Dynamic Logic with Binders and Its Application to the Development of Reactive Systems

Alexandre Madeira[1(✉)], Luis S. Barbosa[1], Rolf Hennicker[2], and Manuel A. Martins[3]

[1] HASLab INESC TEC, University of Minho, Braga, Portugal
madeira@ua.pt
[2] Ludwig-Maximilians-Universität München, Munich, Germany
[3] CIDMA - Department of Mathematics, University of Aveiro, Aveiro, Portugal

Abstract. This paper introduces a logic to support the specification and development of reactive systems on various levels of abstraction, from property specifications, concerning e.g. safety and liveness requirements, to constructive specifications representing concrete processes. This is achieved by combining binders of hybrid logic with regular modalities of dynamic logics in the same formalism, which we call $\mathcal{D}^{\downarrow}$-logic. The semantics of our logic focuses on effective processes and is therefore given in terms of reachable transition systems with initial states. The second part of the paper resorts to this logic to frame stepwise development of reactive systems within the software development methodology proposed by Sannella and Tarlecki. In particular, we instantiate the generic concepts of constructor and abstractor implementations by using standard operators on reactive components, like relabelling and parallel composition, as constructors, and bisimulation for abstraction. We also study vertical composition of implementations which relies on the preservation of bisimularity by the constructions on labeleld transition systems.

1 Introduction

The quest for suitable notions of *implementation* and *refinement* has been for more than four decades on the research agenda for rigorous Software Engineering. It goes back to Hoare's paper on data refinement [16], which influenced the whole family of model-oriented methods, starting with VDM [18]. A recent reference [30] collects a number of interesting refinement case studies in the B method, probably the most industrially successful in the family.

Almost 30 years ago, D. Sannella and A. Tarlecki claimed, in what would become a most influential paper in (formal) Software Engineering [28], that *"the program development process is a sequence of implementation steps leading from a specification to a program"*. Being rather vague on what was to be understood either by specifications (*"just finite syntactic objects of some kind"* which *"describe a certain signature and a class of models over it"*) or programs (*"which for us are just very tight specifications"*), the paper focuses entirely on the development process, based on a notion of refinement. In model-oriented approaches it is consensual that a specification refines to another if every model of the latter is a model

A. Sampaio and F. Wang (Eds.): ICTAC 2016, LNCS 9965, pp. 422–440, 2016.
DOI: 10.1007/978-3-319-46750-4_24

of the former. Sannella and Tarlecki's work complemented and generalised this idea with the notions of *"constructor"* and *"abstractor implementations"*. The idea of a constructor implementation is that for implementing a specification *SP* one may use one or several given specifications and apply a construction on top of them to satisfy the requirements of *SP*. Abstractor implementations have been introduced to deal with the fact that sometimes the properties of a requirements specification are not literally satisfied by an implementation but only up to an abstraction which usually involves hiding of implementation details. Over time, many others contributed along similar paths, with Sannella and Tarlecki's specific view later consolidated in their landmark book [29]. All main ingredients were already there: (i) the emphasis on *loose* specifications; (ii) correctness by construction, guaranteed by vertical compositionality and (iii) genericity, as the development process is independent, or parametric, on whatever logical system better captures the requirements to be handled.

Our paper investigates this approach in the context of reactive software, *i.e.* systems which interact with their environment along the whole computation, and not only in its starting and termination points [1]. The relevance of such an effort is anticipated in Sannella and Tarlecki's book [29] itself: *"An example of an area for which a satisfactory, commonly accepted solution still seems to be outstanding (despite numerous proposals and active research) is the theory of concurrency"* (page 157). Different approaches in that direction have been proposed, of which we single out an extension to concurrency in K. Havelund's Ph.D. thesis [15]. The book, however, focused essentially on functional requirements expressed by algebraic specifications and implemented in a functional programming language.

On the other hand, the development of reactive systems, nowadays the norm rather than the exception, followed a different path. Typical approaches start from the construction of a concrete model (e.g. in the form of a transition system [31], a Petri net [26] or a process algebra expression [4,17]) upon which the relevant properties are later formulated in a suitable (modal) logic and typically verified by some form of model-checking. Resorting to old software engineering jargon, most of these approaches proceed by *inventing & verifying*, whereas this paper takes the alternative *correct by construction* perspective.

Our hypothesis is that also in the domain of reactive systems, loose specification has an important role to play, because they support the gradual addition of requirements and implementation decisions such that verification of the correctness of a complex system can be done piecewise in smaller steps. Thus also a documentation keeping trace of design decisions is available supporting maintenance and extensibility of systems. Therefore, our challenge was twofold. First to design a logic to support the development of reactive systems at different levels of abstraction. Second, to follow Sannella and Tarlecki's recipe according to which *"specific notions of implementation (...) corresponds to a restriction on the choice of constructors and abstractors which may be used"* [28]. The paper's contributions respond to such challenges:

– Borrowing modalities indexed by regular expressions of actions, from dynamic logic [14], and state variables and binders, from hybrid logic [6], a new logic,

$\mathcal{D}^{\downarrow}$, is proposed to express properties of computations of reactive systems. $\mathcal{D}^{\downarrow}$ is able to express abstract properties, such as liveness requirements or deadlock avoidance, but also to describe concrete, recursive process structures implementing them. Note that our focus is actually on computations, and therefore on transition structures over reachable states with an initial point, rather than on arbitrary relational structures with global satisfaction, as usual in modal logic. Symbol $\downarrow$ in $\mathcal{D}^{\downarrow}$ stands for the *binder* operator borrowed from hybrid logic: $\downarrow x.\phi$ evaluates ϕ and assigns to variable x the current state of evaluation.

– Then, a particular pallete of constructors and abstractors found relevant to the development of reactive systems, is introduced. Interestingly, it turns out that requirements of Sannella and Tarlecki's methodology for vertical composition of abstractor/constructor implementations is just the congruence property of bisimulation w.r.t. constructions on labelled transition systems, like parallel composition and relabelling.

The new $\mathcal{D}^{\downarrow}$ logic is introduced in Sect. 2. Then, the two following sections, 3 and 4, respectively, introduce the development method, with a brief revision of the relevant background, and its tuning to the design of reactive systems. Finally, Sect. 5 concludes and points out some issues for future work. To respect the page limit fixed for the Conference, all proofs were removed from the paper. They appear in the accompanying technical report [21].

2 $\mathcal{D}^{\downarrow}$ - A Dynamic Logic with Binders

2.1 $\mathcal{D}^{\downarrow}$-logic: Syntax and Semantics

$\mathcal{D}^{\downarrow}$ logic is designed to express properties of reactive systems, from abstract safety and liveness properties, down to concrete ones specifying the (recursive) structure of processes. It thus combines modalities with regular expressions, as originally introduced in Dynamic Logic [14], and binders in state variables. This logic retains from Hybrid Logic [6], only state variables and the binder operator first studied by V. Goranko in [11]. These motivations are reflected in its semantics. Differently from what is usual in modal logics, whose semantics are given by Kripke structures and the satisfaction evaluated globally in each model, $\mathcal{D}^{\downarrow}$ models are reachable transition systems with initial states where satisfaction is evaluated.

Definition 1 (Model). Models for a finite set of atomic actions A *are reachable* A*-LTSs, i.e. triples* (W, w_0, R) *where* W *is a* set of states, $w_0 \in W$ *is the* initial state *and* $R = (R_a \subseteq W \times W)_{a \in A}$ *is a family of* transition relations *such that, for each* $w \in W$*, there is a finite sequence of transitions* $R_{a^k}(w^{k-1}, w^k)$, $1 \leq k \leq n$, *with* $w_k \in W$, $a^k \in A$, *such that* $w_0 = w^0$ *and* $w^n = w$.

