The Ceptre Editor: A Structure Editor for Rule-Based System Simulation

1st Alexander Card
Principles of Expressive Machines Lab
Department of Computer Science
North Carolina State University
Raleigh, USA
acard@ncsu.edu

2nd Chris Martens
Principles of Expressive Machines Lab
Department of Computer Science
North Carolina State University
Raleigh, USA
crmarten@ncsu.edu

***Abstract*—Systems understanding is a skill required to solve many of the world's most important problems, from climate change to immunotherapy to social decision-making. However, these problems also require communication among experts with diverse skill sets and academic backgrounds. Our long-term goal is to facilitate systems understanding across a range of disciplines through end-user computational modeling tools. This paper presents the Ceptre Editor, a structure editor for the rule-based programming language Ceptre. The Ceptre Editor runs in the browser and offers a visual interface and integrated development environment for Ceptre, following design recommendations from end-user programming, with the goal of providing discoverable affordances for program construction and maintaining syntactic well-formedness at each edit state. We performed a preliminary evaluation of the tool through a qualitative study, assessing the editors effectiveness at helping users understand and extended a system model, and found promising results regarding learnability and mental model accuracy.**

***Index Terms*—rule-based programming, structure editors, end user programming**

I. Introduction

Systems thinking, defined by the ability to achieve deep causal understanding of the relationship between parts in a complex system, is a critical skill for problem-solving in many disciplines, including physical sciences, economics, medicine, social and decision sciences, and urban planning. One activity hypothesized to support the development of systems thinking skills is computational modeling, or building virtual simulations of systems that can be executed, analyzed, and queried. A number of existing computational modeling tools support tasks in specific domains, such as Kappa [1] for molecular signaling networks and Kodu [2] for digital arcade games, but these systems provide the user with a fixed set of language primitives that cannot be combined or easily extended. General-purpose modeling tools, such as GoldSim, Simulink and MATLAB, rely on some combination of general-purpose imperative programming skills and mathematical reasoning techniques. These prerequisites can hinder audiences such as: (1) members of interdisciplinary teams with diverse training; (2) novice scientists or practitioners within a discipline trying to get up to speed with current theories; (3) policymakers or members of the general public who want to understand the impact of systemic changes.

Rule-based programming has emerged as a promising computational modeling paradigm for a number of domains that require systems thinking, such as molecular interactions [3]–[5], security protocols [6], and multi-agent systems [7]. Therefore, we are evaluating *rule-based programming* as an appropriate programming model for systems understanding. In rule-based programming, programmers specify sets of possible program states as logical assertions, then define rewrite rules that update the state based on a logical description of the rule's conditions and effects. We use the Ceptre programming language [8], based on forward-chaining linear logic programming corresponding to multiset rewriting, as a starting point for this work.

In this paper, we describe our efforts to bring end-user programming tool design principles to the Ceptre programming language through a new editing environment. We created a GUI programming tool that allows users to develop Ceptre programs in a scaffolded way that maintains the well-formedness of the syntax tree, based on the principles of *structure editing* (editing the structure of the program, rather than the text). This editor is similar to the block-based programming environments that have been successful at introducing novices to procedural programming in that it (a) provides discoverable affordances that scaffold the user's program creation process; (b) maintains the well-formed structure of partial programs by enforcing type constraints through the user interface.

Our contributions are as follows: (1) A structure editor for a multiset rewriting-based programming language, designed for learnability and usability (2) A human-centered evaluation of such a language, resulting in evidence for use in rapidly constructing and modifying new system models for end users.

II. Related Works

Many modern introductory programming tools implement designs which guide the end user in manipulating the program structure, instead of directly editing code in a text file. One common instrument for maintaining program structure is known as block programming. Block programming breaks the language semantics into separate blocks, helping guide the user during the

978-1-7281-0810-0/19/$31.00 2019 IEEE

creation of the program. As block shape determines placement, the user cannot place a block in a syntactically incorrect place.

Block based languages such as Scratch [9], or Alice2 [10], are used to introduce newcomers to programming, as well as mitigate issues which can arise from language syntax. Scripting languages or specialized languages are frequently seen in the introductory tools. This allows for a more friendly environment for novices which has a positive effect on learning [11].

Research on introductory programming languages such as Scratch, Greenfoot [12], or Alice2 has shown that maintaining syntactic well-formedness has a positive effect on the user's ability to learn to program in an imperative language, and that modifying program structure instead of text-based code provides benefits for programming novices. In this work we take a similar design and implement it as an editor for a rule-based programming language.

