idDL2DL – Interval syntax to $d\mathcal{L}^\star$

Jaime Santos[1,2], Daniel Figueiredo[2,3], and Alexandre Madeira[2]

[1] Universidade do Minho, Braga, Portugal
[2] CIDMA - Dept. of Mathematics, University of Aveiro, Portugal
[3] Association for Biomedical Research and Innovation in Light and Image, Portugal

Abstract. A wide range of methods from computer science are being applied to many modern engineering domains, such as synthetic biology. Most behaviors described in synthetic biology have a hybrid nature, in the sense that both discrete or continuous dynamics are observed. Differential Dynamic Logic ($d\mathcal{L}$) is a well-known formalism used for the rigorous treatment of these systems by considering formalisms comprising both differential equations and discrete assignments. Since the many systems often consider a range of values rather than exact values, due to errors and perturbations of observed quantities, recent work within the team proposed an interval version of $d\mathcal{L}$, where variables are interpreted as intervals. This paper presents the first steps in the development of computational support for this formalism by introducing a tool designed to models based on intervals, prepared to translate them into specifications ready to be processed by the `KeYmaera X` tool.

Keywords: Synthetic biology · Formal verification · Dynamic Logic.

Introduction and preliminaries

Hybrid systems – those composed of continuous and discrete components – are everywhere, from the medical devices to the aerospace artifacts we have. Due to the critical role that some of them play in our life, the scientific community was pushed to develop theories and tools to support the trustworthy conception of these systems, not only via simulation techniques (e.g. with `Simulink`) but also using program verification techniques and logic.

Differential Dynamic Logic [1], with its supporting tool `KeYmaera X` (`KX`) [2], represents a core formalism in this context. This approach brings principles and techniques from program verification to hybrid systems developers, namely from dynamic logic [3]. The formalism has been successfully applied to several computational hybrid systems scenarios [1]; but also in other less obvious domains, including the specification and analysis of models in biology [4].

Differential dynamic logic is a quantified dynamic logic with two kinds of atomic programs: assignments (discrete state transitions); and continuous evolutions. The syntax combines both kinds of atomic programs to express and prove the correctness of assertions of the so-called cyber-physical or hybrid systems.

⋆ This work is supported by FCT, the Portuguese funding agency for Science and Technology with the projects PTDC/CCI-COM/4280/2021, UIDB/50014/2020 and UIDB/04106/2020.

Because of this, a $d\mathcal{L}$ program is called a *hybrid program*. With the sound proof calculus for $d\mathcal{L}$, and the KX tool – a semi-automatic prover – one can prove the correctness of such systems. When the user provides a formula of $d\mathcal{L}$ as input, KX either generates its proof or retrieves some simpler formulas - preconditions that are required to be valid - to prove the formula.

We follow with a brief description of $d\mathcal{L}$ syntax.

Definition 1. *Let Σ be a signature containing n-ary functions, propositions, and state variables; and X be a set of logical variables.*

- *The set $Trm(X,\Sigma)$ of terms is the least set containing X such that $f(t_1,...,t_n) \in Trm(X,\Sigma)$ iff $f \in \Sigma$ is a n-ary function and $t_1,...,t_n \in Trm(X,\Sigma)$;*
- *$p(t_1,...,t_n)$ is a predicate if $p \in \Sigma$ is a n-ary proposition and $t_1,...,t_n \in Trm(\Sigma,X)$;*
- *$Fml_{FOL}(X,\Sigma)$ is the least set containing every predicate, and such that $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $\forall\varphi$, $\exists\varphi \in Fml_{FOL}(X,\Sigma)$ whenever $\varphi,\psi \in Fml_{FOL}(X,\Sigma)$.*

Note that constants are 0-ary functions and other Boolean operators can be introduced as usual abbreviations. Also, Σ_{fl} denotes the set of state variables.