The *set of (structured) actions*, $\mathrm{Act}(A)$, induced by a set of atomic actions A is given by

$$\alpha ::= a \mid \alpha; \alpha \mid \alpha + \alpha \mid \alpha^*$$

where $a \in A$.

Let X be an infinite set of variables, disjoint with the symbols of the atomic actions A. A valuation for an A-model $\mathcal{M} = (W, w_0, R)$ is a function $g : X \to W$. Given such a g and $x \in X$, $g[x \mapsto w]$ denotes the valuation for $\mathcal{M}$ such that $g[x \mapsto w](x) = w$ and $g[x \mapsto w](y) = g(y)$ for any other $y \neq x \in X$.

Definition 2 (Formulas and sentences). *The set* $\mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A)$ *of* A-formulas *is given by*

$$\varphi ::= \mathbf{tt} \mid \mathbf{ff} \mid x \mid \downarrow x.\varphi \mid @_x\varphi \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

where $x \in X$ *and* $\alpha \in \mathrm{Act}(A)$. $\mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A) = \{\varphi \in \mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A) | \mathrm{FVar}(\varphi) = \emptyset\}$ *is the set of* A*-sentences, where* $\mathrm{FVar}(\varphi)$ *are the free variables of* φ*, defined as usual with* $\downarrow$ *being the unique operator binding variables.*

$\mathcal{D}^{\downarrow}$ retains from Hybrid Logic the use of binders, but omits nominals: only state variables are used, even as parameters to the satisfaction operator ($@_x$). By doing so, the logic becomes restricted to express properties of reachable states from the initial state, i.e. processes.

To define the satisfaction relation we need to clarify how composed actions are interpreted in models. Let $\alpha \in \mathrm{Act}(A)$ and $\mathcal{M} \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$. The interpretation of an action α in $\mathcal{M}$ extends the interpretation of atomic actions by $R_{\alpha;\alpha'} = R_\alpha \circ R_{\alpha'}$, $R_{\alpha+\alpha'} = R_\alpha \cup R_{\alpha'}$ and $R_{\alpha^*} = (R_\alpha)^\star$, with the operations $\circ$, $\cup$ and $\star$ standing for relational composition, union and Kleene closure.

Given an A-model $\mathcal{M} = (W, w_0, R)$, $w \in W$ and $g : X \to W$,

- $\mathcal{M}, g, w \models \mathbf{tt}$ is true; $\mathcal{M}, g, w \models \mathbf{ff}$ is false;
- $\mathcal{M}, g, w \models x$ iff $g(x) = w$;
- $\mathcal{M}, g, w \models \downarrow x.\varphi$ iff $\mathcal{M}, g[x \mapsto w], w \models \varphi$;
- $\mathcal{M}, g, w \models @_x\varphi$ iff $\mathcal{M}, g, g(x) \models \varphi$;
- $\mathcal{M}, g, w \models \langle\alpha\rangle\varphi$ iff there is a $w' \in W$ with $(w, w') \in R_\alpha$ and $\mathcal{M}, g, w' \models \varphi$;
- $\mathcal{M}, g, w \models [\alpha]\varphi$ iff for any $w' \in W$ with $(w, w') \in R_\alpha$ it holds $\mathcal{M}, g, w' \models \varphi$;
- $\mathcal{M}, g, w \models \neg\varphi$ iff it is false that $\mathcal{M}, g, w \models \varphi$;
- $\mathcal{M}, g, w \models \varphi \wedge \varphi'$ iff $\mathcal{M}, g, w \models \varphi$ and $\mathcal{M}, g, w \models \varphi'$;
- $\mathcal{M}, g, w \models \varphi \vee \varphi'$ iff $\mathcal{M}, g, w \models \varphi$ or $\mathcal{M}, g, w \models \varphi'$.

We write $\mathcal{M}, w \models \varphi$ if, for any valuation $g : X \to W$, $\mathcal{M}, g, w \models \varphi$. If φ is a sentence, then the valuation is irrelevant, i.e., $\mathcal{M}, g, w \models \varphi$ iff $\mathcal{M}, w \models \varphi$. For each sentence $\varphi \in \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A)$, we write $\mathcal{M} \models \varphi$ whenever $\mathcal{M}, w_0 \models \varphi$. Observe again the pertinence of avoiding nominals: if a formula is satisfied in the standard semantics of Hybrid Logic, then it is satisfiable in ours. Obviously, this would not happen in the presence of nominals.

The remaining of the section discusses the versatility of $\mathcal{D}^{\downarrow}$ claimed in the introductory section. Here and in the following sentences, in the context of a set of actions $A = \{a_1, \ldots, a_n\}$, we write A for the complex action $a_1 + \ldots + a_n$ and for any $a_i \in A$, we write $-a_i$ for the complex action $a_1 + \ldots + a_{i-1} + a_{i+1} + \ldots + a_n$.

By using regular modalities from Dynamic Logic [13,14], $\mathcal{D}^{\downarrow}$ is able to express liveness requirements such as "*after the occurrence of an action a, an action b can be eventually realised*" with $[A^*; a]\langle A^*; b\rangle\mathbf{tt}$ or "*after the occurrence of an action a, an occurrence of an action b is eventually possible if it has not occurred before*" with $[A^*; a; (-b)^*]\langle A^*; b\rangle\mathbf{tt}$. Safety properties are also captured by sentences of the form $[A^*]\varphi$. In particular, *deadlock freeness* is expressed by $[A^*]\langle A\rangle\mathbf{tt}$.

Example 1. As a running example we consider a product line with a stepwise development of a product for compressing files services, involving compressions of text and of image files. We start with an abstract requirements specification SP_0. It is built over the set $A = \{inTxt, inGif, outZip, outJpg\}$ of atomic actions *inTxt*, *inGif* for inputting a txt-file or a gif-file, and actions *outZip*, *outJpg* for outputting a zip-file or a jpg-file. Sentences (0.1)–(0.3) below express three requirements: (0.1) Whenever a txt-file has been received for compression, the next action must be an output of a zip-file, (0.2) whenever a gif-file has been received, the next action must be an output of a jpg-file, and (0.3) the system should never terminate.

(0.1) $[A^*; inTxt]\big(\langle outZip\rangle\mathbf{tt} \wedge [-outZip]\mathbf{ff}\big)$
(0.2) $[A^*; inGif]\big(\langle outJpg\rangle\mathbf{tt} \wedge [-outJpg]\mathbf{ff}\big)$
(0.3) $[A^*]\langle A\rangle\mathbf{tt}$

Obviously, SP_0 is a very loose specification of rudimentary requirements and there are infinitely many models which satisfy the sentences (0.1)–(0.3). □

$\mathcal{D}^{\downarrow}$-logic, however, is also suited to directly express process structures and, thus, the implementation of abstract requirements. The binder operator is crucial for this. The ability to give names to visited states, together with the modal features to express transitions, makes possible a precise description of the whole dynamics of a process in a single sentence. Binders allow to express recursive patterns, namely loop transitions (from the current to some visited state). In fact we have no way to make this kind of specification in the absence of a feature to refer to specific states in a model, as in standard modal logic. For example, sentence

$$\downarrow x_0.\big(\langle a\rangle x_0 \wedge \langle b\rangle \downarrow x_1.(\langle a\rangle x_0 \wedge \langle b\rangle x_1)\big) \tag{1}$$

specifies a process with two states accepting actions a and b respectively. As discussed in the sequel, the stepwise development of a reactive system typically leads to a set of requirements defining concrete transition systems and expressed in the fragment of $\mathcal{D}^{\downarrow}$ which omits modalities indexed by the Kleene closure of actions, that can be directly translated into a set of FSP [22] definitions. Figure 1 depicts the translation of the formula above as computed by a proof-of-concept implementation of such a translator[1]. Note, however, that sentence (1) loosely specifies the purposed scenario (e.g. a single state system looping on a and b also satisfies this requirement). Resorting to full $\mathcal{D}^{\downarrow}$ concrete processes unique up to isomorphism, can be defined, i.e. we may introduce monomorphic specifications.

