Evolving Specification Engineering

Dusko Pavlovic, Peter Pepper, Doug Smith

Email: {dusko,smith}@kestrel.edu, pepper@cs.tu-berlin.de

Kestrel Institute and Technische Universität Berlin

Abstract. The motivation for this work is to support a natural separation of concerns during formal system development. In a development-by-refinement context, we would like to be able to first treat basic functionality and normal-case behavior, and then later add in complicating factors such as physical limitations (memory, time, bandwidth, hardware reliability, and so on) and security concerns. Handling these complicating factors often does not result in a refinement, since safety or liveness properties may not be preserved. We extend our earlier work on evolving specifications (1) to allow the preservation of both safety and liveness properties under refinement, and (2) to explore a more general notion of refinement morphism to express the introduction of complicating factors.

1 Introduction

It is natural for developers to initially focus on essential requirements, and to first consider only "normal" behaviors when writing specifications. Gradually, through refinement, the developer can then strengthen the initial optimistic assumptions, and handle the exceptional, unusual and abnormal cases, as well as introduce stronger requirements.

However, this approach presents a conceptual problem for a formal specification framework. Refining a specification by catching some exceptional behavior may not preserve the liveness properties of the system. Refining a specification by explicitly handling some exceptional behavior may not preserve its safety properties. So the question is: which properties should the useful refinement operations preserve?

In the present paper, this problem is formalized and solved in the framework of Evolving Specifications (especs) [11]. In order to express and characterize the conditions under which the relevant safety and liveness properties are preserved, we extend the framework by temporal modalities. In order to capture the specific preservation properties, combining safety and liveness in a way suitable for exception handling, we use *guard intervals* [9], spanned between the conditions under which an operation *may* fire, and the conditions under which it *must* fire. The capability to separate concerns for normal behavior from the exceptional cases opens an alley towards better understanding and implementing the mechanisms to introduce new safety and security policies in a system, and their semantic effects on a design.

1.1 A simple real-world example.

We illustrate our approach with a running example that is taken from the automotive domain: A modern car contains numerous devices such as radio tuner, CD player, navigation system, mobile phone and so forth. We presume here that all these devices are connected through a MOST bus (a modern optical bus that is often used in European cars) such that the user interaction with all services can take place over a common microphone, amplifiers and graphical display. The MOST bus architecture [1] provides both synchronous and asynchronous channels (and also command channels) for the interconnection of devices. Throughout this paper we will use (admittedly oversimplified) features of the MOST architecture as illustrating examples. The basic concept is illustrated in Fig. 1: All devices – including the MOST bus itself – are considered as "components" (more or less like in UML).

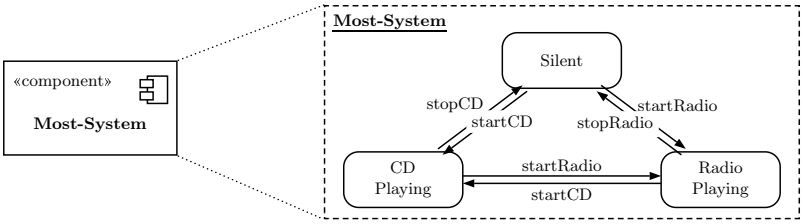


Fig. 1. The modes and transitions of a trivial MOST system

The behavior of a component is described in the espec formalism. For the sake of illustration we consider an oversimplified two-device system consisting of a radio and a CD player. This leads to three general modes, namely *CD-Playing*, *Radio-Playing* and *Silent*. Therefore Fig. 1 essentially describes the system from the viewpoint of the MOST bus: At any given point in time either the radio is playing or the CD player or none of them. There are six transitions, for which we allow overloaded naming, as long as their source or target modes are different.

Each device requires four bus channels to connect with the amplifier. In a natural specification development process, we would like to be able to simply assume that four channels are available when transitioning, say, from *Silent* to *CD-Playing*. Only later would we deal with abnormal situations in which that is not the case, as illustrated in Section 5.

1.2 Background

In previous work we introduced Evolving Specifications (especs) as a framework for specifying, composing, and refining systems [11, 12, 10]. This framework extends our earlier work on the algebraic/categorical specification of software [8], which it still contains as a subframework. Especs add the dimension of stateful

behavior, and thus leads into the realm of system specifications. This approach is in the tradition of many approaches in the literature to address issues of system design by utilizing category-theoretic mechanisms (e.g. [3].)

Modes and Transitions Although formally our evolving specifications resemble state machines, we prefer to speak of "*modes (of operation)*", rather than states. A specification usually describes how a system evolves from mode to mode: e.g., a CD player, may be in the mode *Playing*, performing the various activities within that mode, until a suitable event triggers a transition into another mode, say *Searching*. From an intuitive semantic point of view, a mode $\mathcal{M}$ can be viewed as a *set of (finite or infinite) traces of states*:

$$Beh(\mathcal{M}) = \{\ \mathcal{T} \mid \mathcal{T} = \langle \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \dots \rangle,\ \mathcal{S}_i \models Theory(\mathcal{M})\ \} \tag{1}$$

The modes $\mathcal{M}_1, \mathcal{M}_2, \dots$ are *specified* by logical theories[1] $M_1, M_2, \dots$. Therefore the semantics of a mode $\mathcal{M}_i$ consists of all traces of states, which fulfill the corresponding theory M_i, as is expressed in (1).