Definition 2. *The set of hybrid programs $HP(X,\Sigma)$ is defined as follows:*

- *$(x_1 := t_1,...,x_n := t_n)$, $(x_1' = t_1,...,x_n' = t_n \,\&\, \psi)$, $?\psi \in HP(X,\Sigma)$ for every state variables $x_1,...,x_n \in \Sigma$, $t_1,...,t_n \in Trm(X,\Sigma)$ and $\psi \in Fml_{FOL}(X,\Sigma)$;*
- *$\alpha;\beta$, $\alpha \cup \beta$, $\alpha^* \in HP(X,\Sigma)$ whenever $\alpha,\beta \in HP(X,\Sigma)$.*

Definition 3. *The set $Fml(X,\Sigma)$ of formulas of $d\mathcal{L}$ is defined recursively as the least set containing $Fml_{FOL}(X,\Sigma)$ and such that:*

- *$[\alpha]\varphi$, $\langle\alpha\rangle\varphi \in Fml(X,\Sigma)$ whenever $\varphi \in Fml(X,\Sigma)$ and $\alpha \in HP(X,\Sigma)$*
- *$\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $\forall\varphi$, $\exists\varphi \in Fml(X,\Sigma)$ whenever $\varphi,\psi \in Fml(X,\Sigma)$.*

The truth of a formula in the scope of a modality embedding a hybrid program is evaluated in a scenario obtained after the hybrid program is run. The semantics of $d\mathcal{L}$ are defined over the reals, and a strict interpretation of formulas and propositions from Σ is imposed. Thus, Σ only contain symbols such as $+$ or $\leq$ that are interpreted as "sum" or "less or equal", respectively. The full semantics can be found in [1].

In [5], an interval syntax is developed for $d\mathcal{L}$, regarding its application in contexts where variables are presented in terms of intervals, namely due to errors or uncertainty. This can be useful, for instance, to model a physical system under some measure uncertainties or whenever we want to make an assignment of an irrational number without machine representation. Apart from replacing real numbers for closed intervals, the interval syntax is the same as in $d\mathcal{L}$, and the semantics presented are adapted to an interval context, namely following the work of Moore [6]. In this context real numbers are considered as degenerated intervals. For instance the value $a \in \mathbb{R}$ is represented by the interval $[a,a]$. Under this perspective, the semantics for this adapted syntax considers a "strict" interpretation over closed intervals, *i.e.* a symbol like $+ \in \Sigma$ is interpreted as "interval sum" , for instance (check [6] for additional information in interval

arithmetics). Also, logical and state variables are evaluated over $\mathcal{I}(\mathbb{R})$. Thus, the interpretation of each predicate P (defined for reals) must be adapted to intervals. Particularly, denoting by $P^{\mathcal{I}}(\mathbb{R})$ the interval interpretation of P, a predicate $P^{\mathcal{I}}(\mathbb{R})(X_1, ...X_n)$ is said to be *true* if $P(x_1, ..., x_n)$ holds for every $(x_1, ..., x_n) \in X_1 \times ... \times X_n$. With this definition, we expand the valuation for the full set of formulas, by keeping the coherence, so that the semantics of $d\mathcal{L}$ can be seen as a particular case of the interval one since the interpretation of its formulas is done over real numbers (the set of degenerated intervals).

This paper presents a parser and a translator which accepts interval $d\mathcal{L}$ formulas and retrieves equivalent ones in standard $d\mathcal{L}$. In this way, we take advantage of the sound $d\mathcal{L}$ proof calculus. In particular, we can use `KX` and try to obtain proof for the original interval $d\mathcal{L}$ formula. We illustrate the framework by modeling a biological regulatory network [7] – where there are variables like concentration of a protein are rather expressed in intervals than with a determined value. For this example, we consider a formula in the interval syntax of $d\mathcal{L}$, describing a property of the system, and use our parser and translator to obtain its equivalent formula in $d\mathcal{L}$ standard syntax. We then use the automatic tactic of `KX` to prove the correctness of this example.

The `idDL2dDL` tool

This section introduces a tool to parse and translate specifications from interval $d\mathcal{L}$ to specifications in standard $d\mathcal{L}$, following the theoretical work in [5]. The implementation, developed in `Python`, is structured in five main parts – the *lexer*, the *parser*, the *translator*, a *graphical user interface* (GUI), and the *interpreter*. Detailed user instructions are available in the GitHub repository[4].

We first perform lexical analysis to convert the input text into tokens, contained in a list of predefined symbols, whose order is maintained through a `Position` attribute. Then, we use a parser to analyze the syntax of these tokens and generate an abstract syntax tree (AST) as output. The priority of operations in the AST is determined by the depth of the nodes, which are constructed with unary, and binary operations according to Definition 2 of hybrid programs [1]. This was one of the core challenges with the parser, since it required some language design and testing to ensure the legality of the nodes.

