Towards a Unified Model of Spatial Computing

Jacob Beal
Raytheon BBN Technologies, USA
Email: jakebeal@bbn.com

Mirko Viroli
University of Bologna, Italy
Email: mirko.viroli@unibo.it

Ferruccio Damiani
University of Torino, Italy
Email: ferruccio.damiani@unito.it

***Abstract*—In spatial computing, there is a fundamental tension between discrete and continuous models of computation: computational devices are generally discrete, yet it is often useful to program them in terms of the continuous environment through which they are embedded. Aggregate programming models for spatial computers have attempted to resolve this tension in a variety of different ways, generating a profusion of approaches that are difficult to compare or combine. Recently, however, two minimal models have been proposed: continuous *space-time universality* and a discrete *field calculus*. This paper unifies these two models by proving that field calculus is space-time universal, and thus provides the first formal connection between continuous and discrete approaches to spatial computing.**

I. Introduction

One of the ongoing challenges in the study of spatial computing is being able to understand and compare the profusion of approaches to specifying and controlling computation on spatial computers. For example, a recent survey [1] identified more than 100 significant languages and programming models for spatial computers. Partially, this is likely due to the fact that spatial computing cuts across many disciplines with different requirements and biases in their approach to computational models (the survey [1] identified eight major domains, including sensor networks, high-performance computing, and synthetic biology). We believe, however, that another important contributor to this confusion of approaches is lack of a unified model for computations distributed over space and time.

Traditional models of distributed computation (e.g., [2], [3]) are formulated in terms of the message passing behavior of individual devices. An important focus of spatial computing research, however, has been to raise the abstraction level, such that adaptive and resilient programs can be specified over aggregates of devices [4], [5], [6], [7]. In these efforts, there has been a major tension between discrete and continuous models of computation. On the one hand, a spatial computer is comprised of devices distributed through some region of continuous space, evolving in continuous time, and spatial computing algorithms and programming models are typically designed to take advantage of this circumstance. At the same time, however, in most cases the computation is actually carried out by a discrete collection of individual computational devices executing discrete rounds of computation.

Recently, minimal models have been proposed for both continuous and discrete approaches. A continuous definition of *space-time universality* is proposed in [8], along with a minimal basis set of space-time universal operators on continuous fields. For the discrete world, a minimal *field calculus* was recently developed in [9]. It is not yet certain whether either model can be further reduced, and there are mathematical questions still to be resolved regarding the proposed continuous model of space-time universality. Regardless, between them these two models appear to cover the breadth of existing aggregate programming approaches. Unifying these two models provides the first formal connection between continuous and discrete approaches to spatial computing. It is our hope that this will provide a critical step toward resolving the tension between continuous and discrete models and toward a general basis for comparison and investigation across all aggregate programming models for spatial computers.

We now prove that field calculus is space-time universal. We accomplish this using the Proto programming language [10], [11] as an intermediate stage, based on the prior analysis of Proto's relationship to space-time universality in [8]. Following a brief review of field calculus, space-time universality, and Proto in Section II, we determine criteria for when a Proto program can be finitely approximated in Section III. We then show that field calculus covers a broad class of finitely approximable Proto programs in Section IV and use this result to prove space-time universality of field calculus.

II. Foundations

In this paper, we will work with three concepts in space-time computation: field calculus, space-time universality, and the Proto programming language. All of these are based around the notion of a *computational field*, which is defined as a function that maps every point in a space to some computational object. Examples include temperatures measured by a sensor network (a scalar field), routes toward a destination (a vector field), the area near an object of interest (a Boolean indicator field), or people allowed access to an area (a set-valued field).

We begin with a brief review of the key concepts that will be used for field calculus, space-time universality, and Proto. All three models are closely related, since Proto was one of the key inspirations for both theoretical models, but with critical differences arising from their differing goals.

A. Field Calculus

The computational field calculus, introduced in [9], is a core calculus intended to capture the key ingredients of programming languages that create and manipulate computational fields. Field calculus is based around a set of five operators for manipulating fields, which were chosen with the aim of finding

```
e ::= x | l | (b ē) | (f ē)                          expression
    | (rep x w e) | (nbr e) | (if e e e)  special constructs
w ::= x | l                                  variable or value
F ::= (def f(x̄) e)                                   function
P ::= F̄ e                                            program
```

Fig. 1. Syntax of field calculus

a minimal set of operators capable of describing any field-based computation. This paper's proof that field calculus is space-time universal confirms that this aim has been satisfied.

The syntax for field calculus is presented in Figure 1, using the overbar notation to denote lists, e.g., $\overline{\mathtt{e}}$ is a list of expressions, $\mathtt{e}_1\ \mathtt{e}_2\ \ldots\ \mathtt{e}_n$. The basic element of field calculus is an expression `e` that can be evaluated on a spatial computer to produce a field. The terminal expressions of field calculus are literal values `l`, which are fields mapping every device to a local value such as a number, a Boolean, or a tuple, and variables `x`, which reference a function parameter or a state variable created by a `rep` construct (see below).

These terminal expressions can then be composed into more complex programs using five constructs:

- *Functional composition:* Using the ordinary rules of mathematical function composition, $(\mathtt{b}\ \mathtt{e}_1\ \mathtt{e}_2\ \ldots\ \mathtt{e}_n)$ is the field obtained by composing together all the fields $\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_n$ by a built-in operator `b`, where the built-in operators are some set of instantaneous functions that can be performed purely locally, such as addition, sine, sensors, actuators, measuring the progress of time and the distance to neighbors, etc.
- *Function definition and call:* Abstraction and recursion are supported by function definition: functions are declared Lisp-style with expressions of the form $(\mathtt{def}\ \mathtt{f}(\overline{\mathtt{x}})\ \mathtt{e})$ and applied by expressions of the form $(\mathtt{f}\ \mathtt{e}_1\ \mathtt{e}_2\ \ldots\ \mathtt{e}_n)$.
- *Time evolution:* Program state is created and tracked over time by a "repeat" construct `(rep x w e)`. The state variable `x` is initialized with the field `w` (a local value or a variable) and updated at each step in time to a new field computed by `e` using the prior value of `x`.[1]
- *Neighborhood values:* Information moves between devices by a construct `(nbr e)` that maps each device to a field of its neighbors' values of field `e`. This is thus a field of fields, and implies that messages containing `e` must be sent to neighbors.[2] Neighborhood fields may then be manipulated and summarized with built-in operators. For example, `(min-hood (nbr e))` maps each device to the minimum value of `e` amongst its neighbors.
- *Domain restriction*: Distributed branching is implemented by the construct $(\mathtt{if}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2)$, which computes $\mathtt{e}_1$ where $\mathtt{e}_0$ is true and $\mathtt{e}_2$ where $\mathtt{e}_0$ is false. This is done by changing the effective domain of the fields, which prevents the unexpected spreading of a distributed computation outside of the devices in the branch, even within arbitrarily nested function calls. Restriction is also necessary for termination of recursion. Domain restriction is thus critically important for controlling the composition of distributed programs,[3] yet it is currently supported by very few models of distributed programming.