Various introductory programming tools approach the problem of well-formedness differently. Scratch for example, assumes default values for any program element which has not been initialized, while other languages such as micro:bit [13] select values from the drop-down immediately when created. These options do not separate code being edited from code which the user is done editing, and may execute code currently being edited. We introduce an locking system which we hypothesize will help reduce frustrations caused by syntactic and typographical errors, while not executing unfinished code.

AgentSheets [14] and Kodu [15] are end-user programming tools which use rule-based languages, where the rules are condition/action pairs. These condition/action pairs are attached to an agent or object in the system and when the condition is met, will execute the associated action. Our approach differs from AgentSheets and Kodu by using multiset rewriting: rules are not attached to any construct in the system, instead rewriting the multiset to create successive states.

System simulation tools, such as COMSOL [16], Simulink [17], and GoldSim [18], utilize a drag and drop interface which allows the end user to create a system by dragging components into the editor and connecting the components in a domain-specific way. This allows the end user to define certain structures without needing to modify the underlying language. However, since the ways to connect components are defined in domain-specific ways, introducing new component types require a modification to the underlying language [19].

Kappa [1], a rule-based system designed for molecular systems biology modeling, provides the end-user-programmer with an integrated development environment for creating molecular systems in the textual language. We differ in this work by incorporating design decisions from introductory programming tools to facilitate novice end-user-programming, and do not specifically target any specific modeling domain.

In addition, much of the existing research on these tools are: extensions of the language [20], simulations created using the tool [21], [22], or uses a differing visual language for design and modeling [23], [24]. In this work we strive to maintain a friendly, generalized environment for the end-user-programmer.

III. Background

Before introducing the editor and discussing the evaluation, we introduce the Ceptre language through an example in an ecology-inspired domain. In this example, system states consist of predator (fox) and prey (rabbit) populations, and rules manipulate these populations by modeling predation (foxes eat rabbits) and reproduction (both rabbits and foxes multiply).

Ceptre is a rule-based specification language [8] which uses logic to represent the rules of a system. A Ceptre program represents system states (configurations) as multisets of logical predicates and defines rules that can manipulate those multisets, replacing certain facts by others. The structure of a Ceptre program consists of type and predicate definitions, an unordered set of rules, and a description of the initial state.

In Ceptre, simulation states are represented by multisets of ground predicates. These multisets contain all the information that is true in the current simulation state. Using these multisets, the simulation is progressed by the use of rules, which change the state by taking preconditions in the current simulation state, and replacing the preconditions with new ground predicates that follow the fixed rule structure. This allows the states to change constrained by rules set forth by the author.

Describing the rules requires the definition of two symbols: * called tensor, which conjoins predicates, and ⊸ called lolli, which is the transition operator. In the predator/prey model, there could be a rule `rabbit_grow_mature :rabbit young ⊸ rabbit mature` which requires a young rabbit as a condition, which is replaced by a mature rabbit.

Ceptre's types describe the domains over which program terms can range. Terms represent the nouns in the language, or the objects we want to refer to in predicates. In the ecological predator/prey model, we will use two types, `age` and `hunger`, representing age and hunger for the animals in the simulated system. We represent three ages (`young`, `mature`, and `old`) and two hunger levels (`hungry` and `sated`) to represent the states of individuals in the population that the rules modify.

Predicates can represent additional information about types, or information which is true in the world. For example, in an ecological predator/prey model, there could be a predicate `rabbit age`, denoting a rabbit's age.

IV. Ceptre Editor

As we intend the Ceptre editor to be approachable by programming novices and researchers communicating across domains, the editor should primarily focus on providing the user with an integrated development environment which minimizes the frustrations caused by syntactic and typographical errors. The editor is hosted at microceptre.glitch.me, and the source is at https://github.ncsu.edu/acard/microceptre. Images of the editor are included in the supplementary materials.

A. Design Philosophy

In this work we hypothesize that a structure editor built using similar features as the previous editors would afford an integrated development environment which would minimize frustration caused by syntactic and typographical errors for a rule-based programming language. For this purpose, the Ceptre Editor uses a modified subset of Ceptre for the language, representing a world through sets, predicates, rules, and atoms. Sets are analogous to Ceptre's types, functioning as groupings of objects and things in the world.