The translator converts interval dynamic logic expressions into regular $d\mathcal{L}$ formulas using `visit` methods for each type of node. These methods evaluate tokens and preserve the priority degree of the AST. For instance, when an interval token is detected, a `TranslatedInterval` object creates an inequation between a fresh variable and the interval bounds. To comply with `KX` syntax, many other expressions are converted according to its specifications. Ensuring each interval generates a unique variable and that the final expression maps the variable to the correct interval came at a surprising performance cost, indicating that our method for generating infinite strings requires improvement.

The user interface was created using `Python`'s `Tkinter` library and has two pages: a translation page and a translation history page. On the translation

[4] github.com/JaimePSantos/idDL2DL

page, shown in Fig. 2, users can load a file containing multiple formulas and save the resulting translations as *.kyx* files. If only one formula is translated, the user can specify a file name and location. For multiple translations, users can choose to save them in a single *.kyx* file or multiple files. The history page saves all translated statements, which is useful for analyzing complex models step by step. Displaying errors in a user-friendly way is still a work in progress for this component.

Interpreter and performance. The latest feature of the software is an initial version of an interpreter. This component, similarly to the translator, converts an AST into a string. The implementation logic is to visit each node and apply one of the interval arithmetic rules from [6,8], depending on the type of operation associated with the node

$$[a,b] + [c,d] = [a + c, b + d] \tag{1}$$

$$[a,b] - [c,d] = [a - d, b - c] \tag{2}$$

$$[a,b] \times [c,d] = [min(P), max(P)] \text{ where } P = \{a \times c, a \times d, b \times c, b \times d\} \tag{3}$$

$$[a,b] \div [c,d] = [min(P), max(P)] \text{ where } P = \{\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\}, 0 \notin [c,d]. \tag{4}$$

As we visit and interpret the nodes of the AST, the resulting output must be used to rebuild the structure of the AST. Thus, the current version of the program recreates and re-parses the tokens before feeding them to the translator, which results in an inherent performance cost. Fig. 1 shows the performance comparison between a simple translation and an interpreted translation of a formula that involves the sum of consecutive divisions between two intervals. Each formula was sampled 100 times to calculate the mean of each execution. Despite the interpreter adding a small performance cost, it was expected to be significantly larger due to the second round of lexing and parsing. This suggests that the true bottleneck is indeed in the translation process, most likely due to an inefficient method of generating unique variables.

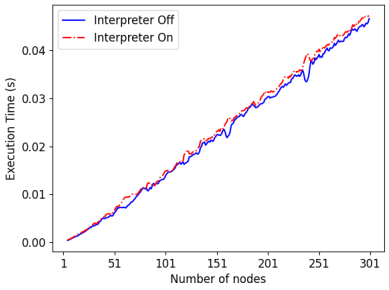

Fig. 1. Execution time with and without the interpreter vs the number of nodes.

This component posed several challenges during implementation, particularly in ensuring that the visit methods properly apply the defined operations while ignoring and preserving nodes containing operations between intervals and non-intervals. However, a limitation of the current version is that operations between intervals and expressions inside parentheses are kept separate, which will be addressed in the future. Another limitation of the software is the recursion limits imposed by `Python`, which can be extended but is not recommended. In the future, this limitation can be resolved by replacing recursion with a stack.

An illustration. We illustrate the application of the `idDL2dDL` tool with the analysis of a *PieceWise linear (PWL) model* of a biological regulatory network

[9]. Biological regulatory networks are complex systems describing biological phenomena such as cell metabolism. This kind of model describes the physical and chemical interaction between cell proteins, mRNA, and other cell organelles. The more detailed deterministic formalism used to model these systems are nonlinear differential equations. Numerical methods, such as simulations, are then applied to study the complex behavior and interactions between the components of a cell. These systems of differential equations often are subjected to a preliminary study to fully understand the major dynamics of a biological process. They are firstly simplified by proper methods, resulting in models such as PWL models that preserve the major dynamics of the original one and are easier to study. A PWL model is composed of several domains containing a system of linear differential equations which are obtained by proper simplifications of the (nonlinear) original one (cf. [9] for details). In [7] we can found an example of a PWL model, illustrated in Table 1.

