Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A logic for the stepwise development of reactive systems

Alexandre Madeira [a,*], Luís S. Barbosa [a,b], Rolf Hennicker [c], Manuel A. Martins [d]

[a] *QuantaLab and HASLab INESC TEC, Univ. Minho, Portugal*
[b] *United Nations University, UNU-EGOV*
[c] *Ludwig-Maximilians-Universität München, Germany*
[d] *CIDMA – Dep. Mathematics, Univ. Aveiro, Portugal*

ARTICLE INFO

Article history:
Received 13 April 2017
Received in revised form 14 November 2017
Accepted 1 March 2018
Available online xxxx

Keywords:
Specification
Reactive systems
Dynamic logic
Hybrid logic

ABSTRACT

$\mathcal{D}^{\downarrow}$ is a new dynamic logic combining regular modalities with the binder constructor typical of hybrid logic, which provides a smooth framework for the stepwise development of reactive systems. Actually, the logic is able to capture system properties at different levels of abstraction, from high-level safety and liveness requirements, to constructive specifications representing concrete processes. The paper discusses its semantics, given in terms of reachable transition systems with initial states, its expressive power and a proof system. The methodological framework is in debt to the landmark work of D. Sannella and A. Tarlecki, instantiating the generic concepts of constructor and abstractor implementations by standard operators on reactive components, e.g. relabelling and parallel composition, as constructors, and bisimulation for abstraction.

1. Introduction

Almost 30 years ago, D. Sannella and A. Tarlecki claimed, in what would become a most influential paper in (formal) Software Engineering [31], 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.

Indeed, the quest for suitable notions of *implementation* and *refinement* has been for more than four decades on the research agenda for rigorous Software Engineering. This goes back to Hoare's paper on data refinement [19], which influenced the whole family of model-oriented methods, starting with VDM [21]. A recent reference [33] collects a number of interesting refinement case studies in the B method, probably the most successful member of the family in what concerns industrial applications.

In such model-oriented approaches, a specification is said to refine another one if every model of the latter is a model of the former. Sannella and Tarlecki's work complemented and generalised this view with the notions of *"constructor"* and *"abstractor implementations"*:

> "***constructor*** *implementations which involve a construction 'on top of' the* ***implementing*** *specification, and* ***abstractor*** *implementations which additionally provide for abstraction from some details of the* ***implemented*** *specification*" [31].

* Corresponding author.
E-mail address: madeira@ua.pt (A. Madeira).

https://doi.org/10.1016/j.tcs.2018.03.004
0304-3975/

The idea behind a constructor implementation is that for representing 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*. On the other hand, abstractor implementations capture the fact that sometimes the requirements for a system are only satisfied 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 [32]. 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.

The present article 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 [32] 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 PhD thesis [17]. His work, however, focused essentially on functional requirements expressed by algebraic specifications and implemented in a functional programming language.

As a matter of fact, the development of reactive systems, which are 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 [34], a Petri net [29] or a process algebra expression [20,4]) 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.

Actually, our research hypothesis is that also in the domain of reactive systems, loose specification has an important role to play, because it supports the gradual incorporation of further requirements and implementation decisions such that verification of the correctness of a complex system can be done piecewise in smaller steps. Additionally, this allows for the systematic documentation of design decisions, as a support to systems' maintenance and refactoring.

Therefore, the challenge undertaken here is twofold. First, we propose a new logic to support the development of reactive systems at different levels of abstraction. Then, we show how to adapt to this context 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"* [31].

To address these challenges, we introduce a new logic, $\mathcal{D}^{\downarrow}$, which is able not only to express abstract properties, such as liveness requirements or deadlock avoidance, but also to describe the concrete, recursive process structures which implement them. The logic combines modalities indexed by regular expressions of actions, as in dynamic logic [16], and state variables and binders, characteristic of hybrid logic [7].

As a second contribution, the paper introduces a number of constructors and abstractors relevant to the development of reactive systems. Interestingly, it turns out that requirements of Sannella and Tarlecki's methodology for vertical composition of abstractor/constructor implementations boils down to the congruence property of bisimilarity w.r.t. constructions on labelled transition systems, like parallel composition and relabelling.

This article is an extended version of our previous work [24], presented at ICTAC'2016. As such it includes the complete proofs of all results, and two new sections: Section 5 discusses the expressive power of $\mathcal{D}^{\downarrow}$, while Section 6 introduces a sound proof calculus for it.

Apart from those new sections, section 2 introduces $\mathcal{D}^{\downarrow}$, and sections 3 and 4, respectively, characterise the development method, with a brief revision of the relevant background, and its tuning to the design of reactive systems. Finally, section 7 concludes and points out some issues for future work.

2. A dynamic logic with binders

2.1. $\mathcal{D}^{\downarrow}$: syntax and semantics

$\mathcal{D}^{\downarrow}$ logic is designed to express properties of reactive systems, from abstract safety and liveness requirements, down to concrete design decisions specifying the (recursive) structure of processes. It thus combines modalities with regular expressions, as originally introduced in dynamic logic [16], and binders in state variables. This logic retains from hybrid logic [7], only state variables and the binder operator first studied by V. Goranko in [13]. These motivations are reflected in its semantics. Differently from what is usual in modal logics, whose semantics is given by Kripke structures and satisfaction evaluated globally in each model, $\mathcal{D}^{\downarrow}$ models are reachable transition systems with initial states at which satisfaction is evaluated.

Definition 1 *(Model).* For a finite set of atomic actions A, *models* are reachable A-labelled transition systems, 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 $w^n = w$.

Please cite this article in press as: A. Madeira et al., A logic for the stepwise development of reactive systems, Theoret. Comput. Sci. (2018), https://doi.org/10.1016/j.tcs.2018.03.004