Finally, a program is a sequence of function definitions followed by a base expression to be evaluated. As the program is evaluated, the field calculus semantics track "domain alignment" between the ongoing evaluation and the information shared by its neighbors. This process ensures that when a field of neighborhood values is used, it contains information only from those neighbors that have followed the same branch in every domain restriction, even if the information was gathered in a less-restricted branch of the program.

As an example of field calculus, the following field calculus program uses the common "distance gradient" coordination pattern to compute the minimum distance to a high temperature:

```
(def distance-to (source)
  (rep d
   infinity
   (mux source 0
    (min-hood (+ (nbr d) (nbr-range))))))

(distance-to (> (temperature) 25))
```

where `temperature` is an assumed built-in operator for a local temperature sensor, `nbr-range` is a built-in operator that determines the distance from every device to its neighbors, `+` is a built-in addition that is also overloaded to apply to fields, and `mux` is a built-in operator that is a purely functional multiplexer that computes all three of its inputs and then uses the first to select whether to return the second or third. For full details of field calculus and more examples, see [9].

Note that the syntax of field calculus is not tied very tightly to the discrete model, and that most of the constructs could be easily adapted to continuous space-time. The operational semantics of field calculus developed in [9], however, is entirely discrete, depending critically on each device having a finite set of neighbors and on computation advancing in discrete rounds.

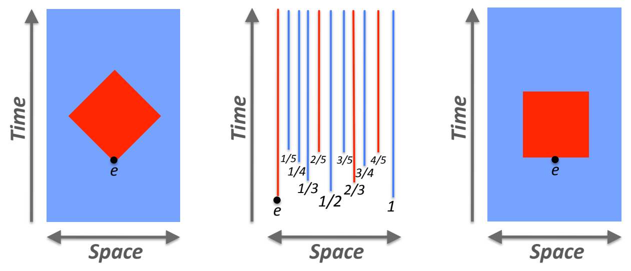
B. Space-Time Universality

The notion of *space-time universality* as proposed in [8], considers computations in terms of arbitrary fields mapping each point in a space-time manifold to some value. The set of such computations includes many that can be readily defined and computed in theory but cannot be implemented or even

[1]Note that if `rep` constructs are nested, the inner `rep` is resolved using the prior value of the outer, while the outer is resolved using the new value from the inner.

[2]Field calculus, being a theoretical construct, finds it more elegant to send each device's entire evaluation state and ignore everything not needed.

[3]Note that while such behavior can theoretically be implemented without restriction. The distinction is similar to the difference between preemptive and cooperative multitasking: domain restriction, like preemptive multitasking, imposes an implicit control framework on every computation. Without domain restriction, we must count on every function being extended with a compatible implementation of an intricate architecture for managing program scope, greatly increasing the complexity of code and the fragility of any program.



(a) Physical (b) Non-approximable (c) Acausal

Fig. 2. Space-time universality applies only to physically realisable computations, meaning they are both causal and finitely approximable. For example, if an otherwise `false` field (blue) contains a region of `true` values (red) triggered by an event e, then (a) is both causal and finitely approximable. An example of a function that is not finitely approximable (b) measures distance from an event and is `true` on rational distances with an even numerator and `false` for odd numerators. An acausal example is shown in (c), which requires devices to switch state before information from e has time to move to them across space.

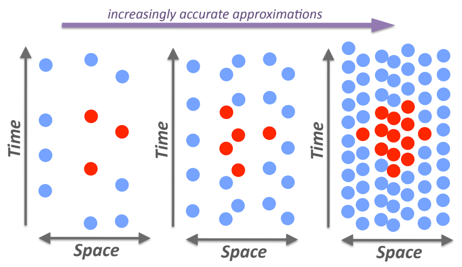


Fig. 3. Example sequence of increasingly accurate discrete approximations of the computation example in Figure 2(a). A function is finitely approximable if the ϵ-approximations of every such sequence converge to the continuous limit for every possible set of inputs.

approximated by any real computing system, to the best of our current understanding of the laws of physics. The notion of space-time universality is therefore restricted to only those computations that are physically realizable: causal and finitely-approximable functions (Figure 2).

Causal computations are simply defined as those in which the value at any point in space-time depends only on information that can have reached it propagating across space at some maximum velocity c. An example of an acausal computation is one in which every device always reports the instantaneous current value sensed by a designated device in the network, with no delay for transmission.

Intuitively, a function is finitely approximable if increasing the density of a discrete set of devices brings the values they compute closer to the continuous ideal. We formalize this notion using the following two definitions adapted from [8] (with some small modifications):

Definition 1 (ϵ-approximation)**.** *Consider a finite set A_ϵ of points in a manifold M of finite diameter, chosen such that no point in M is more than distance ϵ from a point in A_ϵ. An ϵ-approximation of a field F with regards to the set A_ϵ is any field F_ϵ with the same domain as F, in which every point has the value of F at the nearest point in A_ϵ (choosing arbitrarily for equidistant points).*

Note that although the definition of ϵ-approximation specifies a finite set and a finite diameter (meaning finite space and finite time), there is no bound on how large these sets might be. Note also that although the set A_ϵ is a finite set, the ϵ-approximation constructed from it is not, which will be important for the next definition.