[1] See `translator.nrc.pt`.

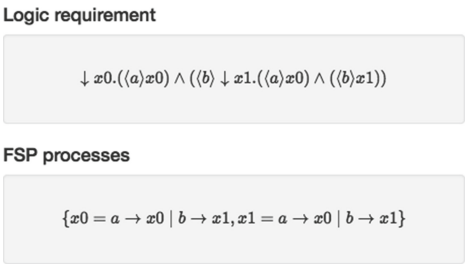

Fig. 1. D2FSP Translator: Translating $\mathcal{D}^{\downarrow}$ into FSP processes.

For this specific example, it is enough to consider, in the conjunction in the scope of x_1, the term $@_{x_1}\neg x_0$ (to distinguish between the states binded by x_0 and x_1), as well as to enforce determinism resorting to formula (det) in Example 2.

2.2 Turning $\mathcal{D}^{\downarrow}$-logic into an Institution

In order to fit the necessary requirements to adopt the Sannella Tarlecki development method, logic $\mathcal{D}^{\downarrow}$ has to be framed as a logical institution [10].

In this view, our first concern is about the *signatures category*. As suggested, signatures for $\mathcal{D}^{\downarrow}$ are finite sets A of *atomic actions*, and a signature morphism $A \xrightarrow{\sigma} A'$ is just a function $\sigma : A \to A'$. Clearly, this entails a category to be denoted by $\mathrm{Sign}^{\mathcal{D}^{\downarrow}}$.

Our second concern is about the *models functor*. Given two models, $\mathcal{M} = (W, w_0, R)$ and $\mathcal{M}' = (W', w_0', R')$, for a signature A, a *model morphism* is a function $h : W \to W'$ such that $h(w_0) = w_0'$ and, for each $a \in A$, if $(w_1, w_2) \in R_a$ then $(h(w_1), h(w_2)) \in R'_a$. We can easily observe that the class of models for A, and the corresponding morphisms, defines a category $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$.

Definition 3 (Model reduct). *Let* $A \xrightarrow{\sigma} A'$ *be a signature morphism and* $\mathcal{M}' = (W', w_0', R')$ *an* A'*-model. The* σ-reduct of $\mathcal{M}'$ *is the* A*-model* $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}') = (W, w_0, R)$ *such that*

- $w_0 = w_0'$;
- W *is the largest set with* $w_0' \in W$ *and, for each* $v \in W$*, either* $v = w_0'$ *or there is a* $w \in W$ *such that* $(w, v) \in R'_{\sigma(a)}$*, for some* $a \in A$;
- *for each* $a \in A$, $R_a = R'_{\sigma(a)} \cap W^2$.

Models morphisms are preserved by reducts, in the sense that, for each models morphism $h : \mathcal{M}_1' \to \mathcal{M}_2'$ there is a models morphism $h' : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}_1') \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}_2')$, where h' is the restriction of h to the states of $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}_1')$. Hence, for each signature morphism $A \xrightarrow{\sigma} A'$, a functor $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma) : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A') \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$ maps models and morphisms to the corresponding

reducts. Finally, this lifts to a contravariant *models functor*, $\mathrm{Mod}^{\mathcal{D}^{\downarrow}} : (\mathrm{Sign}^{\mathcal{D}^{\downarrow}})^{op} \to \mathbb{C}at$, mapping each signature to the category of its models and, each signature morphism to its reduct functor.

The third concern is about the definition of the *functor of sentences*. Each signature morphism $A \xrightarrow{\sigma} A'$ can be extended to formulas' translation $\hat{\sigma} : \mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A) \to \mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A')$ identifying variables and replacing, symbol by symbol, each action by the respective σ-image. In particular, $\hat{\sigma}(\downarrow x.\varphi) = \downarrow x.\hat{\sigma}(\varphi)$ and $\hat{\sigma}(@_x\varphi) = @_x\hat{\sigma}(\varphi)$. Since $\mathrm{FVar}(\varphi) = \mathrm{FVar}(\hat{\sigma}(\varphi))$ we can assure that, for each signature morphism $A \xrightarrow{\sigma} A'$, we can define a translation of sentences $\mathrm{Sen}^{\mathcal{D}^{\downarrow}}(\sigma) : \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A) \to \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A')$, by $\mathrm{Sen}^{\mathcal{D}^{\downarrow}}(\sigma)(\varphi) = \hat{\sigma}(\varphi)$, $\varphi \in \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A)$. This entails the intended functor $\mathrm{Sen}^{\mathcal{D}^{\downarrow}} : \mathrm{Sign}^{\mathcal{D}^{\downarrow}} \to \mathbb{S}et$, mapping each signature to the set of its sentences, and each signature morphism to the corresponding translation of sentences.

Finally, our forth concern is on the agreement of the satisfaction relation w.r.t. *satisfaction condition*. This is established in the following result:

Theorem 1. *Let $\sigma : A \to A'$ be a signature morphism, $\mathcal{M}' = (W', w_0', R') \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A')$, $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}') = (W, w_0, R)$ and $\varphi \in \mathrm{Fm}^{\mathcal{D}^{\downarrow}}(A)$. Then, for any $w \in W (\subseteq W')$ and for any valuations $g : X \to W$ and $g' : X \to W'$, such that, $g(x) = g'(x)$ for all $x \in \mathrm{FVar}(\varphi)$, we have*

$$\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), g, w \models \varphi \text{ iff } \mathcal{M}', g', w \models \hat{\sigma}(\varphi)$$

In order to get the satisfaction condition, we only have to note that for any $\varphi \in \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(A)$, we have $\mathrm{FVar}(\varphi) = \emptyset$, and hence, by Theorem 1, for any $w \in W$, $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}'), w \models \varphi$ iff $\mathcal{M}', w \models \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(\sigma)(\varphi)$. Moreover, by the definition of reduct, $w_0 = w_0' \in W$. Therefore, $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}') \models \varphi$ iff $\mathcal{M}' \models \mathrm{Sen}^{\mathcal{D}^{\downarrow}}(\sigma)(\varphi)$.

3 Formal Development *á la* Sannella and Tarlecki

Developing correct programs from specifications entails the need for a suitable logic setting in which meaning can be assigned both to specifications and their refinement. Sannella and Tarlecki have proposed a formal development methodology [28,29] which is presented in a generic way for arbitrary logical systems forming an institution. As already pointed out in the Introduction, Sannella and Tarlecki have studied various algebraic institutions to illustrate their methodology and they presume the lack of a satisfactory solution in the theory of concurrency. In this section we briefly summarize their crucial principles for formal program development over an arbitrary institution and we illustrate the case of simple implementations by examples of our $\mathcal{D}^{\downarrow}$-logic institution. The more involved concepts of constructor and abstractor implementations will be instantiated for the case of $\mathcal{D}^{\downarrow}$-logic later on in Sect. 4.

