On the role of temporal and spatial representations in light of the ETS formalism

by

Benjamin Reuben Peter-Paul

Bachelor of Computer Science, UNB, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

This thesis is accepted by the Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2008

Dedication

To my daughter, Leyla.

Abstract

The Evolving Transformation System (ETS) is *the first class(ification)-oriented representational formalism*. In ETS, all objects are viewed and represented as processes, where object representation is a temporal sequence of structured events called a *struct*. Compared to the conventional mathematical, i.e. "spatial", representations, it appears to be a more basic form of representation, which however, can be spatially instantiated.

The main objective of this thesis is to illustrate with implementation the process of spatially instantiating ETS structs. This involves the design and implementation of abstract data types, data structures and algorithms for the basic concepts of ETS. While the central concept of class generating system and its specification is implemented as a finite state machine. The actual structs used for the spatial instantiation process, are obtained by applying the above implementations to a system of interacting class generating systems called "Bubble Man". Finally, the structs generated by the "Bubble Man" class generating systems are spatially instantiated via a system of finite state transducers and a system of programmable "spheres" (or "bubbles"), such that given a struct each transducer produces a "program", which is then executed by a particular "bubble". I hope that this thesis will help one thinking about the nature of the ETS object representation.

Acknowledgements

Studying ETS has been a “roller coaster ride” of an experience. There have been many challenging moments, when I have tried to wrap my brain around some strange ideas, and there have been moments of pure excitement at the great potential offered by this unique language. I am grateful to Dr. Lev Goldfarb for the invaluable experience of getting to work with such a great thinker and an interesting and passionate individual. I would also like to extend my thanks to three ETS group members in particular: David Gay, Muhammad Al-Digeil and Ian Scrimger.

I am also grateful to Dr. Weichang Du whom has also made this experience as a grad student at UNB a positive one. My thanks are also extended to my committee members, Dr. Mike Fleming, Dr. Bruce Spencer and Dr. Yevgen Biletskiy. Thank you for your time reading my thesis and helping me improve it.

Most importantly, I would like acknowledge my beautiful wife, Isis, for her patience, love and support as I have spent many hours away in ETS group meetings, writing, programming, reading and thinking. I could not have done this without you. Finally, I would like to thank my daughter Leyla, your birth has provided me with a great source of motivation to “step on it”!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

That representation makes a difference is evident for a different reason. All mathematics exhibits in its conclusions only what is already implicit in its premises . . . Hence all mathematical derivation can be viewed simply as change in representation, making evident what was previously true but obscure.

This view can be extended to all of problem solving—solving a problem simply means representing it so as to make the solution transparent. If the problem solving could actually be organized in these terms, the issue of representation would indeed become central.

H. A. Simon, [1]

We are unconsciously conditioned by *all* conventional formalisms to expect to see spatial information somehow present in our various forms of representation.[1] While working with the ETS group on developing the "Bubble Man", an application illustrating the main concepts of the radical new formalism named the Evolving Transformation System, I realized I was missing something important conceptually. I realized that I was focused on incorporating an objects structural features using the ETS formalism, rather than focusing on identifying temporal features, in the form of structured temporal events.

[1]As is the case in mathematics.

This realization led to the following question: “What is the role of temporal and spatial representation in light of the ETS formalism?”

1.1 Origin of ETS

ETS is an entirely new approach to the concept of representation. The original motivation for “temporal (structural) representation” originated in an attempt to generalize the “temporal” process of constructing the set, $\mathbb{N}$, of natural numbers:

> One can gain an initial intuitive understanding of the proposed representation by generalizing the temporal process of the (Peano) construction of natural numbers: replace the single structureless unit out of which a number is built by multiple structural ones. An immediate and important consequence of the distinguishability (or multiplicity) of units in the construction process is that we can now see which unit was attached and when. Hence, the resulting (object) representation for the first time embodies temporal structural information in the form of a formative, or generative, object “history” recorded as a series of (structured) events. Each such event stands for a “standard” interaction of several objects/processes. [2, p. 1]

In figure 1.1 (left), a structured event denoted, $\boldsymbol{\pi}_1$, represents the restructuring of a single object/process. To illustrate further, one might think of the object as being an infinitely long piece of rope (without any knots or tangles in it), we then might say that this straight piece of rope represents the number 0. The Peano “successor operation” may be realized by tieing a knot in the rope and the observation of a knot appearing on the rope is an event and represented by the primitive $\boldsymbol{\pi}_1$. In Fig. 1.1 (right), the first appearance of a knot, is represented with a primitive denoted

π_{1a}, the resulting representation represents the natural number 1. The appearance of a second knot on the rope is represented by a primitive denoted π_{1b}, the resulting representation represents the natural number 2. Likewise for the construction of the number 3. The representations, shown in Fig. 1.1, record sequences of knots appearing on a rope, such that the entire construction process of the numbers 1, 2 and 3 are recorded as a temporal sequence of events. These sequences of events store not only the information that several knots were tied on a rope, but when they were tied (relatively).

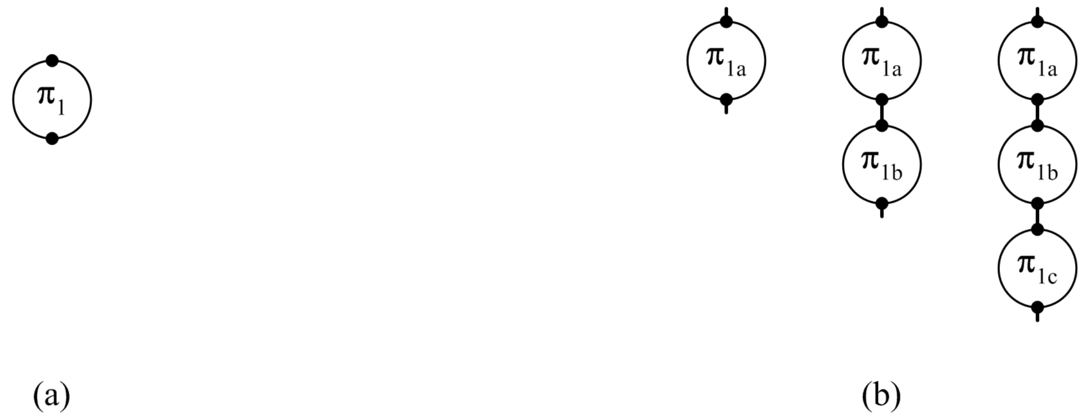


Figure 1.1: (a) The single primitive involved in the ETS representation of natural numbers. (b) Structs representing the numbers 1, 2, and 3.

Of course, the development of the Evolving Transformation System has been further motivated by the need for classification. Since it appears that structural representation lends itself to the inductive recovery of a generative schema for producing a family of similar structural representations. Such that, given a small "training set" of structural representations, the entire class of these structural representations may be learned inductively.

1.2 ETS concept of class in a nutshell

ETS is an event-based class(ification)-oriented formal language for representing reality as a hierarchical system of evolving classes of structural processes. These structural processes are represented in ETS as temporal sequences of structured events called *structs*, and the structured events are referred to as *primitive transformations* (*cf.* Sec. 2.1.1 p. 9, and for an example of an ETS struct *cf.* Fig. 2.3 p. 15).

ETS views the concept of class as a collection of “structurally similar” processes, and is defined in ETS via the concept of *class generating system.* A class generating system, in ETS, is a generative mechanism used to output all the structural entities belonging to a particular class. More specifically, class generating systems *assemble* class elements according to a given step-wise specification. Such that, given an initial piece of structure (contributed by the environment) and a step-wise specification, for each step in the specification a “structurally formulated extension”[2] is assembled to the “working struct”. The result is a struct representing a particular class element (e.g. cf. Fig.’s 2.8 p. 25).

To help further clarify the ideas of class and class element generation, the following illustrative analogy is offered. Assume that there exists a special abstract machine whose internal memory stores a struct. To borrow from the theory of finite state automata, this special machine can be used to accept/generate a set of structs. To generate a struct, at each state, the machine nondeterministically selects a “structurally formulated extension” from a subset of possible extensions and assembles it to the class element (“working struct”) under construction.

This machine describes an interesting relationship among the outputs (structs) that is analogous to the one described by finite state machines for regular languages, where

[2]A struct satisfying some “structural formula” (structural constraint) selected by the generating system.

the struct-generating machine may be thought of as defining a “language” such that each struct generated is a “word” in that “language”. However, in ETS, we say that such a machine defines a “class” and each struct produced is an “object/instance” of that class. ETS suggests, that a strong relationship exists between class elements (the outputs of the above machines), which referred to as “structural similarity” [3], such that two structs are structurally similar if (and only if) they are outputs of the same class generating system.

1.3 Motivation

This thesis is an attempt to elucidate the concept of ETS temporal representation compared to more familiar forms of representation, e.g. spatial representation. It appears that ETS structural representation are more fundamental, or basic, in nature compared to spatial representations, e.g. pictures and computer graphics, based on the idea that ETS structs record less information by recording interactions between objects in a system, as structured events called “primitive transformations”. Moreover, it appears that “more *is* less”, and that ETS structs encode sufficient information to reconstruct the original system of objects, i.e. ETS structs may be “spatially instantiated”.

In [2, pp. 22-23], the conventional view of reality (maintained in science) is portrayed as being “object-centred”, whereas the ETS model suggests a process view of reality and emphasizes the “transformative” event, rather then on the objects themselves. The object-centered view of reality forms what is referred to as an object environment, the process-centered view forms what is referred to as event environment, “by admitting the above two environments and creating an interface between them: the ETS model operates with **ideal events** that correspond to **real events** in the object environment. A real event is accounted for [...] by its idealized version[...], while in the object environment a real event is accounted for by the realization of an ideal event."

In my attempts to study the role of temporal representation and its relation to conventional spatial representations, a software system was developed (the implementations of which have been generalized and are discussed in Ch. 4), the software system contains a subsystem that implements the "event environment" as a system of several running and interacting class generating system *processes*[3]. The software system also contains two additional subsystems, one that implements the process of spatial instantiation of ETS structs as a system of *finite state transducers.* The other subsystem implements the object environment as a system of 3D models and involves the use of "Microsoft's XNA content pipeline" for rendering and manipulating 3D models. These two subsystems are tightly coupled such that a *finite state transducer* is implemented and associated with each 3D model.

1.4 Scope

In order to apply the software system, an ETS class description was needed, and for simplicity the illustrative example found in [2, Sec. 8] referred to as the class of "2D Bubble Men" was selected and extended. The class of "2D bubble men" is a three-level class, such that it is recursively defined via several two-level classes, which are then recursively defined by several single-level classes. Discussion of multi-level classes and their associated concepts (e.g. multi-level structs, structural constraints and generating systems) is left out, since, given the simplistic nature of the example, the organizational roles played by the higher-level classes are limited and may be implemented with nested queues. It was, therefore, unnecessary to implement ADTs

[3]The term process is used to emphasize the idea that a single class generating system may be specified, and have several processes/instantiations generating structs simultaneously.

and data structures for the multi-level features of the ETS formalism, and so, for the purpose of this thesis work, the “Bubble Man” class is viewed as a system of single-level classes rather than as a *single* three-level class. Although, in Ch. 3, no attempt is made to portray the example in any other way than as three-levels of class representations. Moreover, this thesis will limit discussion of temporal and spatial representation to single-level structs and associated concepts. With regard to the process of spatial instantiation, only spatial instantiation of single-level structs are considered.

1.5 Organization of this thesis

First I would like to remind the reader that the Evolving Transformation System is a radically new “structural” representational formalism, and moreover, it is considered to be the first language to provide a formal definition of the concept of class. With this in mind, the ideas, concepts and formal definitions I rely upon are not only novel but several are recent additions to the ETS formalism.

I cannot hope to provide the reader with a comprehensive understanding of ETS and the many implications associated with it. Instead I hope to guide the reader in the right direction. To this end, I have organized the thesis as follows:

- In Chapter 2, an overview of the key features and concepts of ETS, relied upon for application and implementation, are semi-formally presented here. Note that many of the figures, in this chapter have been adapted from [2].
- Discussed in Chapter 3, the Bubble Man, which is also the main illustrative example in [2], and initial motivation for this work, is the application chosen for simulating the systematic class element generation and spatial instantiation of ETS structural representations. Also, given that the Bubble Man was initially

developed for illustrating ETS concepts in *Idem.*, many of the figures in this chapter have been adapted from *Idem.*

- The abstract data types and data structures developed for the implementation of level 0 object and class representation are presented in Chapter 4. Along with the implementations of the ETS class generating system and the interface, between event and object environments, used to spatially instantiate ETS structs.

- Finally, in Chapter 5, a summary, results of this study, and suggestions for future work are presented.

Chapter 2

On the Evolving Transformation System

Thus nature is a structure of evolving processes. The reality is the process. It is nonsense to ask if the colour red is real. The colour red is ingredient in the process of realisation. The realities of nature are . . . the events in nature.

A. N. Whitehead, [4]

2.1 Object representation

2.1.1 Structured events

The evolving transformation system formalism addresses a universal view of nature, whereby objects are understood to be structural processes, which may be observed over time—alternatively, their evolution may be reasoned about—and represented as a temporal sequence of "structured events". These structured events signify an interruption in the flow of predetermined primal processes.

When defining a set of primitive transformations one must first decide at which stage of representation one will begin. This stage is called **stage 0**, or the **initial stage**, and its critical feature is the specification of a set of "primal disjoint" classes of processes,

$$\mathbb{C} = \{C_1, C_2, \ldots, C_m\} \ .$$

Where each element of such a class, $c_i \in C_i$, is a structural process whose structure is suppressed. This effectively makes each element, of such a class, unstructured and indistinguishable from another. For this reason, when describing the concept of primitive, in the following section, the structural processes involved in a particular interaction/event will be referred to as **primal classes** (de-emphasizing the notion of structure). However, it is still helpful to remember that each such element *is a structural representation of an observed process*, and it has, time and again, proven helpful to reason about this structure when defining a set of primitives.

2.1.1.1 Concrete and abstract primitives

A basic concept and first definition of ETS [2, Def. 1], is that of a **primitive transformation**, or **primitive**, and it is the basic building unit of an ETS object representation. Basically, a primitive is defined in terms of the set of primal classes they transform,

$$\mathrm{Init}(\widehat{\pi_i})\langle C_{j_1}, C_{j_2}, \ldots, C_{j_{p(i)}}\rangle, \quad p(i) > 0,$$

called **initials**, and the resulting set of primal classes,

$$\mathrm{Term}(\widehat{\pi_i})\langle C_{k_1}, C_{k_2}, \ldots, C_{k_{q(i)}}\rangle, \quad q(i) > 0,$$

called **terminals**. Other properties of a primitive include a name, $\widehat{\pi_i}$, and a set of labels,

$$\mathcal{L}_i, \mathcal{L}_i \quad \subseteq C_{j_1} \times C_{j_2} \times \ldots \times C_{j_{p(i)}}$$

associated with $\widehat{\pi_i}$.

As a consequence of suppressing the structure of the initial and terminal primal classes, there are two kinds of primitives: abstract and concrete. A **concrete primitive**, π_{ia}, has a label, a, from the above mentioned set of labels, $a \in \mathcal{L}_i$, such that, a is a tuple consisting of concrete class elements,

$$a = \langle c_1, c_2, \ldots, c_{p(i)} \rangle$$

one from each initial primal class, i.e. $c_1 \in C_{j_1}$. More formally a concrete primitive π_{ia} is defined as a tuple

$$\pi_{ia} = \langle \widehat{\pi_i},\ \mathrm{Init}(\widehat{\pi_i}),\ \mathrm{Term}(\widehat{\pi_i}),\ a \rangle.$$

See Fig. 2.1, notice that the initial primal classes are marked as various shapes on the top while the terminal primal classes are shown on the bottom. Unfortunately, the corresponding shape does not distinguish between the class, in an abstract primitive, and its element, in the concrete primitive. (The only processes identified are the initials of $\pi_{2b} \ : \ b \ = \ \langle c_i^1, c_j^2, c_k^3 \rangle$, where c_t^s is the t^{th} process from the primal class C_s.)

An **abstract primitive**, e.g. $\boldsymbol{\pi}_i$, then, as its name suggests, is representative of a family of concrete primitives (see Fig. 2.1):

$$\boldsymbol{\pi}_i = \{\boldsymbol{\pi}_i(a) \mid a \in \mathcal{L}_i\}.$$

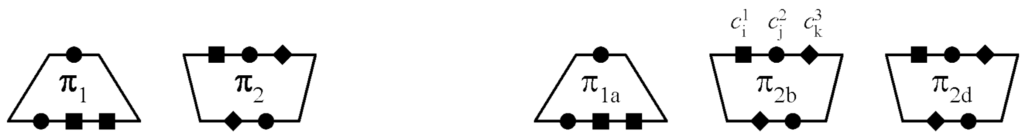


Figure 2.1: Pictorial illustration of two abstract primitives (*left*) and three concrete primitives (*right*).

2.1.1.2 Primitive-classes

Primitive classes are considered to be very useful generalizations of the concept of primitive transformation. It is understood that such generalization affords one the freedom to choose more appropriate primal classes without being concerned about an *explosion in the number of primitives.* Also, the concept of primitive-class is the most recent addition to the ETS formalism, and therefore, is not as fully developed as the other concepts such as the *abstract and concrete primitive transformations.*

Basically, among a set of predefined abstract primitives $\mathbf{\Pi}$, a family $\mathbf{\Pi}_*$ of *structurally similar* primitives:

$$\mathbf{\Pi}_* = \{\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_t\} \ ,$$

is considered to be a class of abstract primitives, referred to as a **primitive-class**, and denoted $[\boldsymbol{\pi}_*]$. Thus, $\mathbf{\Pi}_*$ may be referred to as the name of the class induced by the corresponding equivalence relation, $[\boldsymbol{\pi}_*]$, on $\mathbf{\Pi}$, see Fig. 3.2 and [2, Appendix].

2.1.2 Structs and their operations

2.1.2.1 Structs

The second basic concept of ETS is that of the **(level 0) struct**, see Fig. 2.3 and [2, Def. 3], which is understood to represent macro-events, or *pieces of formative history.* More formally structs consist of a set of primitives, Π_σ, and a relation **struct**

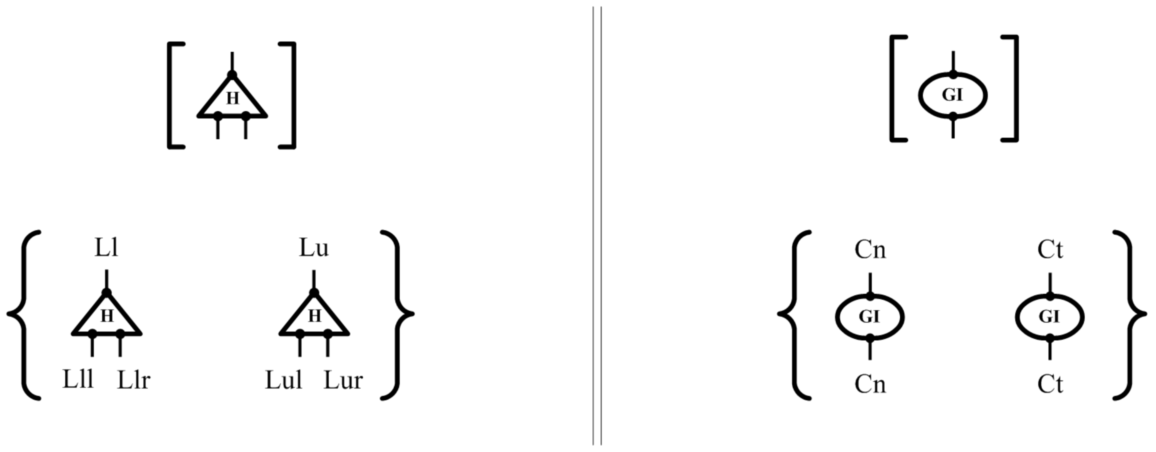


Figure 2.2: Primitive-classes and their elements.

link, SL_σ, which is a finite subset of the relation **class link between concrete primitives**[1] $\mathrm{CL}_{\mathbb{C},\Pi}$,

$$\mathrm{SL}_\sigma \subseteq \mathrm{CL}_{\mathbb{C},\Pi} \subseteq \Pi \times \mathbb{N}_{\mathrm{Term}} \times \Pi \times \mathbb{N}_{\mathrm{Init}}.$$

For example, given the following tuple:

$$\langle \pi_{ia}, u_i, \pi_{jb}, v_j \rangle,$$

if π_{ia}'s ${u_i}^{th}$ terminal primal class is equivalent to π_{jb}'s ${v_j}^{th}$ initial primal class, then the above tuple represents an observed concrete primal class element connecting concrete primitives π_{ia} and π_{jb}. A struct link then is a particular element from the set of possible class links between two concrete primitives.