Definition 2 (Finitely Approximable)**.** *Consider any function f that maps fields to fields, and a countable sequence of ϵ_i-approximations of its inputs with $\epsilon_i < \epsilon_{i-1}$. Let O be the output for a given set of inputs, and O_i the output using ith approximation of the inputs. The function f is finitely approximable if for every well-defined application of f to a set of input fields producing a well-defined output O, and for every converging sequence of ϵ_i-approximations, it is the case that $|M \cup M_i - M \cap M_i|$ and the Lebesgue integral $\int_{M \cap M_i} |O - O_i|$ both converge to zero, where M and M_i are the domains of O and O_i respectively.*

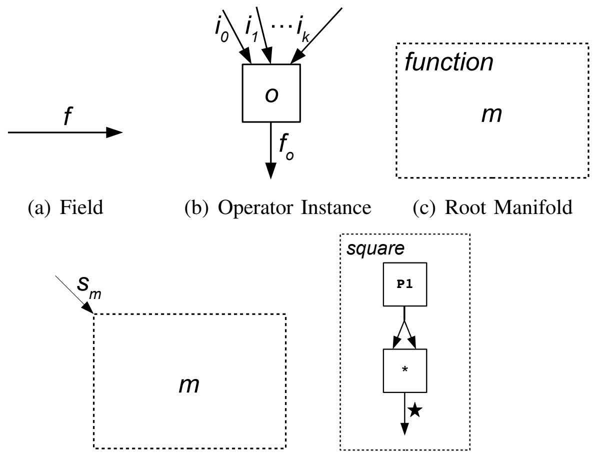
For functions where the value at each device is a field, such as `nbr` or `nbr-range`, we extend the definition of finitely approximable to integrate the difference across these "per-device" fields as well.

Note that M and M_i need not be the same, because the function f may output fields with different domains depending on the values of its inputs. The definition requires that the difference in domains converge to a set of measure zero. This definition also requires that some metric of difference be imposed on the range of output values. The definition then requires that the locations in which values differ by a non-trivial value (all differences being non-trivial for non-numerical values such as sets) must converge to a set of measure zero.

A model of computation is then space-time universal if it can finitely approximate any causal and finitely approximable function. Figure 3 illustrates the concept of finite approximability, using the computation example in Figure 2(a).

In [8], a small set of operators is proposed to be space-time universal, and the sketch of a proof by construction is given. Although said proof has not yet been fully worked, for purposes of this paper, we will assume that it is correct, and use the proposed operators as our target representation for proving space-time universality.

Note that the definition of finite approximability does not imply that the function f is actually computable. For any portion of the input space that f is non-computable, the output O is not well-defined, and the finite approximability property thus asserts no constraint on the behavior of ϵ-approximations. A slightly stronger property, which we do not consider here, would assert that the sequence of ϵ_i-approximations converge to having equivalent regions where outputs are ill-defined. In practice, this is often the case, but as we are primarily interested in computations that can be executed, we leave this property for future investigation.



(a) Field (b) Operator Instance (c) Root Manifold

(d) Dependent Manifold (e) Function Definition

Fig. 4. Diagrammatic representation of elements of a Proto program.

C. Proto

We will use the Proto spatial computing language [10], [11] as our intermediary point between field calculus and space-time universality. The primary reason for this choice is that analysis of Proto in [8] determined that it is nearly space-time universal. The only capability that Proto is missing is the ability to measure certain properties of the local structure of a spatial manifold. This capability can be fulfilled in field calculus through built-in operators, and thus if field calculus can approximate the class of Proto programs that are nearly universal, then field calculus is universal.

Proto is also useful as a bridge because it is based on an abstraction called the amorphous medium [12], which is used to map continuous aggregate models into a discrete approximation, which is assumed by construction to be appropriate. In this paper, one consequence of our proofs will be to determine that this assumption is guaranteed to hold for a broad class of programs.

For this paper, we will use a definition of Proto programs adapted from [13], which provides a formal representation of Proto programs in terms of a collection of operators on manifolds, and proves that all such continuous-space programs are well-defined. As defined in that paper, a Proto program may be represented as a tuple $P = (M, F, O, R, D)$, where:

- M is a set of compact Riemannian manifolds with both space and time dimensions. Each $m \in M$ is either a root manifold or a dependent manifold. Root manifolds are instantiated at runtime from the invocation of the program or by function calls. Dependent manifolds are defined as restrictions of root manifolds by a Boolean-valued selector field $s_m \in F$ (playing the same role as `if` in field calculus). We also assume that no $m \in M$ is infinite in time.
- F is a set of fields $f : m \to V$ mapping manifolds to values.
- O is a set of operator instances $o : f_{i_0} \times f_{i_1} \times \cdots \times f_{i_k} \to f_o$, each mapping from a set of zero or more input fields to a single output field.
- R is a return value function $R : M \to F$ mapping each root manifold m to a unique field f with m as its domain.
- D is a set of function definitions, each of which is a tuple $d = (o, m, \{(i, o_i)\}, r)$, associating an operator o for the function with the function's root manifold m, return value $r = R(m)$, and a set of mappings from the ith function input to a "parameter" operator o_i that references its values.

Figure 4 illustrates these elements diagrammatically: such diagrams will be used in presentation of our proofs. Note that these definitions abuse terminology for clarity: technically, these program elements are not the mathematical objects themselves, but descriptions of the objects that are instantiated in any particular evaluation of a program.

Within this representation, we consider only well-defined Proto programs, which follow the "natural" constraints of generating a program as a composition of operator instances, in essence simply asserting that there are no missing or conflicting elements:

- For every operator $o \in O$, its input and output fields have compatible types and domains (see [13] for details; we also present these conditions as needed in our proofs).
- Every field $f \in F$ is the output of precisely one operator instance o_i.[4]
- The set of manifolds M is precisely equal to the set of domains of fields in F.
- All but precisely one return value in R (the "base expression") are associated with their corresponding root manifolds by function definitions in D.
- For each function definition d in D, with k parameter mappings, the indices uniquely cover 1 to k and the operator instances are unique and have outputs with a domain equal to the function's root manifold.

Under this model, a Proto program is evaluated by identifying the root manifold of the base expression with a space-time region and a neighborhood function N that maps each point to a neighborhood contained in a ball of radius r around it.

The operators in Proto fall into four classes, which are closely matched with those in field calculus:

- Pointwise operators are a Turing-universal set of operators acting only instantaneously on the values at a device, like the built-in operators `b` in field calculus.
- The `restrict` operator takes two inputs, reducing the domain of the first to only those points with Boolean value `true` in the second. This plays the same role as domain-alignment in field calculus.
- The `delay` operator shifts values in time by some small value Δt, where Δt is a continuous positive function over the manifold. This plays the same role as the variable annotation in `rep` constructs in field calculus.