We remain completely abstract w.r.t. the specific nature of states in order to encompass all kinds of underlying systems; therefore we only require them to be models of the given theories. (Actually we also look into variants, where the state traces are replaced or complemented by continuous behaviors, comparable to hybrid automata.) Hence we focus solely on the modes from now on.

The modes of a system are connected by transitions (as illustrated in Fig. 2). These transitions t are usually guarded, which we denote here as $g \Rrightarrow t$.

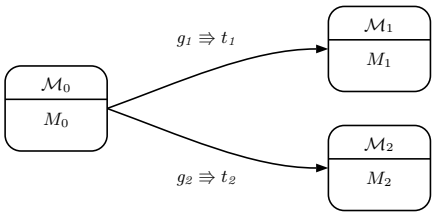


Fig. 2. Modes and transitions

Note: Each mode is assumed to have an *identity transition* with guard *true* and transition *id* (nothing changes). This transition – which we do not draw explicitly in our illustrations – corresponds to "stuttering", and is left to be specified in later refinements.

Semantically, a transition such as $g_1 \Rrightarrow t_1$ in Fig. 2 usually means that, whenever the *guard* g_1 holds in some state $\mathcal{S}_j$ of $\mathcal{M}_0$, then the transition *may* be taken. But it can only be taken in states where the guard holds. For reasons to

[1] We essentially identify the modes with their theories; therefore we purposely distinguish them only by the font.

be seen in a moment we refer to these kinds of guards as *safety guards*. Safety guards represent very weak and liberal constraints: They may hold arbitrarily often during a mode without their transition being taken. In particular, guards g_1 and g_2 of competing transitions (such as in Fig. 2) need not be disjoint.

But there is a second view of guards, where a transition such as $g_1 \Rrightarrow t_1$ in Fig. 2 means that, whenever the *guard* g_1 holds in some state $\mathcal{S}_j$ of $\mathcal{M}_0$, then the transition *must* be taken. Consequently, competing transitions must have disjoint guards g_1 and g_2. For reasons to become clear in a moment, we refer to guards of this kind as *liveness guards*.

These semantic intuitions have led to the formalism of evolving specifications [11]. Its main conceptual components are:

Transitions. The transition $\mathcal{M}_1 \xrightarrow{g \Rrightarrow t} \mathcal{M}_2$ is captured as an interpretation $M_2 \xrightarrow{t} M_1$, which rewrites the theory M_2 in terms of the theory M_1:

$$M_2 \models q \quad \Longrightarrow \quad M_1 \models (g \Rightarrow t(q)) \tag{2}$$

Within the category of specifications, such guarded transitions are modeled as opspans of interpretations in the form

$$M_1 \longrightarrow (M_1 \wedge g) \xleftarrow{t} M_2 \tag{3}$$

The formal details of categorical semantics of evolving specifications can be found in [11, 12]. Intuitively, the action t performed by a transition can be construed as a predicate transformer.

Within this formal and intuitive framework, the guards allow two semantically relevant interpretations:

Safety guards. An occurrence of the transition $\mathcal{M}_1 \xrightarrow{g \Rrightarrow t} \mathcal{M}_2$ in an execution $\mathcal{Q}$ is *enabled*, when $M_1 \wedge g$ is satisfied for the variable assignments at that point of the execution.

Liveness guards. An occurrence of the transition $\mathcal{M}_1 \xrightarrow{g \Rrightarrow t} \mathcal{M}_2$ in an execution $\mathcal{Q}$ is *forced*, when $M_1 \wedge g$ is satisfied for the variable assignments at that point of the execution.

While the framework of [12] left the choice between these two interpretations to the designer, deciding if the refinements should preserve safety or liveness, in Section 3 below, we shall present a unified semantical framework, subsuming both of the above interpretations.

Definition 1 *A* run *of a system is a sequence of modes* $\mathcal{M}_0 \rightarrow \mathcal{M}_1 \rightarrow \mathcal{M}_2 \rightarrow \ldots$ *for which there are transitions* $\mathcal{M}_i \xrightarrow{g \Rrightarrow t} \mathcal{M}_{i+1}$. *The* behavior *Beh(Spec) of a system specification is the set of all its runs.*

1.3 Refinement

The main point of especs is to provide a precise and convenient framework to specify the functions and behavior of software systems *incrementally*. The

main point of their categorical semantics is to provide a formal underpinning for refinement and composition, in terms of morphisms and colimits.

The basic principle can be summarized as follows: As usual, a refinement adds details, but preserves certain properties. Hence, the theory increases and the set of models becomes smaller:

$$\left(Spec_a \xrightarrow{\varphi} Spec_c\right) \implies \left(Beh\,(Spec_a) \supseteq Beh\,(Spec_c)\right) \tag{4}$$

Due to the added details, one often refers to the original specification $Spec_a$ as the *abstract model* and to the refined specification $Spec_c$ as the *concrete model*.

