# A Self-Replication Algorithm to Flexibly Match Execution Traces

Paul Leger    Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile - Santiago, Chile
{pleger,etanter}@dcc.uchile.cl

## Abstract

Stateful aspects react to the history of a computation. Stateful aspect developers define program execution patterns of interest to which aspects react. Various stateful aspect languages have been proposed, each with non-customizable semantics for matching a join point trace. For instance, most languages allow multiple matches of a sequence when the associated context information is different. Obtaining a different matching semantics requires ad hoc modifications of the aspects, if at all possible. In order to allow flexible customization of the matching semantics of a given aspect, this paper presents a self-replication algorithm called MatcherCells. Through the composition of simple reaction rules, MatcherCells makes it possible to express a wide range of matching semantics, per aspect. In addition, we present an initial implementation of our proposal.

***Categories and Subject Descriptors*** D.3.3 [*Software*]: Programming Languages

***General Terms*** Algorithms, Design

***Keywords*** Self-replication algorithms, execution traces, stateful aspects.

## 1. Introduction

Aspect-Oriented Programming (AOP) [5] allows developers to use a set of language features to modularize crosscutting concerns. Aspects can only react to the current execution event, called *join point*. The standard *matching process* of an aspect just consists in the evaluation of a *pointcut* with a join point. If the pointcut matches the join point, the associated *advice* is executed [8].

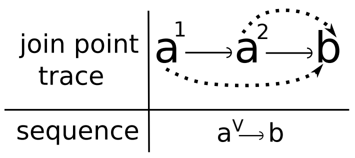

**Figure 1.** Possible matches of a sequence.

In most languages, pointcuts can also refer to the execution context as represented by the call stack. However, some crosscutting concerns cannot be directly expressed through the reaction to the current join point, even considering stack inspection, *e.g.* error detections [7]. Stateful aspects [2] can react to join point *traces*, *i.e.* to the whole history of a computation. The matching process of a stateful aspect consists in the evaluation of a *sequence* (instead of a pointcut) with a join point trace. A sequence can match and bind free variables multiple times. For example, Figure 1 shows two possible matches of the sequence $a^v \to b$ in a join point trace. Apart from the definition of the trace "$a$ followed by $b$", this sequence specifies that the free variable $v$ is bound when an $a$ join point is matched. Depending on the semantics of the matching process, the sequence can match either once or twice, where the first match binds $v$ to 1, and the second binds $v$ to 2.

In existing stateful aspect languages [1, 3, 4, 7, 11], stateful aspects have the same fixed semantics for the matching process. For example, EventJava [3], HALO [4], and tracematches [1] only support multiple matches of a sequence if the values bound are different. Therefore, if the desired matching process of a particular stateful aspect does not fit the default, the developer ends up overburdening the definitions of the sequence and/or advice.

To illustrate this, let us consider an *autosave* feature of a text editor application; this feature saves the document every three editions. The stateful aspect that implements this feature needs to only match once every three editions. Using tracematches [1], we could define the aspect as follows:

```
tracematch() {
  sym edit after: call (* Editor.edit());
  edit edit edit {
    Editor.save();
  }
}
```

However this definition is not correct because tracematches performs multiple matches. As a result, once three edits happen, each subsequent edit triggers a save. The programmer has to tweak the aspect definition to artificially introduce another symbol, `save`, which is then excluded from the regular expression (tracematches require contiguous occurrences of the events denoted by the symbols in the regular expression):

```
tracematch() {
  sym edit after: call (* Editor.edit());
  sym save after: call (* Editor.save());
  edit edit edit {
    Editor.save();
  }
```

This showcases the fact that the default matching semantics of a stateful aspect language is not adequate in all cases. This motivates the need to open up the language in order to customize its semantics when needed.

In this paper, we propose that the use of a self-replication algorithm [9], named MatcherCells, allows developers to flexibly express the semantics of a matching process. The following two sections present MatcherCells and an initial implementation, and Section 4 concludes.

## 2. MatcherCells

Through small entities with simple rules, self-replication algorithms [9] allow developers to flexibly express the semantics of a process. This is because each rule defines a portion of the semantics of a process and the combination of them defines the full semantics. In this section, we define a self-replication algorithm, named MatcherCells, to match join point traces and express different semantics of the matching process of a stateful aspect.