[4]Because of their ongoing temporal extent, Proto programs are conceived of similar to agents, obtaining external input not through their invocation but through "sensor" operators.

- The `nbr` operator gathers the most recent available values from the neighborhood of a device, just as it does in field calculus. Values move across space at a fixed velocity c. This means that the value for device i gathered by a `nbr` operator at device j is lagged by time $d(i, j)/c$.

For a full treatment of Proto formalized as continuous field operators, see [13].

III. Finite Approximability of Proto

As the first stage of our proof, we focus on finite approximability of Proto, whether by field calculus or other means. Most pointwise operators in Proto are finitely approximable. For example, addition and subtraction are finitely approximable. So is measuring distance to neighbors with `nbr-range` or progress of time with `dt`. Some simple operations, however, are not finitely approximable: for example, testing for equality between real numbers is not in general finitely approximable. Testing for equality between discrete values such as integers, however, is. Thus, for example, the multiplexing `mux` operator, which selects between two inputs based on whether a third is true or false, is finitely approximable.

Critically, the three special operators of Proto, `restrict`, `delay`, and `nbr`, are all finitely approximable.[5] For `restrict`, the values of the approximated output are identical to the input, and only the domain may be different. The domain is set by a test for equality between discrete values, however, so if the input differences converge to zero, then the difference in domains must also go to zero. The `nbr` operator is similar: values are copied directly from the input approximation, and the neighborhood domains at each device converge. The `delay` operator also copies values directly, but problems could be caused by the fact that the Δt shift can vary over space and time. The continuity constraint for Δt given above, however, ensures that it too converges.

Given these facts, we can identify a large class of Proto programs which are finitely approximable:

Theorem 1 (Finite Approximability of Proto Programs)**.** *Any well-defined Proto program $P = (M, F, O, R, D)$ composed only of finitely approximable operator instances is finitely approximable.*

Proof Sketch (full proof in Appendix A): For any Proto program without feedback or function calls, all of the operator instances O can be ordered, so that each operator draws its inputs (if any) only from operators before it in the order. We then consider a sequence of "partial programs" beginning with only the first operator instance in the order (which is finitely approximable by definition), and adding each operator instance in order. If the ith partial program is finitely approximable, then all of its fields converge, which means that the output field of the $i + 1$ operator instance converges, and so does the $i + 1$ partial program. Thus, by induction, such a program is finitely approximable. This argument may then be extended to cover function calls (using a substitution model), and feedback loops (using a block diagram and induction over time).

[5]This is not surprising, since Proto was designed around these operators explicitly for the purpose of creating approximate distributed algorithms.

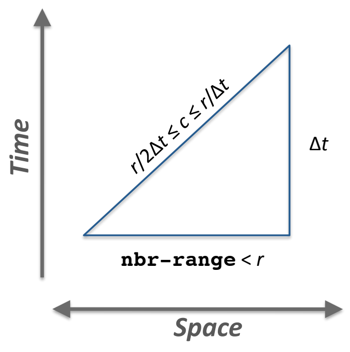


Fig. 5. In field calculus, information moves across space and time in discrete steps, while in Proto it flows at a fixed velocity c. The field calculus approximation of c is determined by the ratio between neighborhood range r and time step Δt. Thus, any computation that depends on both exchanging information and neighborhood size can be approximated for only a restricted range of time-steps Δt.

IV. Field Calculus is Space-Time Universal

We now show that field calculus implements a broad class of Proto programs. This will be enough to prove that field calculus is universal, a corollary of which is that it can also implement every finitely-approximable Proto program (Proto is strictly causal). First, we need one more restriction to the class of Proto programs that we will attempt to approximate:

Definition 3 (Neighborhood Independence)**.** *Consider a sequence of neighborhood functions N_i with strictly decreasing bounding radii r_i. A Proto program $P = (M, F, O, R, D)$ is neighborhood independent if the values of every field $f \in F$ converge for every such sequence N_i.*

We need this restriction because of communication is quantized in field calculus (Figure 5). In Proto's continuous model, information flows across space at a velocity c. In field calculus, assuming synchronous rounds[6] of length Δt and a communication range of r, information moves at maximum velocity $c \leq \frac{r}{\Delta t}$ and minimum velocity $c > \frac{r}{2\Delta t}$ (the subsequent step being just outside of the communication range). Constructing an ϵ-approximation using field calculus requires $\Delta t < \epsilon$, so either c or r must change. It is possible to create a Proto program that depends on both c and r, such as:

```
(def depends-on-speed-and-radius (x)
  (* (any-hood (nbr (sense 1)))
   (distance-to (sense 1))))
```

In this program, the `any-hood` expression depends on r, while the `distance-to` function depends on c. Neighborhood independence eliminates the dependence on r, which we choose as these programs are less often of interest. What this

[6]Asynchronous evaluation is somewhat more complex, depending also on the time lag between computing a round and state arriving at neighbors, but yields the same asymptotic bound with different constants.

then means is that, as the density of devices in space rises, we can also increase the density of evaluations in time, so long as the density in space increases faster than the density in time, so that the average number of neighbors continues to increase.

We can now define a class of Proto programs that can be readily approximated by field calculus, and from that show its space-time universality and coverage of all other finitely approximable Proto programs.

Theorem 2. *Any well-defined neighborhood-independent Proto program $P = (M, F, O, R, D)$ composed only of finitely approximable operator instances can be approximated using field calculus.*

Proof Sketch (full proof in Appendix B): This theorem can be proved by construction; using the same partial program ordering as in Theorem 1, we create a function definition for each Proto operator instance. The function for the ith partial program has as arguments the fields computed by the prior partial programs, uses the appropriate ones as inputs for the ith operator instance, and then includes the result in the call to the next partial program function. At the end of this sequence, the proper field is returned. By providing similar constructions for functions, restrictions, feedback loops, and neighborhood computations, we can cover the set of all Proto programs satisfying the preconditions of this theorem.