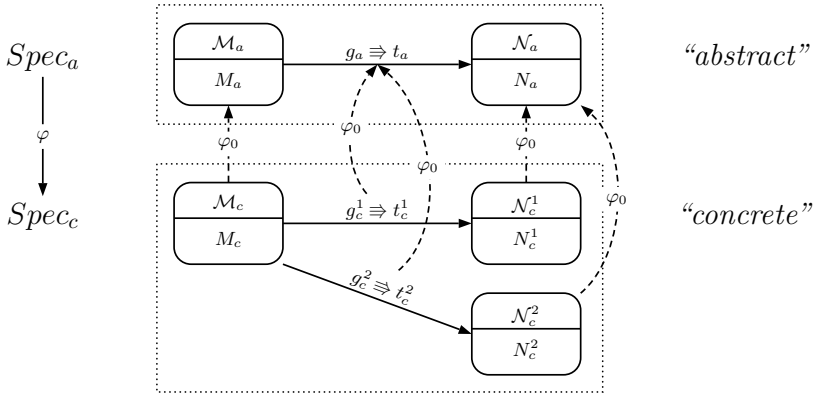


Fig. 3. Refinement of modes and transitions

As is illustrated in Fig. 3 it is possible that several modes of the concrete model correspond to ("refine") a single mode of the abstract model. And also several concrete transitions may correspond to ("refine") a single abstract transition (see [12]).

Definition 2 *A* refinement $\varphi : Spec_a \longrightarrow Spec_c$ *as depicted in Fig. 3 consists of two components:*

- *a graph morphism* $\varphi_0 : Diag_c \longrightarrow Diag_a$*, assigning to each concrete mode an abstract mode, and to each concrete transition an abstract transition, which it refines;*
- *a tuple of traditional specification morphisms* $\varphi_1^N : \varphi_0(N) \longrightarrow N$*, one for each concrete mode* $N \in Diag_c$*, telling how the specification of the mode* N *refines the specification of the mode* $\varphi_0(N)$*.*

Whereas this bipartite view of the structure preservation is rather familiar, e.g. from theory of institutions [5], the treatment of the *guards under refinement* is more intricate, since we need to distinguish *safety guards* and *liveness guards*.

Safety morphisms are required to preserve safety properties: the refinement and the composition steps along the safety morphisms must not introduce new runs — every run in the concrete system is a refinement of (or is simulated by) some run of the abstract system.

To preserve only the enabled executions, the specification components ς_1 of a safety morphism $\varsigma : Spec_a \longrightarrow Spec_c$ must satisfy, for every concrete transition $\mathcal{M}_c \xrightarrow{g_c \Rightarrow t_c} \mathcal{N}_c$,

$$M_c \models (g_c \Rightarrow \varsigma_1(g_a)) \tag{5}$$

where g_a is the guard of the φ_0-image of this transition.
This formalizes the fact that a concrete transition may only be taken if the corresponding abstract transition may have been taken as well.

Liveness morphisms are required to preserve liveness properties: the refinement and the composition steps along the morphisms must not introduce new deadlocks, but guarantee that every trace in the abstract system induces some refined trace in the concrete system.

To preserve all the forced executions, the specification components λ_1 of a liveness morphism $\lambda : Spec_a \longrightarrow Spec_c$ must satisfy, for every concrete transition $\mathcal{M}_c \xrightarrow{g_c \Rightarrow t_c} \mathcal{N}_c$,

$$M_c \models (\lambda_1(g_a) \Rightarrow g_c) \tag{6}$$

This formalizes the fact that whenever an abstract transition must be taken then the corresponding concrete transition must be taken as well.

Every first order trace property can be expressed as a conjunction of a safety property and a liveness property [2]. In order to specify refinements preserving arbitrary first order properties of interest, it is therefore sufficient to assure that both safety and liveness properties are preserved. A general method to realize this by combining the two types of espec morphisms described above is presented in Section 3. To motivate it, we first summarize a more special problem that drives this paper.

1.4 The Problem to be Solved

In rare cases, the process of system design can be subdivided into refinement steps where only safety or only liveness is preserved. In most cases, however, a property required from the system inextricably combines liveness and safety aspects. See [6] or [7] for examples.

Related to this is another issue: In many situations it is natural to first specify the *normal* behaviors of the system, under some simplifying assumptions, and to handle separately the *exceptional* behaviors, when these assumptions are not satisfied. The refinement step where the exceptions are recognized does not preserve liveness (since it blocks some runs), whereas the refinement step where they are handled does not preserve safety (since it adds new runs).

This leaves us with two complementary tasks of refining the notion of espec refinement, respectively capturing

1. general properties, which combine safety and liveness properties, and
2. exception recognition and handling.

The solutions of these two tasks will be outlined in Sections 3 and 4.2. As a preparatory step we formalize in Section 2 the above remarks about the safety guards and the liveness guards by defining an obvious interpretation of temporal logics of especs.

