme. The second author was partially supported by CONICET and the Agencia Córdoba Ciencia. The third author was partially supported by McMaster University, the Canada Research Chair programme, and the Natural Sciences and Engineering Council of Canada.

References

1. Chandy, K., Misra, J.: Parallel Program Design - A Foundation. Addison-Wesley, Reading (1988)
2. Clarke, E., Grumberg, O., Long, D.: Model Checking and Abstraction. In: ACM Trans. on Programming Languages and Systems, vol. 16(5), ACM Press, New York (1994)

3. Fiadeiro, J.: On the Emergence of Properties in Component-Based Systems. In: Nivat, M., Wirsing, M. (eds.) AMAST 1996. LNCS, vol. 1101, Springer, Heidelberg (1996)
4. Fiadeiro, J., Maibaum, T.: Categorical Semantics of Parallel Program Design. In: Science of Computer Programming, vol. 28(2-3), Elsevier, Amsterdam (1997)
5. Francez, N., Forman, I.: Interacting Processes. Addison-Wesley, Reading (1996)
6. Garlan, D., Monroe, R., Wile, D.: ACME: An Architecture Description Interchange Language. In: Proc. of CASCON'97, Toronto, Ontario (1997)
7. Goguen, J.: Categorical Foundations for General System Theory. In: Advances in Cybernetics anda Systems Research, Transcripta Books (1973)
8. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, Springer, Heidelberg (1997)
9. Katz, S., Gil, J.: Aspects and Superimpositions. In: Moreira, A.M.D., Demeyer, S. (eds.) Object-Oriented Technology. ECOOP'99 Workshop Reader. LNCS, vol. 1743, Springer, Heidelberg (1999)
10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, Springer, Heidelberg (1997)
11. Lopes, A., Wermelinger, M., Fiadeiro, J.: Higher-Order Architectural Connectors. ACM Trans. on Software Engineering and Methodology, vol. 12(1). ACM Press, New York (2003)
12. Lopes, A., Fiadeiro, J.: Using Explicit State to Describe Architectures. In: Finance, J.-P. (ed.) ETAPS 1999 and FASE 1999. LNCS, vol. 1577, Springer, Heidelberg (1999)
13. Lopes, A., Fiadeiro, J.: Superposition: Composition vs. Refinement of Non-Deterministic, Action-Based Systems. In: Formal Aspects of Computing, vol. 16(1), Springer, Heidelberg (2004)
14. Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., Mann, W.: Specification and Analysis of System Architecture Using Rapide. IEEE Trans. on Software Engineering. IEEE Press, New York (1995)
15. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, Springer, Heidelberg (1995)
16. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, Heidelberg (1991)
17. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems - Safety. Springer, Heidelberg (1995)
18. McMillan, K.: Symbolic Model Checking - An Approach to the State Explosion Problem, PhD thesis, SCS, Carnegie Mellon University (1992)
19. McMillan, K.: The SMV Language, Cadence Berkeley Labs, Cadence Design Systems (1998)
20. Medvidovic, N., Oreizy, P., Robbins, J., Taylor, R.: Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In: Proc. of ACM SIGSOFT '96, San Francisco, CA, ACM Press, New York (1996)
21. Wermelinger, M., Lopes, A., Fiadeiro, J.: Superposing Connectors. In: Proc. of the 10th International Workshop on Software Specification and Design, IEEE Press, Los Alamitos (2000)
22. Wermelinger, M., Lopes, A., Fiadeiro, J.: A Graph Based Architectural (Re)configuration Language. In: Proc. of ESEC/FSE'01, ACM Press, New York (2001)
23. Wermelinger, M., Oliveira, C.: The CommUnity Workbench. In: Proc. of ICSE 2002, Orlando (FL), USA, ACM Press, New York (2002)