Theorem 3 (Space-Time Universality)**.** *Field calculus is space-time universal.*

Proof Sketch (full proof in Appendix C): A set of space-time universal operators have been proposed in [8]. All but one of these operators have been shown to be covered by Proto. The last operator can be covered by the built-in operators `b` of field calculus. We then show that a set of Proto programs covering these operators satisfy the preconditions of Theorem 2; thus, field calculus can implement a program equivalent to a Proto program that is equivalent to to any program constructed with the space-time universal operators proposed in [8].

Corollary 4 (Approximation of Proto by Field Calculus)**.** *Any well-defined finitely approximable Proto program can be approximated using field calculus.*

Proof. This corollary follows directly from space-time universality. Any Proto program not covered by Theorem 2, however, might be extremely inefficient in its implementation. □

We thus have the result that we were looking for: field calculus can serve as a core calculus for both continuous and discrete models of space-time computation.

V. Contributions

This paper presents a key result towards unifying continuous and discrete approaches to spatial computing, demonstrating that field calculus is universal for continuous space-time computation. One of the challenges in spatial computing has been the difficulty of comparing the many different approaches to programming computing systems that extend across space-time. With the recent development of field calculus and the addition of this result connecting space-time universality and field calculus, it will now be possible to at least answer the question of whether any given language is sufficiently strong.

Another important area not yet addressed is the computation *of* spaces, in addition to computation *on* spaces. This includes, for example, the development of manifold topologies using MGS [14], [15] and formation control in swarm systems (e.g., [16], [17], [18]). Necessary future work includes investigation of how to formally connect the models of computation developed so far with a notion of universality in the formation of manifolds and discrete device distributions.

References

[1] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, M. Mernik, Ed. IGI Global, 2013, ch. 16, pp. 436–501, a longer version available at: `http://arxiv.org/abs/1202.5509`.

[2] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press and McGraw-Hill, 2009.

[4] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss, "Amorphous computing," MIT, Tech. Rep. AIM-1665, 1999.

[5] D. Yamins, "A theory of local-to-global algorithms for one-dimensional spatial multi-agent systems," Ph.D. dissertation, Harvard, Cambridge, MA, USA, December 2007.

[6] J. Beal, "Engineered self-organization approaches to adaptive design," in *1st International Conference on Through-life Engineering Services*, R. Roy, E. Shehab, C. Hockley, and S. Khan, Eds. Cranfield University Press, November 2012, pp. 35–42.

[7] J. Fernandez-Marquez, G. Marzo Serugendo, S. Montagna, M. Viroli, and J. Arcos, "Description and composition of bio-inspired design patterns: a complete overview," *Natural Computing*, vol. 12, no. 1, pp. 43–67, 2013.

[8] J. Beal, "A basis set of operators for space-time computations," in *Spatial Computing Workshop*, 2010, available at: `http://www.spatial-computing.org/scw10/`.

[9] M. Viroli, F. Damiani, and J. Beal, "A calculus of computational fields," in *Advances in Service-Oriented and Cloud Computing*, ser. Communications in Computer and Information Sci., C. Canal and M. Villari, Eds. Springer Berlin Heidelberg, 2013, vol. 393, pp. 114–128.

[10] J. Beal and J. Bachrach, "Infrastructure for engineered emergence in sensor/actuator networks," *IEEE Intelligent Systems*, vol. 21, pp. 10–19, March/April 2006.

[11] "MIT Proto," software available at `http://proto.bbn.com/`, Retrieved January 1st, 2012.

[12] J. Beal, "Programming an amorphous computational medium," in *Unconventional Programming Paradigms*, ser. Lecture Notes in Computer Science, J.-P. Banatre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds. Springer Berlin Heidelberg, 2005, vol. 3566, pp. 121–136. [Online]. Available: http://dx.doi.org/10.1007/11527800_10

[13] J. Beal, K. Usbeck, and B. Benyo, "On the evaluation of space-time functions," *The Computer Journal*, 2012, doi: 10.1093/comjnl/bxs099.

[14] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz, "Computational models for integrative and developmental biology," Univerite d'Evry, LaMI, Tech. Rep. 72-2002, 2002.

[15] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher, "Computation in space and space in computation," Univerite d'Evry, LaMI, Tech. Rep. 103-2004, 2004.

[16] J. Bachrach, J. Beal, and J. McLurkin, "Composable continuous space programs for robotic swarms," *Neural Computing and Applications*, vol. 19, no. 6, pp. 825–847, 2010.

[17] P. Ogren, E. Fiorelli, and N. Leonard, "Formations with a mission: Stable coordination of vehicle group maneuvers," *Proc. 15th Int'l Symposium on Mathematical Theory of Networks and Systems*, vol. 170, 2002.

[18] M. Ji and M. Egerstedt, "Distributed coordination control of multiagent systems while preserving connectedness," *Robotics, IEEE Transactions on*, vol. 23, no. 4, pp. 693–703, aug. 2007.

Appendix A
Finite Approximability of Proto Programs

Restatement of Theorem 1 (Finite Approximability of Proto Programs). Any well-defined Proto program $P = (M, F, O, R, D)$ composed only of finitely approximable operator instances is finitely approximable.

Proof. We first consider programs without feedback or function calls. For such a program, well-definedness means that it is always possible to construct a total order of operator instances such that the input fields of o_i are outputs only of operators o_j with $j < i$. We can then consider a sequence of partial programs P_i restricted to contain only the first i operator instances and their associated fields, manifolds, and returns.

Assume that partial program P_i is finitely approximable. This means that for operator instance o_{i+1}, all of its input fields will converge for any converging sequence of ϵ_j-approximations. Since o_{i+1} is finitely approximable as well, this means that both $|M \cup M_j - M \cap M_j|$ and $\int_{M \cap M_j} |O_i - O_{i,j}|$ converge to zero, and thus P_{i+1} is also finitely approximable. Since we assume that every operator instance in P is finitely approximable, P_1, which contains only a single operator with no inputs, is finitely approximable. Thus, by induction any Proto program with finitely-approximable operators and no feedback or function calls is also finitely approximable.

For a Proto program with function calls, but no recursion, the number of function calls that can be made is bounded and can be determined statically from the program definition. It is thus possible to construct an equivalent program in which every function call is made precisely once: for any function that is called more than once, create a unique function name, but with the same definition, for each body. Proto does not currently support iteration or higher order function calls, so without recursion this process is guaranteed to be bounded. Once all function calls are unique, operators can once again be given a total order following a substitution model of function evaluation[7].