2 Temporal evolving specifications (tespecs)

The temporal statements in an espec are expressed in a global language, common to all modes. That is, the atomic formulas are given by the intersection of (the signatures of) all mode theories M_i. Over these we build temporal formulas using the usual connectors from propositional logic and the two tense operators

$$\begin{array}{ll} \bigcirc q & (\textbf{next}) \\ q \,\mathcal{W}\, r & (\textbf{waiting-for}) \end{array}$$

We define the validity of a (temporal) formula q in a certain mode $\mathcal{M}_0$ based on its validity for all runs $\mathcal{M}_0 \twoheadrightarrow \mathcal{M}_1 \twoheadrightarrow \mathcal{M}_2 \twoheadrightarrow \ldots$ that begin with $\mathcal{M}_0$. Based on the standard notion of validity ($M_i \models q$) for non-temporal formulas q, we define the validity of the temporal formalas in the usual way:

$$M_0 \models \bigcirc q \iff M_1 \models q \tag{7}$$

$$M_0 \models q \,\mathcal{W}\, r \iff (\forall i.\ M_i \models q) \vee \exists k.\ (M_k \models r) \ \wedge\ (\forall j < k.\ M_j \models q) \tag{8}$$

Remark: Temporal formulas quantify over the coarse-grained modes (runs) and not over the fine-grained internal states (traces) inside the modes.
Together with the usual connectors of classical logic, we can introduce the well-known further temporal modalities

$$\begin{array}{lll} \Box\, q & = q \,\mathcal{W}\, \bot & (\textbf{henceforth}) \\ \Diamond\, q & = \neg \Box \neg q & (\textbf{eventually}) \\ q \,\mathcal{U}\, r & = q \,\mathcal{W}\, r \ \wedge\ \Diamond\, r & (\textbf{until}) \end{array} \tag{9}$$

Now we can formalize the statements from the Introduction. Actually, much stronger and more precise statements could be proved.

Definition 3 *A* safety property *has the form* $\Box\, q$. *A* liveness property *has the form* $\Diamond\, q$.

The following lemma points to the way in which the (global) safety and liveness properties are logically related to the (local) guards of transitions.

Lemma 1 *(i) A system described by an espec* ***satisfies a safety property*** $\Box\, q$*, if and only if in each run (i) the property q is satisfied at the initial mode $\mathcal{M}_0$, and (ii) it is invariant under every enabled transition, i.e.*

$$\begin{array}{c} M_0 \models q \\ \textit{and} \quad (M \models q \wedge g) \Longrightarrow (N \models q), \qquad \textit{for all } \mathcal{M} \xrightarrow{g \Rrightarrow t} \mathcal{N} \end{array} \tag{10}$$

(ii) A system ***satisfies a liveness property*** $\Diamond\, q$*, if and only if in each run either the property q is satisfied at the initial mode, or there is an enabled transition, where q is established.*

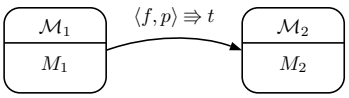
$$\begin{array}{c} M_0 \models q \\ \textit{or} \quad (M \models \neg q \wedge g) \ \wedge \ (N \models q), \qquad \textit{for some } \mathcal{M} \xrightarrow{g \Rrightarrow t} \mathcal{N} \end{array} \tag{11}$$

Proposition 1 *An espec morphism preserves safety (resp. liveness) properties if and only if it preserves all safety (resp. liveness) guards.*

Note: The characterization of liveness in the definitions (8) and (9) and in Lemma 1(ii) reflects the liveness view of branching-time logic in the Manna-Pnueli style, where $\Diamond\, q$ essentially means that *in every run* there is at least one state where q holds. If we would instead internalize the quantification over all runs into the definition of the validity, we would obtain the view of the temporal logic CTL*. This view represents the very weak property that *there is at least one possible run containing at least one state, where q holds.* So our approach could be geared towards both variants without much effort.

3 Guard Intervals

In the previous sections we have been working with the *concepts* of safety and liveness guards, but *without notational means* to distinguish them. Since many specifications combine both safety *and* liveness aspects,it turns out to be useful to bring them together. This leads to the idea of *guard intervals*, originally implemented in the CommUnity system [9].



Definition 4 *A* ***guard interval*** *is given in the form $\langle f, p\rangle$, where as an additional constraint the implication $f \Rightarrow p$ must hold.*

- *f is the* ***forcing guard****, i.e. the liveness guard that determines which* (good) things must *happen;*
- *p is the* ***permitting guard****, i.e. the safety guard that says which things are* not bad *and* may *happen.*

Let $\mathcal{S}$ be some state in a run of $\mathcal{M}_1$. Then the transition $\mathcal{M}_1 \xrightarrow{\langle f,p\rangle \Rrightarrow t} \mathcal{M}_2$ is

– *enabled* if $\mathcal{S} \models p$;
– *forced* if moreover $\mathcal{S} \models f$.

As we shall see next, under refinement the interval monotonically tightens, but not necessarily to a singleton.