In the following we assume given an arbitrary institutions with category Sign of signatures and signature morphisms, with sentence functor Sen : Sign $\to \mathbb{S}et$,

and with models functor Mod : $\mathrm{Sign}^{op} \to \mathbb{C}at$ assigning to any signature $\Sigma \in |\mathrm{Sign}|$ a category $\mathrm{Mod}(\Sigma)$ whose objects in $|\mathrm{Mod}(\Sigma)|$ are called Σ-models. As usual, the class of objects of a category C is denoted by $|C|$. If it is clear from the context, we will simply write C for $|C|$.

3.1 Simple Implementations

The simplest way to design a specification is by expressing the system requirements by means of a set of sentences over a suitable signature, i.e. as a pair $SP = (Sig(SP), Ax(SP))$ where $Sig(SP) \in |\mathrm{Sign}|$ and $Ax(SP) \subseteq |\mathrm{Sen}(Sig(SP))|$. The (loose) semantics of such a *flat* specification SP consists of the pair $(Sig(SP), Mod(SP))$ where

$$Mod(SP) = \{M \in |Mod(Sig(SP))| : \ M \models Ax(SP)\}.$$

In this context, a refinement step is understood as a restriction of an abstract class of models to a more concrete one. Following the terminology of Sannella and Tarlecki, we will call a specification which refines another one an *implementation*. Formally, a specification SP' is a *simple implementation* of a specification SP over the same signature, in symbols $SP \rightsquigarrow SP'$, whenever $Mod(SP) \supseteq Mod(SP')$. Transitivity of the inclusion relation ensures the *vertical composition* of simple implementation steps.

Example 2. We illustrate two refinement steps with simple implementations in the $\mathcal{D}^{\downarrow}$-logic institution. Consider the specification SP_0 of Example 1 which expresses some rudimentary requirements for the behavior of compressing files services. The action set A defined in Example 1 provides the signature of SP_0 and the axioms of SP_0 are the three sentences (0.1)–(0.3) shown in Example 1.

First refinement step $SP_0 \rightsquigarrow SP_1$. SP_0 is a very loose specification which would allow to start a computation with an arbitrary action. We will be a bit more precise now and require that at the beginning only an input (of a text or gif file) is allowed; see axiom (1.1) below. Moreover whenever an output action (of any kind) has happened then the system must go on with an input (of any kind); see axiom (1.4). This leads to the specification SP_1 with $Sig(SP_1) = Sig(SP_0) = A$ and with the following set of axioms $Ax(SP_1)$:

(1.1) $\langle inTxt + inGif \rangle \mathbf{tt} \wedge [outZip + outJpg]\mathbf{ff}$
(1.2) $[A^*; inTxt]\big(\langle outZip \rangle \mathbf{tt} \wedge [-outZip]\mathbf{ff}\big)$
(1.3) $[A^*; inGif]\big(\langle outJpg \rangle \mathbf{tt} \wedge [-outJpg]\mathbf{ff}\big)$
(1.4) $[A^*; (outZip + outJpg)]\big(\langle inTxt + inGif \rangle \mathbf{tt} \wedge [outZip + outJpg]\mathbf{ff}\big)$

It is easy to check that $SP_0 \rightsquigarrow SP_1$ holds: Axioms (0.1) and (0.2) of SP_0 occur as axioms (1.2) and (1.3) in SP_1. It is also easy to see that non-termination (axiom (0.3) of SP_0) is guaranteed by the axioms of SP_1.

The level of underspecification is, at this moment, still very high. Among the infinitely many models of SP_1, we can find, as an admissible model the LTS shown in Fig. 2 with initial state w_0 and with an alternating compression mode.

Second refinement step $SP_1 \rightsquigarrow SP_2$. This step rules out alternating behaviours as shown above. The first axiom (2.1) of the following specification SP_2

is equivalent to axiom (1.1) of SP_1. Alternating behaviours are ruled out by axioms (2.2) and (2.3) which require that after any text compression and after any image compression the initial state must be reached again. To express this we need state variables and binders which are available in $\mathcal{D}^{\downarrow}$-logic. In our example we introduce one state variable x_0 which names the initial state by using the binder at the beginning of axioms (2.2) and (2.3). Moreover, we only want to admit deterministic models such that in any (reachable) state there can be no two outgoing transitions with the same action. It turns out that $\mathcal{D}^{\downarrow}$-logic also allows to specify this determinism property with the set of axioms (det) shown below. This leads to the specification SP_2 with $Sig(SP_2) = Sig(SP_1) = A$ and with axioms $Ax(SP_2)$:

(2.1) $(\langle inTxt\rangle \mathbf{tt} \vee \langle inGif\rangle \mathbf{tt}) \wedge [outZip + outJpg]\mathbf{ff}$
(2.2) $\downarrow x_0.\, [inTxt]\big(\langle outZip\rangle x_0 \wedge [-outZip]\mathbf{ff}\big)$
(2.3) $\downarrow x_0.\, [inGif]\big(\langle outJpg\rangle x_0 \wedge [-outJpg]\mathbf{ff}\big)$
(det) For each $a \in A$, the axiom: $[A^*] \downarrow x.(\langle a\rangle \mathbf{tt} \Rightarrow (\langle a\rangle \downarrow y.\, @_x[a]y))$

Clearly SP_2 fulfills the requirements of SP_1, i.e. $SP_1 \rightsquigarrow SP_2$. SP_2 has three models which are shown in Fig. 3. (Remember that models can only have states reachable from the initial one.) The first model allows only text compression, the second one only image compression, and the third supports both. The signature of all models is A, though in the first two some actions have no transitions.

Let us still discuss some variations of SP_2 to underpin the expressive power of $\mathcal{D}^{\downarrow}$. If we want only the model where both text and image compression are possible, then we can simply replace in axiom (2.1) $\langle inTxt\rangle \mathbf{tt} \vee \langle inGif\rangle \mathbf{tt}$ by $\langle inTxt\rangle \mathbf{tt} \wedge \langle inGif\rangle \mathbf{tt}$. If we would like to require that text compression must be possible in any model but image compression is optional, i.e. we rule out the second model in Fig. 3, then we would simply omit $\vee\langle inGif\rangle \mathbf{tt}$ in axiom (2.1). This is an interesting case since this shows that $\mathcal{D}^{\downarrow}$-logic can express so-called "may"-transitions offered by modal transition systems [20] to specify options for implementations. □

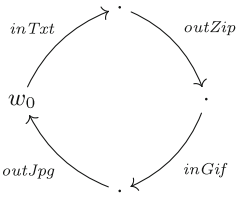

Fig. 2. A model of SP_1

3.2 Constructor Implementations

The concept of simple implementations is, in general, too strict to capture software development practice, along which, implementation decisions typically introduce new design features or reuse already implemented ones, usually entailing a change of signatures along the way. The notion of *constructor implementation* offers the necessary generalization. The idea is that for implementing a

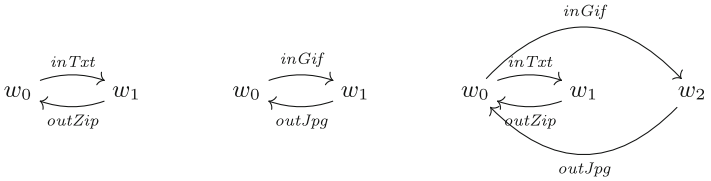

Fig. 3. Models of SP_2

specification SP one may use a given specification SP' and apply a construction to the models of SP' such that they become models of SP. More generally, an implementation of SP may be obtained by using not only one but several specifications $SP'_1, \ldots, SP'_n$ as a basis and applying an n-ary constructor such that for any tuple of models of $SP'_1, \ldots, SP'_n$ the construction leads to a model of SP. Such an implementation is called a *constructor implementation with decomposition* in [29] since the implementation of SP is designed by using several components. These ideas are formalized as follows, partially in a less general manner than the corresponding definitions in [29] which allow also partial and higher-order functions as constructors.