In addition to the definitions of class link and struct link, there are two restrictions

[1]See [2, Def. 2] for the definition of relation $\mathrm{CL}_{\mathbb{C},\Pi}$.

constraining the structure of structs: first, the directed graph[2] of

$$\mathrm{ATTACH}_\sigma : \Pi_\sigma \times \Pi_\sigma \to \mathcal{P}(\mathrm{SL}_\sigma)^3$$

representing the projection of SL_σ onto $\Pi \times \Pi$ is connected and acyclic. Second, any terminal primal class can be connected to at most one initial primal class, and visa versa:

$$\forall \langle \pi_{ia}, u_i, \pi_{jb}, v_j \rangle,\ \langle \pi_{i'a'}, u_{i'}, \pi_{j'b'}, v_{j'} \rangle \in \mathrm{SL}_\sigma$$
$$\pi_{ia} = \pi_{i'a'},\ u_i = u_{i'} \Longleftrightarrow \pi_{jb} = \pi_{j'b'},\ v_j = v_{j'}\ .$$

For example, given a set of primitives Π_σ and a relation SL_σ, one may graphically represent the struct link relation using the primitives as nodes and class links as arcs. The tuple

$$\langle \Pi_\sigma, \mathrm{SL}_\sigma \rangle$$

is a struct, σ, if the resulting graph is acyclic and connected and no terminal primal classes are connected to more than one initial primal class and vice versa.

2.1.2.2 Substruct

Given two structs $\alpha = \langle \Pi_\alpha, \mathrm{SL}_\alpha \rangle$ and $\beta = \langle \Pi_\beta, \mathrm{SL}_\beta \rangle$, it is said that α is a substruct of β, denoted $\alpha \Subset \beta$, if

$$\Pi_\alpha \subseteq \Pi_\beta \quad \text{and} \quad \mathrm{SL}_\alpha \subseteq \mathrm{SL}_\beta\ ,$$

see [2, Def. 4].

[2]The vertices of the graph are the elements of Π_σ and the edges are defined by the ordered pairs, $\Pi_\sigma \times \Pi_\sigma$, of ATTACH_σ, see [2, Def. 3, p. 29]

[3]$\mathcal{P}(\mathrm{SL}_\sigma)$ represents a powerset of SL_σ.

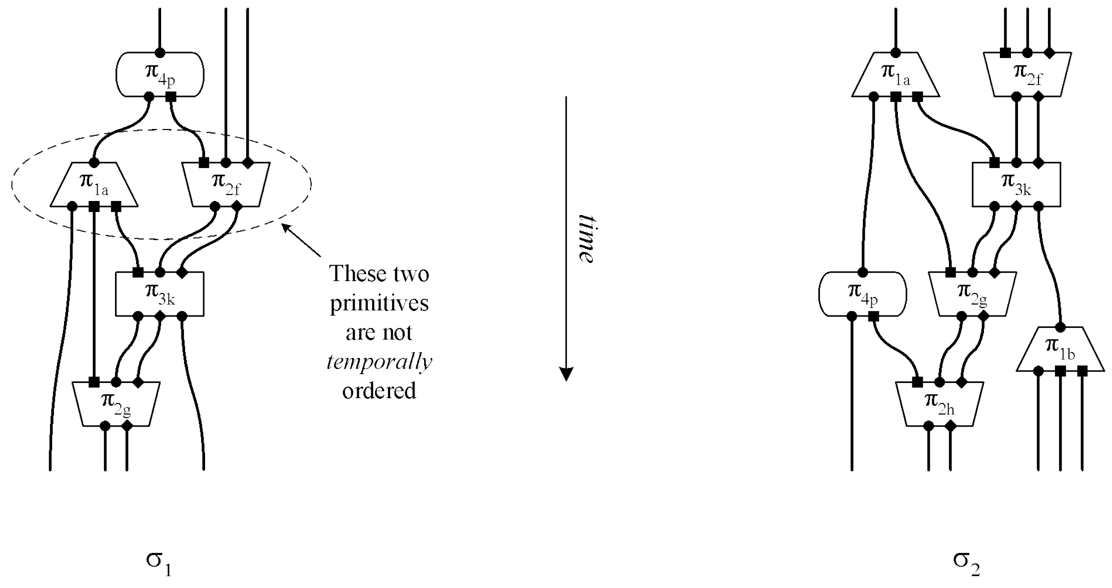


Figure 2.3: Two structs σ_1 and σ_2.

2.1.2.3 Struct assembly

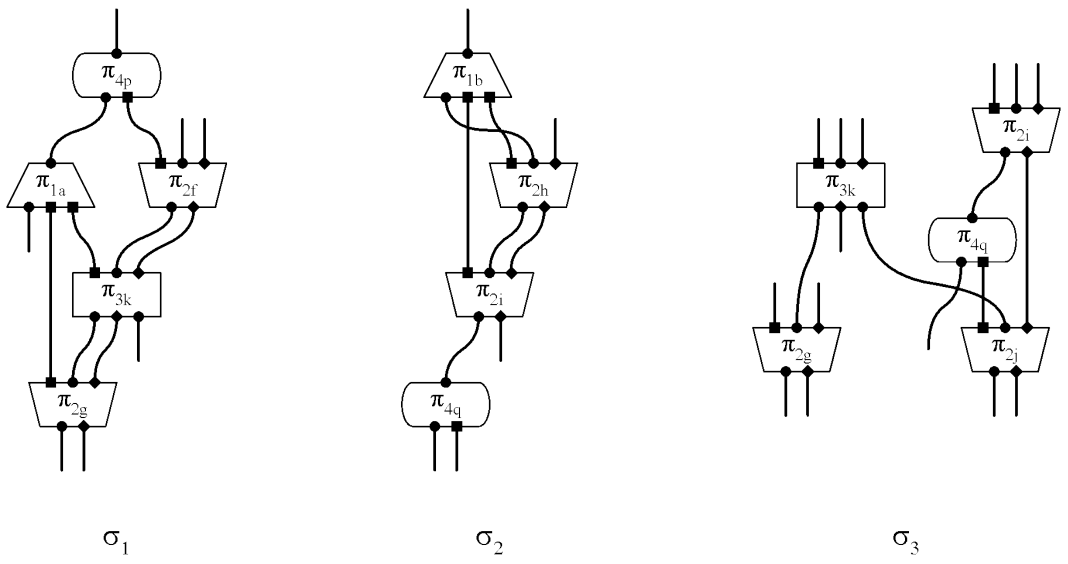
If structs overlap it is understood that the objects represented are, themselves, interacting. Overlapping level 0 structs, e.g. $\sigma_1, \sigma_2, \ldots, \sigma_r$ ($\sigma_i = \langle \Pi_{\sigma_i}, \mathrm{SL}_{\sigma_i} \rangle$), can then be assembled into a larger struct $\mathcal{A}(\sigma_1, \sigma_2, \ldots, \sigma_r)$, if the pair

$$\sigma = \langle \bigcup_{i=1}^{r} \Pi_{\sigma_i}, \bigcup_{i=1}^{r} \mathrm{SL}_{\sigma_i} \rangle$$

is a valid level 0 struct called the **assembly of level 0 structs** $\sigma_1, \sigma_2, \ldots, \sigma_r$, denoted as $\sigma = \mathcal{A}(\sigma_1, \sigma_2, \ldots, \sigma_r)$ or

$$\sigma = \sigma_1 \, \lhd \, \sigma_2 \, \lhd \, \ldots \, \lhd \, \sigma_r \ ,$$

see Fig. 2.4 and [2, Def. 8].



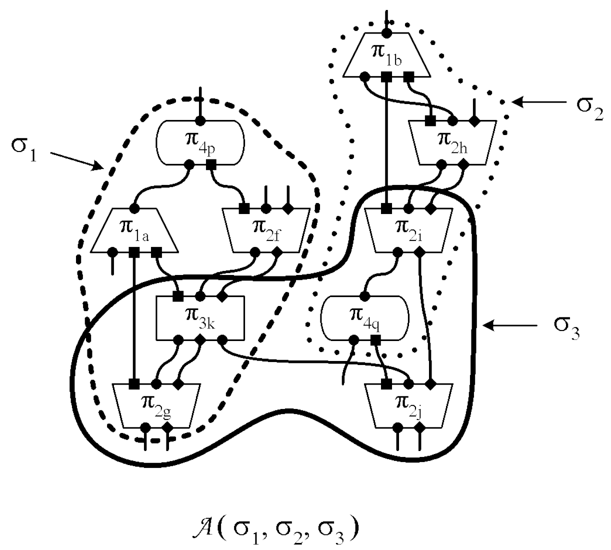


Figure 2.4: Three structs (top row) and their assembly.

2.2 Class representation

2.2.1 Specifying structure

2.2.1.1 Unit-constraints

Although the unit-constraint is a recent development in ETS it is now a central concept and may be thought of as an extension to the concept of a **class link between two primitives** [2, Def. 2]. In addition to describing class links between primitives—**unit-constraints** are used to constrain admissible structure to a family of structural units fixed between two **pivot primitives**. Such a mechanism affords the designer more flexibility by allowing one to specify how two primitives are to be connected without spelling out all such possible ways. Furthermore, it appears that the unit-constraint is a natural way to account for environmental influences (class noise).

Basically, to specify a unit-constraint between two primitives a designer simply states that a unit-constraint exists between a certain terminal primal class of a primitive, π_{ia}, and the initial primal class of another primitive, π_{jb}:

$$\mathrm{UCon}\langle\pi_{ia}, u_i\rangle\langle\pi_{jb}, v_j\rangle(\mathbf{\Pi}^*, \mathrm{FR})$$

Where $\mathbf{\Pi}^*$ is a subset of Π, called **noise primitives** and the set, FR, called **formation rules** for specifying various admissible sets Π^*'s, consisting of concrete primitives from $\mathbf{\Pi}^*$, which may be "absorbed" into the structure between the two *pivots*. See Fig. 2.5 for an illustrative example, and [2, Def. 9] for a formal definition.

In Fig. 2.5, a unit-constraint $\mathrm{UCon}\langle\pi_{1a}, 1\rangle\langle\pi_{2b}, 2\rangle(\mathbf{\Pi}^*, \mathrm{FR})$ is shown, along with several structs. Three structs satisfy the given unit-constraint, e.g. $(\sigma_1, \sigma_2, \sigma_3)$, and three do not, e.g. $(\sigma_4, \sigma_5, \sigma_6)$. In σ_5, π_{4q} is not a descendant of π_{1a} and in σ_6 , π_{3k} is not an ancestor of π_{2f} . The rules FR are not described in this figure, however on can think of a simple example: at most two occurrences of $\boldsymbol{\pi}_2$, at most five occurrences of $\boldsymbol{\pi}_3$, one occurrence of $\boldsymbol{\pi}_4$, and at most three occurrences of $\boldsymbol{\pi}_6$. Also notice, at the bottom-left of the figure is a pictorial representation of the unit-constraint with pivot primitives π_{1a} and π_{2f} , the light gray ellipse connecting them stands for a corresponding admissible substruct. On the right, non-pivot primitives are shown shaded in light gray.

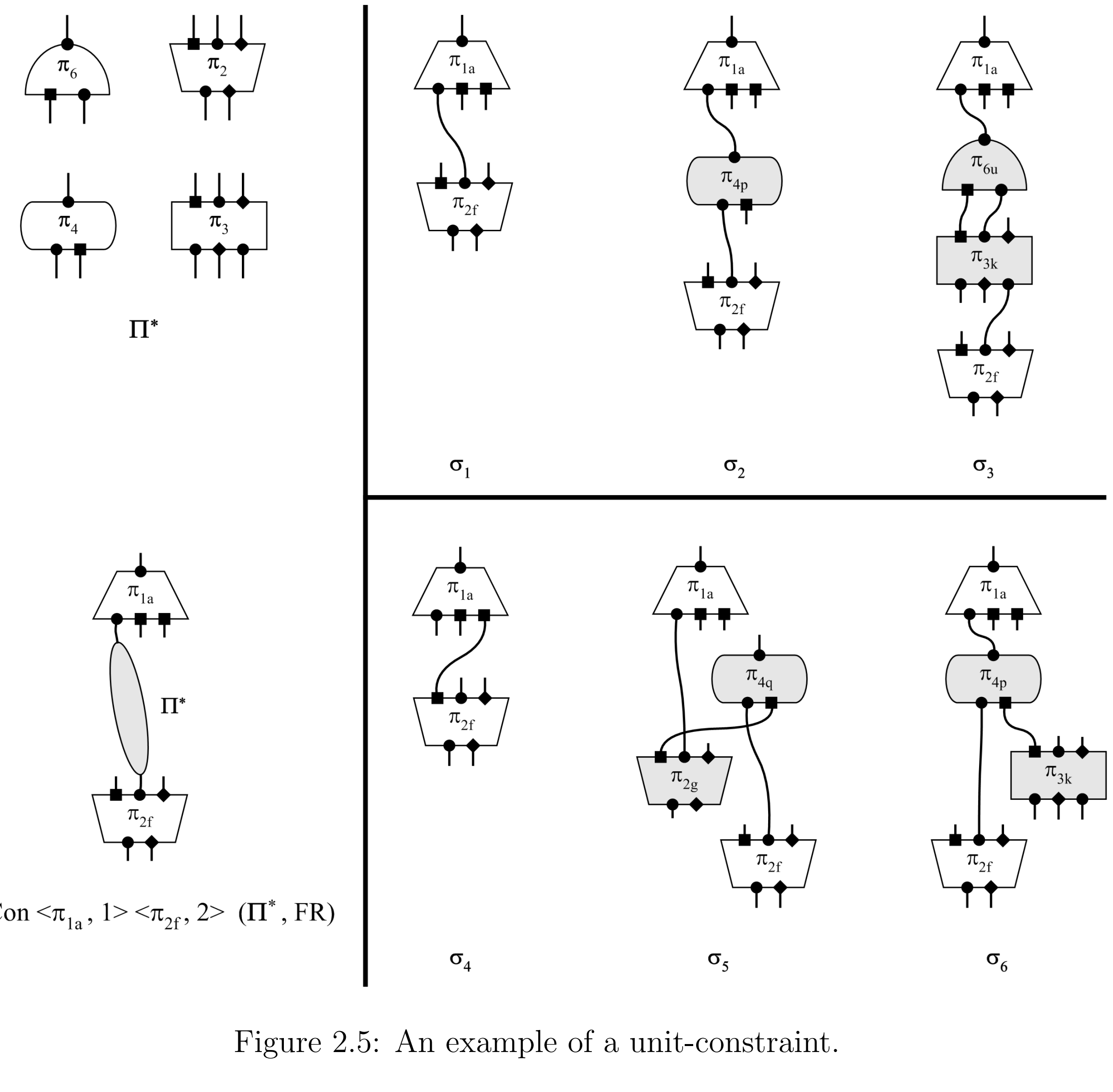


Figure 2.5: An example of a unit-constraint.

2.2.1.2 Structural constraints

A central definition for single level class representation is that of the **(level 0) structural constraint**, which is a formal mechanism for specifying a family of structs sharing a particular structural "backbone", which may be thought of as a kind of "structural formula".

Formally, the structural constraint consists of a set of pivot primitives $\overline{\Pi}$ and a tuple of sets of primitives $\mathcal{UT}$

$$\mathrm{Con}(\overline{\Pi}, \mathcal{UT}) = \Big\langle \mathrm{UCon}\langle \pi_{1a}, u_1\rangle\langle \pi_{2b}, v_2\rangle(\mathbf{\Pi}_1^*, \mathrm{FR}_1)\ , \ldots$$
$$\ldots\ , \mathrm{UCon}\langle \pi_{2_{k-1},c}, u_{2_{k-1}}\rangle\langle \pi_{2_k,b}, v_{2_k}\rangle(\mathbf{\Pi}_k^*, \mathrm{FR}_k)\ \Big\rangle\ ,$$

Conceptually, it is easier to think of a structural constraint as a collection of unit constraints.

Then, when specifying such structural formulas, it is helpful to visualize the structure and proceed as if one were constructing the constraint out building blocks. Each such building block is a particular unit-constraint and may be "snapped" together at the pivots forming a complex structure[4], see Fig. 2.6 and [2, Def. 10]. In Fig. 2.6 an example of a structural constraint is illustrated, e.g. $\mathrm{Con}(\overline{\Pi}, \mathcal{UT})$, with $\overline{\Pi} = \{\ \pi_{1a}\ ,\ \pi_{2f}\ ,\ \pi_{3k}\ ,\ \pi_{6u}\ \}$ and Π_i^*'s as shown on the top left; FR_i's are not included. To the right are two sets of structs, σ_1 and σ_2 that satisfy the constraint (the two light-shaded ellipses correspond to the dashed ellipse at bottom left, and σ_3 and σ_4 do not. A primitive, π_{3l} in struct σ_3, is attached to the wrong terminal of π_{1a}, and in σ_4, a pivot primitive π_{6v} is not an ancestor of pivot primitive π_{2f}, according to the corresponding unit-constraint shown on the left.

[4]Note that the graph, whose vertices correspond to the pivot primitives and edges to the unit-constraints, must be connected, i.e. all pivot primitives are connected to one another via one or more unit-constraints.

For example, given a struct α and some arbitrary structural constraint, the noise primitives specified within each unit-constraint of that constraint, describe incidental events[5] that may be attached to the specified substructures of α, such that α will still satisfy the constraint, e.g. σ_1 and σ_2 in *cf.* Fig. *Ibid.* In other words, α may contain other primitives so long as the "structural backbone" is maintained, thus facilitating a reasonable degree of (localized) variation.

2.2.1.3 Active structural constraints

An extension of the structural constraint is the **active (level 0) structural constraint**, such that in addition to the set of pivot primitives and $\mathcal{UT}$, the active structural constraint appends two additional sets: a set of **anchor primitives** and a set of **open primitives**. This extension to the concept of constraint enables the designer to "mark" the pivot primitives of active constraints, both as open and/or as an anchor, which affords a greater degree of control over the generating system via their restrictive roles in the **active extensions of (level 0) working structs**[6].

2.2.2 Specifying class generating systems

A **(level 0) class representation**, $\mathfrak{R}$, consists of a set of *constituent pivot primitives* $\overline{\Pi}$, a set of *constituent noise primitives* Π^*, and a (partial) **class generating system specification** $\mathscr{G}_{\mathfrak{R}}$:

$$\mathfrak{R} = \langle \overline{\Pi}, \Pi^*, \mathscr{G}_{\mathfrak{R}} \rangle \ .$$

The class generating system specification is a step-wise description for assembling (level 0) structs, where each step consists of a non-empty set of active (level 0)

[5]Incidental events are considered to be "noise" produced by other classes in the environment.
[6]See [2, Def. 12] for a formal definition.