Remark. The idea to capture the safety and the progress properties of executions by pairs of guards goes back to Fiadeiro's and Lopez' work on the CommUnity system [9]. Like especs, CommUnity belongs to the broad family of categorical specification systems [3, 5], where the property preservation under refinements is enforced as the structure preservation, imposed on the morphisms. However, the differences between the tasks supported by CommUnity and the tasks set out in this paper lead to different treatments of guard intervals. In particular, while the superposition morphisms of CommUnity only allow strengthening of both safety and liveness guards [9, Def./Prop. 4.1], and their refinement morphisms add a further constraint [9, Def./Prop. 5.1], leading to the equivalence of each abstract liveness guard with the disjunction of its concretizations, a simpler preservation requirement will turn out to be more appropriate in our framework. This requirement is the subject of the next section.

Refinement with guard intervals

Let us consider a refinement of guard intervals as depicted in Fig. 4 below (where, as earlier, φ denotes the specification morphism and ψ the opposite morphism on the diagrams). When will it preserve both liveness and the safety? The answer is a direct consequence of equations (5) and (6) in Section 1.3:

– The forcing guard f has to be *weakened*;
– The permitting guard p has to be *strengthened*.

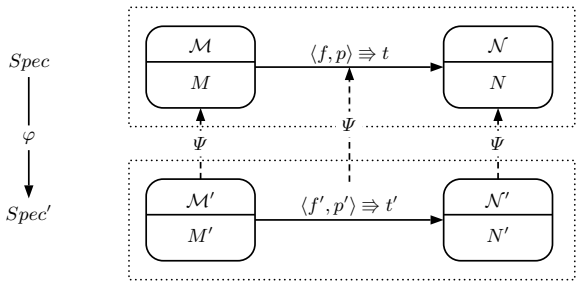


Fig. 4. Refinement of guard intervals

Gathering these implications, together with the constraint of Def. 4, in the form

$$\mathcal{M}' \models \varphi(f) \Rightarrow f' \wedge f' \Rightarrow p' \wedge p' \Rightarrow \varphi(p) \tag{12}$$

we see that the refinement actually *tightens* the guard interval, just like it does in real number computation, so that above implications can be construed as the interval inclusion $\langle f', p' \rangle \subseteq \langle \varphi(f), \varphi(p) \rangle$. For especs, this just captures the fact that the behaviors in which the refined transition is taken is strictly included, modulo the interpretations, among the behaviors where the abstract transition is taken. The effect of a nontrivial liveness guard (i.e. not always *false*) is to force the inclusion of the transition in *all* refinements.

If liveness properties are proved relative to the liveness guards (forced transitions) then since the liveness guards are only weakened under refinement, they will be preserved under any refinement. Similarly, if safety properties are proved relative to the safety guards (enabled transitions) then since the safety guards are only strengthened under refinement, they will be preserved under any refinement.

This extension of the espec formalism by the pairs of guards $\langle f, p \rangle$ is easily seen to be yet another instance of the abstract framework of [12]. The procedure of adjoining guards to the category Spec of specification, described in Section 3 of that paper, only needs to be modified by taking

$$\mathbb{G}(K, M) = \{\langle f, p \rangle \in \mathcal{L}_K^2 \mid K \wedge f \Rightarrow M \Rightarrow K \wedge p\} \tag{13}$$

Defining the espec morphisms as above then allows capturing the suitable combinations of safety and liveness properties, expressible by the guard intervals. The language of especs with guard intervals is more expressive than the ordinary guarded language, as it can express certain combinations of temporal modalities. The exact characterization of its expressiveness appears to be nontrivial.

The well-known topological analysis of liveness and safety properties [2] tells that every first order trace property can be expressed as an intersection of a safety and a liveness property, i.e. in the form $\Box\, q \wedge \Diamond\, r$.

Based on Lemma 1 we know that by representing a first order property in the form $\Box\, q \wedge \Diamond\, r$, and setting up the guard intervals in an abstract espec to realize this property, we can be sure that the espec morphisms preserving the guard intervals will preserve this property.

Proposition 2 *An espec morphism with guard intervals preserves all first order properties.*

4 "Normally" Modality

When they explain the functioning of a system (be it existing or planned), engineers usually begin in the style: *"Disregarding pathological borderline cases, the normal behavior is ..."*. In practice, there is a healthy distinction between the essential purpose of the system and all the nitty-gritty details of possible complications and unwanted effects. As soon as one tries to transfer this principle to the rigorous world of mathematical specifications, severe problems arise. Specifying the essential features without explicitly precluding the undesired exceptional

situations often leads to inconsistencies. On the other hand, enumerating the undesired situations, and their interactions, is known to lead to "formal noise" that exceeds the specification proper by an order of magnitude.

We attempt to mitigate this situation by introducing a special operator *normally*, denoted by $\natural$. We could define this operator as some kind of modality, but our framework allows us to introduce it as a simple abbreviation.