### 2.1 Self-Replication Algorithms

Self-replication algorithms are inspired by the cellular behavior. Concretely, these algorithms emulate the reactions of a set of biological *cells* into a *solution* to a trace of *reagents*. Figure 2 shows the different possible reactions of a cell to a reagent. The reaction of a cell to a reagent can be *a)* the creation of an identical copy of itself with a small variation in order to persist in the solution, *b)* nothing, *c)* death, *d)* or some of these combinations. An algorithm that follows self-replicating behavior is defined by a pair $(C_0, R)$, where $C_0$ is the set of first cells into a solution (*a.k.a.* seeds) and $R$ is the set of rules that govern the evolution of the solution.

### 2.2 Using MatcherCells to Match Traces

In MatcherCells, a cell contains the sequence that should be matched, the free variables bound during this matching,

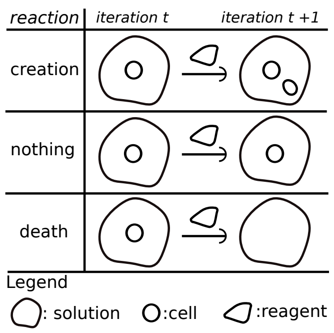

**Figure 2.** Different reactions of a cell to a reagent.

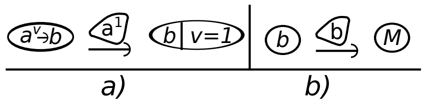

**Figure 3.** *a)* The cell creates a cell that expects to match the next join point and keeps the bindings. *b)* When a cell matches the last join point specified, the cell creates a match cell.

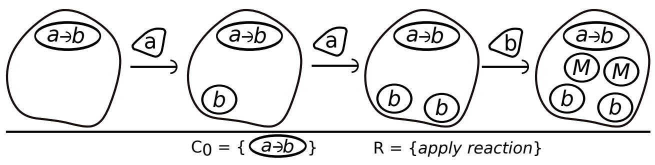

**Figure 4.** Using MatcherCells to match multiple times.

and a reference to its creator. Cells react to join points, which corresponds to reagents. As Figure 3a) shows, if a cell matches a join point, it creates a new cell that expects to match the next join point specified by the sequence. In addition, this new cell contains the possible values bound when the join point was matched. These bound values are stored in an environment, which can be accessed by the sequence. As Figure 3b) shows, when there is no next join point to expect, a *match cell* is created to indicate a match of the join point trace.

A stateful aspect in MatcherCells is defined by a sequence, an advice, and an optional set of rules $(R)$[1]. Each rule defines a portion of the semantics of the matching process and the set of rules defines entirely the semantics. Figure 4 shows a stateful aspect in MatcherCells that matches twice. The solution begins with a set of only one seed $(C_0)$, which contains the specified sequence[2], and evolves according to the set of rules defined for this stateful aspect. For example, Figure 4 shows that the "apply reaction" rule only applies the reaction of each cell.

[1] If $R$ is not defined, a stateful aspect uses a default $R$.

[2] If the stateful aspect language supports the definition of a stateful aspect with multiples pairs sequence-advice, the solution begins with more seeds.