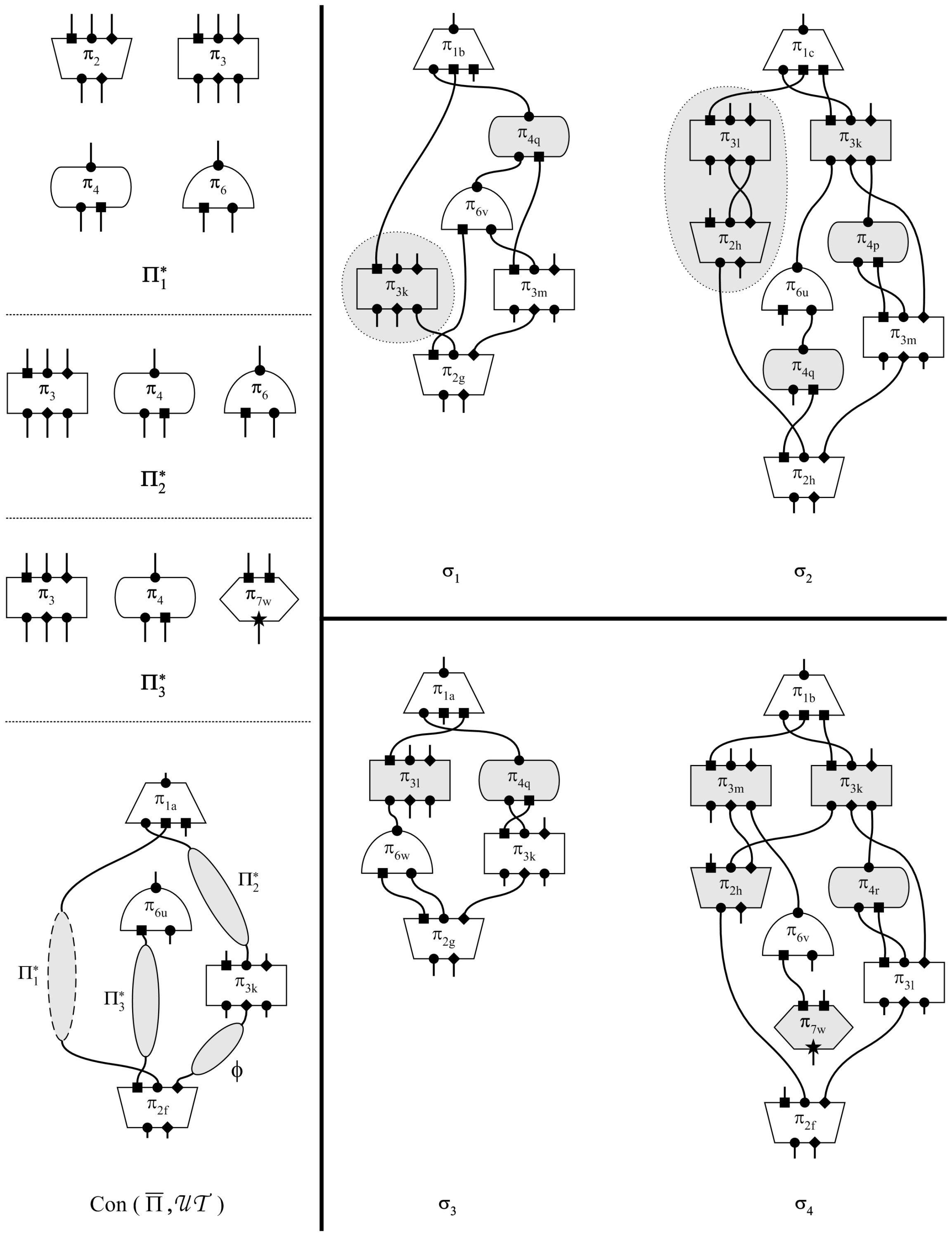


Figure 2.6: An example of a structural constraint.

constraints:

$$\mathscr{G}_{\mathfrak{R}} = \Big\langle \Big\{ \mathrm{ACon}_{1,i}(\overline{\Pi}_{1,i}, \mathcal{UT}_{1,i}, \overline{\Pi}^{anc}_{1,i}, \overline{\Pi}^{opn}_{1,i}) \Big\}_{i \in I_1}, \Big\{ \mathrm{ACon}_{2,i}(\overline{\Pi}_{2,i}, \mathcal{UT}_{2,i}, \overline{\Pi}^{anc}_{2,i}, \overline{\Pi}^{opn}_{2,i}) \Big\}_{i \in I_2}, \ldots \ldots, \Big\{ \mathrm{ACon}_{t,i}(\overline{\Pi}_{t,i}, \mathcal{UT}_{t,i}, \overline{\Pi}^{anc}_{t,i}, \overline{\Pi}^{opn}_{t,i}) \Big\}_{i \in I_t} \Big\rangle .$$

2.3 Generating class elements

2.3.1 Class generating system operations

To help with the following discussion three additional concepts are required upfront: the concept of current working struct; the concept of working class element; and the concept of active extension. A **current working struct** is the struct being produced by a particular class generating system process operating in an environment with other class generating system processes, such that the struct being produced may contain primitives not contributed by the class generating system producing the working struct. These extra primitives are referred to as *noise primitives*, which reflect environmental influences, and are contributed by external class generating system processes. A **working class element**, γ_{j-1}, is a substruct of some current working struct σ^k_{2j-1},

$$\gamma_{j-1} \Subset \sigma_{2j-1} ,$$

and is formed only by the primitives that have been contributed so far by a generating process following some given class generating system specification $\mathscr{G}_{\mathfrak{R}}$, *cf.* Fig. 2.7. An **active extension**

$$\mathrm{Ext}\big(\sigma^w_{j-1}, \mathrm{ACon}_{j,i}(\overline{\Pi}_{j,i}, \mathcal{UT}_{j,i}, \overline{\Pi}^{anc}_{j,i}, \overline{\Pi}^{opn}_{j,i})\big)$$

of working struct σ^w_{j-1} with respect to some given active structural constraint $\mathrm{ACon}_{j,i}$ is defined as a set of active[7] structs α, such that (i) each struct α satisfies the given constraint; (ii) the assembly $\mathcal{A}(\sigma^w, \alpha)$ exists, (iii) the primitives marked as "anchors" on α have corresponding primitives marked as "open" on σ^w_{j-1},[8] and (iv) α contributes at least one additional primitive to the working struct.

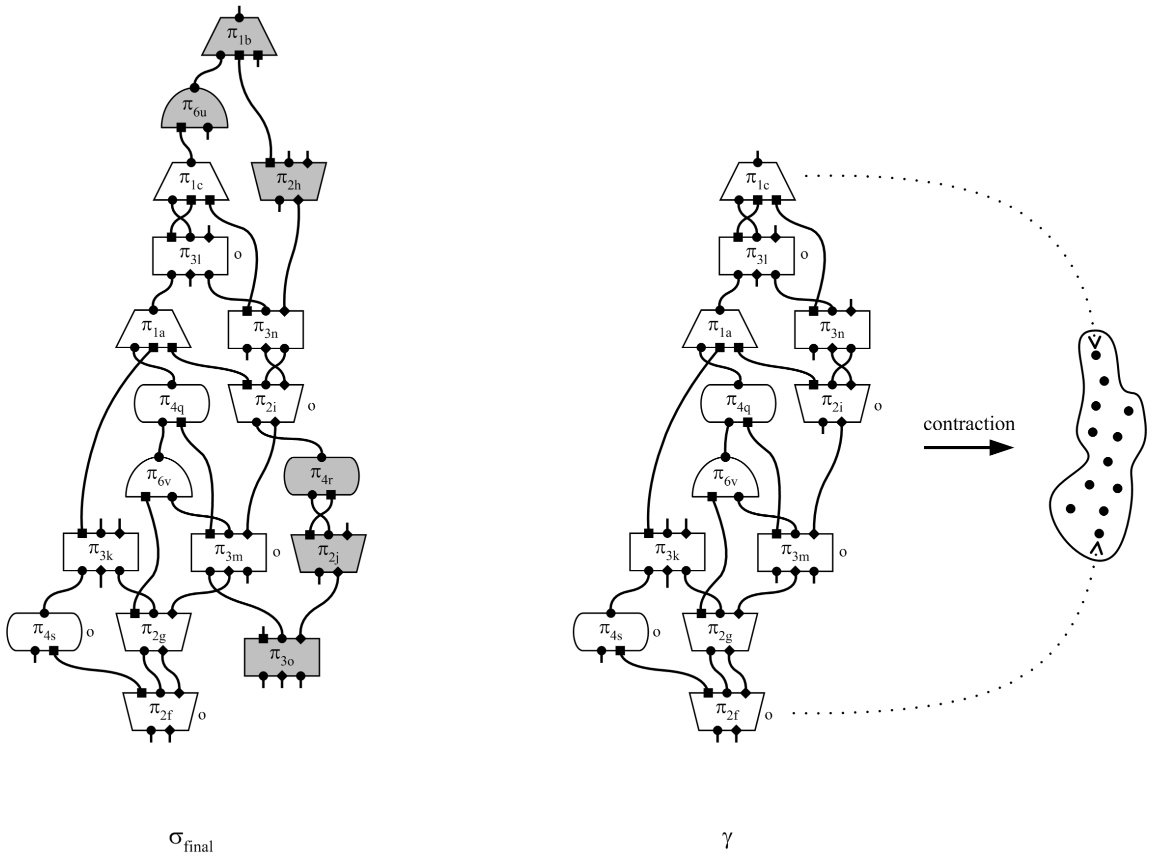


Figure 2.7: A pictorial illustration of a class element struct as a substruct of a *final* working struct.

The actions of a $\mathscr{G}_{\mathfrak{R}}$–generating system consist of the following two operations.

- First, *a struct β_j is (nondeterministically) selected* from one of the non-empty

[7]The reader is referred to [2, Def. 12, p. 43] for more details about active extensions, such as active structs.

[8]These markings together specify how two structs must overlap.

sets of active extensions,

$$\beta_j \in \left\{ \mathrm{Ext}\big(\sigma^w_{j-1}, \mathrm{ACon}_{j,i}(\overline{\Pi}_{j,i}, \mathcal{UT}_{j,i}, \overline{\Pi}^{anc}_{j,i}, \overline{\Pi}^{opn}_{j,i})\big) \right\}_{i \in I_j} .$$

- Second, *the active extension, β_j, is assembled to the current working struct*, σ_{2j-1}, in order to produce the next working struct, σ_{2j}. Incidently, this assembly accomplishes another assembly

  $$\gamma_j = \gamma_{j-1} \underset{\mathrm{Ext}}{\triangleleft} \beta_j ,$$

  since $\gamma_{j-1} \Subset \sigma_{2j-1}$, where γ_j is the *next working class element.* The latter assembly appropriately modifies and allocates the "open" markings. Such that the primitives previously marked as open remain open, if so specified in the constraint, and new markings are inherited from β_j.

A visual example of the above two operations performed by a class generating system is shown in Fig. 2.8, also the reader is referred to [2, pp. 43–45] for a more in depth discussion.

In Fig. 2.8, a two-step generative unit, comprised of a step by the environment and a corresponding step by the class generating system, attaches a piece of substructure to a *working class element.* The dark shaded primitives are those added by the environment and the dotted lines delineate the selected active struct β_j assembled to σ_{2j-1}. Primitives that remain dark shaded at the end of the generative process are not part of the working class element, e.g. γ_{2j}. Note that the lightly shaded primitives are incidental primitives that happened to be in β_j and were subsequently absorbed into the γ_{2j}.

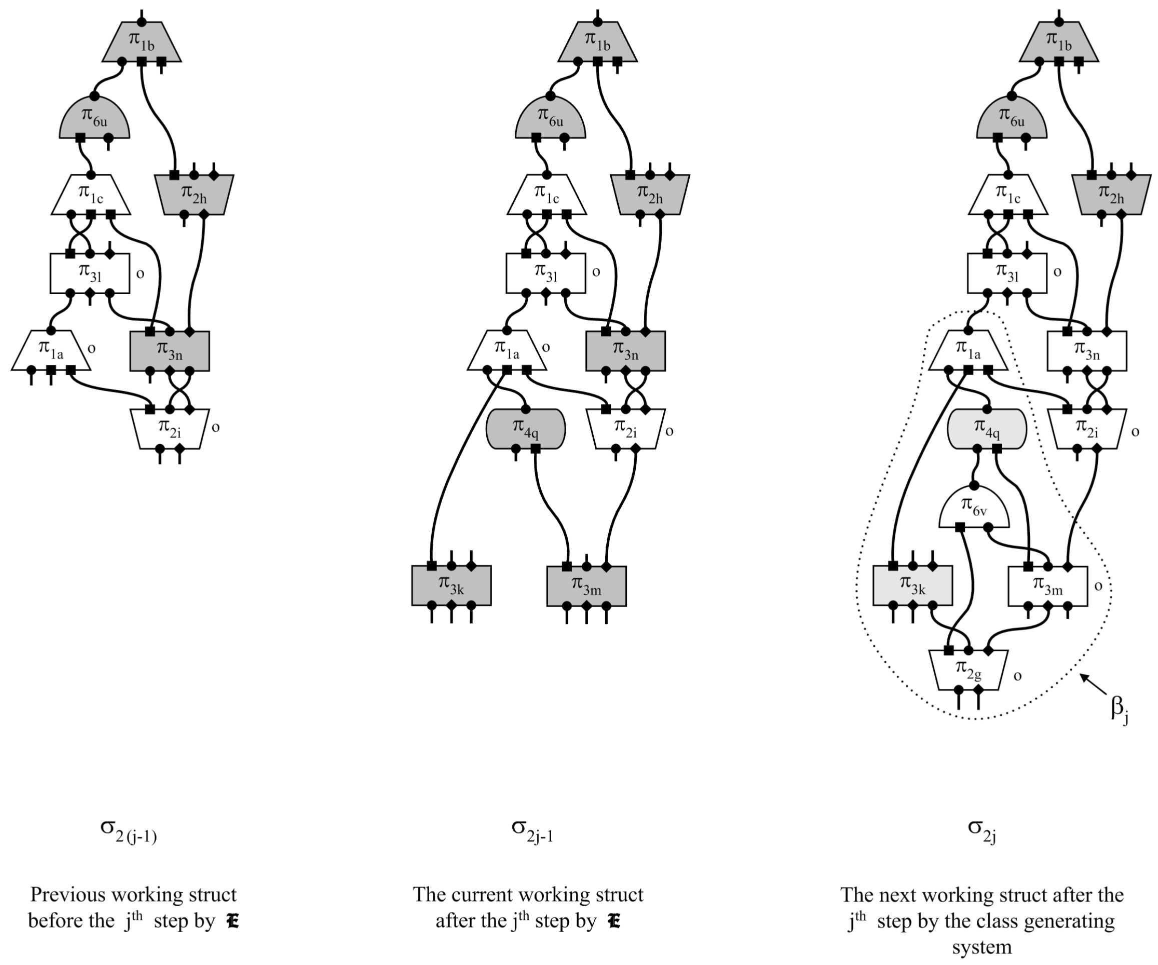


Figure 2.8: A pictorial illustration of a two-step generative unit executed by a class generating system process involved in the construction of a class element.

2.3.2 Active role of the (event) environment

ETS includes definitions for multi-level class representation and their class generating systems, however for the purposes of this paper, higher-level classes are not discussed. Instead attention is focused on single-level (or level 0) object and class representation. Also notice that the following discussion on the role of environment, is confined to the event environment (first introduced in Sec. 1.4).

The environmental role in the generation of class elements is understood as follows:

- A **(level 0) environment** $\mathfrak{E}$ is comprised of all (level 0) class generating systems, i.e. a system of class generating systems.
- A step by any instance of these class generating systems is understood to follow, first, a step by $\mathfrak{E}$.
- Each step, by the environment is understood to be specified by sets of active constraints.
- Time-homogeneity for the duration of time a generating system remains in some step j may not presumed.
- Each step of the environment, relative to some class generating system process, "manifests" itself in the *environmental structs* being assembled to the current *working class element struct.*

For example, relative to a given class generating system, at some step $2(j-1)$ the environment $\mathfrak{E}$ may assemble several structs (each selected by another generative process) to the *previous working struct* $\sigma_{2(j-1)}$, producing the current **working struct** σ_{2j-1}, whose pivot primitive *markings are unchanged.* Such a step by $\mathfrak{E}$, is likely to consist of several "actual steps" depending on the state of the environment, e.g. some classes involved may have faster running generating systems (spend less time in each step), while some may have slower running generating processes and not contribute any primitives at all by remaining in a particular step/state for long periods of time.

Class generating systems act independently from the structure of $\mathfrak{E}$, i.e., the class generating systems in $\mathfrak{E}$ cannot interfere with another generative process directly. However, via the concept of environment, external class generating systems in $\mathfrak{E}$ may interfere with an internal generative process indirectly, if the class elements being produced by both processes overlap, which involves assembling structs to one another's working struct.

Chapter 3

Application

3.1 Bubble Man object representation

3.1.1 A single primal class and its sub-classes

The Bubble Man example contains a single primal class, called a Bubble Man Basic Cell. These cells are oval-like, and we used them to draw an analogy with biological cells, such that over time our oval-like cells become more "specialized", i.e. "[w]e have "cells" at the beginning, and we have (many more specialized) "cells" at the end, but the representational language is the same for all stages". [2, p. 61]

3.1.2 Primitive-classes

As mentioned in Sec. 2.1.1.2 and specified in [2, Appendix], primitive-classes are a family of structurally similar abstract primitives. Such that the initial and terminal primal classes of each abstract primitive, in such a family, are subclasses of those specified on the primitive-class. This notion of sub-classing primal classes not only promotes careful specification of primitives but decreases the size of the set of primitives one needs to work with when building class representations.

$$|\{[\boldsymbol{\pi}_{*,1}],[\boldsymbol{\pi}_{*,1}],\ldots,[\boldsymbol{\pi}_{*,r}]\}| \;\leq\; |\boldsymbol{\Pi}| \;\ll\; \Pi$$

In Fig. 3.1, the set of primitive classes used in this application are shown. Notices that all the primal processes (except the initials for germination) belong to the Basic Cell class of oval-like "cells", i.e., there is only one primal class where its "sub-classes" are shown in Fig. 3.2 and listed in Table 3.1.

The *external, hand and foot growth* primitives are structured, in such a way, to illustrate how complexity may be modeled (even within such a simple example), e.g. the *left initial* is associated with a particular single cell and the *right initial* is associated with another particular single cell. These three primitives represent a growth in the first cell that induces a simultaneous growth in the second cell, thus, the *left and right terminals* correspond to larger cells. Obviously, one could have introduced similar primitives with more initial and terminal sites, responsible for the modification of several neighboring cells, and in general, one could have split each of these primitives into two.

3.1.3 Primal subclasses

In Fig. 3.2, a pictorial illustration showing how corresponding equivalence relations are used to partition $\boldsymbol{\Pi}$. Note that the *germination* primitive class is not specified because it is not considered to be critical, however, we do specify that the terminal class of this primitive is a subclass named *germ cell*, denoted g.

Listed in Table 3.1, are the twenty-nine subclasses of the "Basic Cell" primal class. In this application, where the "Basic Cell" is an exaggeration of the "stem cell", it is not difficult to make a case for each of the listed sub-classes. Note that, in the

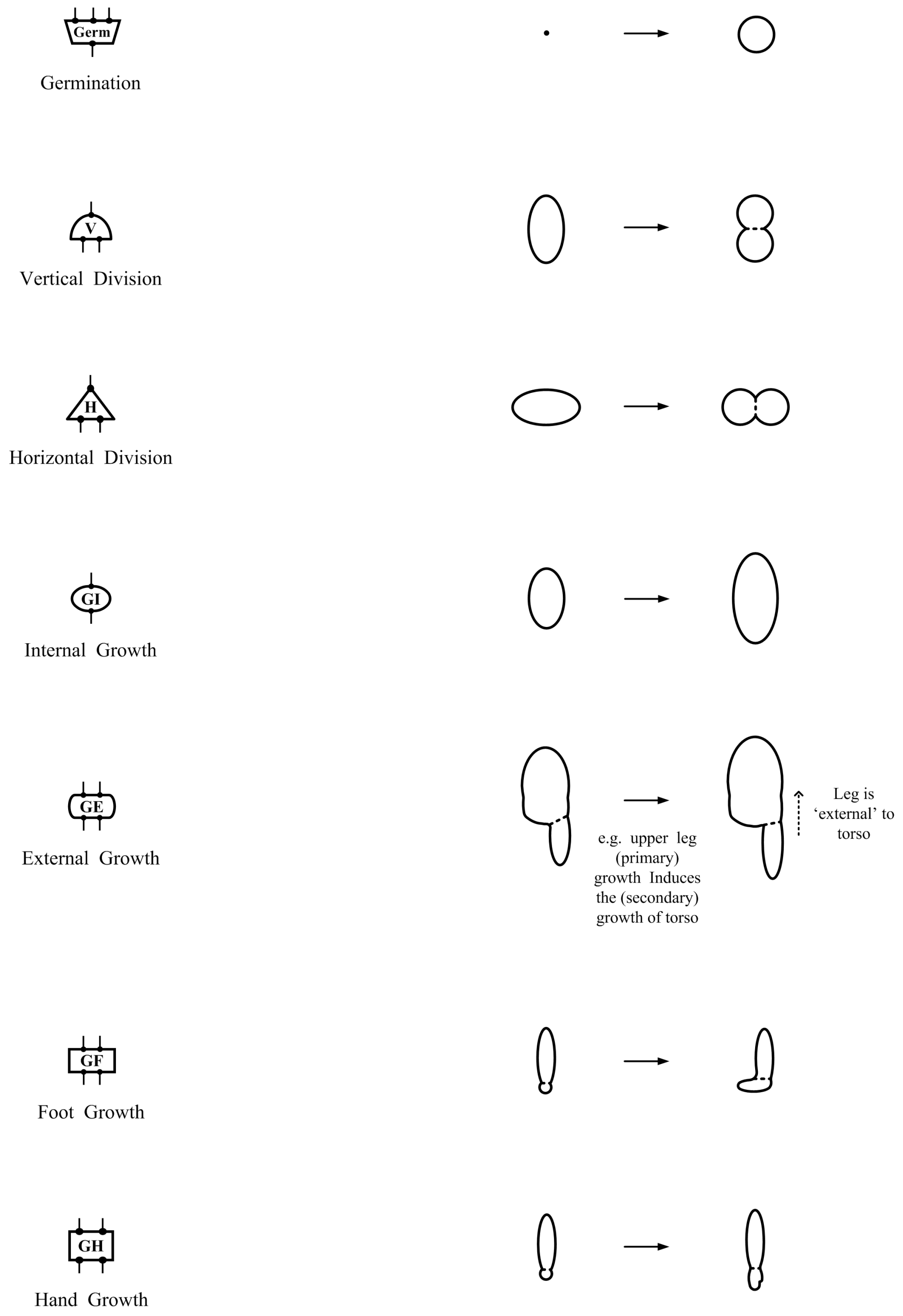


Figure 3.1: The primitive-classes used in this application (*Left*) and the corresponding "Geometric" encapsulations, i.e. 2D spatial representations (*Right*).