Definition 5 (Normally) *The* normally *operator* $\natural$ *is an abbreviation for an uninterpreted guarding predicate* n*:*

$\natural$ *property*	*abbreviates*	$nrml \Rightarrow property$
$\natural$ *guard* $\Rrightarrow$ *transition*	*abbreviates*	$nrml \wedge guard \Rrightarrow transition$

Note that there is a fresh predicate symbol nrml for each occurrence of the operator $\natural$*.*

As a shorthand notation we may qualify a whole specification or a whole mode or transition with the normally operator. This means that every single axiom and transition is implicitly preceded by the operator $\natural$*.*

In the later course of the development of the model this variable *nrml* can be made explicit and then be more and more concretely interpreted by giving axioms for it. This way, one can successively add exception handling to an originally "purely optimistic" model.

Together with our concept of refinement, this operator stratifies specifications considerably. The following program illustrates our use of the normally operator.

ESPEC *Player* IS
  ...
  MODE *Playing* IS
    $\natural$ $\#(channels) = 4$
    ...
END-ESPEC *Player*

This specification says that an active CD player "normally" has four channels available for streaming (thus enabling stereo). However, there may be situations in which the MOST bus does not have enough free channels. Then we have to take appropriate measures in order to build a workaround (e.g. changing to mono). But we do not want to clutter our specification of the essential behavior with that kind of exception handling in the early stages of our development. These kinds of complications need to be worked into the specification at some later stage – and it needs to be done in a systematic way; this is achieved in our approach by employing suitable refinement morphisms.

4.1 Refinement of "Normally"

Many of the occurrences of the normally operator $\natural$ can be refined by the standard mechanisms developed so far. Since the operator usually corresponds to the addition of uninterpreted predicate symbols, we simply need to define axioms

that interpret these symbols in order to make the specification more concrete. This is a classical refinement morphism.

However, there is one additional activity that we need to add for purely pragmatic reasons, even though it partly conflicts with our notion of refinement morphisms: If in a specification Syst a whole transition is qualified as "normally", i.e. $\natural(\mathcal{M} \xrightarrow{g \Rrightarrow t} \mathcal{N})$, then the designer often wants to express the fact that there may be further transitions out of $\mathcal{M}$, which are not yet relevant at this stage of the development.

A later refinement Syst' may then add another transition $\mathcal{M} \xrightarrow{g' \Rrightarrow t'} \mathcal{K}$ out of $\mathcal{M}$. The problem is that this need not correspond to any transition of the original specification Syst. Hence, the mapping φ_0 (see Section 1.3) is *not* a proper diagram morphism.

There are a number of ways out of this dilemma. In the next section we show a special instance of this paradigm in order to demonstrate how the principal mechanism works.

4.2 Especs of exceptions

The normally operator $\natural$ allows a relatively fine-grained qualification of those aspects that are in the core of a system (as opposed to borderline cases such as errors or rare events). However, it does not really help to solve another unpleasant feature of real systems: *exceptions*.

Since exceptions can happen anywhere and anytime, the whole specification would have to be qualified by $\natural$, meaning that every single axiom, mode and transition is qualified as "normally". This would make the refinement effort to successively eliminate all occurrences of $\natural$ unbearable. Hence we need other means to systematically cope with this kind of global pathology.

Raising an exception interrupts some existing computation flow, and therefore may not preserve liveness properties. *Catching* an exception introduces some new computation flows, and therefore may not preserve safety properties. That is why imposing policies, to distinguish normal behaviors and to handle exceptional behaviors, is a challenge for systematic system design.

More precisely, we are given a basic system $\mathsf{Syst}_\natural$ satisfying a behavior B under "normal" circumstances, i.e. as long as there are no exceptions: $\mathsf{Syst}_\natural \models B$. From this system we want to derive a system Syst_E satisfying B whenever the norm $\neg E$ (no exceptions) is satisfied, otherwise satisfying the handling requirement H. Formally, we require:

$$\mathsf{Syst}_E \models (\neg E \Rightarrow B) \ \wedge \ (E \Rightarrow H) \tag{14}$$

This is realized by building system Syst_E with

$$\mathsf{Syst}_E \models (\neg E \wedge B) \ \mathcal{W} \ (E \wedge H) \tag{15}$$

The system Syst_E is systematically obtained from $\mathsf{Syst}_\natural$ as follows:

– the modes of Syst_E are:

  - the modes of $\mathsf{Syst}_\natural$,
  - an adjoined *handling* mode $\mathcal{H}$,
- the transitions of Syst_E are:
  - for each transition $\mathcal{M} \xrightarrow{\langle f,p\rangle \Rrightarrow t} \mathcal{N}$ in $\mathsf{Syst}_\natural$ a transition $\mathcal{M} \xrightarrow{\langle f,\neg E \wedge p\rangle \Rrightarrow t} \mathcal{N}$ in Syst_E, provided $f \Rightarrow \neg E$, and
  - for each mode $\mathcal{M}$ of $\mathsf{Syst}_\natural$ a new transition $\mathcal{M} \xrightarrow{E \Rrightarrow \widehat{t}} \mathcal{H}$ in Syst_E, where $\widehat{t}$ initializes the variables of H.