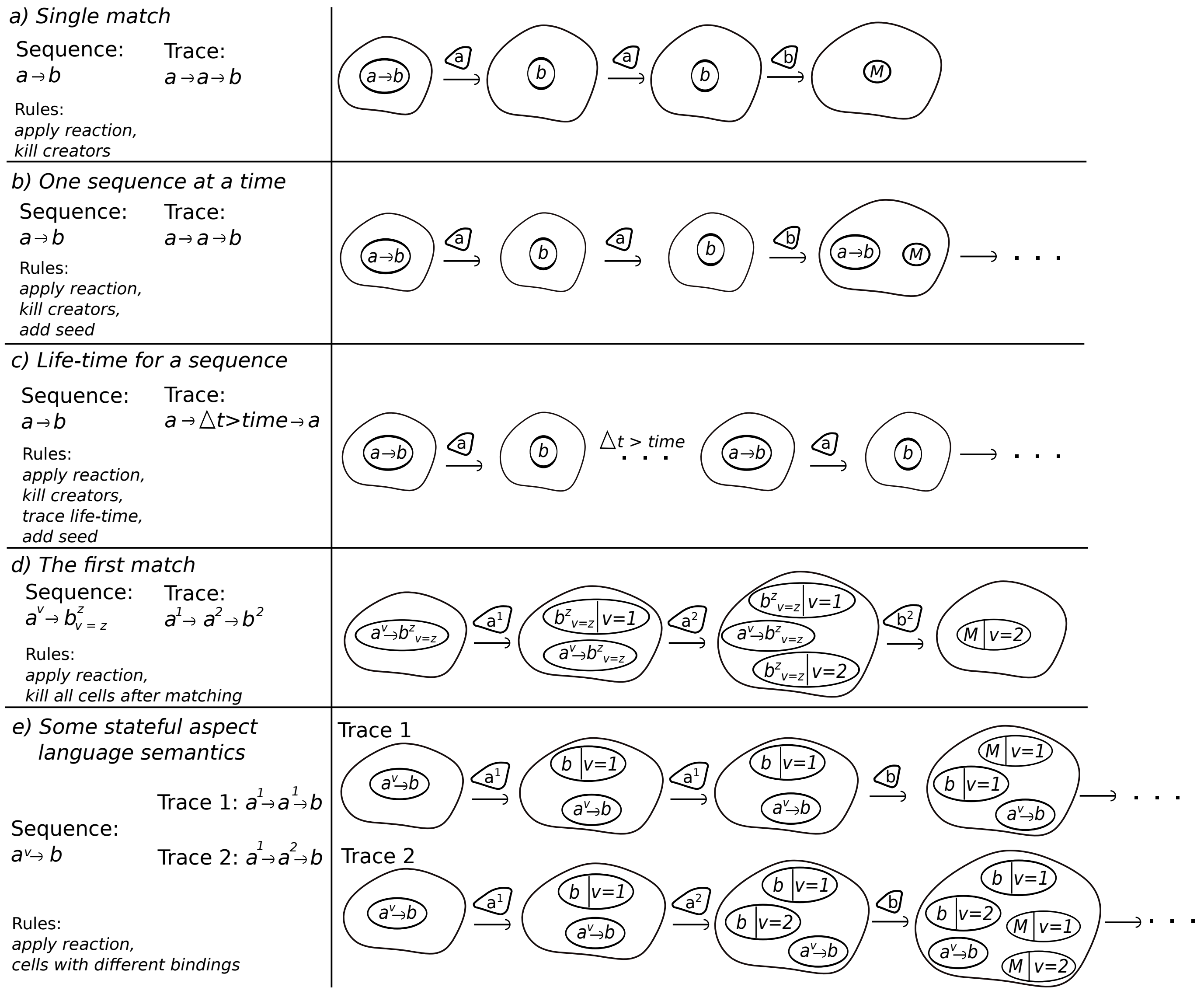
a) Single match
Sequence: a→b
Trace: a→a→b
Rules: apply reaction, kill creators
b) One sequence at a time
Sequence: a→b
Trace: a→a→b
Rules: apply reaction, kill creators, add seed
c) Life-time for a sequence
Sequence: a→b
Trace: a→△t>time→a
Rules: apply reaction, kill creators, trace life-time, add seed
△t > time
d) The first match
Sequence: a^v→b^z_{v=z}
Trace: a^1→a^2→b^2
Rules: apply reaction, kill all cells after matching
e) Some stateful aspect language semantics
Trace 1: a^1→a^1→b
Sequence: a^v→b
Trace 2: a^1→a^2→b
Rules: apply reaction, cells with different bindings
Trace 1
Trace 2
M
v=1
v=2

**Figure 5.** Different semantics for the matching process of a stateful aspect.

## 2.3 Using MatcherCells to Express Matching Process Semantics

Figure 5 shows that using simple rules, it is possible to achieve different semantics of a matching process:

- Figure 5a) The "kill creators" rule kills the cells that create a new cell. Adding this rule, a stateful aspect cannot match multiple times anymore.
- Figure 5b) The "add seed" rule adds a seed if there are no cells or only match cells in the solution. This rule allows a stateful aspect to match again.
- Figure 5c) The "trace life-time" rule kills all cells whose time of the join point trace that are matching has exceeded a specific period. Adding this rule, developers can define stateful aspects that only match join points traces that occurs at a period of time.
- Figure 5d) The "kill all cells after matching" rule kills all cells when a match cell is found. This rule allows a stateful aspect to have various possible matches until the first join point trace is matched. In this figure, the match cell contains the value 1, which was bound when the $a^1$ join point was matched[3].
- Figure 5e) The "cells with different bindings" rule only allows cells to create new cells with different values bound. For example, the reaction to the second $a^1$ join point in the trace 1 does not create a new cell. Instead, the reaction to the $a^2$ join point in the trace 2 creates a new cell. This rule emulates the semantics of HALO [4],

[3] Like in pointcuts, the definition of a sequence also involves its constrains to match. In favor of flexibility to define sequences, we assume that these constrains are explicitly developed by a programmer (*e.g.* $v = z$ and $v > z$).