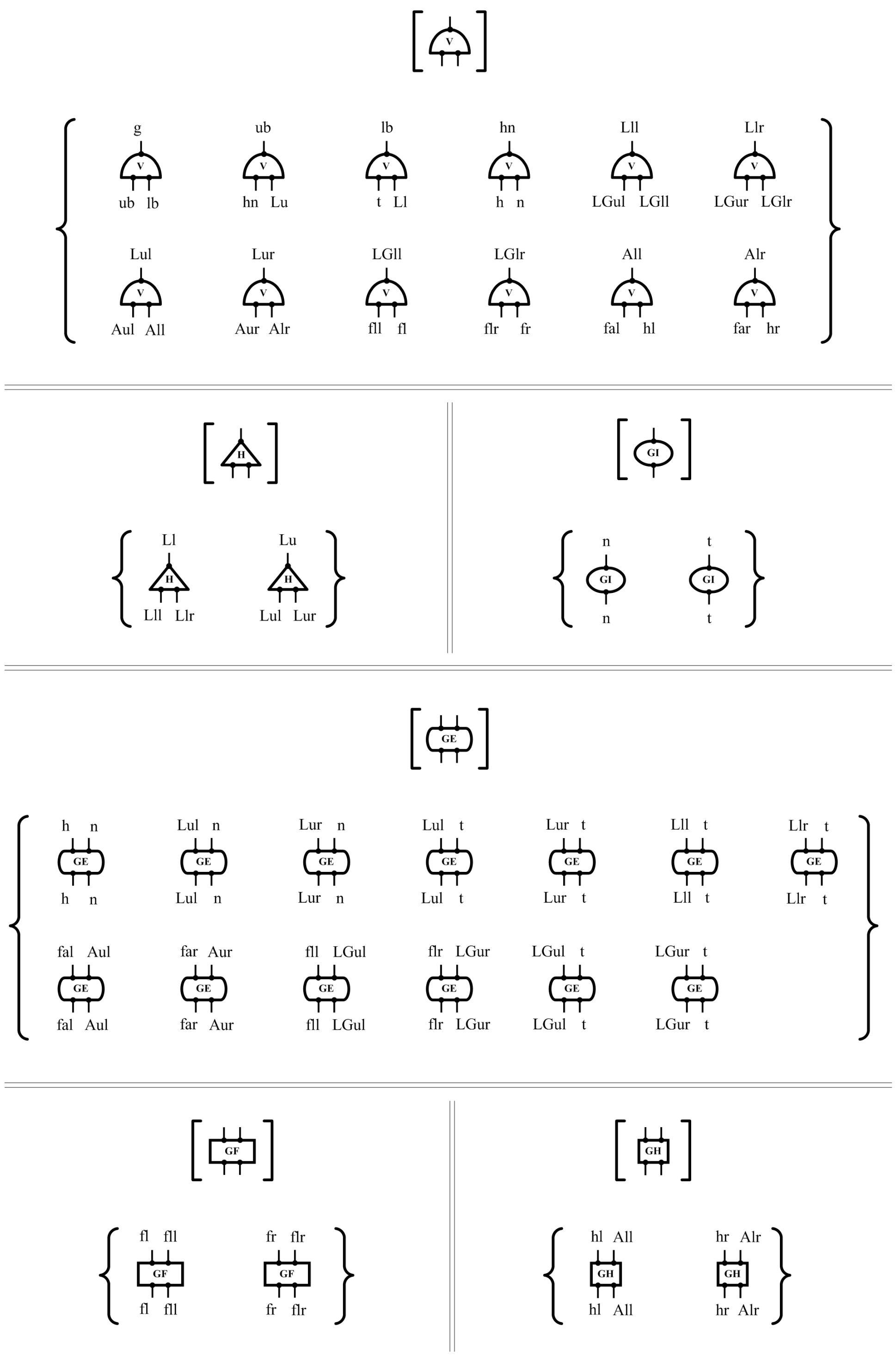


Figure 3.2: Primitive-classes and their elements, except for the germination primitive-class.

abbreviations, capitals may be interpreted as: L for "limb", LG for "leg", and A for "arm".

g – germ cell	Lur – upper right Limb cell	Lul – upper left Limb cell
ub – upper body cell	Llr – lower right Limb cell	Lll – lower left Limb cell
lb – lower body cell	LGur – upper right Leg cell	LGul – upper left Leg cell
hn – head-neck cell	LGlr – lower right Leg cell	LGll – lower left Leg cell
t – torso cell	Aur – upper right Arm cell	Aul – upper left Arm cell
h – head cell	Alr – lower right Arm cell	All – lower left Arm cell
n – neck cell	flr – right foreleg cell	fll – left foreleg cell
Lu – upper Limb cell	far – right forearm cell	fll – left forearm cell
Ll – lower Limb cell	fr – right foot cell	fl – left foot cell
	hr – right hand cell	hl – left hand cell

Table 3.1: All postulated primal "sub-classes" of the single primal class that we call Basic Cell.

3.2 Bubble Man class representation

In this section two class representations from level 0, two class representations from level 1 and the level 2 Bubble Man class representation are presented, starting with the single-level (level 0) class representations and ending with the three-level Bubble Man class representation. The remaining class representations not discussed in this chapter and shown in Fig. 3.3, may be found in Appendix A.

As mentioned in Sec. 1.4, no attempt is made to portray the "Bubble Man" example in such a way as to avoid discussion of higher-level classes. Rather, the "Bubble Man" example is presented in its entirety between this chapter and Appendix A. However, when referring to the "Bubble Man" as a system of classes, the classes that are being referred to are those shown in the right-most column of Fig. 3.3. (With regard to Fig. 3.3, lines point to the constituent classes.)

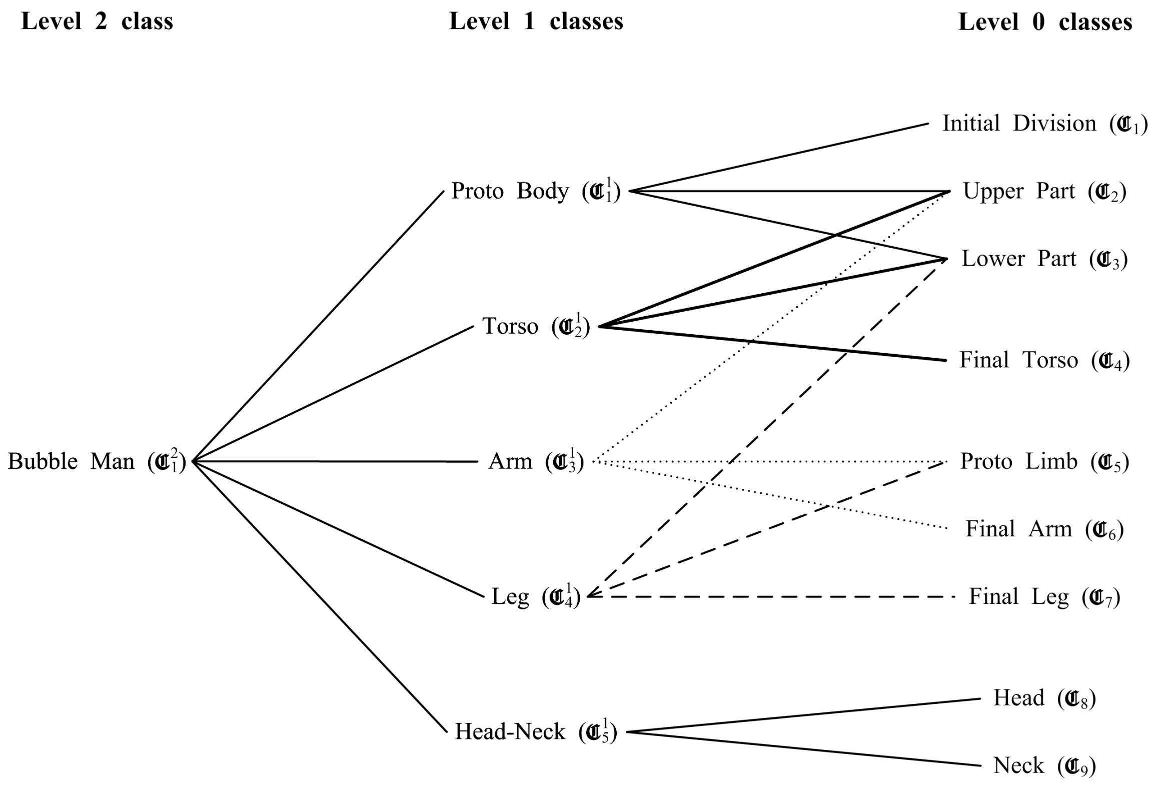

Level 2 class
Level 1 classes
Level 0 classes
Initial Division ($\mathfrak{C}_1$)
Proto Body ($\mathfrak{C}^1_1$)
Upper Part ($\mathfrak{C}_2$)
Lower Part ($\mathfrak{C}_3$)
Torso ($\mathfrak{C}^1_2$)
Final Torso ($\mathfrak{C}_4$)
Bubble Man ($\mathfrak{C}^2_1$)
Arm ($\mathfrak{C}^1_3$)
Proto Limb ($\mathfrak{C}_5$)
Final Arm ($\mathfrak{C}_6$)
Leg ($\mathfrak{C}^1_4$)
Final Leg ($\mathfrak{C}_7$)
Head ($\mathfrak{C}_8$)
Head-Neck ($\mathfrak{C}^1_5$)
Neck ($\mathfrak{C}_9$)

Figure 3.3: Names of various classes in this example (at three levels).

3.2.1 Level 0 classes

The following two figures depict class representations, $\mathfrak{R}_4$ and $\mathfrak{R}_7$, of the "Final Torso" and the "Final Leg" level 0 classes. In general, they contain the following main elements: an initial (first) step by the level 0 environment $\mathfrak{E}$; a set of noise primitives, a set of pivot primitives, and a class generating system specification. These class generating system specifications are comprised of several steps, each a set containing one or more (level 0) active constraints (and Θ denotes the **level 0 null active constraint** [2, Def. 11]). The set of *pivot primitives*, denoted $\overline{\Pi}$, is specified implicitly within the constraints of the n-step specification, which are all illustrated in the figures. The set of *noise primitives*, denoted $\mathbf{\Pi}^*$, is also specified implicitly within the non-empty unit-constraints, also illustrated using primitive-classes at the top of the figures

$$\mathbf{\Pi}^* = \bigcup_p \mathbf{\Pi}^*_p \,.$$

In general, the class generation system specifications are specified using primitive-classes, denoted as primitives with no distinguishing labels on the primal processes. However, some primitives have labels on all of their primal processes, this singles out a particular (admissible) abstract primitive from the corresponding primitive-class.

3.2.2 Level 1 classes

The next two figures depict level 1 class representations, $\mathfrak{R}^1_2$ and $\mathfrak{R}^1_4$, of the "Torso" and "Leg" level 1 classes. In general, they contain the following main elements: an initial (first) step by the level 1 environment $\mathfrak{E}^1$; a set of pivot class elements, and a class generating system specification. The class generating system specification is comprised of a number of steps, each containing a single (level 1) active constraint. Note that because the level 1 class specifications are comprised of such simple steps,

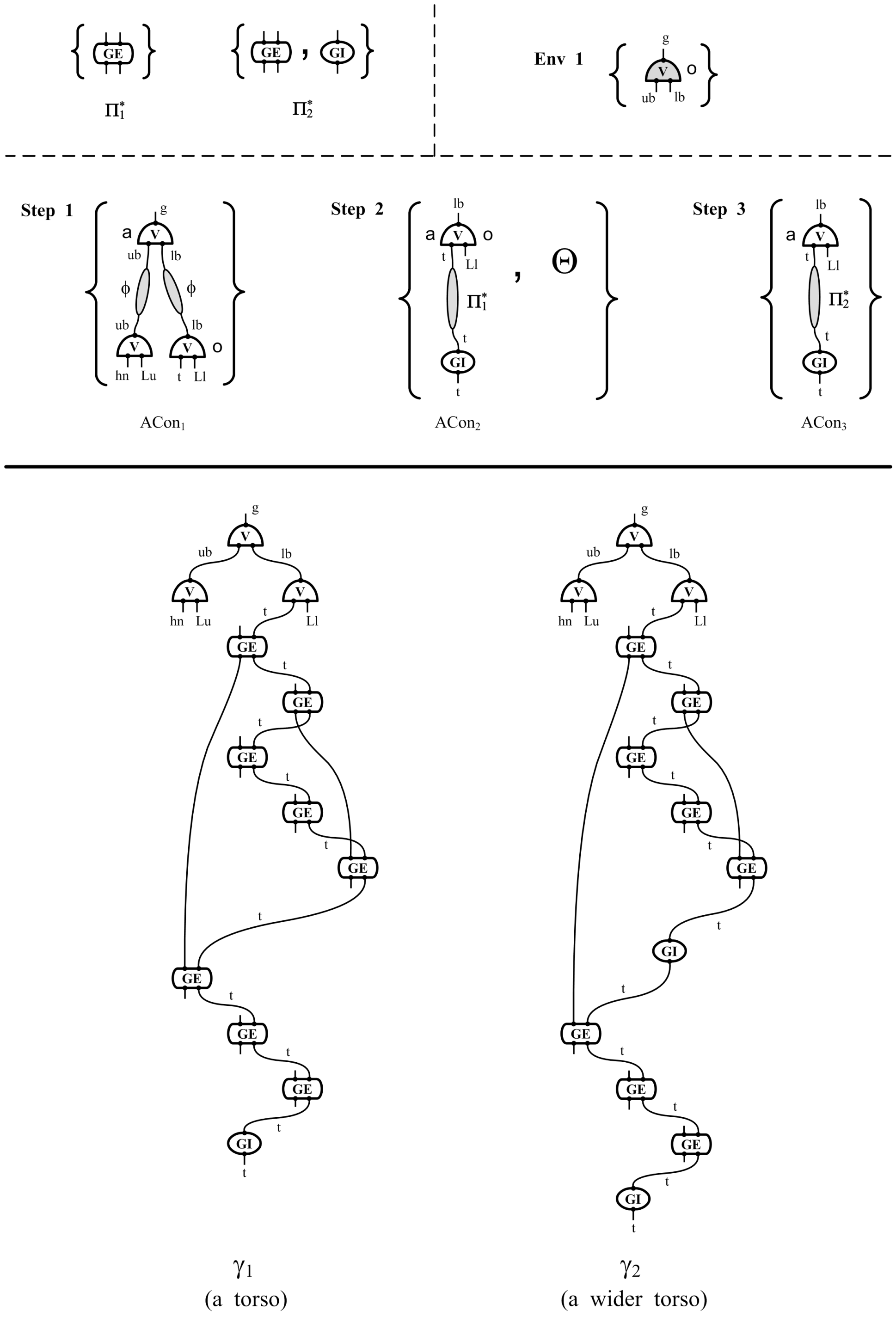


Figure 3.4: Pictorial description of $\mathfrak{R}_4$, the level 0 class representation of **Final Torso**, also denoted $\mathfrak{C}_4$.

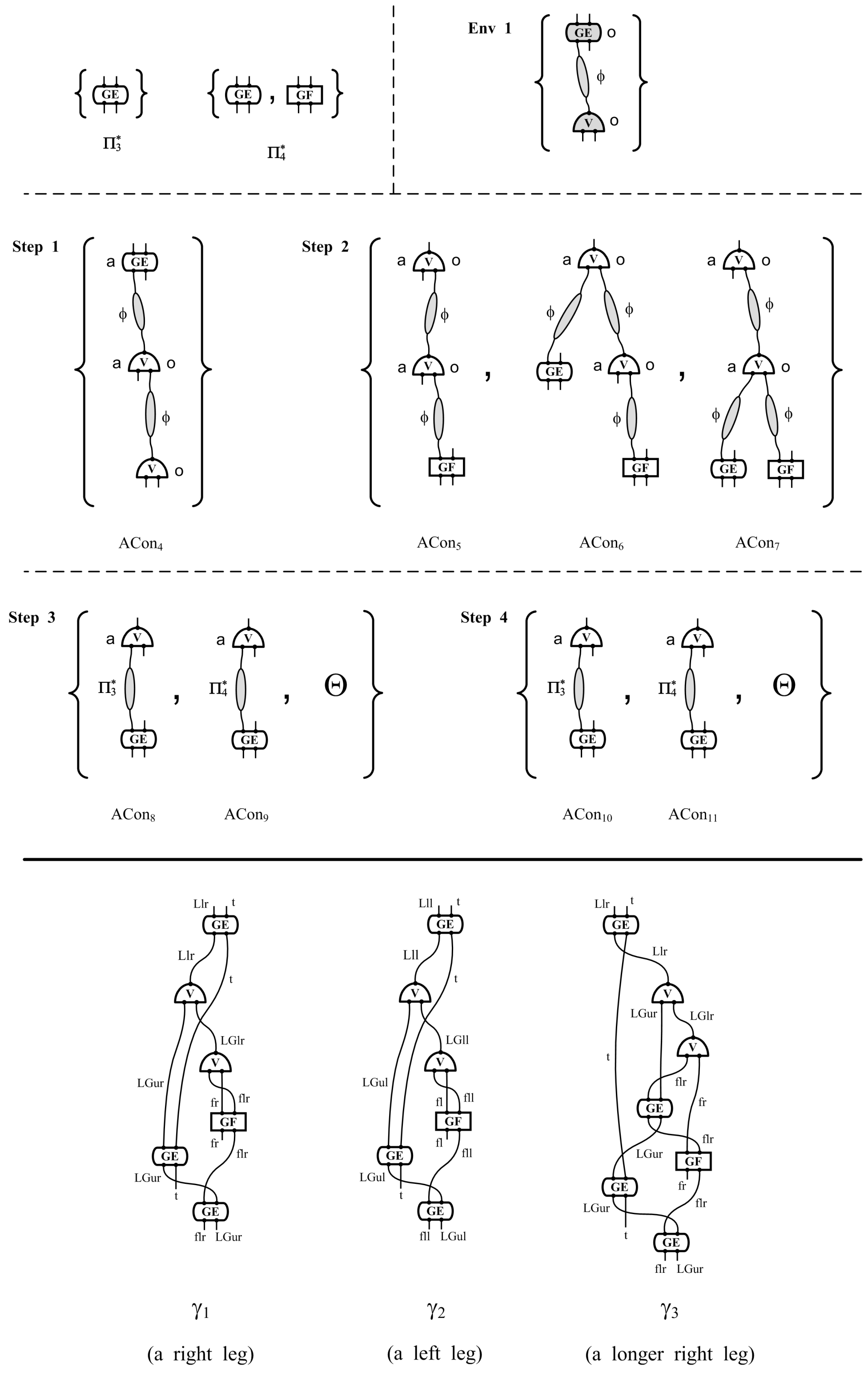


Figure 3.5: Pictorial description of $\mathfrak{R}_7$, the level 0 class representation of **Final Leg**, also denoted $\mathfrak{C}_7$.

variation is subsequently limited to the initial level only. The set of *pivot class elements*, denoted $\overline{\mathcal{C}}_{\mathfrak{R}^1_k}$, is specified implicitly within the constraints of the n-step specification, which are all illustrated in the figures.

In Fig.'s 3.6 and 3.7, the dots are contracted constituent primitives, and their difference in number (between γ^1_1 and γ^1_2) is a reflection of variation in the corresponding previous level structures. Since our set of primitive transformations models growth events in general, the structural elements shown to the right of each figure correspond to temporally and spatially larger elements/objects. However, the set of *constituent noise classes* $\mathcal{C}^*_{\mathfrak{R}^1_k}$ is empty.

3.2.3 The Bubble Man class

Finally, the Bubble Man level 2 class representation shown below, Fig. 3.8, contains the following main components: the initial step by the environment, $\mathfrak{E}^2$; a set of pivot class elements, and a partial step-wise generating system specification (note that the solid squares are contracted constituent classes). This class generating specification is comprised of a single step, which contains a single constraint. The set of noise primitives is empty, and the set of pivot primitives are illustrated in the single constraint of the Bubble Man's only step. Given the simplistic nature of the level 1 and 2 class generating systems, the only variation obtained in the generation of Bubble Man elements is found at level 0. Also note that several level 1 and level 0 classes have been left out and are presented in Appendix A only.