Proposition 3 Syst_E *satisfies (15) and hence (14).*

As can be seen here easily, this is a global treatment of a global normally operator: Such an operator implicitly qualifies all modes and transitions by $\natural$. And the above construction simply refines all the different predicate symbols (to which these $\natural$ operators correspond) by one single predicate $\neg\, E$.

The question is: *What kind of espec morphism supports refinements in the form* $\mathsf{Syst}_\natural \longrightarrow \mathsf{Syst}_E$*?* The problem is that the mode $\mathcal{H}$, adjoined in Syst_E does not arise from any mode present in $\mathsf{Syst}_\natural$.

One possible answer is to first extend $\mathsf{Syst}_\natural$ by an *unreachable* mode $\mathcal{H}$, with a transition from each mode $\mathcal{M}$, but guarded by $\bot$. Such an *unreachable abstraction* $\mathsf{Syst}_\natural \longleftarrow \mathsf{Syst}_\bot$ leads to an espec $\mathsf{Syst}_\bot$ semantically equivalent to $\mathsf{Syst}_\natural$; that is, both systems have the same traces. So $\mathsf{Syst}_\natural$ and $\mathsf{Syst}_\bot$ satisfy the same properties.

On the other hand, $\mathsf{Syst}_\bot$ can be refined to Syst_E. This refinement, of course, does not preserve safety, since $\mathcal{H}$ is not unreachable in Syst_E. However, it is not hard to prove that

Proposition 4 *The span* $\mathsf{Syst}_\natural \longleftarrow \mathsf{Syst}_\bot \longrightarrow \mathsf{Syst}_E$*, viewed as a generalized morphism, preserves liveness, and moreover it preserves all properties* B *of* $\mathsf{Syst}_\natural$*, relativized to* $\neg\, E$*, in the sense of (15).*

5 Parameterized Especs: Modeling the Environment

In system design, the need arises to specify the properties and behavior of a component's environment, including required behaviors, invariant properties, and required services. The correctness of the component's behavior follows from the assumption that the environment behaves as specified, together with the internal structure and behavior of the component. This is sometimes referred to as the "rely-guarantee" paradigm. Parameterized especs neatly satisfy this need.

A parameter to an espec is an espec that models the environment – what behavior and properties the component expects of the environment, and what services it requires. The binding of a parameter to the environment is given by an espec morphism π – the environment is expected to be a refinement of the parameter. The environment will typically have much more structure and

behavior than is specified by the parameter, but it must have at least as much as is required for the correct operation of the component.

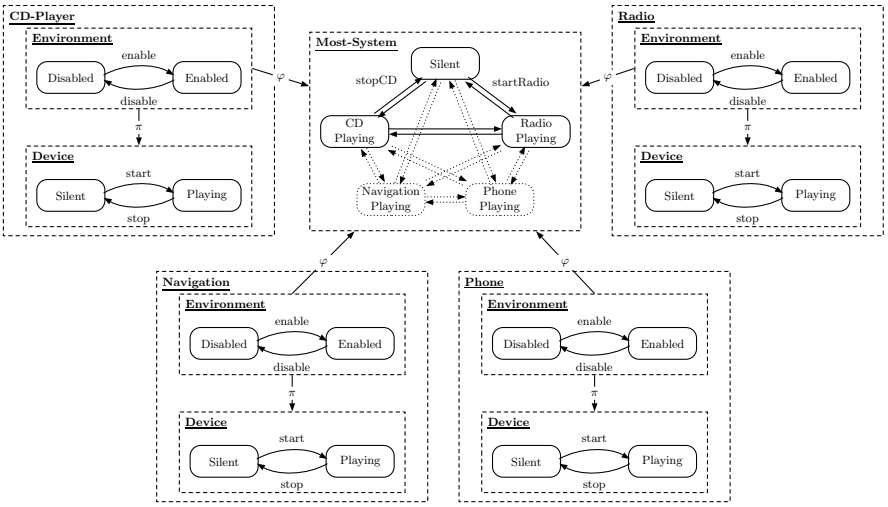


Fig. 5. Modularization through parameter morphisms

In our running example of the MOST bus this will typically lead to situations as depicted in Fig. 5. Each device has a *body specification* for the device proper and a *parameter specification* for its interface to the context; both are linked through a parameter morphism π. The interfaces are then linked to the overall system, i.e. the MOST bus, through refinement morphisms φ. This makes it relatively easy to add any number of components to some MOST system without running into an unmanageable combinatorial explosion of the size of the specifications and, above all, of the number of interconnections.

This raises the question of the refinement of parameterized specifications. Fig. 6 shows an example of such a refinement.

In the original abstract model an audio device is simply assumed to switch between the two modes *Silent* and *Playing*. Accordingly the environment is expected to consider the device as *Disabled* or *Enabled* (with the appropriate matchings).

However, in the MOST bus the enabling of a device is performed by a full-fledged *connection protocol*: In order to become playing, the device needs a number of channels to be allocated by the MOST bus. After having received them, the device still cannot play, since the channels also need to be allocated to the amplifiers. Therefore the device has to go into an intermediate mode *Ready*, while the environment is in the mode *Connecting*. (It is only by chance that the number of modes and transitions in the parameter and the body coincide in this example. In general they will be different.)