This system is characterized by having continuous dynamics within each domain but discrete reconfigurations when we move from one domain to another. Continuous variables describe the concentration of intracellular components, such as proteins and RNA. This hybrid dynamics can be expressed by a hybrid program of $d\mathcal{L}$.

$\begin{cases} x' = -x \\ y' = -y \end{cases}$ $x \leq 2$ $2 \leq y$	$\begin{cases} x' = -x \\ y' = -y \end{cases}$ $2 \leq 4$ $2 \leq y$	$\begin{cases} x' = -x \\ y' = 3 - y \end{cases}$ $4 \leq x$ $2 \leq y$
$\begin{cases} x' = -x \\ y' = -y \end{cases}$ $x \leq 2$ $y \leq 2$	$\begin{cases} x' = 5 - x \\ y' = -y \end{cases}$ $2 \leq x \leq 4$ $0 \leq y \leq 2$	$\begin{cases} x' = 5 - x \\ y' = 3 - y \end{cases}$ $4 \leq x$ $0 \leq y \leq 2$

Table 1. PWL model.

Making the same for the other five domains (and aggregating bio_{00} and bio_{01} in bio_0), we have:

$bio_0 \equiv ?(x \leq 2); (x' = -x, y' = -y \& (x \leq 2))$

$bio_{10} \equiv ?(2 \leq x \wedge x \leq 4 \wedge 0 \leq y \wedge y \leq 2); (x' = 5 - x, y' = -y \& (2 \leq x \wedge x \leq 4 \wedge 0 \leq y \wedge y \leq 2))$

$bio_{20} \equiv ?(4 \leq x \wedge 0 \leq y \wedge y \leq 2); (x' = 5 - x, y' = 3 - y \& (4 \leq x \wedge 0 \leq y \wedge y \leq 2))$

$bio_{11} \equiv ?(2 \leq x \wedge x \leq 4 \wedge 2 \leq y); (x' = -x, y' = -y \& (2 \leq x \wedge x \leq 4 \wedge 2 \leq y))$

$bio_{21} \equiv ?(4 \leq x \wedge 2 \leq y); (x' = -x, y' = 3 - y \& (4 \leq x \wedge 2 \leq y))$

The hybrid program describing the full dynamics of the biological system is:

$$bio \equiv (bio_0 \bigcup bio_{10} \bigcup bio_{20} \bigcup bio_{11} \bigcup bio_{21})^*$$

Then we can take advantage of the interval syntax to express biological properties like "when the concentrations x and y are around 5.5 and 3.5, the biological system will never reach a state where $x < 2$".

$$\big[x := [5,6]\,;\, y := [3,4]\big]\,\big[bio\big]\; x > 2$$

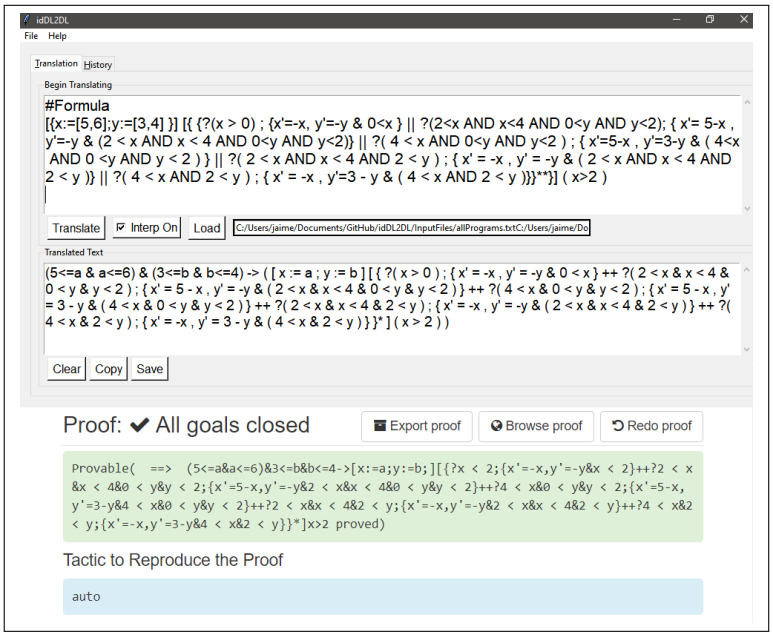

Fig. 2. The translation of the biological regulatory network example in `idDL2dDL` and the respective proof in `KX`.