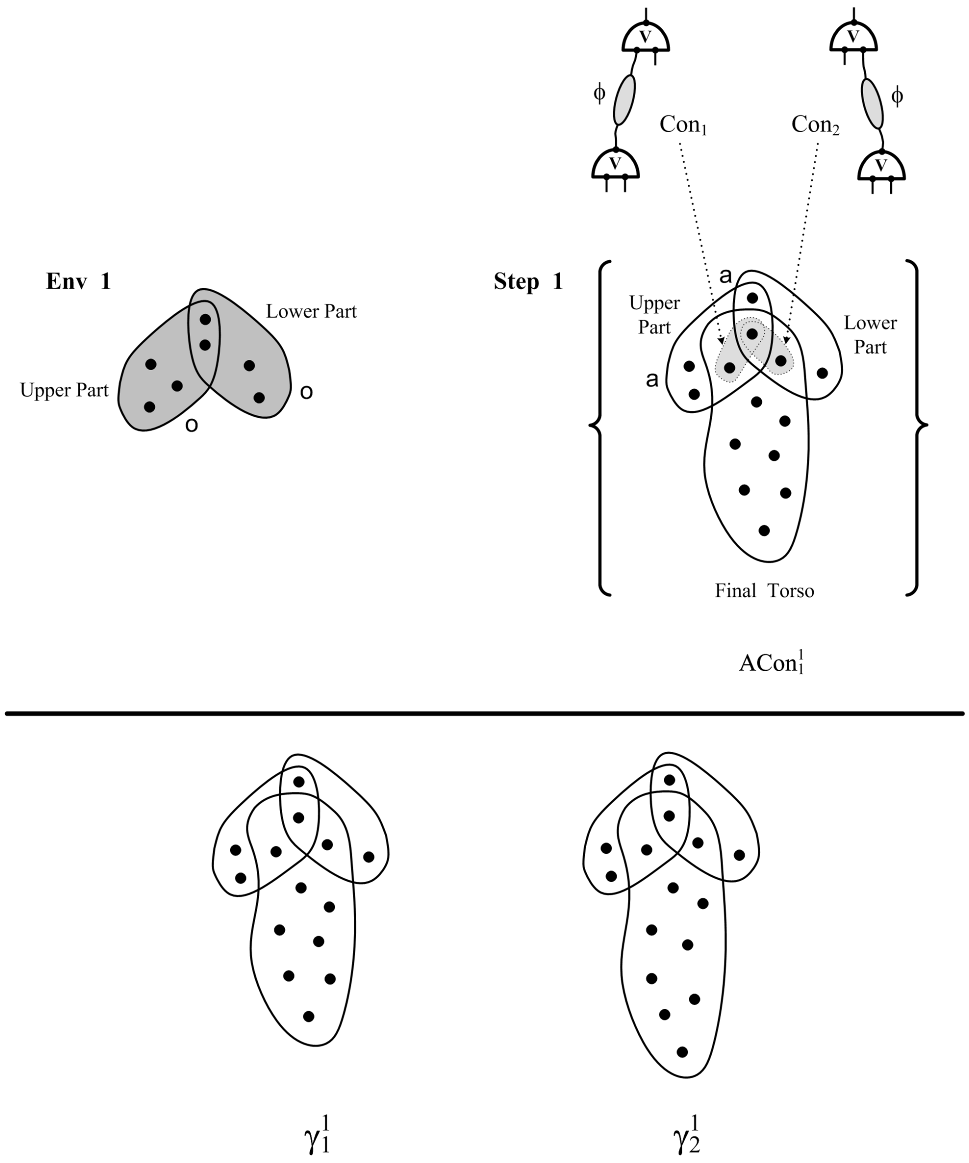


Figure 3.6: Pictorial description of $\mathfrak{R}^1_2$, the level 1 class representation of **Torso**, also denoted $\mathfrak{C}^1_2$, and two example class element structural representations.

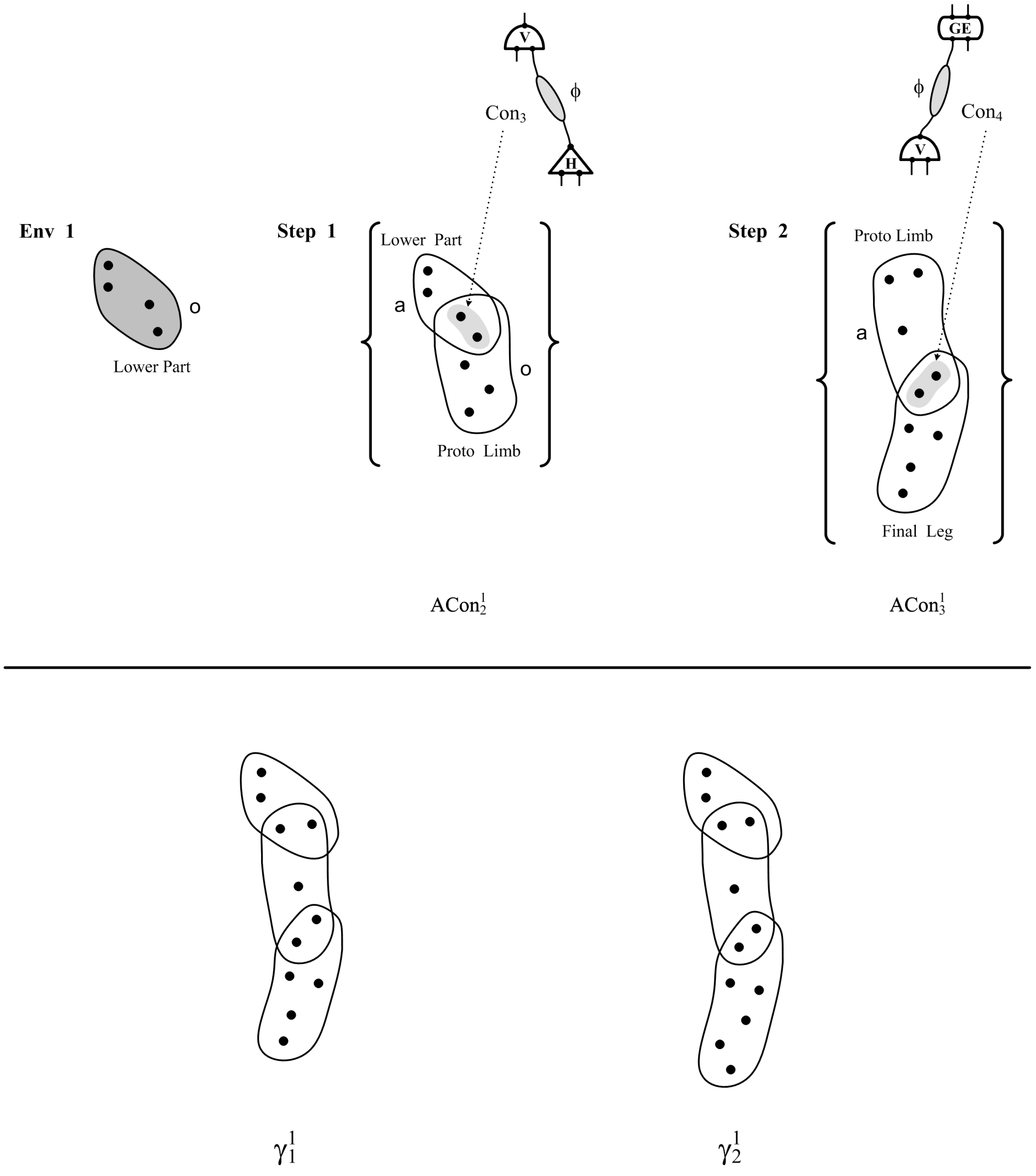
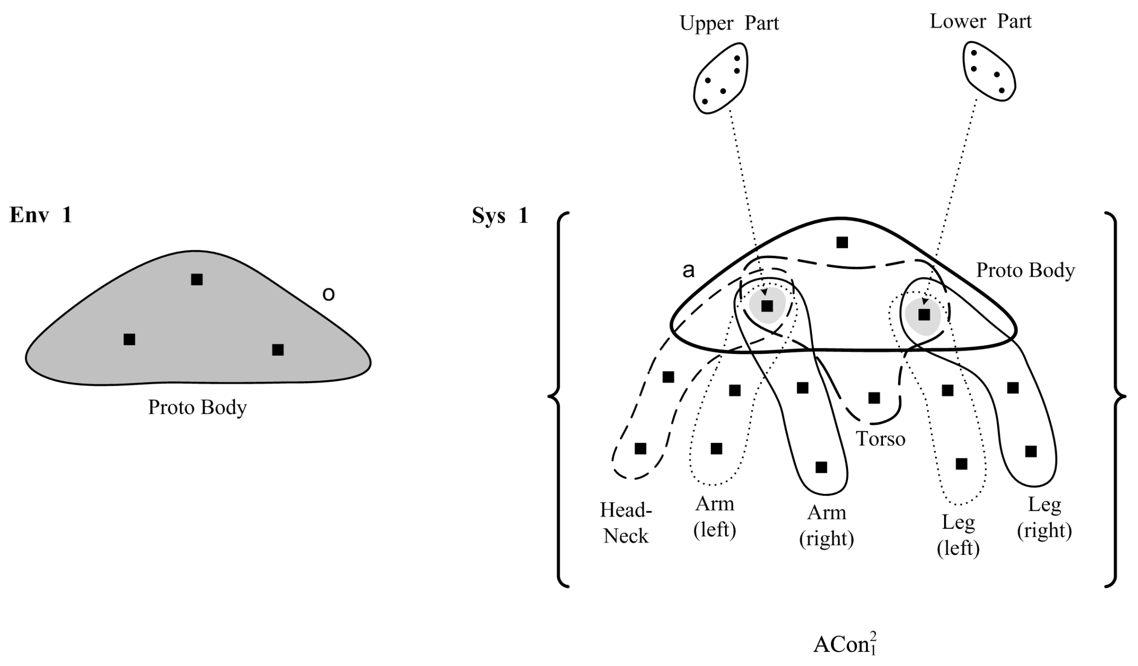


Figure 3.7: Pictorial description of $\mathfrak{R}_4^1$, the level 1 class representation of **Leg**, also denoted $\mathfrak{C}_4^1$, and two example class element structural representations.



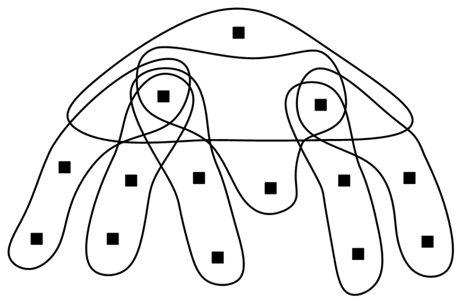


Figure 3.8: Pictorial description of $\mathfrak{R}_1^2$, the level 2 class representation of **Bubble Man**, also denoted $\mathfrak{C}_1^2$.

Chapter 4

Implementation

The following discussion concerning the development of several abstract data types (ADT) and data structures, for the basic concepts of ETS, rely upon and specify operations, some of whose algorithmic implementation and rationale may be found in Appendix B.

4.1 Single-level structs

4.1.1 Primitive ADT

Since ETS structural representations are built out of primitive transformations, the development of a primitive ADT is the (most obvious) first step towards further development of ADT's and algorithms for the other more complex ETS concepts. The ETS primitive transformations, in Fig. 4.1, form the data set underlying the development of the primitive ADT and data structure.

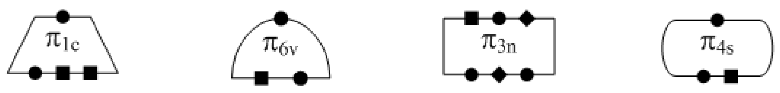


Figure 4.1: A pictorial illustration of four concrete primitives.

A **concrete primitive transformation** is defined as the following four-tuple

$$\pi = \langle \widehat{\pi}, \mathrm{Init}(\pi), \mathrm{Term}(\pi), a \rangle \ ,$$

where $\widehat{\pi}$ is a given name of the primitive, Init(π) and Term(π) are tuples of primal classes, and a is a tuple of primal class elements, according to [2, Def. 1 and pp. 26–28]. Several properties and at least one operation on primitive transformations have been identified, and the properties are listed as follows:.

- Name(π) is a given name of an abstract primitive.
- Init(π) is a tuple of primal classes called the tuple of initial classes, or initials.

$$\langle C_{j_1}, C_{j_2}, \ldots, C_{j_{p(i)}} \rangle_{0\,<\,p(i)}$$

- Term(π) is a tuple of primal classes called the tuple of terminal classes, or terminals.

$$\langle C_{k_1}, C_{k_2}, \ldots, C_{k_{q(i)}} \rangle_{0\,<\,q(i)}$$

- Label(π) is a tuple of primal class elements (such that no two constituent elements are equal).

$$\langle c_{j_1}, c_{j_2}, \ldots, c_{j_{p(i)}} \rangle_{0\,<\,p(i)} \ .$$

The single operation on the primitive transformation identified is called Relabel and it represents the substitution of one label for another.

4.1.2 Primitive data structure

Primitives are stored in (computer) memory using a structure called the primitive data structure, and consists of four arrays, *cf.* Fig. 4.2.

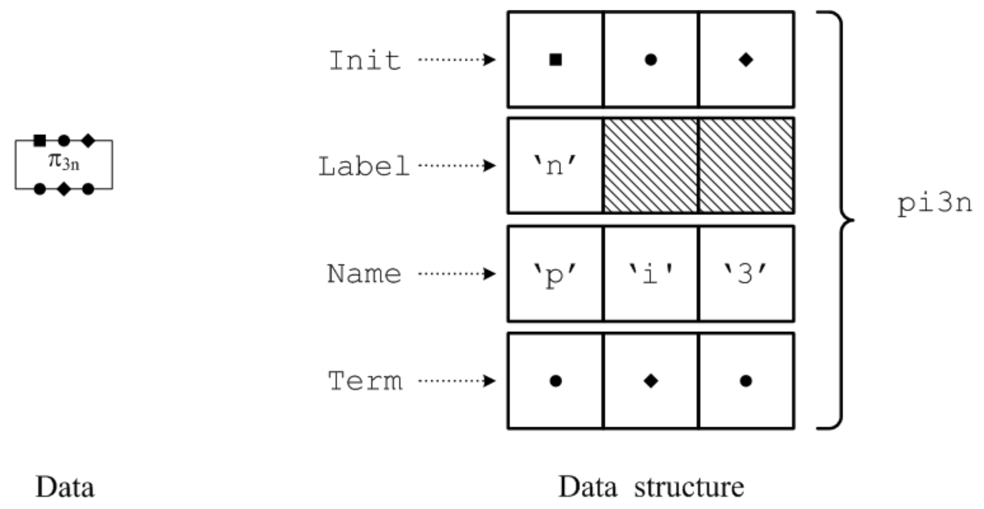


Figure 4.2: A pictorial illustration of a primitive (*left*) stored in memory (*right*).

- `Init` is an array of primal class enumerated types, for example

  $$\texttt{Init[ ]} = \{\ \blacksquare,\ \bullet,\ \blacklozenge\ \}$$

  (the primal class enumerated type may be specified as:

  $$\texttt{enum primalClass \{ } \blacktriangle,\ \bullet,\ \blacklozenge,\ \spadesuit,\ \blacksquare,\ \texttt{etc}, \ldots \})$$

- `Label` is an array of characters, e.g. `a[ ] = { 'n' }`.
- `Name` is an array of characters, e.g. `char[ ] = { 'p', 'i', '3' }`.
- `Term` is also an array of primal class enumerated types, e.g. `Term[ ] = {` $\bullet$, $\blacklozenge$, $\bullet$ `}`.

Finally, the single identified operation on concrete primitives, Relabel, is specified algorithmically as follows:

```
Relabel(PrimitiveADT P, char[ ] Label'):char[ ]
for $i \Leftarrow$ 1 to Length(Label) do
  Label[i] $\Leftarrow$ ‘’
for $j \Leftarrow$ 1 to Length(Label') do
  Label[j] $\Leftarrow$ Label'[j]
```

4.1.3 Linked primitive ADT

A class link between a terminal primal process of one primitive and the initial primal process of another primitive is considered to be a concrete class element connecting the two primitives, *cf.* Fig. 4.3.

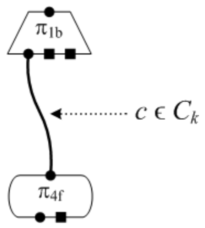


Figure 4.3: A class element $c \in C_k$ connecting (concrete) primitives π_{ia} and π_{jb}.

It may be helpful to think of the class link between two concrete primitives, as an object (or structural process) produced by the first primitive, which eventually became an input to the second primitive. This notion of input and output, motivated an extension of the *primitive* ADT referred to as the *Linked Primitive* ADT. *Linked Primitives* specify an additional operation over concrete primitives, called *class link* (or CL), such that given the four-tuple $\langle \pi_{ia}, u_i, \pi_{jb}, v_j \rangle$ a class link is created *if* $\underline{class}(\boldsymbol{\pi}_i, u_i) = \overline{class}(\boldsymbol{\pi}_j, v_j)$, $u_i \leq |\text{Term}(\pi_{ia})|$ and $v_j \leq |\text{Init}(\pi_{jb})|$.

4.1.4 Linked primitive data structure

The extension of the *primitive* data structure, is called the *Linked Primitive* data structure and it is used to implement the *Linked Primitive* ADT. It consists of a *primitive*, and two tables,

$$\texttt{Input} : \mathbb{N}_{\texttt{Init}} \to \texttt{LinkedPrimitiveADT} \times \mathbb{N}$$

and

$$\texttt{Output} : \mathbb{N}_{\texttt{Term}} \to \texttt{LinkedPrimitiveADT} \times \mathbb{N}$$

such that for each row of the table is a pair consisting of:

- a reference to another *Linked Primitive* (in memory);
- and an index used to select primal processes out of the other *primitive* data structure's `Init` (or `Term`) array.

The *Linked Primitive* data structure is designed in such a way that the onus of storing of a class link, e.g. $\langle \pi_{ia}, u_i, \pi_{jb}, v_j \rangle$, is split between the two primitives involved, e.g. the pair $\langle \pi_{jb}, v_j \rangle$ is stored in `Output[` u_i`]` and the other pair $\langle \pi_{ia}, u_i \rangle$ is stored in `Output[` v_j`]`, *cf.* Fig. 4.4.

The operation on *Linked Primitives*, `Class Link`, stores a class link connection in the primitives involved only if it is valid to do so, which is specified algorithmically as follows:

```
CL(LinkedPrimitiveADT P, int i, LinkedPrimitiveADT Q, j)
if P.Term[i] $\equiv$ Q.Init[j], i $\leq$ Length(P.Term) and j $\leq$ Length(Q.Init)
then
  P.Output[i] $\Leftarrow$ { Ref(Q), j }
  Q.Input[j] $\Leftarrow$ { Ref(P), i }
```

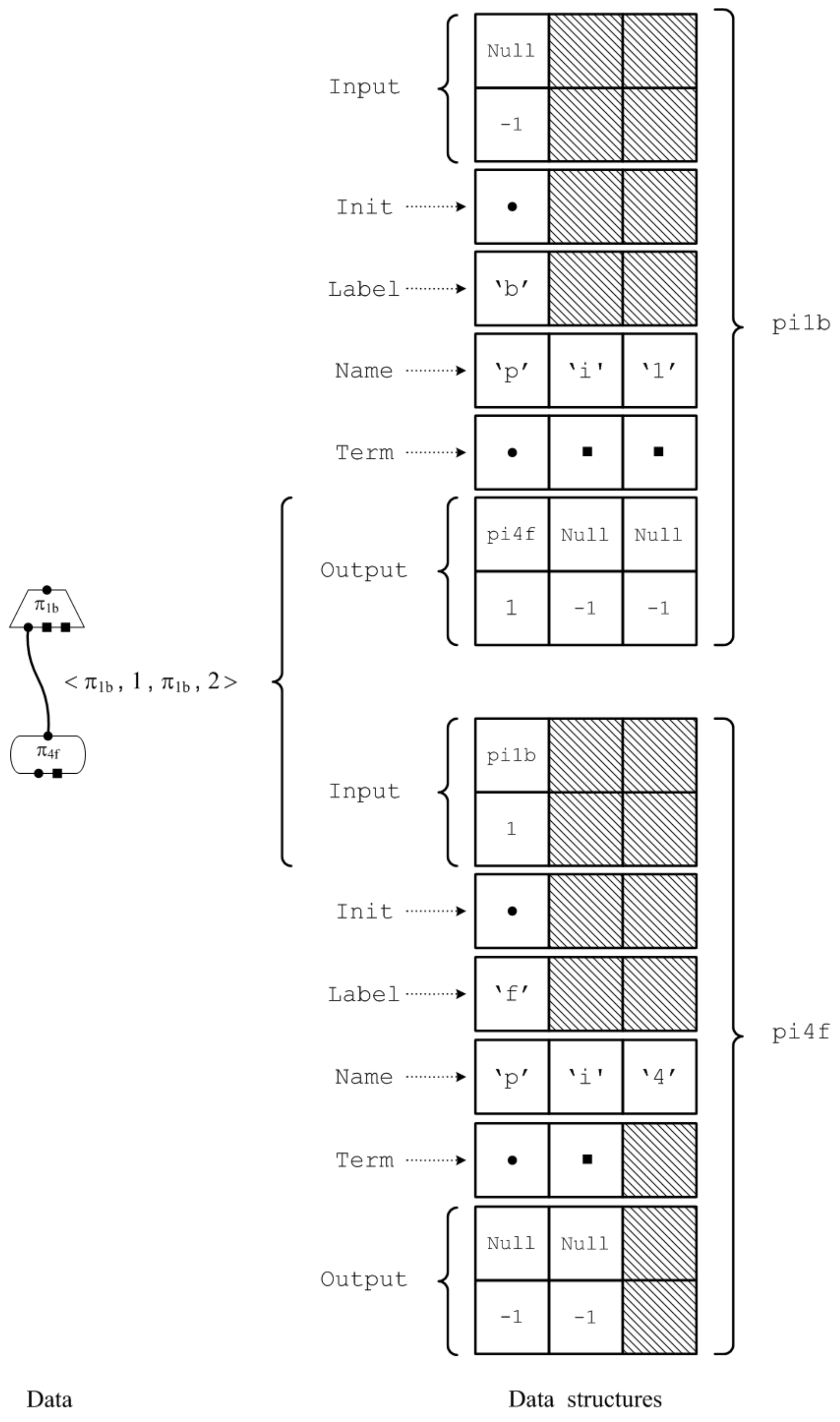


Figure 4.4: Pictorial illustration of two *Linked Primitive* data structures and how the data (*left*) is stored.

4.1.5 Struct ADT

The (single-level) struct is a temporal sequence of primitive transformations, see Fig. 4.5, as such, the temporality, i.e. the order in which the events are "sensed/observed" *must* be preserved.