B. *Editing the Model*

In an attempt to minimize syntactic and typographical errors, the editor only allows users to type information when naming a set or predicate, and adding a set element or rule variable. In each case, once the user has named the program component, the component appears in menus of options which contain program components of that type. Any time an edit is made to a program component, all associated menus of options are updated, but their current selection is held if possible.

In the Ceptre Editor, both sets and their elements must be fully typed by the user. Predicates only permit the user to type the name. As any argument to a predicate is a set, these are presented to the user via a menu of options which allow the user to select the set from those that currently exist.

Rules consist of two groups of predicates: the first being a list of conditions for the rule to fire, and the second the effects of the rule on the successive state. When adding a predicate to either group, the predicate is selected through a menu of options consisting of predicates which are syntactically well-formed. Selecting a predicate automatically updates the user interface and provides the user with the appropriate number of arguments. These arguments are also menus of options, consisting of the appropriate set elements, variables which have been created, or if the predicate is a condition, creation of a new variable.

The initial state allows creation of ground predicates which are instantiated predicates which are constructed entirely through menus of options. First, the predicate is selected from a list of syntactically well-formed predicates, updating the user interface with the appropriate number of arguments. The menu of options for arguments are elements taken from the appropriate set as described by the predicate.

C. *Executing the Model*

When execution is started, the program runs with the current initial state and rules, displaying the initial program state as a multiset of ground predicates, and the rules which can be used to transition to the next state. The transitions present in the execution are presented as the rule name, followed by a list of the instantiated variables in the rule. Selecting one of the available options allows the user to execute the selected transition and generate the next program state. Additionally, the user may execute the program to quiescence (a program state where no more rules can fire, if one exists) or enter a number of steps which the program can automatically execute by selecting the successive transitions at random.

D. *Locking and Unlocking Components*

The locking mechanism was implemented to avoid partial programs which are not well-formed. This is done by having two modes for every element in the program: unlocked and locked. An unlocked element may be modified in any valid way by the user at any point, but is not available in a menu of options anywhere in the program and thus cannot be referenced.

If the lock function is toggled, the program element is checked to ensure well-formedness. If the program element is well formed, it will appear in menus of options for that program element. If the program element is not well-formed, the user will receive immediate feedback regarding the issue and the program element remains in unlocked mode.

The choice for using the locking mechanism primarily involved the requirement of feedback and conscious user action. In order to lock a program component, the user must take a conscious action. When the user takes this action, the editor provides immediate feedback in case of failure. This feedback makes it considerably more difficult to miss components which are not well-formed.

V. Formative Study

To evaluate the editor we constructed a pilot study to answer the following questions: (1) Are users able to rapidly construct and understand models using the editor? (2) Is a formal logic background required for the success of a user?

We observed eight participants recruited from our university campus. This study, alongside previous literature on structure editors, has led to design recommendations for improving the user experience while constructing and modeling systems using a structure editor.

A. *Study Setup*

We gave the participants a brief tutorial on creating a model limited to 30 minutes. Then, the participants were asked to modify the model they constructed during the tutorial during a think aloud protocol while talking through their decisions in a 30 minute session. The session was concluded with an interview.

The tutorial for the study provided a walk-through to set up a blocks world model. The blocks world model consisted of three blocks on a table, and a robotic arm which could pick up and set down blocks. The tutorial walked the participant through adding the sets, predicates, rules, and starting world state, ending with stepping through an execution of the program.

Extending the blocks world model followed the tutorial. The first extension was to add a second robotic arm which could pick up and set down blocks. This was designed to evaluate the participant's understanding of how the arms were represented and how they interacted with blocks. The second extension to the model was to add another block to the world. This needed to be done in two parts, first adding a new block to the appropriate set, demonstrating understanding of block representation. The second part required the participant to add the block in the initial state, demonstrating understanding of the initial state's purpose.

The third and final extension was for the participant to add a rule which could allow the system to quiesce. The conditions to do this were given: stack the blocks in alphabetical order and remove them from the world. This task evaluated the participant's understanding of rules and transitions, the predicates in the model, as well as program termination.

These tasks address the first research question as the participants construct an extension to the model. These model extensions give us insight on the level of model understanding the participants have regarding the blocks world model.

After the model extension, an interview was conducted alongside a demographic survey. Participants were asked if they had taken a course on logic, and the number of years of programming experience the participant had, and were asked about the tool and supporting materials. This addresses research question 2, and informs future designs of the editor.