Adding in recursion, we now have two possibilities: either a program halts for a certain set of inputs or it does not (where halting means its computation requires only a finite number of recursions). Obviously, the sets of inputs which fall into each category is not generally computable, but it will suffice to prove finite approximability holds for both sets. For the set of non-halting inputs, finite approximability is trivially true: since the output is not well-defined, finite approximability makes no assertion about the behavior of ϵ-approximations. For any input that does halt, we can use a substitution model of evaluation to find an equivalent recursion-free program, for which the prior proof of finite approximability holds.

We now extend to include programs with feedback. A Proto program with feedback cannot be totally ordered, since there

[7]This follows the usual intuitive definition; formal details of the mapping may be found in [13].

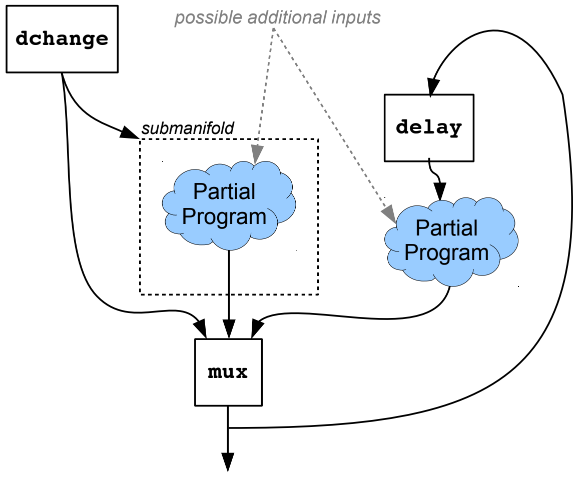


Fig. 6. Proto feedback loops are based on a `delay` operator that time-shifts values by some small duration Δt. Initial values are given by a subprogram run in the submanifold where the domain has changed (identified by operator `dchange`, and are updated by computing using the delayed value.

are `delay` operators that form feedback loops. These are not arbitrary, however, but in order to ensure well-formedness (per [13]) are always generated following the template shown in Figure 6 (or a similar parallel template, which is equivalent to applying the template shown to a tuple). Initial values are given by a subprogram run in the submanifold where the domain has changed, and are updated by computing using the output value time-shifted by Δt by the `delay` operator. If we consider the complex of delay, the partial program for update, and the multiplexer `mux` as a single composite function, we can still totally order the remainder of the program with regards to this function. We need then only check that this looping partial program is finitely-approximable.

The `mux` and `delay` operators are both finitely approximable, and so, by assumption are all of the operators in the update partial program. Letting m be the output of the `mux`, if this partial program does not converge, then there must be some measurable set $X \subset m$ where the values do not converge. Since all of the operators are finitely approximable, then it must be the case that there is an input to the `delay` (or a `delay` within the update partial program) where the approximated values do not converge. By the definition of `delay`, if X does not converge, then the input space that does not converge must be some X' that is X shifted backward in time by the appropriate set of Δt values. Since the input to a `delay` is the output of its corresponding `mux`, this means that $X = X'$. Since Δt is greater than zero at every point, however, this cannot be the case for any finite X. Thus, we have by contradiction that all Proto programs composed of finitely approximable operator instances are finitely approximable.[8] □

[8]Note that this does not mean the Proto program will necessarily converge as Δt goes to zero—that is an entirely different problem. For example, `(rep x 1 (- 1 x))` does not converge as $\Delta t \to 0$ but for any given choice of Δt it is finitely approximable.

Appendix B
Proof Field Calculus Approximates Proto

We approach proof of the approximation of Proto programs by construction, beginning with simple pointwise programs and adding in classes of operators one at a time.

Lemma 5. *Any well-defined Proto program (M, F, O, R, D) composed only of finitely approximable pointwise operator instances and no dependent manifolds or function definitions can be approximated using field calculus.*

Proof. Field calculus has a free choice of a set of built-in local operators `b` and local values `l`. We can thus choose this set to contain the set of finitely approximable pointwise operators in Proto, using `l` for all literals and `b` for all other operators. Since we also assume there are no functions and thus no function calls, it is therefore the case that for any operator instance $o \in O$, we can choose an equivalent built-in operator $\texttt{b} \cup \texttt{l}$.

We now show by construction that it is possible to choose a field calculus expression that generates an equivalent set of fields and operator instances. The main challenge here is that the same Proto field can be consumed by multiple operator instances, while field calculus uses each subexpression precisely once. The solution is to use functions to bind each output value to a function variable, where it can be freely referenced as many times as needed (similar to how a "let" expression is constructed in lambda calculus).

As in Theorem 1, given only pointwise operator instance it is always possible to construct a total order over operators, such that the input fields of o_i are outputs only of operators o_j with $j < i$. For the ith operator instance in such a sequence, let p_k be the sorted index of the operator instance producing the kth input field (e.g., if the third input is the field output by o_5, then $p_3 = 5$).

In the case where o_i maps to a built-in operator in `b`, we define the following function:

```
(def f_$i$ (v_1 v_2 ... v_{$i-1$})
   (f_{$i+1$} v_1 v_2 ... v_{$i-1$}
                  ($b$_$i$ v_$p_1$ v_$p_2$ ...)))
```

where b_i is the built-in operator corresponding to o_i. Otherwise, it must be the case that o_i maps to a literal l_i in `l`, in which case we define `f_i` as:

```
(def f_$i$ (v_1 v_2 ... v_{$i-1$})
   (f_{$i+1$} v_1 v_2 ... v_{$i-1$} $l_i$))
```

In other words, we construct a function that "wraps" each operator instance with the set of values created by all prior operator instances, and passes these along with the output it generates to the next operator instance in sequence.

Now let us consider the base cases: the first and last functions in this sequence. The first function is in essence no different from the ith function, but for the fact that its set of pre-existing variables happens to be empty, and it is guaranteed to be a literal or a function with no input:

```
(def f_1 () (f_2 ($b_i$)))
(def f_1 () (f_2 $l_i$))
```

The last function, on the other hand, will be used to establish the return value. Since we assume no function calls, there is precisely one return value pair $r = (m, f_j)$. For a program with n operator instances, we thus define `f_`$\{n+1\}$ as:

```
(def f_{$n+1$} (v_1 v_2 ... v_n) v_$j$)
```

thereby returning the jth field.