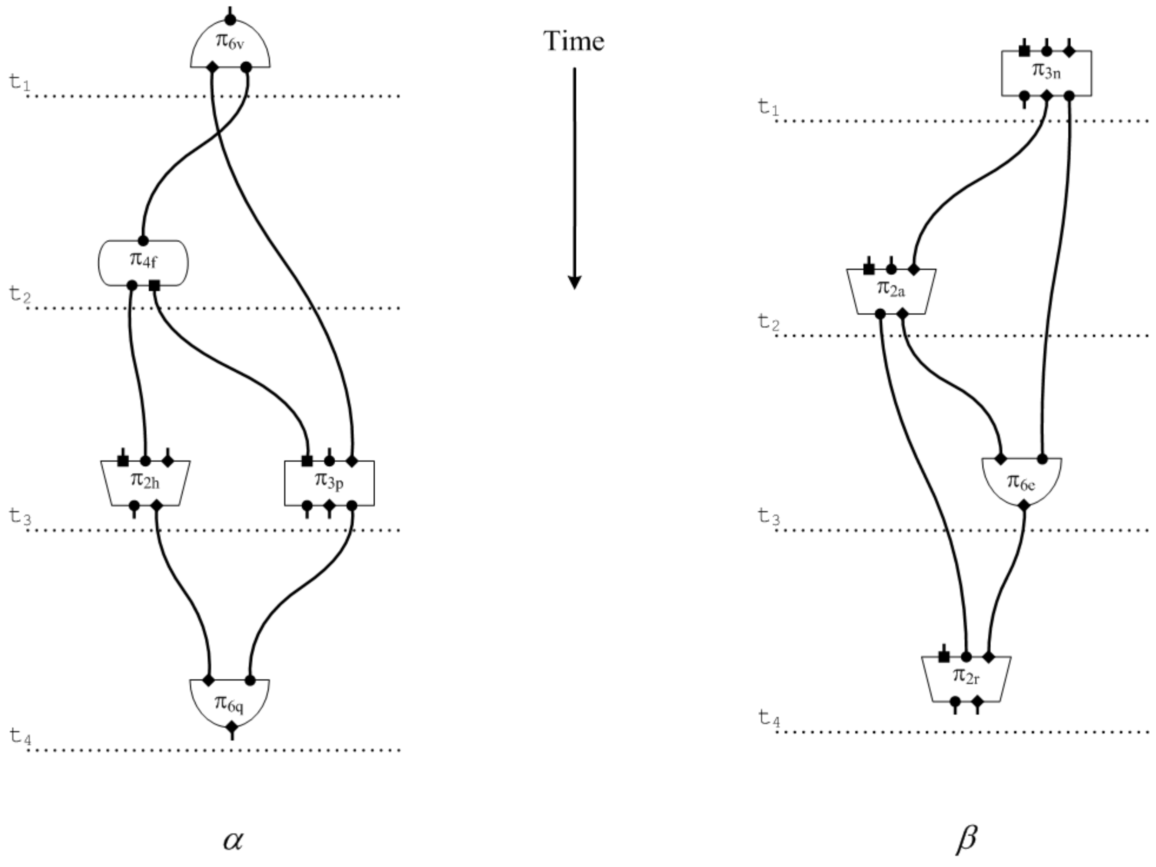


Figure 4.5: Two level 0 concrete structs, α and β.

There are four operations that have been be specified over structs (their formal definitions may be found in [2, Def.'s 3–4, 8]) and are listed as follows:

- *Struct Link, or SL*, may be thought of as an extension of CL that further restricts the creation of class links between primitives. When drawing two primitives with a struct link between them: the first primitive, in the four-tuple, should be drawn on-top so that its terminal primal process(es) connects downward to the initial primal process(es) of the other primitive. Additionally, any terminal primal process may only connect to at most one initial process, and vice-versa.

- *Primitive attachment* is a simple mechanism involving the generation of struct links between an ordered pair of primitives, if a valid struct link exists between two primitives, then the link is added to the struct along with the pair of primitives.

- *Substruct*, given two structs σ_1 and σ_2 ($\sigma_i = \langle \Pi_{\sigma_i},\ \mathrm{SL}_{\sigma_i} \rangle$), if

$$\Pi_{\sigma_1} \subseteq \Pi_{\sigma_2} \quad \text{and} \quad \mathrm{SL}_{\sigma_1} \subseteq \mathrm{SL}_{\sigma_2} ,$$

  then σ_1 is a *substruct* of σ_2.

- Given a set of structs, *struct assembly* is accomplished if the superimposition of each struct onto one another results in a valid struct. Primitives may only be superimposed onto one another if they share the same label[1].

4.1.6 Single-level struct data structure

The data structure designed for storing level 0 structs, is composed of a queue whose elements are lists, which are used to store *Linked Primitives*. The *position types* of these lists are of type `char[]`, *cf.* Fig. 4.6. When adding primitives to these lists, the primitive label is used as a key.

Tentative algorithms implementing the *Struct* ADT operations listed above have been included in Appendix B. Note that these algorithms are only initial implementations of the *Struct* ADT, and are subject to change and modification.

[1]Two primitives are structurally similar if they share the same "main" shape, as well as the same initial and terminal shapes (including the same ordering along the top and bottom of the

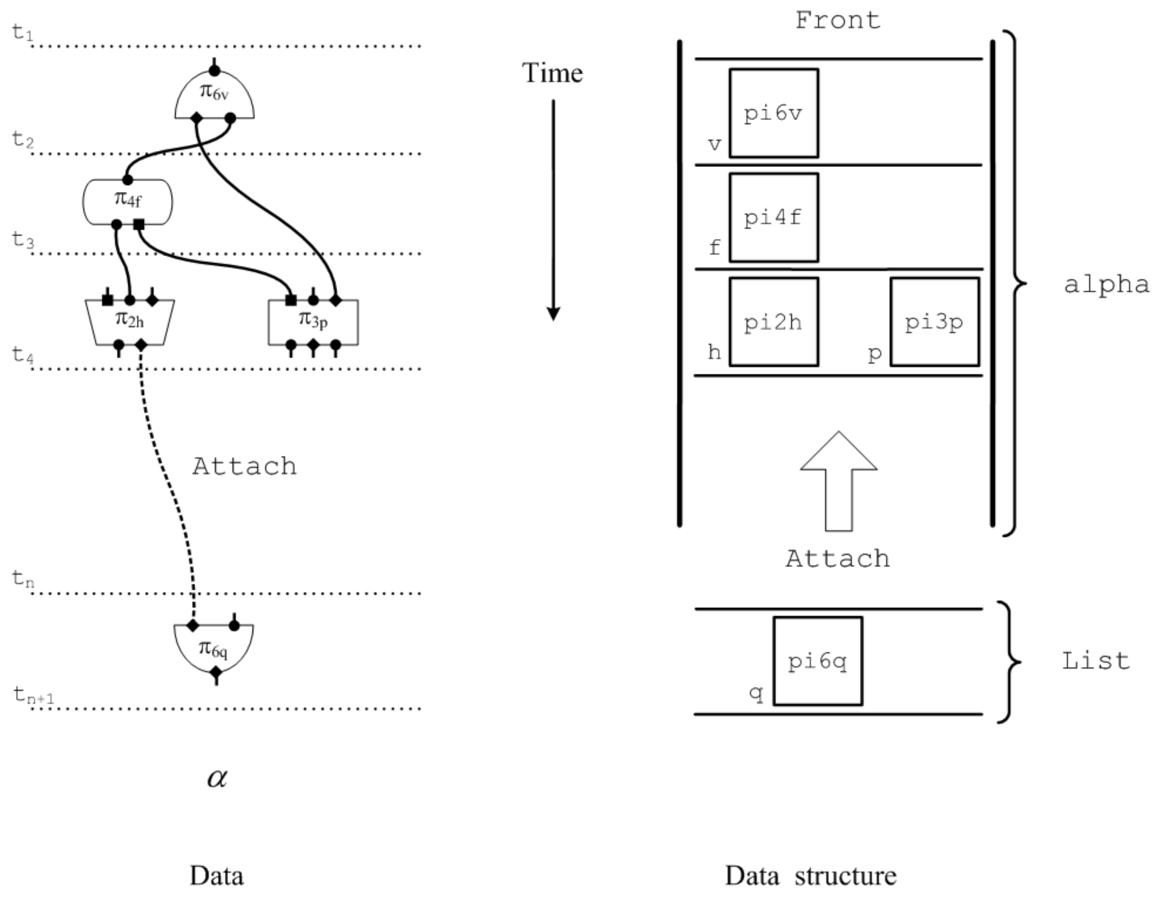


Figure 4.6: Pictorial illustration of how a struct α undergoing attachment of a fifth primitive π_{6q} to another primitive π_{2h}, already an element of the struct, is implemented using a (tentative) struct data structure.

4.2 Single-level structural constraints

4.2.1 Unit-constraint ADT

The unit-constraint is a very useful concept in ETS for specifying simple admissible structure between to primitives called pivots and it is formally specified in [2, Def. 9], *cf.* Fig. 4.7.

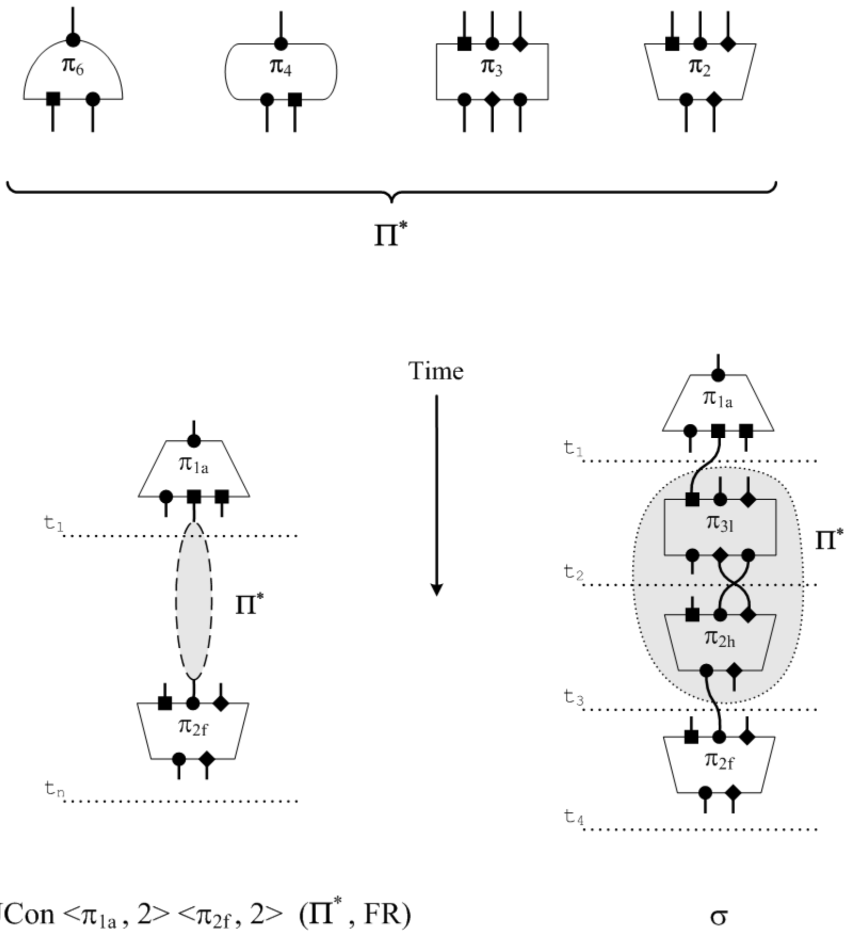


Figure 4.7: Pictorial illustration of a unit constraint (*left*) and a struct satisfying it (*right*).

Unit-constraints are used to constrain structure, so that when given a simple struct, e.g. σ in Fig. *Ibid.*, the struct is said to satisfy the unit-constraint if the extra struc-

primitive “main” shape). Labels are used to ensure structural similarity of primitives.

ture (outlined with a shaded ellipse in the Fig. *Ibid.*) lying between the two given pivot primitives connects to the specified terminal and initial primal processes. With respect to generativity, the concept of unit-constraint is a useful one for specifying a family of (level 0) structs *sharing a particular structural backbone.*

The *Unit-Constraint* ADT is simple and consists of a single unit-constraint operation, FR, which is an application-specific set of rules for specifying various admissible sets of Π^*'s, where each Π^* consists of a set of concrete primitives from $\mathbf{\Pi}^*$.

4.2.2 Primary primitive data structure

The *Primary Primitive* data structure was designed as an extension of the *Primitive* data structure. *Primary Primitives* are similar to the design of the *Linked Primitive* data structure, in their use of tables, however, instead of adding two tables, only a single table is used:

$$\texttt{UCon} : \mathbb{N}_{\text{Term}} \rightarrow \texttt{FR} \times \mathbb{N}_{|\mathbf{\Pi}^*|} \times \mathbb{N}_{|\mathbf{\Pi}^*|} \times \ldots \times \mathbb{N}_{|\mathbf{\Pi}^*|} \times \texttt{PrimaryPrimitive} \times \mathbb{N} \; .$$

Essentially, each row of `UCon` is a vector storing the following elements:

- a memory reference to given set of formation rules, FR;
- an ordered sequence of indices used by FR to index into a set of abstract primitives, e.g. $\boldsymbol{\pi_i}$ is the i^{th} abstract primitive in $\mathbf{\Pi}^*$;
- a memory reference to the other pivot component of the unit constraint;
- the sequence ends with an index into the other primitive's `Init` array, see Fig. 4.8.

The reason for naming this data structure, "primary primitive" is because the first primitive specified in the unit-constraint, or "primary primitive", is the only pivot primitive stored along with the unit-constraint specification. This is done since the unit-constraint is not an explicit connection between pivot primitives, but rather it represents an implicit connection. Since the ETS model incorporates environmental influences into is representations, the unit constraint provides a natural mechanism for handling the presence of "noise" (primitives contributed by other generating systems).

The sequence of abstract primitive indices stored in UCON, after the memory reference to FR, are used by FR and their ordering is meant to serve as a mechanism for specifying of formation rules. FR is tentatively implemented as a "hardwired mapping" from an index to a set of concrete primitives stored in memory, see Fig. 4.8. Notice the shaded ellipses emphasizing the correspondence between the data set, used to model the unit-constraint ADT (*c.f.* Fig. 4.7), and the data structure. Also, shown in *Ibid.* is an example of how FR may be specified via explicit ordering of abstract primitive indices and a "hardwired mapping".

$$\mathtt{FR} : \mathbb{N}_{|\mathbf{\Pi}|} \rightarrow \Pi \times \Pi \times \ldots \times \Pi \ .$$

4.2.3 Active structural constraint ADT

The concepts of structural constraint and active structural constraint are very similar, and differ only with regard to markings on primitives (active constraints mark primitives as anchors or open, whereas the former concept does not). Implementation of the active structural constraint will be discussed here, since the implementation of the "inactive" structural constraint can be deduced (simply disregard discussion

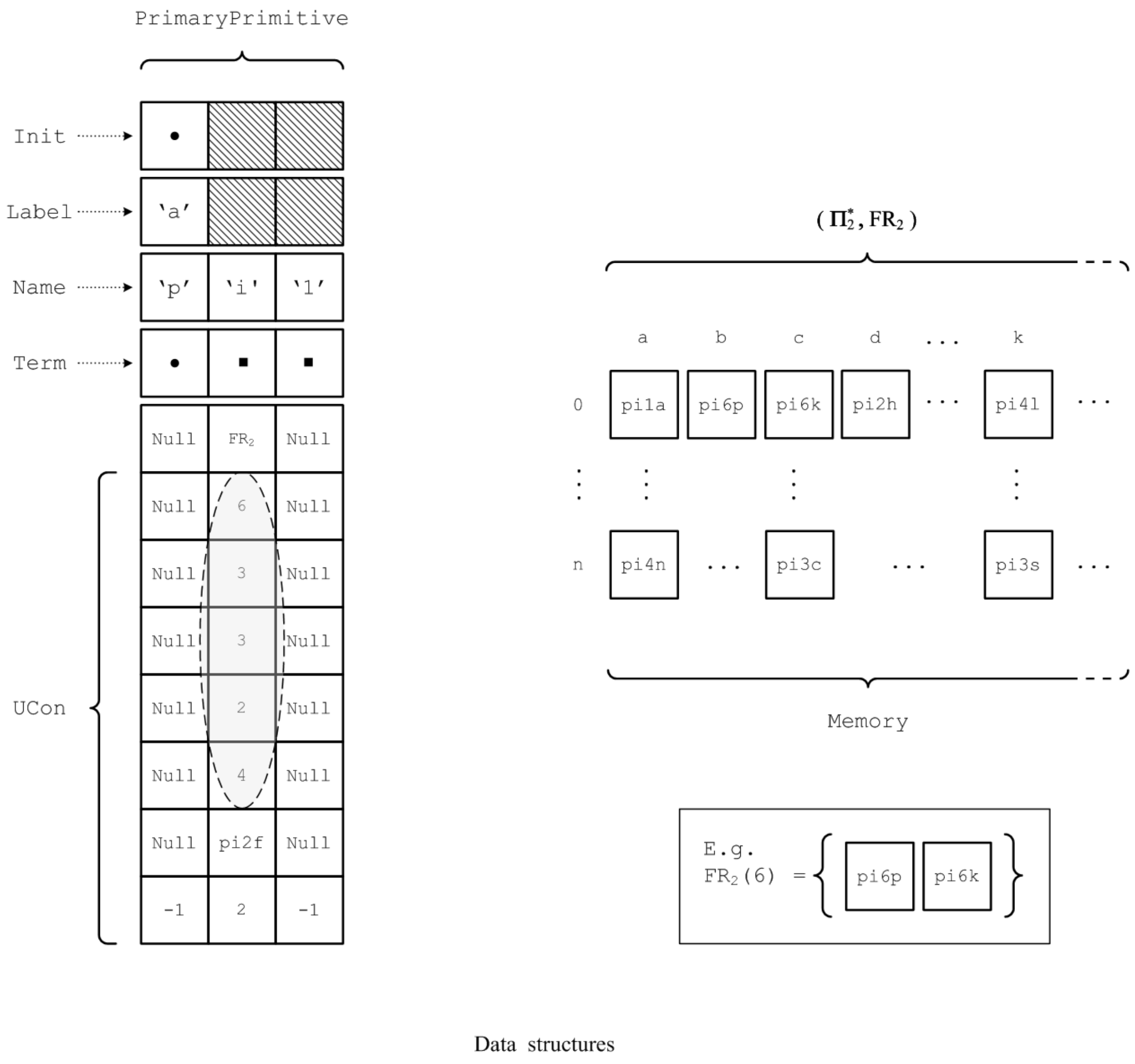


Figure 4.8: Pictorial illustration of the *Primary Primitive* data structure, as an extension of the *Primitive* data structure with the addition of a single two-dimensional array called `UCon`.

related to primitive markings).

An active structural constraint may be thought of as a tuple of unit-constraints, *cf.* [2, Def. 10]. In addition to the tuple of unit-constraints are two relations over the set of pivot primitives, denoted $\overline{\Pi}$. The pivot primitives may be derived from the constraint's tuple of unit-constraints or their diagrams (being the only primitives shown), *cf.* Fig. 4.9 and [2, Def. 11]. One subset is called the set of **anchor primitives**, denoted $\overline{\Pi}^{anc}$ where $\overline{\Pi}^{anc} \subset \overline{\Pi}$, and the other is called the set of **open primitives**, denoted $\overline{\Pi}^{opn}$ such that $\overline{\Pi}^{opn} \subset \overline{\Pi}$.

Since structural constraints are more or less a tuple of unit-constraints, it is important to recognized that the connections between the given pivot primitives of a constraint are also implicit, and so do not represent structs explicitly. Instead the structural constraint specifies a family of structs sharing a structural "backbone".

The active structural constraint ADT is also simple and consists of a single operation, called *active extension*,

$$\texttt{Ext}(\sigma^{w},\ \texttt{ACon})\ ,$$

and is defined in the following way. An active extension, produces a struct that satisfies the given constraint, assembles to the given working struct and adds at least one additional primitive to the working struct.

4.2.4 Active structural constraint data structure

Like the *Struct* data structure, the *Active (Structural) Constraint* data structure is implemented using a queue whose elements are lists of *Primary Primitive's*. Also, like the *Struct* data structure, the lists' position types are `char[ ]`'s such that each *Primary Primitive* is positioned by its own uniquely identifying label.