EventJava [3], and tracematches [1]: a sequence can match multiple times if values bound are different.

## 3. An Implementation of MatcherCells

In MatcherCells, rules are functions that can be composed in order to define the semantics of the matching process of a stateful aspect. This section presents an initial implementation of our proposal.

### 3.1 Cell Reaction

$$\text{react}: \text{Cell} \times \text{JP} \rightarrow \text{Cell}$$

The react function carries out the reaction of a cell to a join point. If the cell matches the join point, the function returns a new cell, otherwise the function returns the same cell. Our proposal does not impose restrictions about how a cell matches a join point. Therefore, the implementation of react only depends on the expressiveness of the stateful aspect language to define sequences. MatcherCells only requires that this function follows this semantics.

### 3.2 Rules

$$\text{rule}: \text{List<Cell>} \times \text{JP} \rightarrow \text{List<Cell>}$$

A rule is a function that takes as parameters a list of cells and a join point, and returns the list of cells of the next iteration. For example, the implementation of the "apply reaction" rule is:

```
var applyReaction = function(cells,jp) {
    return removeDuplicates(
            append(cells,map(cells,react,jp)));
};
```

The applyReaction function returns the cells and their reactions. A cell, whose reaction is itself, is in the list of cells and the list of reactions, meaning that this cell is duplicated when both lists are joined. To prevent this duplication, the removeDuplicates function is used. Using rule designators (*i.e.* functions that return rules), developers are able to create rules that can be composed:

```
var killCreators = function(rule) {
  return function(cells,jp) {
    var newCells = rule(cells,jp);
    return difference(newCells,getCreators(newCells,cells));
} };


var addSeed = function(sequence) {
  return function(rule) {
    return function(cells,jp) {
      var newCells = rule(cells,jp);
      return length(newCells) == 0||onlyMatchCells(newCells)?
        append(newCells,[createSeed(sequence)]): newCells;
} } };
```

```
var traceLifeTime = function(delta) {
  return function(rule) {
    return function(cells,jp) {
      var newCells = rule(cells,jp);
        return filter(function(cell) {
          return currentTime() − cell.time <= delta;
        },newCells);
} } };


var killAllCellsAfterMatching = function(rule) {
  return function(cells,jp) {
    var newCells = rule(cells,jp);
    return isAMatchCell(newCells)? []: newCells;
} };


var cellsWithDifferentBindings = function(rule) {
  return function(cells,jp) {
    var newCells = rule(cells,jp);
    return filter(function(cell) {
       return hasDifferentBindings(cell,newCells);
    },newCells);
} };
```

These rule designators are parametrized by a rule, which corresponds to the rule that should be applied before the current one. The rule returned by killCreators first applies a previous rule (*e.g.* applyReaction) to obtain a list of cells, where the cells that created new cells are removed for the next iteration. The addSeed[4] rule designator returns a rule that adds a seed if there are no cells or only match cells. Some rules can need that cells contain additional information. For example, traceLifeTime needs cells that contain the time when the matching of the trace begins (cell.time). To add information to cells, the MatcherCells framework supports a custom cell creation functions that can be used to annotate newly-created cells; this custom cell creation is a function creation: Cell $\rightarrow$ Cell. To annotate a cell with the trace time, one needs to provide the following function:

```
var creation = function(cell){
  if (!isSeed(cell))
    cell.time = isSeed(cell.parent)? currentTime(): cell.parent.time;
  return cell;
};
```

The piece of code above adds the current time when the matching of a join point trace is beginning. The rule returned by killAllCellsAfterMatching returns an empty list of cells if it finds a match cell in the list, otherwise returns the same list. The last rule designator returns a rule that only keeps the cells with different bindings or expect to match a different join point.

Developers can compose these functions to obtain one rule that represents the composition. For example, the rule that only permits one match of a sequence at a time (Figure 5b) is:

```
var oneAtATime = addSeed(sequence)(killCreators(applyReaction));
```

[4] Notice that addSeed is in fact a higher-order rule designator, parameterized by the original sequence.