Finally, we complete our field calculus program with an expression that invokes the first function in the chain, making the complete field calculus program:

```
(def f_1 () ...
(def f_2 (v_1) ...
...
(def f_{$n+1$} (v_1 v_2 ...
(f_1)
```

We thus have a field calculus program that invokes a set of operators equivalent to O producing fields equivalent to F and returning a value equivalent to the single pair in R. Since we assume no dependent manifolds or function calls, there is precisely one manifold in M, and since we have used only literals and pointwise operators in our field calculus there are guaranteed to be no domain changes, thus giving the same manifold there as well. Likewise, with no function definitions D is empty.

Thus, we have shown it is possible to construct a field calculus expression that has an equivalence mapping to every element of a well-defined Proto program (M, F, O, R, D) composed only of pointwise operator instances and no dependent manifolds or function calls. By Theorem 1, we know that the entire program is finitely approximable if each operator instance is finitely approximable. Thus, this Proto program is finitely approximated by the field calculus expression. □

Lemma 6. *Any well-defined Proto program (M, F, O, R, D) composed only of finitely approximable pointwise operator instances and no `restrict` operator instances or dependent manifolds can be approximated using field calculus.*

Proof. To the construction in Lemma 5, we now introduce function definitions and calls. This changes Proto programs by adding a set of function definitions D, operator instances that invoke these functions, and one additional root manifold in M and return value in R for each function. Note that since we are still not yet considering `restrict` operator instances, this proof only covers functions with no references to externally defined variables.

Let us start with the function definitions. Where previously we sorted all operator instances together, we will now partition them into sets, one for the base expression as before, plus one set for the operator instances associated with each function (i.e., those whose outputs are contained within its root manifold). We then impose an arbitrary total order on the set of function definitions in D.

We may then apply the same construction of "wrapping" functions for operator instances to each of these sets, with the following modifications:

- We name the function associated with the ith definition as function_i. Operator instances for this function will be constructed just as for built-in operators, except that function_i will be used instead of a built-in operator from b.
- The jth operator instance for the ith function will be wrapped with a function named function_i_j, and invoke a function function_i_$\{j+1\}$.
- The variable sets for operator instances wrapper functions will start with the parameters p_1, p_2, etc. The first wrapper will not be directly invoked, but will be called by a definition of the function:

```
(def function_i (p_1 p_2 ... p_n)
  (function_i_1 p_1 p_2 ... p_n))
```

Just as the construction in Lemma 5 created a finite approximation of a Proto program "base expression," this construction adds an finite approximation for each function definition and appropriate calls to said functions. The field calculus entry point and final return call for each function correspond to its equivalent entries in M and R, and the parameter associations in its definition are equivalent to the parameters passed in the entry point definition of function_i.

Note that not all function definitions are necessarily invoked; in such cases, the equivalent field calculus construction will still create the same functions and invoke or not invoke them in the same pattern. Note also that the equivalence does not necessarily mean that the program can be successfully evaluated. For example, it is possible to create a Proto program containing an infinite recursion. In most (perhaps all) such cases, the field calculus construction will also fail to evaluate, though details may vary because of the total order imposed in constructing the field calculus program. Regardless, finite approximability holds because it is trivially satisfied for any condition where the Proto program cannot be evaluated. □

Lemma 7. *Any well-defined Proto program* (M, F, O, R, D) *composed only of finitely approximable pointwise and* restrict *operator instances can be approximated using field calculus.*

Proof. To the construction in Lemma 6, we now introduce restrict operator instances and dependent manifolds.

In Proto, restrict operator instances and dependent manifolds are generated in only three ways:

- When a function references an externally defined variable, this reference passes through a restrict operator instance.
- As part of a two-way branch template equivalent to the field calculus if construct, shown in Figure 7. This template creates two dependent manifolds selected complementarily from a test field, and inserts restrict operators for all references to externally defined fields.
- For the initialization of a feedback loop, following the template in Figure 6; this will be addressed in the next proof, when feedback is introduced.

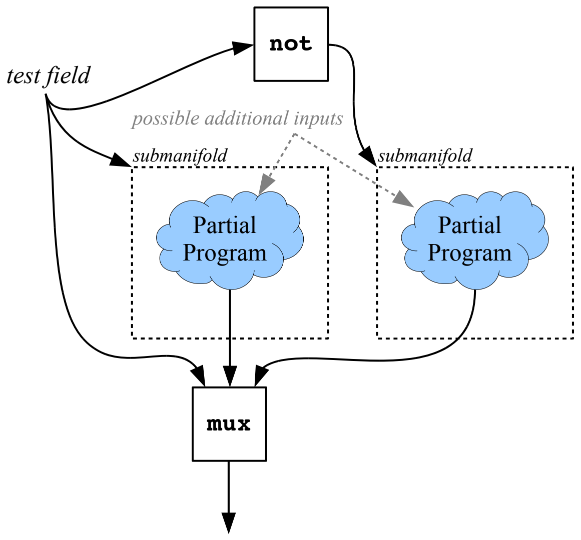


Fig. 7. Proto dependent manifold constructs are based on either the delay template in Figure 6 or the two-way branch template shown above.

We will deal with each of the first two by extending our field calculus construction, as before, while deferring the third to our next proof.

With regards to external references: in Proto, these are generated by lexical nesting of function declarations, which then compiles into the observed restriction relations. Field calculus supports no such notion, since the only variable declarations are in "top level" functions. To implement such references in field calculus, we will transform lexical scope into additional function parameters. For any function_i containing a restrict operator implementing an external reference, that value will be transformed into an additional parameter of the function. Any operator invoking function_i either is in the function that is the source of the external variable, in which case that field is immediately available to supply to the function call, or else it must also have the new parameter added to its definition. Any such chain of definitions and invocations must ultimately reach the function where the external variable is defined, because the original reference was lexically scoped and Proto does not allow first-class functions that might be invoked outside of the scope of their definition. The functionality of the restrict operator itself is then implemented in field calculus by its domain alignment semantics when the new parameter is referenced.