This example was then translated and proven in `KX` version `4.9.5`, using the default automatic proof tactic according to Fig. 2. Note that, although the example presented is representative of biological scenarios treatable with `KX` in real biological systems, human intervention may be required while using the semi-automatic prover, to aid the closing of the proof goals.

Discussion and conclusion

This paper introduces `idDL2DL`, a parser and translator from interval dynamic logic formulas to $d\mathcal{L}$. An example of how to use the tool is presented with a case study for the synthetic biology field. As aforementioned, the formalism introduced consists of a syntax directed to interval contexts along with adapted semantics, to inherit the soundness from $d\mathcal{L}$ (see [1,5]). We note that interval arithmetic has already been considered in $d\mathcal{L}$, through a different approach, in [10]. In that work, a third truth-value U is considered for uncertain statements like $[0, 2] < [1, 3]$. These kinds of propositions are evaluated as $false$ in the present work, to carry a conservative approach. Consequently, our semantical interpretation of continuous evolutions was adapted, not being so restrictive to catch every punctual n-dimension initial state. This allows `KX` to consider every possible continuous evolution as in $d\mathcal{L}$ and, in this way, preserve the soundness (see [5] for details). This tool still has room for multiple improvements, mainly when it comes to extending its pre-processing capabilities. With this user-friendly interface, we aim to develop a user-friendly tool for those without experience in formal verification, as is the case of synthetic biology.

References

1. André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.
2. Andreas Müller, Stefan Mitsch, Wieland Schwinger, and André Platzer. A component-based hybrid systems verification and implementation tool in keymaera x (tool demonstration). In Roger D. Chamberlain, Walid Taha, and Martin Törngren, editors, *Cyber Physical Systems. Model-Based Design, CyPhy 2018*, volume 11615 of *Lecture Notes in Computer Science*, pages 91–110. Springer, 2018.
3. David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.
4. Daniel Figueiredo, Manuel A. Martins, and Madalena Chaves. Applying differential dynamic logic to reconfigurable biological networks. *Mathematical Biosciences*, 291:10 – 20, 2017.
5. Daniel Figueiredo. Introducing interval differential dynamic logic. In Hossein Hojjat and Mieke Massink, editors, *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods*, volume 12818 of *Lecture Notes in Computer Science*, pages 69–75. Springer, 2021.
6. Ramon E. Moore. *Interval Arithmetic and Automatic Error Analysis in Digital Computing*. PhD thesis, Stanford University, 1962.
7. Daniel Figueiredo and Luís Soares Barbosa. Reactive models for biological regulatory networks. In Madalena Chaves and Manuel A. Martins, editors, *Molecular Logic and Computational Synthetic Biology, MLCSB 2018*, volume 11415 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2018.
8. Bartlomiej Jacek Kubica. *Interval Methods for Solving Nonlinear Constraint Satisfaction, Optimization and Similar Problems - From Inequalities Systems to Game Solutions*, volume 805 of *Studies in Computational Intelligence*. Springer, 2019.
9. Hidde De Jong. Modeling and simulation of genetic regulatory systems: a literature review. *Journal of computational biology*, 9(1):67–103, 2002.
10. Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. Veriphy: Verified controller executables from verified cyber-physical system models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 617–630. Association for Computing Machinery, 2018.
11. Ricardo G. Sanfelice, David A. Copp, and Pablo Nanez. A toolbox for simulation of hybrid systems in matlab/simulink: hybrid equations (hyeq) toolbox. In Calin Belta and Franjo Ivancic, editors, *Proceedings of the 16th international conference on Hybrid systems:computation and control, HSCC 2013*, pages 101–106. ACM, 2013.
12. Regivan H. N. Santiago, Benjamín R. C. Bedregal, Alexandre Madeira, and Manuel A. Martins. On interval dynamic logic: Introducing quasi-action lattices. *Sci. Comput. Program.*, 175:1–16, 2019.
13. Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. Veriphy: Verified controller executables from verified cyber-physical system models. *SIGPLAN Not.*, 53(4):617–630, jun 2018.