The implementation of the active structural constraint data structure, also consists

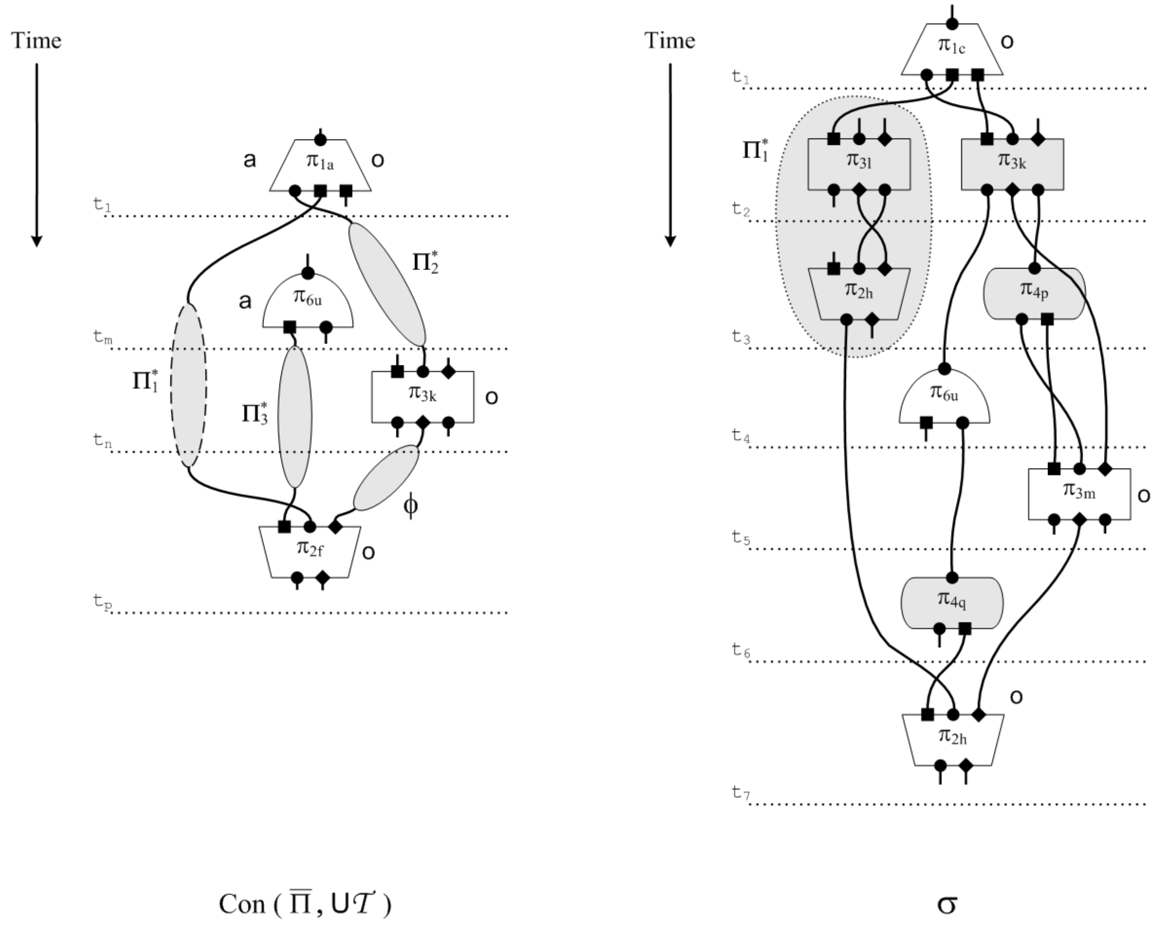


Figure 4.9: Pictorial illustration of a single active structural constraint (*left*) and an active struct (*right*), which satisfies it.

of a table called `Marks`

$$\texttt{Marks} : \texttt{char[ ]} \to \{0,1\} \times \{0,1\} ,$$

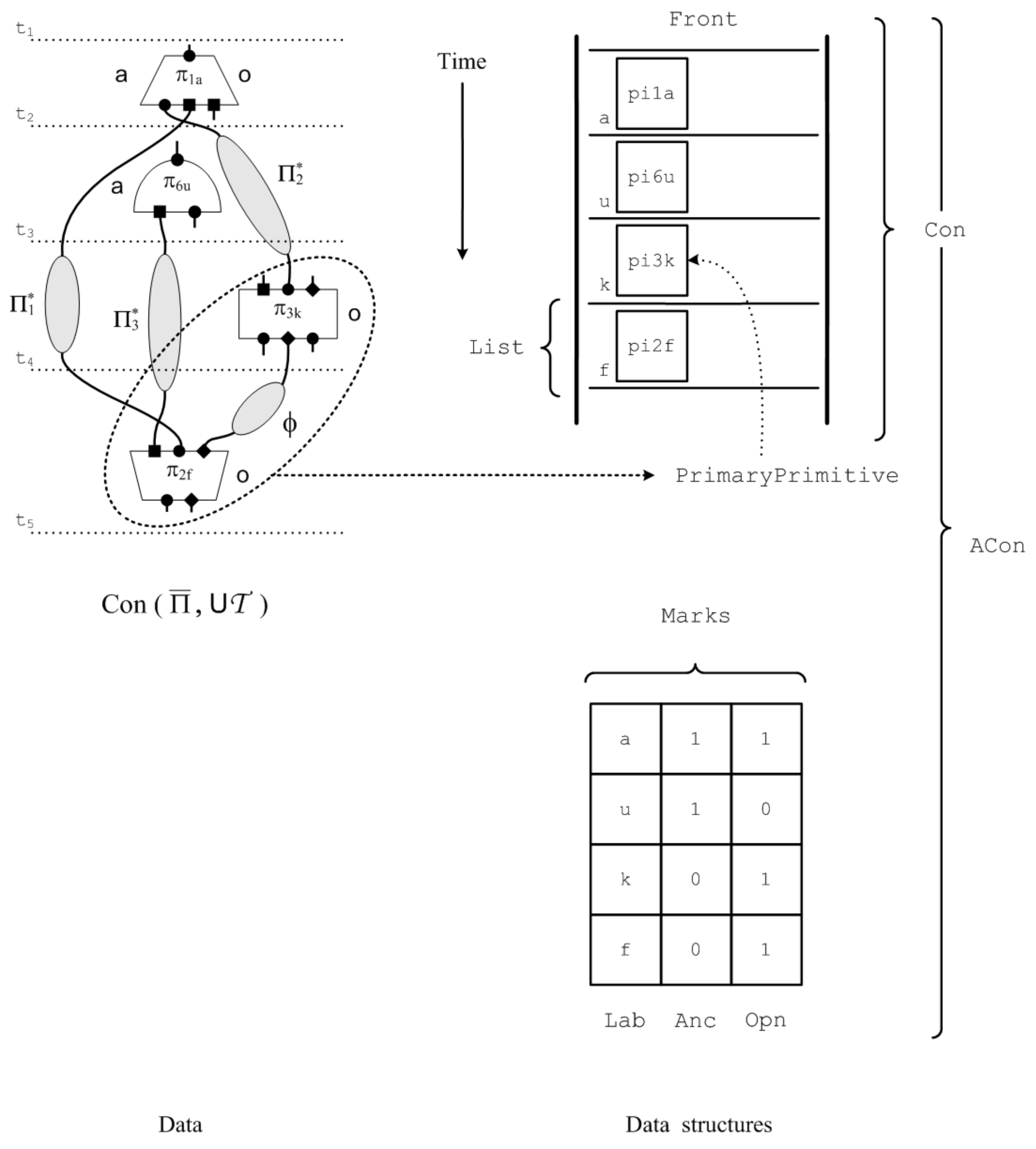
such that, given some *Primary Primitive* label, `Marks` returns one of the following pairs: $\langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle$ and $\langle 1,1\rangle$. The first pair signifies that the primitive is not marked, the second that the primitive is marked as an anchor, the third that the primitive is marked as open, and the last signifies that the primitive is marked as both an anchor and open. Note that since the implementation of non-active structural constraint is accomplished minus the table `Marks`, the *Active Constraint* is defined as a pair of data structures:

$$\texttt{ActiveConstraintADT ACon = } \langle \texttt{ ConstraintADT Con, char[][] Marks } \rangle ,$$

where the non-active structural constraint, Con, is the queue of lists, *cf.* Fig. 4.10 (notice that a dashed ellipse is used to emphasize that a particular unit-constraint is being stored within a *Primary Primitive*, `pi3k`, data structure.)

4.3 Single-level class generating machines

The previous two sections introduced tentative abstract data types and data structures developed to store and manipulate ETS structs and structural constraints. Introduced next is an abstract data type for class representation and a finite state machine used to implement the ETS concept of class generating system.



Lab	Anc	Opn
a	1	1
u	1	0
k	0	1
f	0	1

Figure 4.10: Pictorial illustration of an implementation of the *Active Structural Constraint* ADT, using a queue and lists of *Primary Primitive's*, where the lists *position* their elements according to the *Primary Primitive's* label.

4.3.1 Class representation ADT

A (level 0) Class Representation, $\mathfrak{R}$, consists of three components, c.f. [2, Def. 13], two sets of constituent primitives (referred to as **pivot** and **noise primitives**), and a step-wise **generating system specification**, denoted $\mathscr{G}_{\mathfrak{R}}$. Each step of $\mathscr{G}_{\mathfrak{R}}$ consists of a set of active structural constraints, *cf.* Fig. 4.11 (notice that all the concrete primitive labels are included, along with all the primal subclass labels).

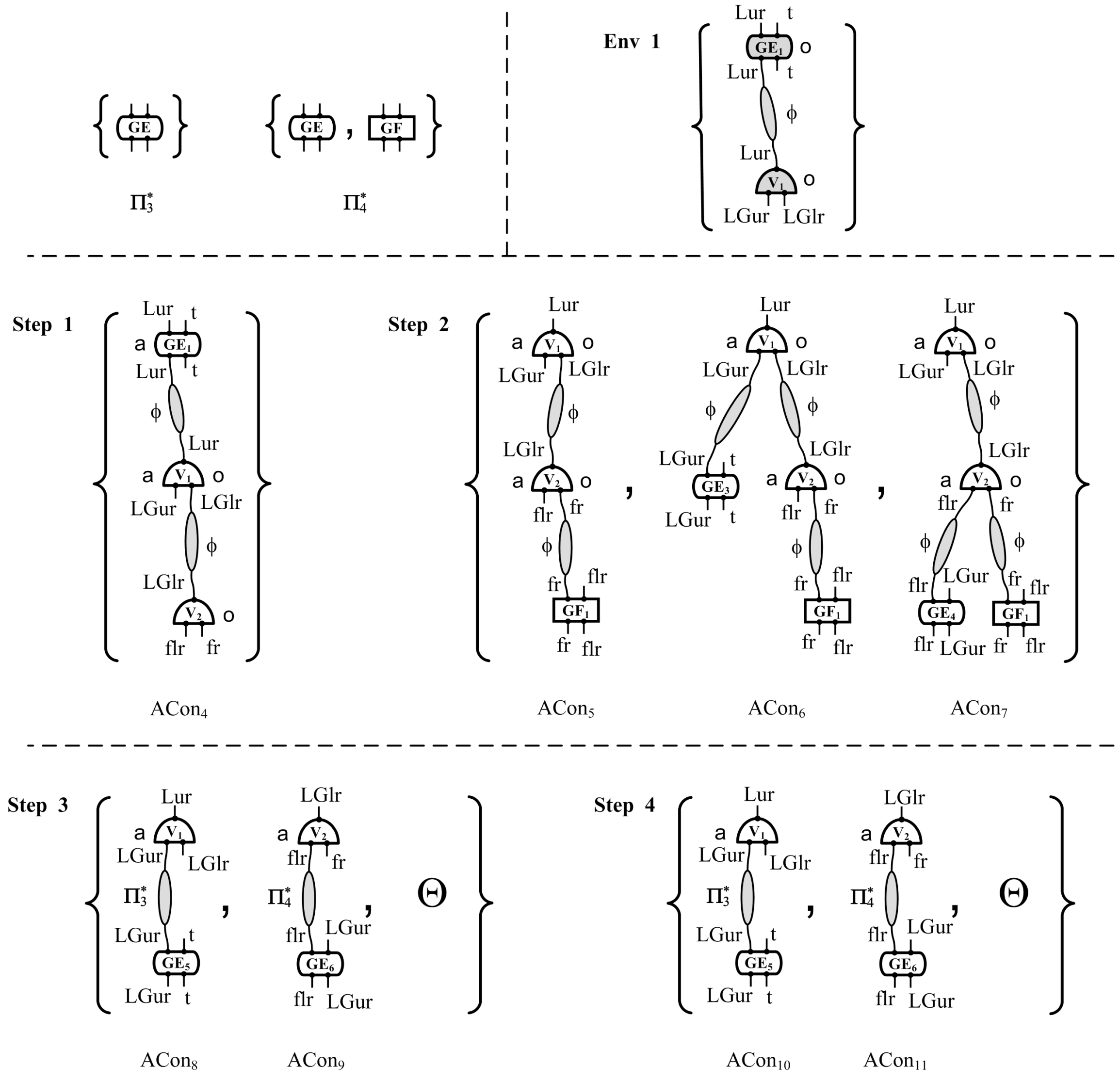


Figure 4.11: Pictorial illustration of the "Final Leg" level 0 class representation.

At least one operation on class representations may be identified, that of **class**

element generation. A class element is not generated per se, but rather an ETS struct is produced. The generation of ETS structs is accomplished via the previously described *Active Extension* operation over *Active Constraints* and a given *(Active) Struct*. The idea is that an entire class element, or *Struct*, may be generated by actively extending the working struct with *Active Constraints* from the given class generating system specification.

4.3.2 Struct generating machines

Finite State Machines are abstract machines that may be described as defining a regular language, such that a string of symbols called a word is written on the machines tape, which is then read, symbol-by-symbol, until the end of the tape is reached. If the machine is in an accept state, when the end of the tape is reached then the language defined by the machine contains the word, otherwise the language does not contain the word and is rejected.

On the other hand *if the machine's tape is viewed as an "output" tape*, we can say the automaton generates a regular language (a set of words). It is along this line of thinking that it was decided to use automata as a model for describing the behavior of the class generating system.

Since, the sets of constituent primitives mentioned above are implicit in the specification of unit-constraints, a single data structure is needed to store the (partial) generating system specification needs. The structure used to implement the concept of class representation the nondeterministic finite state machine (NFSM) with an output tape (instead of an input tape). The NFSM model of a class generating system is defined as the following tuple

$$\langle \Lambda, Q, q_0, \delta, F \rangle$$

and referred to as *Struct Generating Machines for Level 0 Classes cf.* Fig. 4.12, such that:

- The union of the initial working struct σ^w and ϵ with the cartesian product of the *Active Extension* operator $\underset{\text{Ext}}{\triangleleft}$ and the union of the sets of *Active Constraints*, in $\mathscr{G}_{\mathfrak{R}}$, defines the "output" alphabet, denoted Λ

$$\Lambda = \{\sigma^w, \epsilon\} \cup \{\underset{\text{Ext}}{\triangleleft}\} \times \left\{ \bigcup_j \left\{ \text{ACon}_{j,i}(\overline{\Pi}_{j,i}, \mathcal{UT}_{j,i}, \overline{\Pi}^{anc}_{j,i}, \overline{\Pi}^{opn}_{j,i}) \right\}_{i \in I_j} \right\}.$$

- For each element of Q there is a set of *Active Constraints* associated with it that may be written to the output tape, as e.g. $\underset{\text{Ext}}{\triangleleft}$ $\texttt{ACon}_i$, this relationship represents a surjection of states over actions, denoted ω, $\omega : Q \to \Lambda$, similar to that of the generating system specification, i.e.:

$$\mathscr{G}_{\mathfrak{R}}(j) \;\to \left\{ \text{ACon}_{j,i}(\overline{\Pi}_{j,i}, \mathcal{UT}_{j,i}, \overline{\Pi}^{anc}_{j,i}, \overline{\Pi}^{opn}_{j,i}) \right\}_{i \in I_j}.$$

- The state transition relation δ, $\delta : Q \times (\omega(Q) \cup \{\epsilon\}) \to Q$ may be visualized as a directed graph or table, see Fig. 4.12.

- The automata starting state in class generating machines corresponds to the initial environmental step whose action results in the formation of the "initial piece" of the working struct σ^w.

- The automata accept states in class generating machines corresponds to a tentative terminating condition. Since, formally the terminating conditions for class generating systems are not specified, it seems natural to assume that when each step of the generating system specification has been taken, the generating system has since completed the structure it was generating.

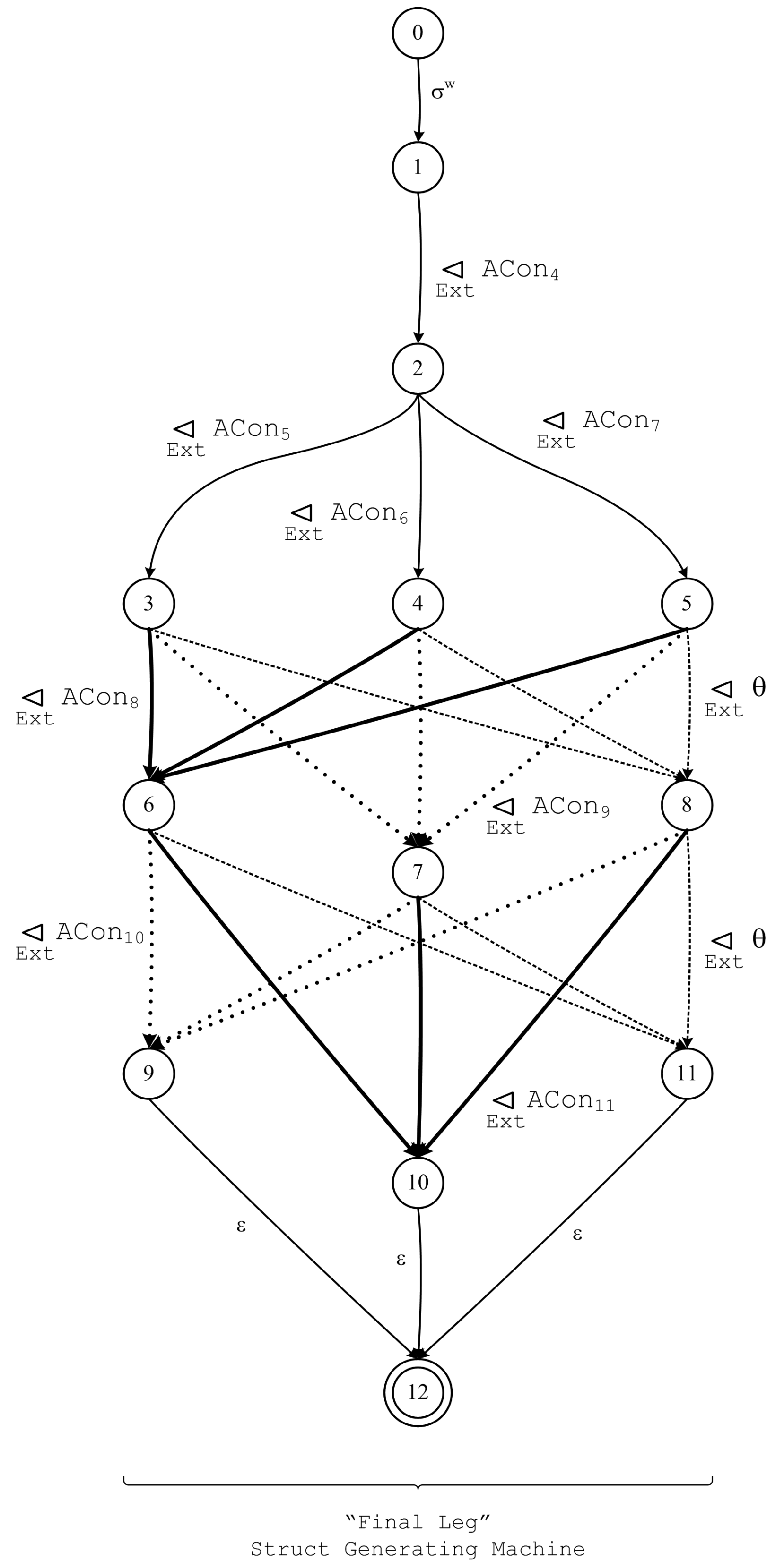


Figure 4.12: Pictorial illustration of the "Final Leg" *Level 0 Struct Generating Machine.*

4.3.3 Class element generation

A single operation is specified over *Class Representations* called *class element generation*, or `Gen`. The operation `Gen` takes a *Struct Generating Machine* `M` and generates a string like the following:

$$\texttt{Gen(SGM M)} = \sigma^w \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_4 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_5 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_8 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_{11}\ ,$$

which represents a series of *active extension* operations (note that these extension operators are associative but not commutative). The result of these active extensions is a class element struct, γ:

$$\begin{aligned}\gamma = \texttt{Gen(SGM M)} &= \sigma^w \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_4 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_5 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_8 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_{11} \\ &\Rightarrow \texttt{Ext(Ext(Ext(Ext(}\sigma^w\texttt{, ACon}_4\texttt{), ACon}_5\texttt{), ACon}_8\texttt{), ACon}_{11}\texttt{)} \\ &\Rightarrow \mathcal{A}(\texttt{T}_1\texttt{, T}_2\texttt{, T}_3\texttt{, T}_4)\end{aligned}$$

Moreover, the environmental actions are incorporated into the implementation of *struct generating machines*, and it is accomplished by having each machine share the same "output" tape, e.g.

$$\texttt{Gen(SGM M)} = \sigma^w \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_4 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_{\mathfrak{E}} \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_5 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_8 \underset{\text{Ext}}{\triangleleft} \texttt{ACon}_{11}\ .$$

4.4 Spatial instantiation

In [2, p. 12], the process of spatial instantiation is described as follows:

> What is the relation between a "physical" object and its event-based representation? We hypothesize that a "physical" object is a physical

instantiation of its event-based (informational) representation, i.e., as each event is being played out, the physical "flesh" is automatically being put on the informational "bone".