VI. Study Results

Among the participants in the study, all eight had prior programming experience with a mean of 3.75 years, median of 3.5, and mode of 2 and 4. Of the eight participants, seven finished the model extension within the allotted time, while one participant reached a timed cutoff at the final prompt. One participant exhibited language barrier issues during the study.

A. Model Prompt Solutions

Regarding the observed solutions to the model extensions there was a single solution to adding a block to the world, two solutions for the robotic arms, and one solution for quiescence.

All eight participants finished the robotic arm solutions correctly with two different solutions. The two solutions to the robotic arm extension give different amounts of information regarding the participant's model understanding. The first solution involved duplicating the predicates and rules associated with the existing arm. Of the eight participants, six participants started with this solution, and four finished with this solution. This solution demonstrates an understanding of how the arms were represented in the model and how the arms interacted with the blocks.

The second solution involved adding a set named `arm`, in which the participants added elements denoting the two arms, and then extended the existing predicates with a new argument which corresponded to the arm. Of the eight participants, two started with this method, and four finished with this solution. This solution demonstrates an understanding of how the arms are represented in the model as well as how predicates interact with sets, and how rules interact with predicates

Finally, the rule which allowed the program to reach quiescence was completed by six of the eight participants without significant assistance. Of the eight participants, one participant did not understand the model predicates well enough to complete the rule before the time ran out, and we hypothesize one participant exhibited negative transfer from AI planning, significantly increasing the time spent constructing the rule. Additionally, all eight participants opted to test the quiescence rule despite not being requested to do so in the prompt.

As participants walked through rule construction and executions, the participant's descriptions of the model components yield insight into their model understanding. During the rule construction, describing the process of picking up a block demonstrates the participant understands the meaning of the condition and effect predicates in the world state. The descriptions given during the execution show the understanding of predicates in states alongside what rules actually mean when acting upon those states.

B. Discussion

From the formative pilot study, we found that participants exhibited model and logic understanding even when never having been exposed to the model or system prior to the study. These results support the hypothesis that the editor allows users to rapidly construct and understand a model.

Experience with logic is correlated to participant success. From the demographic survey, all the participants who successfully completed the model extension had previously taken a logic course, where the participant who did not complete the model extension had not. This provides a correlation between prior logic experience and user success.

The study we performed reinforced the hypothesis that the Ceptre Editor allows new users to begin programming a system model rapidly, as well as preliminarily supporting that logic experience contributes to user success. Additionally, the study provided data which supports that the users understood the model and logic, and that the users found it easy to do so.

VII. Future Work

The next steps involve conducting additional studies for the editor. The next study currently being designed, is with experts in biological systems whom have system simulation needs but limited programming experience. This study will address prior programming influencing performance, as well as influence further iterations of the program's design by refining based on feedback collected from the participants.

Future studies should provide a look into: (1) how novices and experts in fields with system simulation needs construct and execute programs in the Ceptre Editor; (2) how the users expect the program to run; (3) what information the users expect to receive from the program.

VIII. Conclusion

In this paper we have presented the Ceptre Editor, a integrated development environment designed to assist programming novices with system simulation needs and foster interdisciplinary communication. We presented our pilot study and evaluation, which demonstrated that new users are able to rapidly construct and understand models using the Ceptre Editor, and that a formal logic background is beneficial.

The study we performed provides evidence that combining rule-based languages with introductory programming tool design is beneficial to the end-user-programmer. The Ceptre Editor elicited positive feedback from the participants, and we believe this work furthers assisting end-user-programmers with systems thinking, and facilitating interdisciplinary communication.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1755922. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] P. Boutillier, M. Maasha, X. Li, H. F. Medina-Abarca, J. Krivine, J. Feret, I. Cristescu, A. G. Forbes, and W. Fontana, "The Kappa platform for rule-based modeling," *Bioinformatics*, vol. 34, pp. i583–i592, 06 2018.

[2] A. Fowler, T. Fristce, and M. MacLauren, "Kodu game lab: a programming environment," *The Computer Games Journal*, vol. 1, no. 1, pp. 17–28, 2012.

[3] S. Bistarelli, I. Cervesato, G. Lenzini, R. Marangoni, and F. Martinelli, "On representing biological systems through multiset rewriting," in *International Conference on Computer Aided Systems Theory*, pp. 415–426, Springer, 2003.