Given signatures $\Sigma_1, ..., \Sigma_n, \Sigma \in |\mathrm{Sign}|$, a *constructor* is a total function $\kappa : \mathrm{Mod}(\Sigma_1) \times \cdots \times \mathrm{Mod}(\Sigma_n) \to \mathrm{Mod}(\Sigma)$. Constructors compose as follows: Given a constructor $\kappa : \mathrm{Mod}(\Sigma_1) \times \cdots \times \mathrm{Mod}(\Sigma_n) \to \mathrm{Mod}(\Sigma)$ and a set of constructors $\kappa_i : \mathrm{Mod}(\Sigma_i^1) \times \cdots \times \mathrm{Mod}(\Sigma_i^{k_i}) \to \mathrm{Mod}(\Sigma_i)$, $1 \leq i \leq n$, the constructor $\kappa(\kappa_1, \ldots, \kappa_n) : \mathrm{Mod}(\Sigma_1^1) \times \cdots \times \mathrm{Mod}(\Sigma_1^{k_1}) \times \cdots \times \mathrm{Mod}(\Sigma_n^1) \times \cdots \times \mathrm{Mod}(\Sigma_n^{k_n}) \to \mathrm{Mod}(\Sigma)$ is obtained by the usual composition of functions.

Definition 4 (Constructor implementation). *Given specifications $SP, SP'_1, \ldots, SP'_n$, and a constructor $\kappa : \mathrm{Mod}(Sig(SP'_1)) \times \cdots \times \mathrm{Mod}(Sig(SP'_n)) \to Mod(Sig(SP))$, we say that $\langle SP'_1, \ldots, SP'_n \rangle$ is a* constructor implementation via *κ of SP, in symbols $SP \rightsquigarrow_\kappa \langle SP'_1, \ldots, SP'_n \rangle$, if for all $M_i \in Mod(SP'_i)$ we have $\kappa(M_1, \ldots, M_n) \in Mod(SP)$. We say that the implementation involves a decomposition if $n > 1$.*

3.3 Abstractor Implementations

Another aspect in formal program development concerns the fact that sometimes the properties of a requirements specification are not literally satisfied by an implementation but only up to an admissible abstraction. Usually such an abstraction concerns implementation details which are hidden to the user of the system and which may, for instance for efficiency reasons, not be fully conform to the requirements specification. Then the implementation is still considered to be correct if it shows the desired observable behavior. In general this can be expressed by considering an equivalence relation $\equiv$ on the models of the abstract specification and to allow the implementation models to be only equivalent to models of the requirements specification.

Formally, let SP be a specification and $\equiv \subseteq \mathrm{Mod}(Sig(SP)) \times \mathrm{Mod}(Sig(SP))$ be an equivalence relation. Let $Abs_{\equiv}(Mod(SP))$ be the closure of $Mod(SP)$ under $\equiv$. A specification SP' with the same signature as SP is a *simple abstractor implementation* of SP w.r.t. $\equiv$, whenever $Abs_{\equiv}(Mod(SP)) \supseteq Mod(SP')$. Both concepts, constructors and abstractors can be combined as shown in the definition of an abstractor implementation. (For simplicity, the term constructor is omitted.)

Definition 5 (Abstractor implementation). *Let $SP, SP'_1, \ldots, SP'_n$ be specifications, $\kappa : \mathrm{Mod}(Sig(SP'_1)) \times \cdots \times \mathrm{Mod}(Sig(SP'_n)) \to \mathrm{Mod}(Sig(SP))$ a constructor, and $\equiv \subseteq \mathrm{Mod}(Sig(SP)) \times \mathrm{Mod}(Sig(SP))$ an equivalence relation. We say that $\langle SP'_1, \cdots, SP'_n \rangle$ is an* abstractor implementation of SP via κ w.r.t. $\equiv$*, in symbols $SP \rightsquigarrow^{\equiv}_{\kappa} \langle SP'_1, \cdots, SP'_n \rangle$, if for all $M_i \in Mod(SP'_i)$ we have $\kappa(M_1, \ldots, M_n) \in Abs_{\equiv}(Mod(SP))$.*

4 Reactive Systems Development with $\mathcal{D}^{\downarrow}$

4.1 Constructor Implementations in $\mathcal{D}^{\downarrow}$-logic

This section introduces a pallete of constructors to support the formal development of reactive systems with $\mathcal{D}^{\downarrow}$, instantiating the definitions in Sect. 3.2. The idea is to lift standard constructions on labelled transition systems (see, e.g. [31]) to constructors for implementations. We will illustrate most of the constructors introduced in the following with our running example.

Along the refinement process it is sometimes convenient to reduce the action set, for instance, by omitting some actions previously introduced as auxiliary actions or as options that are no longer needed. For this purpose we use the *alphabet extension constructor*. Remember that constructors always map concrete models to abstract ones. Therefore when omitting actions in a refinement step we need an alphabet extension on the concrete models to fit them to the abstract signature.

Definition 6 (Alphabet extension). *Let $A, A' \in |\mathrm{Sign}^{\mathcal{D}^{\downarrow}}|$ be signatures in $\mathcal{D}^{\downarrow}$, i.e. action sets, such that $A \subseteq A'$. The* alphabet extension constructor $\kappa_{ext} : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A) \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A')$ *is defined as follows: For each $\mathcal{M} = (W, w_0, R) \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$, $\kappa_{ext}(\mathcal{M}) = (W, w_0, R')$ with $R'_a = R_a$ for all $a \in A$ and $R'_a = \emptyset$ for all $a \in A' \setminus A$.*

Example 3. The specification SP_2 of Example 2 has the three models shown in Fig. 3. Hence, it allows three directions to proceed further in the product line.

Third refinement step $SP_2 \rightsquigarrow_{\kappa_{ext}} SP_3$. We will consider here the simple case where we vote for a tool that supports only text compression. The following specification SP_3 is a direct axiomatisation of the first model in Fig. 3 considered over the smaller action set $A_3 = \{inTxt, outZip\}$. Hence, $Sig(SP_3) = A_3$ and the axioms in $Ax(SP_3)$ are:

(3.1) $\downarrow x_0.\,(\langle inTxt\rangle \downarrow x_1.\,(\langle outZip\rangle x_0 \wedge [inTxt]\mathbf{ff}) \wedge [outZip]\mathbf{ff})$
(det) For each $a \in A_3$, the axiom: $[A_3^*] \downarrow x.(\langle a\rangle \mathbf{tt} \Rightarrow (\langle a\rangle \downarrow y.\, @_x[a]y))$

Since the signature of SP_3 has less actions than the one of SP_2, we apply an alphabet extension constructor $\kappa_{ext} : \mathrm{Mod}^{\mathcal{D}^\downarrow}(A_3) \to \mathrm{Mod}^{\mathcal{D}^\downarrow}(A)$ which transforms the model of SP_3 into an LTS with the same states and transitions but with actions A and with an empty accessibility relation for the actions in $A \setminus A_3$. Then, trivially, $SP_2 \rightsquigarrow_{\kappa_{ext}} SP_3$ holds. Specification SP_3 is a simple example that shows how labeled transition systems can be directly specified in $\mathcal{D}^\downarrow$. This could suggest that we are already close to a concrete implementation. But this is not true, since SP_3 is in principle just an interface specification which specifies the system behavior "from the outside", i.e. its interactions with the user. □

The standard way to build reactive systems is by aggregating in parallel smaller components. The following parallel composition constructor synchronising on shared actions caters for this.