The order of the composition determines when a rule is applied. For example, the piece of code above first applies the applyReaction rule and then killCreators and addSeed.

### 3.3 Stateful Aspect Deployment

A stateful aspect is defined by a sequence, an advice, and a composition of rules. For example, suppose that developers need to add the *autosave* feature to an application of a text editor. This feature automatically saves the document when it is edited three times. The stateful aspect that implements this feature is:

```
var threeEditions = seq(edit,edit,edit);

var autosave = {
  sequence: threeEditions,
  advice: function (matchCell) {
    //getting the editor from bindings of the match cell
    var editor = matchCell.bindings.editor;
    editor.save();
  },
  rule: addSeed(threeEditions)(killCreators(applyReaction));
};
//deploy of the stateful aspect
deploy(autosave);
```

The autosave stateful aspect has to only match each three editions; thereby, this stateful aspect uses a rule that avoids multiple matches (KillCreators) and permits to match again a join point trace (addSeed). Although we have provided a piece of code to define a sequence and an advice, theses definitions depend on the expressiveness of the stateful aspect language. In MatcherCells, if the rule property is not defined, a default composition of rules is used. To define a stateful aspect that only matches once (Figure 5d), the rule property must simply be changed:

```
var statefulAspect = {
  //sequence and advice definitions
  rule: killAllCellsAfterMatching(applyReaction);
};
```

### 3.4 Weaving of a Stateful Aspect

The weaving of a stateful aspect consists in evolving its list of cells and executing the advice with each match cell found:

```
var StatefulAspectWeaving = function(sAsp,jp) {
  var tempCells = sAsp.rule(sAsp.cells,jp);
  var matchCells = filter(isAMatchCell,tempCells);
  sAsp.cells = filter(isANotMatchCell,tempCells);
  if (length(matchCells) > 0)
    //execute advice with bindings of each match cell
  else
    //execute the join point proceed
}
```

The evolution of the list of cells is determined by a rule, which is the composition of rules defined for a stateful aspect. If match cells are found after the rule is applied, these cells are removed from the list and the advice is executed for every one of them.

## 4. Conclusion

Languages that support stateful aspects all adopt fixed semantics with respect to how execution traces are matched. We have shown that such a fixed semantics is necessarily not adequate in some cases, for which developers have to overburden their aspect definitions, if at all possible. We have proposed a self-replication algorithm in which the composition of simple rules makes it possible to express a wide range of trace matching semantics.

We plan to work on two lines: performance and practical implementation. Performance is a recurrent subject in stateful aspect languages [1, 4, 11]. We think a static analysis can allow us to create rules at compile time that prevent the creation of cells that will not match a join point trace, therefore, less cells are evaluated at each join point. Implementing a practical stateful aspect language that uses MatcherCells will allow us to compare more exhaustively the approach with the matching processes of existing stateful aspect languages. Hence, we plan to use MatcherCells to implement OTM [6], a model of an open stateful aspect language, for the JavaScript language.

***Availability.*** A proof of concept of our proposal is available on http://pleiad.cl/otm/matchercells.

## References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In OOPSLA 2005 [10], pages 345–364. ACM SIGPLAN Notices, 40(11).

[2] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[3] P. Eugster and K. Jayaram. EventJava: An extension of java for event correlation. In S. Drossopoulou, editor, *Proceedings of the 23rd European Conference on Object-oriented Programming (ECOOP 2009)*, number 5653 in Lecture Notes in Computer Science, Genova, Italy, july 2009. Springer-Verlag.

[4] C. Herzeel, K. Gybels, and P. Costanza. A temporal logic language for context awareness in pointcuts. In D. Thomas, editor, *Workshop on Revival of Dynamic Languages*, number 4067 in Lecture Notes in Computer Science, Nantes, France, July 2006. Springer-Verlag.

[5] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al., 1996.

[6] P. Leger and É. Tanter. An open trace-based mechanism. In J. Aldrich and R. Massa, editors, *Proceedings of the 14th Brazilian Symposium on Programming Languages (SBLP 2010)*, Salvador - Bahia, Brazil, Sept. 2010.

[7] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In OOPSLA 2005 [10], pages 365–383. ACM SIGPLAN Notices, 40(11).

[8] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[9] J. V. Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[10] *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, San Diego, California, USA, Oct. 2005. ACM Press. ACM SIGPLAN Notices, 40(11).

[11] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In A. P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag, 2005.