To implement the branching template, we segregate the set of operator instances in the partial programs for each branch, and treat each as through it were an additional function. The remainder of the construct is then treated as a single operator instance for the purposes of ordering and implemented with the following field calculus function:

```
(def f_i (v_1 v_2 ... v_{i-1})
  (f_{i+1} v_1 v_2 ... v_{i-1}
    (if v_ test
      (function_ true v_p_t1 v_p_t2 ...)
```

```
        (function_false v_pf1 v_pf2 ...))))
```

where v_*test* is the test field, and the two functions invoked are the newly separated branch functions, with parameters for all of their references to additional variables. The alignment semantics of the field calculus `if` construct are equivalent to the construction of the two dependent manifolds, and evaluation of the branch functions is equivalent to evaluation within the submanifolds. □

Restatement of Theorem 2. Any well-defined neighborhood-independent Proto program $P = (M, F, O, R, D)$ composed only of finitely approximable operator instances can be approximated using field calculus.

Proof. To the constructions in Lemma 7, we now add the final two special Proto operators: `delay` and `nbr`. This is a relatively simple extension, given that field calculus `rep` and `nbr` operators are modeled off of their Proto equivalents.

As noted previously in Theorem 1, the `delay` operator is only introduced in Proto feedback loop constructs following the template shown in Figure 6. For single feedback variables, an equivalent construct can be implemented in field calculus using the `rep` construct, segmenting the initialization and update partial programs into new programs as with `if` in the prior Lemma:

```
    (def f_i (v_1 v_2 ... v_{i-1})
      (f_{i+1} v_1 v_2 ... v_{i-1}
        (rep var
          (function_init v_pt1 v_pt2 ...)
          (function_update
            var v_pf1 v_pf2 ...))))
```

In the case of parallel constructions in Proto, this construct is exactly the same, except that `var` will be a tuple constructed from the fields of the parallel constructs, and use of those fields will be implemented with built-in operators `b` referencing elements of the tuple.

As for the `nbr` operator: because the `nbr` operator is used equivalently in both Proto and field calculus, it may be implemented just though it were a pointwise operator in Lemma 5.

With these additional constructions, every possible arrangement of fields and operator instances in a well-defined Proto program is covered by an equivalent construction in field calculus. By the nature of these constructions, they also create equivalent manifolds, function definitions, and return values.

Finally, we can choose along with the decreasing series of ϵ_i-approximations a decreasing series of neighborhood radii r_i and decreasing time steps Δt_i, such that the value of $r_i/\Delta t_i$ is always greater than any finite positive rate of information propagation c used for evaluating the Proto program. We cannot ensure that it is equal to c, because of the loose bounds of average rate of information movement in a discrete network: biased distributions of devices can always cause different portions of the network to have different effective rates of information flow. However, if we shrink r_i more slowly than we shrink ϵ_i, then we can reduce the possible variation by reducing the possible shortest step that information can take in a single round: rather than being bounded below by $\frac{r_i}{2\Delta t_i}$, the effective rate of information movement will be bounded below by $\frac{r_i(1-\epsilon_i)}{\Delta t_i}$, which converges with the upper bound to give a converging rate of information flow.

Because we assume the Proto program is neighborhood independent, the choice of r_i and Δt_i will not affect convergence of the series of ϵ_i-approximations, and thus, because every operator is assumed to be finitely-approximable, we have by Theorem 1 that our extended field calculus construction can finitely approximate any well-defined neighborhood-independent Proto program composed of finitely approximable operator instances. □

APPENDIX C
PROOF OF SPACE-TIME UNIVERSALITY

Restatement of Theorem 3 (Space-Time Universality). Field calculus is space-time universal.

Proof. A definition of space-time universality has been proposed in [8], along with a small set of operators for achieving it. In particular, [8] proposes the following set of operators and sketches a proof that they can composed functionally to implement any finitely-approximable causal computation:

- n_d returns a field mapping each device to a field of vectors giving its displacement in space and time from each neighbor.
- g returns a field giving the metric tensor of space at each device.
- n_v collects state from neighbors, equivalent to the `nbr` operator in field calculus and Proto.
- n_r takes a Boolean field and a field whose value at each device is a field over neighbors and restricts the domain of the neighbor fields to only those devices marked where the Boolean field is true.[9]
- n_m takes a field whose value at each device is a field over neighbors and computes for each device the infimum (generalized minimum) of the neighbor values.
- P is a Turing-universal set of pointwise operators

Also in [8], Proto is shown to be able to implement all of these operators except for g (although it turns out that implementing n_r is somewhat difficult).[10] To prove that field-calculus is space-time universal, we thus need to show the following two things: 1) field calculus can implement g, 2) the subset of Proto that we proved is finitely approximated by field calculus in Theorem 2 covers the remainder of the operators. If both of these hold, then any space-time computation that is

[9]This is similar, but not identical to a `restrict` operator, which also acts on the domain of the field of neighborhood fields.

[10]Note that there is no reason in principle that Proto could not be extended to include measurement of the metric tensor, although to date this has not been done. It is also possible that the metric tensor might be computable from the other operators, but this has not been proven one way or another.

implemented by this set of space-time universal operators can be finitely approximated by a field calculus implementation of the Proto implementation of said computation, extended with implementation of g in field calculus.

The first is trivial: measurement of the metric tensor g is an operation that can potentially be implemented via local observation from a device, so it can be included in the set of built-in operators `b` in field calculus, just like `nbr-range`.

For the second, let us consider the Proto implementation of each of the other operators. The n_v, operator is equivalent to Proto's `nbr` operator, which is already explicitly covered in Theorem 2. The n_d, and n_m operators are equivalent to Proto's `nbr-vec` and `min-hood` operators, respectively: these are finitely-approximable operations, and implemented by built-in operators in field calculus. The n_r operation is implemented using a construct based on discrete-value equality tests, and is also finitely approximable. Finally, we can choose the operations of a Turing machine as a finitely approximable Turing-universal set of operators (though of course in practice one would generally use much more compact implementations based on the rich set of operators allowed in `b`).

This leaves only the question of neighborhood independence. With regards to the operators, P and g are neighborhood independent because they are specified pointwise. The remainder (n_d, n_v, n_r, and n_m) interact with an unspecified neighborhood. Thus, given the universality of the operators, for any choice of discretization there is an equivalent computation that can be implemented using these operators. Thus we satisfy all of the criteria of Theorem 2, which, combined with the implementation of g as a built-in operator in field calculus, implies that field calculus is space-time universal.

$\square$