Definition 7 (Parallel composition). *Given signatures A and A' the* parallel composition constructor $\kappa_\otimes : \mathrm{Mod}^{\mathcal{D}^\downarrow}(A) \times \mathrm{Mod}^{\mathcal{D}^\downarrow}(A') \to \mathrm{Mod}^{\mathcal{D}^\downarrow}(A \cup A')$ *is a function mapping models $\mathcal{M} = (W, w_0, R) \in \mathrm{Mod}^{\mathcal{D}^\downarrow}(A)$ and $\mathcal{M}' = (W', w_0', R') \in \mathrm{Mod}^{\mathcal{D}^\downarrow}(A')$, to the $A \cup A'$-model $\mathcal{M} \otimes \mathcal{M}' = \left(W^\otimes, (w_0, w_0'), R^\otimes\right)$ where $W^\otimes \subseteq W \times W'$ and $R^\otimes = (R_a^\otimes)_{a \in A \cup A'}$ are the least sets satisfying $(w_0, w_0') \in W^\otimes$, and, for each $(w, w') \in W^\otimes$,*

- *if $a \in A \cap A'$, $(w, v) \in R_a$, $(w', v') \in R'_a$, then $(v, v') \in W^\otimes$ and $\left((w, w'), (v, v')\right) \in R_a^\otimes$;*
- *if $a \in A \setminus A'$, $(w, v) \in R_a$, then $(v, w') \in W^\otimes$ and $\left((w, w'), (v, w')\right) \in R_a^\otimes$;*
- *if $a \in A' \setminus A$, $(w', v') \in R'_a$, then $(w, v') \in W^\otimes$ and $\left((w, w'), (w, v')\right) \in R_a^\otimes$.*

Since, up to isomorphism, parallel composition is associative, the extension of this constructor to the n-ary case is straightforward. Parallel composition is a crucial operator for constructor implementations with decomposition; see Definition 4. Remember again that constructors always go from concrete models to abstract ones, i.e. in the opposite direction as the development process. Therefore the parallel composition constructor justifies the implementation of reactive systems by decomposition.

Example 4. We are now going to construct an implementation for the interface specification SP_3. in Example 3. For this purpose, we propose a decomposition into two components, a controller component *Ctrl* and a component *GZip* which does the actual text compression. The controller has the actions $A_{Ctrl} = \{inTxt, txt, zip, outZip\}$. First, it receives (action *inTxt*) a txt-file from the user. Then it hands over the text, with action *txt*, to the *GZip* component and receives the resulting zip-file (action *zip*). Finally it provides the zip-file to the user (action *outZip*) and is ready to serve a next compression. Hence, the controller component has the signature *Sig(Ctrl)* $= A_{Ctrl}$ and the following axioms *Ax(Ctrl)* specify a single model, shown in Fig. 4 (left), with the behavior as described above.

$$(4.1)\quad \downarrow x_0.\,(\langle inTxt\rangle \downarrow x_1.\,(\langle txt\rangle \downarrow x_2.\,(\langle zip\rangle \downarrow x_3.\,(\langle outZip\rangle x_0 \wedge [-outZip]\mathbf{ff}) \wedge [-zip]\mathbf{ff}) \wedge [-txt]\mathbf{ff}) \wedge [-inTxt]\mathbf{ff})$$

$$(\text{det})\quad \text{For each } a \in A_{Ctrl}\text{, the axiom: } [A^*_{Ctrl}] \downarrow x.(\langle a\rangle \mathbf{tt} \Rightarrow (\langle a\rangle \downarrow y.\,@_x[a]y))$$

The *GZip* component has the actions $A_{Gzip} = \{txt, compTxt, zip\}$. First, it receives (action *txt*) the text to be compressed from the controller. Then it does the compression (action *compTxt*), delivers the zip-file (action *zip*) to the controller and is ready for a next round. The *GZip* component has the signature $Sig(Gzip) = A_{Gzip}$ and the axioms $Ax(Gzip)$ are similar to the ones of the controller and not shown here. They specify a single model, shown in Fig. 4 (right). To construct an implementation $\langle Ctrl, GZip\rangle$ by decomposition (see Definition 4), we use the synchronous parallel composition operator "$\otimes$" defined above. According to [29], Exercise 6.1.15, any constructor gives rise to a specification building operation. This means that we can define the specification $Ctrl \otimes GZip$ whose model class consists of all possible parallel compositions of the models of the single specifications. Since *Ctrl* and *GZip* have, up to isomorphism, only one model there is also only one model of $Ctrl \otimes GZip$ which is shown in Fig. 5. Therefore, we know by construction that $Ctrl \otimes GZip \rightsquigarrow_{\kappa_\otimes} \langle Ctrl, GZip\rangle$ is a constructor implementation with decomposition. It remains to fill the gap between SP_3 and $Ctrl \otimes GZip$ which will be done with the action refinement constructor to be introduced in Definition 9. □

Two constructions which are frequently used and which are present in most process algebras are *relabelling* and *restriction*. They are particular cases of the reduct functor of the $\mathcal{D}^{\downarrow}$ institution.

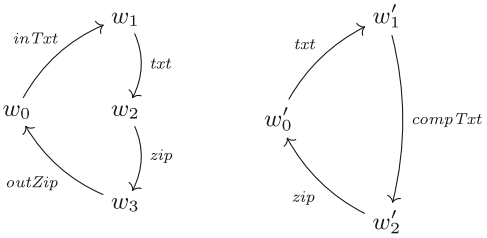

Fig. 4. Models of *Ctrl* and *GZip*

Definition 8 (Reduct, relabelling and restriction). *Let* $\sigma : A \to A'$ *be a signature morphism. The* reduct constructor $\kappa_\sigma : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A') \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$ *maps any model* $\mathcal{M}' \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A')$ *to its reduct* $\kappa_\sigma(\mathcal{M}') = \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\sigma)(\mathcal{M}')$. *Whenever* σ *is a bijective function,* κ_σ *is a* relabelling constructor. *If* σ *is injective,* κ_σ *is a* restriction constructor *removing actions and transitions.*

A important refinement concept for reactive systems is *action refinement* where an abstract action is implemented by a combination of several concrete

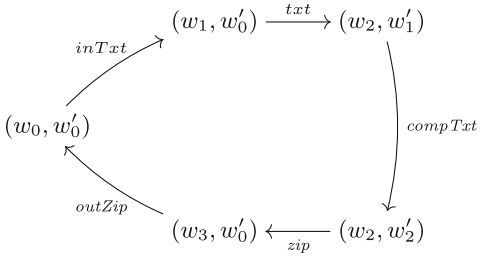

Fig. 5. Model of $Ctrl \otimes GZip$

ones (see [12]). It turns out that an action refinement constructor can be easily defined in $\mathcal{D}^{\downarrow}$-logic if we use the reduct functor for models over a signature consisting of structured actions built over atomic ones.

Definition 9 (Action refinement). *Let* $A, A' \in |\mathrm{Sign}^{\mathcal{D}^{\downarrow}}|$ *be signatures in* $\mathcal{D}^{\downarrow}$*, i.e. sets of actions. Let* D *be a finite subset of* $\mathrm{Act}(A')$ *considered as a signature in* $|\mathrm{Sign}^{\mathcal{D}^{\downarrow}}|$ *and let* $f : A \to D$ *be a signature morphism. The* action refinement constructor $|_f : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(D) \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)$ *maps any model* $\mathcal{M}' \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(D)$ *to its reduct* $\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(f)(\mathcal{M}')$.