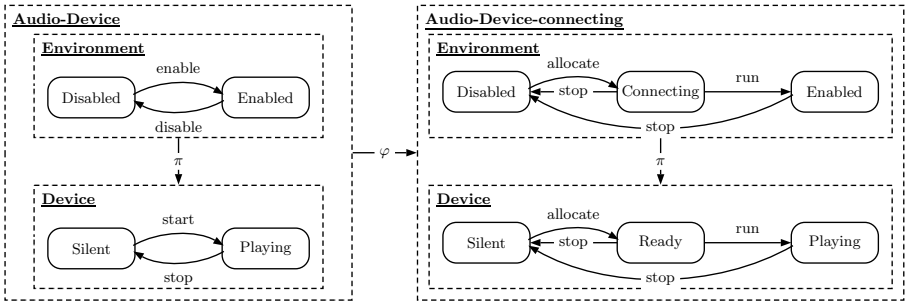


Fig. 6. Refinement of parameterized specifications

This refinement is realized by the morphism that is sketched in Fig. 6. However, this diagram only gives a rough idea of the construction, since it does not express the various compatibility constraints between the two parameter morphisms and the refinement morphism φ. Fortunately most of them are generated automatically by the category-theoretic principles underlying the construction.

Consider the situation of Fig. 5 and the little program in Section 4. Let us assume that the parameter specification establishes $\natural\#(channels) = 4$ in the mode *Enabled*.

Now consider the mode *CD-Playing* of the MOST system in Fig. 5 and suppose that it only contains the assertion $\#(channels) \geq 2$. What does this mean in our overall design?

Due to our various morphisms we need to establish the property

$$CD\text{-}Playing \models \natural\, \#(channels) = 4 \tag{16}$$

This leaves us with the problem of establishing the property (with an uninterpreted symbol *nrml*)

$$\#(channels) \geq 2 \models nrml \Rightarrow \#(channels) = 4 \tag{17}$$

So the further refinements must add interpretations to *nrml* that allow us to complete this required proof. In practice this means that upon connection establishment the CD player needs to obtain the required number of channels from the MOST system, which is stored in a local variable $chNr$. Then *nrml* simply is refined to $chNr = 4$. The span construction of Proposition 4 adds a new transition to a handler mode for the case when $chNr < 4$.

6 Conclusion

The methodology that we have presented in this paper has a number of benefits. It allows the *systematic incremental development of models* as opposed to the predominant current practice of creating monolithic models in a more or less informally crafted process. Moreover, the resulting *models are formally specified*,

which allows not only automatic code generation (at least of prototypes), but also supports all kinds of analyses, ranging from logical consistency or completeness checks to plain testing.

With respect to the underlying formalism, a lot of work still needs to be done. For example, we currently study different approaches to the role of the "normally" operator and its refinement. Also, the role of guard refinement in the context of automotive applications needs to be assessed in greater detail, in particular with respect to liveness vs. safety preservation. Moreover, the role of (dynamic) addition and deletion of components needs to be further investigated. Yet another challenge is to provide a more convenient notation that will be more readily accepted by engineers.

References

1. The MOST cooperation. http://www.mostcooperation.com/home/index.html.
2. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
3. J. A. Goguen. Categorical foundations for general systems theory. In F. Pichler and R. Trappl, editors, *Advances in Cybernetics and Systems Research*, pages 121–130. Transcripta Books, 1973.
4. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for computer science. Technical Report CSLI-85-30, Stanford University, 1985.
5. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for computer science. *Journal of the ACM*, 39(1):95–146, 1992.
6. Huttel and Larsen. The use of static constructs in a modal process logic. In *LFCS: The 1st International Symposium on Logical Foundations of Computer Science*, 1989.
7. J.Fiadeiro, A.Lopes, and M.Wermelinger. A mathematical semantics for architectural connectors. In *FASE-03, LNCS 2793*, pages 190–234, 2003.
8. Kestrel Institute. *Specware System and documentation*, 2003. http://www.specware.org/.
9. A. Lopes and J. L. Fiadeiro. Using explicit state to describe architechtures. In J.-P. Finance, editor, *FASE*, volume 1577 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 1999.
10. D. Pavlovic, P. Pepper, and D. R. Smith. Colimits for concurrent collectors. In N. Dershowitz, editor, *Verification: Theory and Practice: Festschrift for Zohar Manna*, pages 568–597. LNCS 2772, 2003.
11. D. Pavlovic and D. R. Smith. Composition and refinement of behavioral specifications. In *Proceedings of Sixteenth International Conference on Automated Software Engineering*, pages 157–165. IEEE Computer Society Press, 2001.
12. D. Pavlovic and D. R. Smith. Guarded transitions in evolving specifications. In H. Kirchner and C. Ringeissen, editors, *Proceedings of AMAST 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 411–425. Springer Verlag, 2002.