[4] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine, "Rule-based modelling and model perturbation," in *Transactions on Computational Systems Biology XI*, pp. 116–137, Springer, 2009.

[5] M. L. Blinov, J. R. Faeder, B. Goldstein, and W. S. Hlavacek, "Bionetgen: software for rule-based modeling of signal transduction based on the interactions of molecular domains," *Bioinformatics*, vol. 20, no. 17, pp. 3289–3291, 2004.

[6] I. Cervesato, N. Durgin, J. Mitchell, P. Lincoln, and A. Scedrov, "Relating strands and multiset rewriting for security protocol analysis," in *Computer Security Foundations Workshop, 2000. CSFW-13. Proceedings. 13th IEEE*, pp. 35–51, IEEE, 2000.

[7] S. Tisue and U. Wilensky, "Netlogo: Design and implementation of a multi-agent modeling environment," in *Proceedings of agent*, vol. 2004, pp. 7–9, 2004.

[8] C. Martens, "Ceptre: A language for modeling generative interactive systems," in *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[9] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16, 2010.

[10] C. Kelleher, D. Cosgrove, D. Culyba, C. Forlines, J. Pratt, and R. Pausch, "Alice2: programming without syntax errors," *User Interface Software and Technology*, 01 2002.

[11] D. Weintrop and U. Wilensky, "Comparing block-based and text-based programming in high school computer science classrooms," *ACM Trans. Comput. Educ.*, vol. 18, pp. 3:1–3:25, Oct. 2017.

[12] M. Kölling, "The greenfoot programming environment," *Trans. Comput. Educ.*, vol. 10, pp. 14:1–14:21, Nov. 2010.

[13] T. Ball, J. Protzenko, J. Bishop, M. Moskal, J. de Halleux, M. Braun, S. Hodges, and C. Riley, "Microsoft touch develop and the bbc micro: bit," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 637–640, IEEE, 2016.

[14] A. Repenning, A. Ioannidou, and J. Zola, "Agentsheets: End-user programmable simulations," *Journal of Artificial Societies and Social Simulation*, vol. 3, no. 3, pp. 351–358, 2000.

[15] K. T. Stolee, "Kodu language and grammar specification," *Microsoft Research whitepaper, Retrieved September*, vol. 1, pp. 4–6, 2010.

[16] C. Multiphysics, "Introduction to comsol multiphysics®," *COMSOL Multiphysics, Burlington, MA, accessed Feb*, vol. 9, p. 2018, 1998.

[17] J. B. Dabney and T. L. Harman, *Mastering simulink*. Pearson, 2004.

[18] G. U. Manual, "Goldsim probabilistic simulation environment," *GoldSim Technology Group LLC, Issaquah, Washington*, 2013.

[19] K. G. Brown, F. Smith, and G. Flach, "Goldsim dynamic-link library (dll) interface for cementitious barriers partnership (cbp) code integration–11444," in *WM2011 Conference, February*, 2011.

[20] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine, "Rule-based modelling, symmetries, refinements," in *Formal Methods in Systems Biology* (J. Fisher, ed.), (Berlin, Heidelberg), pp. 103–122, Springer Berlin Heidelberg, 2008.

[21] H.-L. Tsai, C.-S. Tu, Y.-J. Su, *et al.*, "Development of generalized photovoltaic model using matlab/simulink," in *Proceedings of the world congress on Engineering and computer science*, vol. 2008, pp. 1–6, San Francisco, USA, 2008.

[22] Y.-M. Lee and Y. Hwang, "A goldsim model for the safety assessment of an hlw repository," *Progress in Nuclear Energy*, vol. 51, no. 6-7, pp. 746–759, 2009.

[23] S. S. Vattam, A. K. Goel, S. Rugaber, C. E. Hmelo-Silver, R. Jordan, S. Gray, and S. Sinha, "Understanding complex natural systems by articulating structure-behavior-function models," *Journal of Educational Technology & Society*, vol. 14, no. 1, pp. 66–81, 2011.

[24] A. Compagnoni, V. Sharma, Y. Bao, M. Libera, S. Sukhishvili, P. Bidinger, L. Bioglio, and E. Bonelli, "Bioscape: A modeling and simulation language for bacteria-materials interactions," *Electronic Notes in Theoretical Computer Science*, vol. 293, pp. 35–49, 2013.