Example 5. Let us now establish a refinement relation between SP_3 (Example 3) and $Ctrl \otimes GZip$ (Example 4). The signature of SP_3 consists of the actions $A_3 = \{inTxt, outZip\}$, the signature of $Ctrl \otimes GZip$ is the set $A_4 = \{inTxt, txt, compTxt, zip, outZip\}$. To obtain an action refinement we define the signature morphism $f : A_3 \to \mathrm{Act}(A_4)$ by $f(inTxt) = inTxt; txt; compTxt$ and $f(outZip) = zip; outZip$. Then we use the action refinement constructor $|_f : \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A_4) \to \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A_3)$ induced by f. Clearly, the application of $|_f$ to the model of $Ctrl \otimes GZip$ leads to the model of SP_3 explained above. Hence, $SP_3 \rightsquigarrow_{|_f} Ctrl \otimes GZip$ and together with Example 4 we have also $Ctrl \otimes GZip \rightsquigarrow_{\kappa_\otimes} \langle Ctrl, GZip \rangle$ which completes our refinement chain

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow SP_2 \rightsquigarrow_{\kappa_{ext}} SP_3 \rightsquigarrow_{|_f} Ctrl \otimes GZip \rightsquigarrow_{\kappa_\otimes} \langle Ctrl, GZip \rangle.$$

Finally, let us discuss how we could implement the last specification of the chain in a concrete process algebra. Translation from $\mathcal{D}^{\downarrow}$ to FSP yields

```
Ctrl = (inTxt -> txt -> zip -> outZip -> Ctrl).
Gzip = (txt -> compTxt -> zip -> Gzip).
```

The FSP semantics of the two processes are just the two models of the *Ctrl* and *Gzip* specifications respectively. They can be put together to form a concurrent system `(Ctrl || Gzip)` by using the synchronous parallel composition of FSP processes. Since the semantics of parallel composition in FSP coincides with our constructor $\kappa_\otimes$, we have justified that the FSP system `(Ctrl || Gzip)` is a correct implementation of the interface specification SP_3. □

4.2 Abstractor Implementations in $\mathcal{D}^{\downarrow}$-logic

Abstractor implementations in the field of algebraic specifications use typically observational equivalence relations between algebras based on the evaluation of terms with observable sorts. Interestingly, in the area of concurrent systems, abstractors have a very intuitive interpretation if we use bisimilarity notions. To motivate this, consider the specification $SP = (\{a\}, \{\downarrow x.\langle a\rangle x\})$. The axiom is satisfied by the first LTS in Fig. 6, but not by the second one. Clearly, however, both are bisimilar and so it should be irrelevant, for implementation purposes, to choose one or the other as an implementation of SP. We capture this with the principle of abstractor implementation using (strong) bisimilarity [24] as behavioural equivalence.

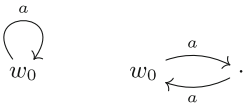

Fig. 6. Behavioural equivalent LTSs

Vertical composition of implementations refers to the situation where the implementation of a specification is further implemented in a next refinement step. For simple implementations it is trivial that two implementation steps compose. In the context of constructor and abstractor implementations the situation is more complex. A general condition to obtain vertical composition in this case was established in [28]. However, the original result was only given for unary implementation constructors. In order to adopt parallel composition as a constructor, we first generalise the institution independent result of [28] to the n-ary case involving decomposition:

Theorem 2 (Vertical composition). *Consider specifications* $SP, SP_1, \ldots, SP_n$ *over an arbitrary institution, a constructor* $\kappa : \mathrm{Mod}(Sig(SP_1)) \times \cdots \times \mathrm{Mod}(Sig(SP_n)) \to \mathrm{Mod}(Sig(SP))$ *and an equivalence* $\equiv \subseteq \mathrm{Mod}(Sig(SP)) \times \mathrm{Mod}(Sig(SP))$ *such that* $SP \rightsquigarrow^{\equiv}_{\kappa} \langle SP_1, \cdots, SP_n\rangle$. *For each* $i \in \{1, \ldots, n\}$, *let* $SP_i \rightsquigarrow^{\equiv_i}_{\kappa_i} \langle SP_i^1, \cdots, SP_i^{k_i}\rangle$ *with specifications* $SP_i^1, \ldots, SP_i^{k_i}$, *constructor* $\kappa_i : \mathrm{Mod}(Sig(SP_i^1)) \times \cdots \times \mathrm{Mod}(Sig(SP_i^{k_i})) \to \mathrm{Mod}(Sig(SP_i))$, *and equivalence* $\equiv_i \subseteq \mathrm{Mod}(Sig(SP_i)) \times \mathrm{Mod}(Sig(SP_i))$. *Suppose that* κ *preserves the abstractions* $\equiv_i$, *i.e. for each* $\mathcal{M}_i, \mathcal{N}_i \in Mod(Sig(SP_i))$ *such that* $\mathcal{M}_i \equiv_i \mathcal{N}_i$, $\kappa(\mathcal{M}_1, \ldots, \mathcal{M}_n) \equiv \kappa(\mathcal{N}_1, \ldots, \mathcal{N}_n)$. *Then,*

$$SP \rightsquigarrow^{\equiv}_{\kappa(\kappa_1, \cdots, \kappa_n)} \langle SP_1^1, \cdots, SP_1^{k_1}, \cdots, SP_n^1, \cdots, SP_n^{k_n}\rangle.$$

The remaining results establish the necessary compatibility properties between the constructors defined in $\mathcal{D}^{\downarrow}$ and behavioural equivalence $\equiv_A \subseteq |\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)| \times |\mathrm{Mod}^{\mathcal{D}^{\downarrow}}(A)|$, $A \in \mathrm{Sign}^{\mathcal{D}^{\downarrow}}$, defined as bisimilarity between LTSs.

Theorem 3. *The alphabet extension constructor* κ_{ext} *preserves behavioural equivalences, i.e. for any* $\mathcal{M}_1 \equiv_A \mathcal{M}_2$, $\kappa_{ext}(\mathcal{M}_1) \equiv_{A'} \kappa_{ext}(\mathcal{M}_2)$.

Theorem 4. *The parallel composition constructor* $\kappa_{\otimes}$ *preserves behavioural equivalences, i.e. for any* $\mathcal{M}_1 \equiv_{A_1} \mathcal{M}'_1$ *and* $\mathcal{M}_2 \equiv_{A_2} \mathcal{M}'_2$, $\mathcal{M}_1 \otimes \mathcal{M}_2 \equiv_{A_1 \cup A_2} \mathcal{M}'_1 \otimes \mathcal{M}'_2$.

Theorem 5. *Let* $f : A \to \mathrm{Act}(A')$ *be a signature morphism. The constructor* $|_f$ *preserves behavioural equivalences, i.e. for any* $\mathcal{M}_1, \mathcal{M}_2 \in \mathrm{Mod}^{\mathcal{D}^{\downarrow}}(\mathrm{Act}(A'))$, *if* $\mathcal{M}_1 \equiv_{\mathrm{Act}(A')} \mathcal{M}_2$, *then* $|_f(\mathcal{M}_1) \equiv_A |_f(\mathcal{M}_2)$.

5 Conclusions and Future Work

We have introduced the logic $\mathcal{D}^{\downarrow}$ suitable to specify abstract requirements for reactive systems as well as concrete designs expressing (recursive) process structures. Therefore $\mathcal{D}^{\downarrow}$ is appropriate to instantiate Sannella and Tarlecki's refinement framework to provide stepwise, correct-by-construction development of reactive systems. We have illustrated this with a simple example using specifications and implementation constructors over $\mathcal{D}^{\downarrow}$. We believe that a case was made for the suitability of both the logic and the method as a viable alternative to other, more standard approaches to the design of reactive software.

Related Work. Since the 80's, the formal development of reactive, concurrent systems has emerged as one of the most active research topics in Computer Science, with a plethora of approaches and formalisms. For a proper comparison with this work, the following paragraphs restrict to two classes of methods: the ones built on top of logics formalised as institutions, and the attempts to apply to the domain of reactive systems the methods and techniques inherited from the loose specification of abstract data types.

In the first class, references [7,9,25] introduce different institutions for temporal logics, as a natural setting for the specification of abstract properties of reactive processes. Process algebras themselves have also been framed as institutions. Reference [27] formalises CSP [17] in this way. What distinguishes our own approach, based on $\mathcal{D}^{\downarrow}$ is the possibility to combine and express in the same logic both abstract properties, as in temporal logics, and their realisation in concrete, recursive process terms, as typical in process algebras.

Our second motivation was to discuss how institution-independent methods, used in (data-oriented) software development, could be applied to the design of reactive systems. A related perspective is proposed in reference [23], which suggests the loose specification of processes on top of the CSP institution [27] mentioned above. The authors explore the reuse of institution independent structuring mechanisms introduced in the CASL framework [3] to develop reactive systems; in particular, process refinement is understood as inclusion of classes of models. Note that the CASL (in-the-large) specification structuring mechanisms can be also taken as specific constructors, as the ones given in this paper.

Future Work. A lot of work, however, remains to be done. First of all, logic $\mathcal{D}^{\downarrow}$ is worth to be studied in itself, namely proof calculi, and their soundness and completeness as well as decidability. In [2] it has been shown that nominal-free dynamic logic with binders is undecidable. Decidability of $\mathcal{D}^{\downarrow}$ is yet an open question: while [2] considers standard Kripke structures and global satisfaction, $\mathcal{D}^{\downarrow}$ considers reachable models and satisfaction w.r.t. initial states. On the other hand, in $\mathcal{D}^{\downarrow}$ modalities are indexed with regular expressions over sets of actions. It would also be worthwhile to discuss satisfaction up to some notion of observational equivalence, as done in [5] for algebraic specifications, thus leading to a behavioural version of $\mathcal{D}^{\downarrow}$.

The study of initial semantics (for some fragments) of $\mathcal{D}^{\downarrow}$ is also in our research agenda. For example, theories in the fragment of $\mathcal{D}^{\downarrow}$ that alternates binders with diamond modalities (thus binding all visited states) can be shown to have weak initial semantics, which becomes strong initial in a deterministic setting. The abstract study of initial semantics in hybrid(ised) logics reported in [8], together with the canonical model construction for propositional dynamic logic introduced in [19] can offer a nice starting point for this task. Moreover, for realistic systems, data must be included in our logic.

A second line of inquiry is more directly related to the development method. For example, defining an abstractor on top of some form of weak bisimilarity would allow for a proper treatment of *hiding*, an important operation in CSP [17] and some other process algebras through which a given set of actions is made non observable. Finally, our aim is to add a final step to the method proposed here in which any constructive specification can be translated to a process algebra expression, as currently done by our proof-of-concept translator D2CSP. A particularly elegant way to do it is to frame such a translation as an institution morphism into an institution representing a specific process algebra, for example the one proposed by M. Roggenbach [27] for CSP.

Acknowledgments. This work is financed by the ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência Tecnologia within project `POCI-01-0145-FEDER-016692` and `UID/MAT/04106/2013` at CIDMA. A. Madeira and L. S. Barbosa are further supported by FCT individual grants `SFRH/BPD/103004/2014` and `SFRH/BSAB/113890/2015`, respectively.

References

1. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge (2007)
2. Areces, C., Blackburn, P., Marx, M.: A road-map on complexity for hybrid logics. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 307–321. Springer, Heidelberg (1999)
3. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: the common algebraic specification language. Theor. Comput. Sci. **286**(2), 153–196 (2002)

4. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. Cambridge University Press, Cambridge (2010)
5. Bidoit, M., Hennicker, R.: Constructor-based observational logic. J. Log. Algebr. Program. **67**(1–2), 3–51 (2006)
6. Braüner, T.: Hybrid Logic and Its Proof-Theory. Applied Logic Series, vol. 37. Springer, Netherlands (2010)
7. Cengarle, M.V.: The temporal logic institution. Technical report 9805, LUM München, Institut für Informatik (1998)
8. Diaconescu, R.: Institutional semantics for many-valued logics. Fuzzy Sets Syst. **218**, 32–52 (2013)
9. Fiadeiro, J.L., Maibaum, T.S.E.: Temporal theories as modularisation units for concurrent system specification. Formal Asp. Comput. **4**(3), 239–272 (1992)
10. Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. J. ACM **39**(1), 95–146 (1992)
11. Goranko, V.: Temporal logic with reference pointers. In: Gabbay, D.M., Ohlbach, H.J. (eds.) ICTL 1994. LNCS, vol. 827, pp. 133–148. Springer, Heidelberg (1994). doi:10.1007/BFb0013985
12. Gorrieri, R., Rensink, A., Zamboni, M.A.: Action refinement. In: Handbook of Proacess Algebra, pp. 1047–1147. Elsevier (2000)
13. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
14. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
15. Havelund, K.: The Fork Calculus -Towards a Logic for Concurrent ML. Ph.D. thesis, DIKU, University of Copenhagen, Denmark (1994)
16. Hoare, C.A.R.: Proof of correctness of data representations. Acta Inf. **1**, 271–281 (1972)
17. Hoare, C.A.R.: Communicating Sequential Processes. Series in Computer Science. Prentice-Hall International, Upper Saddle River (1985)
18. Jones, C.B.: Software Development - A Rigorous Approach. Series in Computer Science. Prentice Hall, Upper Saddle River (1980)
19. Knijnenburg, P., van Leeuwen, J.: On models for propositional dynamic logic. Theor. Comput. Sci. **91**(2), 181–203 (1991)
20. Larsen, K.G., Thomsen, B.: A modal process logic. In: Third Annual Symposium on Logic in Computer Science, pp. 203–210. IEEE Computer Society (1988)
21. Madeira, A., Barbosa, L., Hennicker, R., Martins, M.: Dynamic logic with binders and its applications to the developmet of reactive systems (extended with proofs). Technical report (2016). http://alfa.di.uminho.pt/~madeira/main_files/extreport.pdf
22. Magee, J., Kramer, J.: Concurrency - State Models and Java Programs, 2nd edn. Wiley, Hoboken (2006)
23. O'Reilly, L., Mossakowski, T., Roggenbach, M.: Compositional modelling and reasoning in an institution for processes and data. In: Mossakowski, T., Kreowski, H.-J. (eds.) WADT 2010. LNCS, vol. 7137, pp. 251–269. Springer, Heidelberg (2012)
24. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981). doi:10.1007/BFb0017309
25. Reggio, G., Astesiano, E., Choppy, C.: Casl-ltl: a casl extension for dynamic reactive systems version 1.0. - summary. Technical report disi-tr-03-36. Technical report, DFKI Lab Bremen (2013)

26. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (1985)
27. Roggenbach, M.: CSP-CASL - a new integration of process algebra and algebraic specification. Theor. Comput. Sci. **354**(1), 42–71 (2006)
28. Sannella, D., Tarlecki, A.: Toward formal development of programs from algebraic specifications: implementations revisited. Acta Inform. **25**(3), 233–281 (1988)
29. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. Monographs on TCS, an EATCS Series. Springer, Heidelberg (2012)
30. Sekerinski, E., Sere, K.: Program Development by Refinement: Case Studies Using the B Method. Springer, Heidelberg (2012)
31. Winskel, G., Nielsen, M.: Models for concurrency. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. 4, pp. 1–148. Oxford University Press, Oxford (1995)