The following implementation of the spatial instantiation process, takes the phrase "as each event is being played out, the physical 'flesh' is automatically being put on the informational 'bone"', literally.

To implement a process that fleshes out the informational "bone" a deterministic system was envisioned, such that for each primitive read in a struct one or more 3D models are transformed. In contrast to the single-tape NFSM's used to model struct generation, a dual-tape deterministic finite state automata is used such that one tape serves as input and the other as output for the automata. These machines are called transducers, and are considered to compute relations between two formal languages [5]. However, instead of computing relations between formal languages, a computation of the relationship between an ETS struct and a physical object is desired. Remember that the Bubble Man informational structure is a struct generated from class description, and so is composed of many primal processes, which appear, disappear, grow and interact with each other. For each primal process a finite state transducer is defined. (Examples of finite state transducers and their run-times are shown in Fig.'s 4.13–4.16.)

The interface between the Bubble Man struct and the system of "Bubbles", is modeled as a system of transducers, independently translating a single struct into the various series of rendering and growth instructions that create and transform each primal process. These instructions form an implementable interface that is implemented by each "Bubble" using "Microsoft's XNA content pipline". Also note that, the transducers forming the Event-Object interface are made to share a single input tape, which yields an interesting *one-to-many relationship*[2] between temporal representation and the process of spatial instantiation.

4.4.1 Finite state transducers

Given a particular primal process, e.g. *Head*, a **finite state transducer** may be defined as the tuple

$$\langle Q, q, \Pi, \Lambda, F, \delta, \omega \rangle$$

- The set of states Q correspond to the several states of deformation for *Head*, *cf.* Fig. A.4.
- The set q, a subset of Q are the initial states and correspond to the primitive(s), which instantiate the primal process, e.g. those primitives that represent a division of the single *Head-Neck* primal process into the two primal processes *Head* and *Neck*.
- The input alphabet, Π, is the set of applicable concrete primitives, such that they may instantiate or deform the *Head*.
- The output alphabet, Λ, may be thought of as an interface or "instruction set" for Bubble Man primal process simulators, consisting of three operations/instructions: `Grow`, `Render`, and a null/no-op operation, denoted ϕ, such that further action by the simulator is suspended, e.g. when a primal process divides simulation of that process is no longer required, but rather another two simulators are needed to simulate the resulting two processes.
- The set F is a subset of Q, called the final states, which correspond to either the indefinite suspension of primal process simulation or its indefinite rendering,

[2]This sort of relationship has exciting implications in biology, with regard to DNA and gene expression *cf.* [6].

i.e. the primal process can no longer change, but will continue to exist, or continue to be simulated in its current state of deformation.

- The transition relation, mapping a given state and a primitive to another state, is defined as

$$\delta : Q \times \Pi \to Q$$

- The output function, mapping a given state to a particular output symbol, is defined as

$$\omega : Q \to \Gamma$$

4.4.2 Spatial instantiation example

The following figures illustrate the transduction of the Bubble Man Final Arm struct γ. The first figure, Fig. 4.13, is a pictorial illustration of four transducers used to implement the event-object interface used to spatially instantiate structs. Surrounding each transducer are shapes that are spatially representative of each primal process. Note that the shapes appearing before and after short arrows represent a deformation process, i.e. growth.

In the following example, a struct γ is given and it is spatially instantiated by the system of transducers, which invoke operations specified in Γ and implemented by each of the three "bubbles" shown in the "Object Environment" of Fig.'s 4.14–4.16. These three figures, illustrate *four transductions* of the final three primitives in γ. The results of each transduction are realized by the various "bubbles" comprising the "Bubble Man's" right arm.

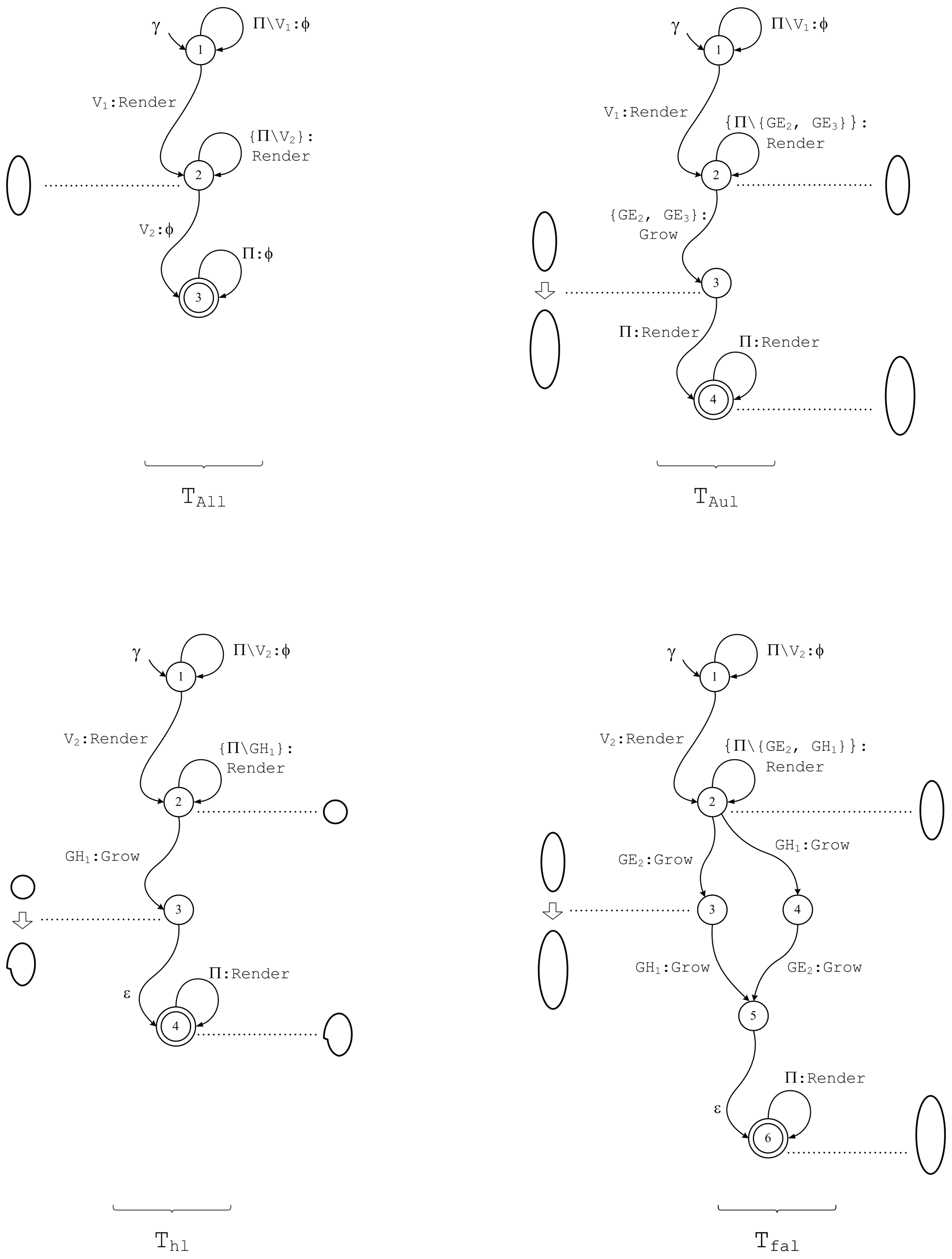


Figure 4.13: Pictorial illustration of four level 0 time-space transducers corresponding to four primal processes: All, Aul, hl and fal.

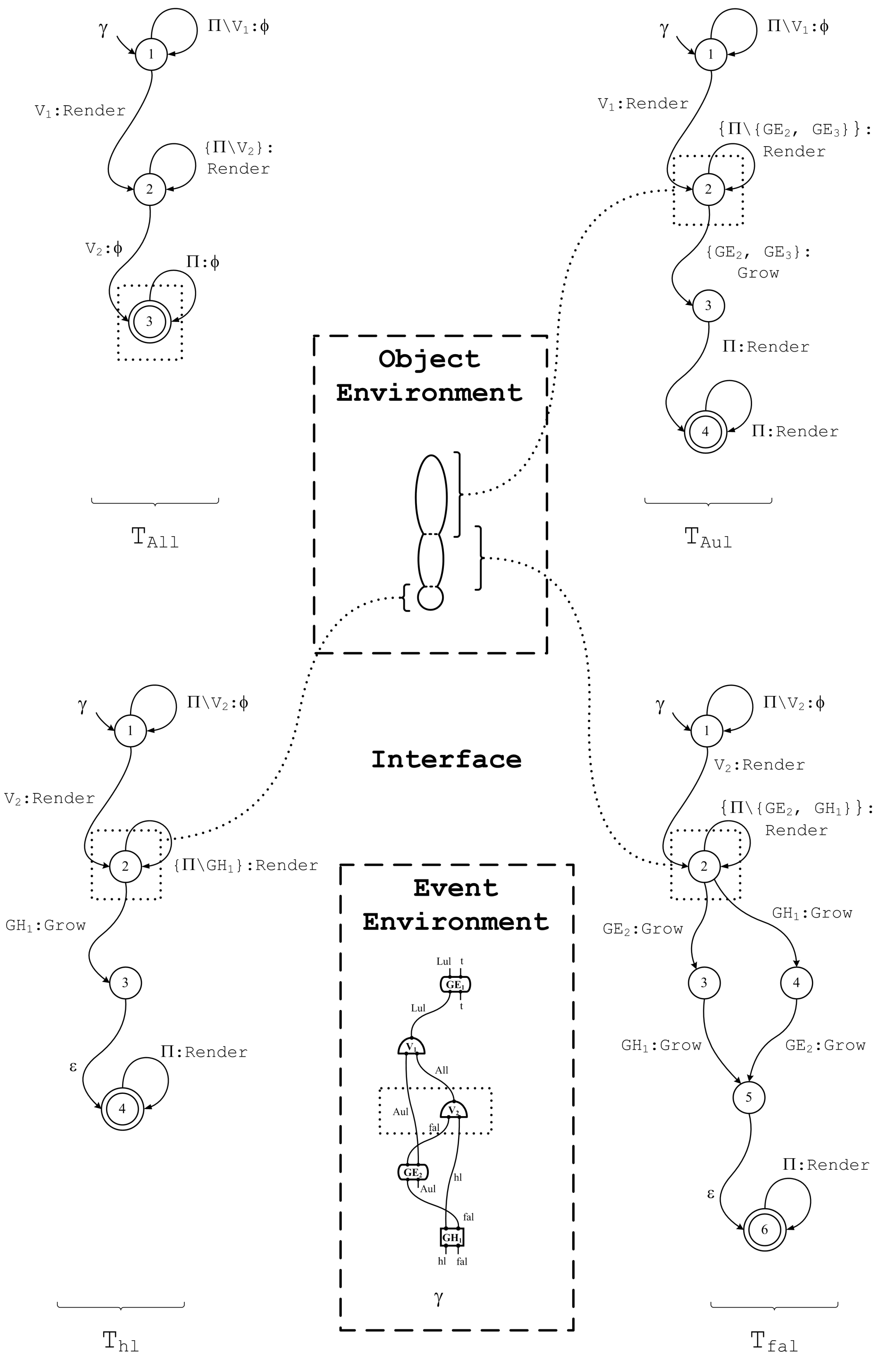


Figure 4.14: Pictorial illustration of four transducers simultaneously transducing the third primitive of the level 0 struct γ, $\mathtt{V}_2$.

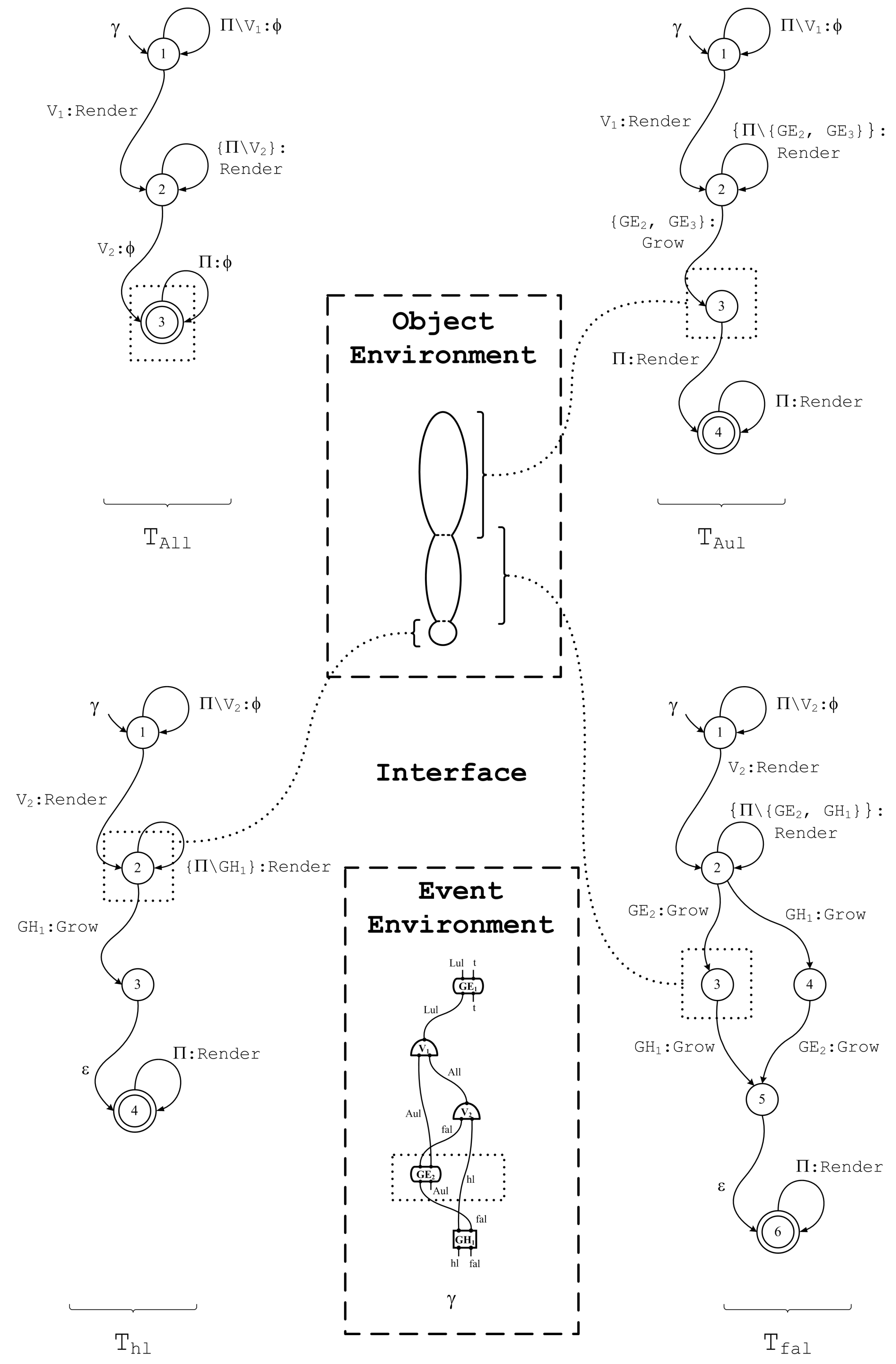


Figure 4.15: Pictorial illustration of four transducers simultaneously transducing the fourth primitive of the level 0 struct γ, $\mathtt{GE}_2$.

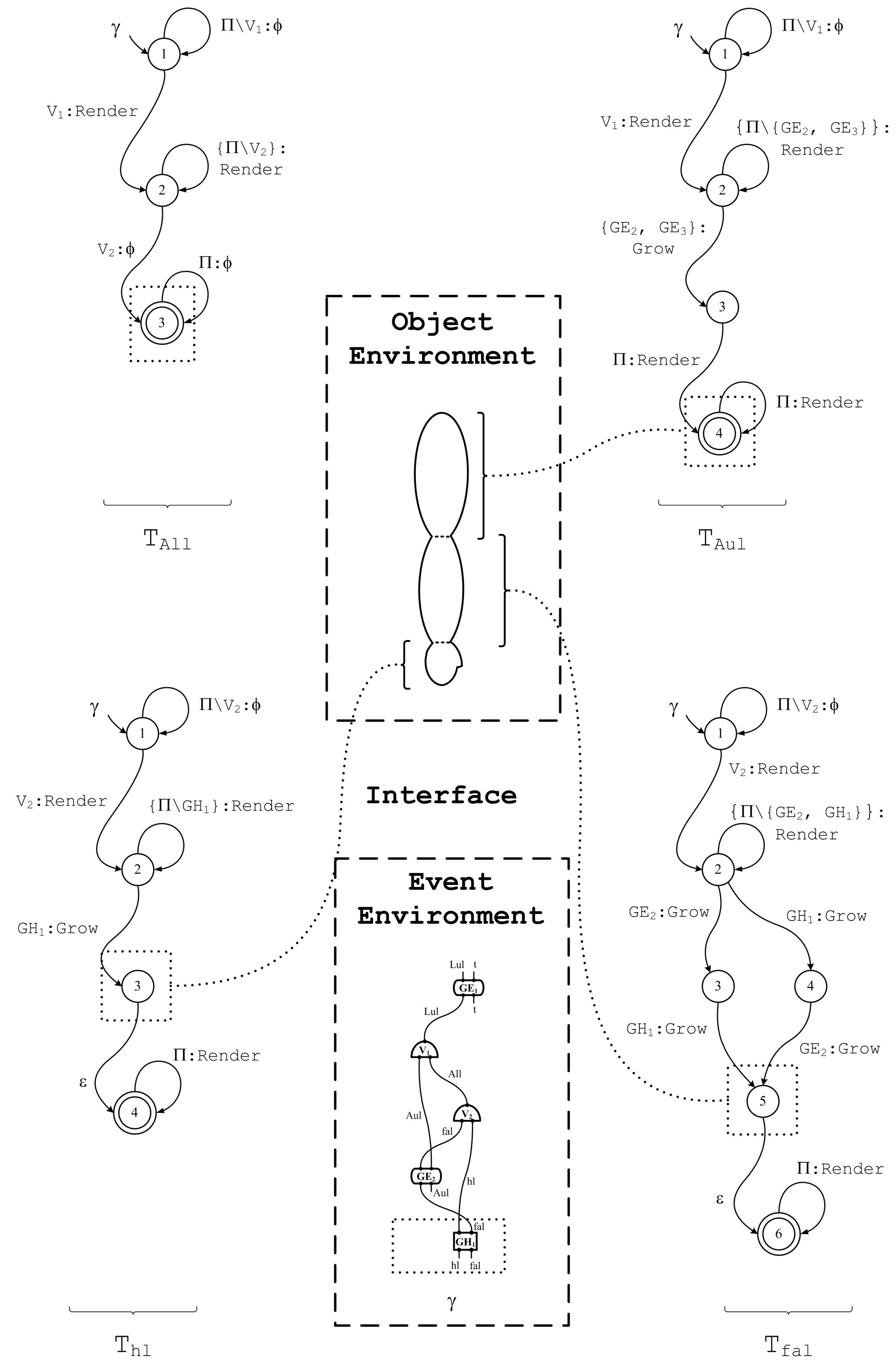


Figure 4.16: Pictorial illustration of four transducers simultaneously transducing the fifth primitive of the level 0 struct γ, $\mathtt{GH}_1$.

Chapter 5

Conclusion

5.1 Summary

An implementation of a system of ETS class generating systems was accomplished by modeling each class generating system and its specification as a nondeterministic finite state machine, which shares the same *output tape* as the other machines in the system. Implementation of the process of spatially instantiating an ETS struct was accomplished by modeling the interface between the concepts of object environment and event environment as a system of finite state transducers that share the same input tape. Essentially, the *shared output tape* of the class generating systems is spliced to the *shared input tape* of the finite state transducers, so that the struct produced by the system of class generating systems is transformed into many sequences of instructions (i.e. programs). The output tape alphabet defined by the transducers, may be thought of as an "instruction set" for the "programs" produced by the transducer. Finally, the "instruction set" defined by the transducers is implemented by several programmable objects, each representing a particular primal process. The result is a system of programmable objects that execute the programs produced by the transducers.

In order to *apply* these developments, I have modified a simple illustrative example of an ETS (multi-level) class, developed for [2] and referred to as the "Bubble Man" class. Essentially, a system of the single-level class generating systems, from which, higher-level Bubble Man class generating system is *recursively* defined, systematically generated structs representing various Bubble man class instances. These structural representations were then spatially instantiated via a system of finite state transducers (whose output tapes consisted of "instructions" such as "grow", "render" and "no-op"), such that the result of each transducer, given a structural representation of the Bubble Man, is a sequence of growth and render instructions. Each such growth-render sequence, is executed by an object that manipulates a 3D computer model of a sphere. The result is a system of "bubbles" that combine to form a 3D representation of the Bubble Man instance.

In conclusion, it appears that ETS temporal representation is more basic than spatial representation, since such temporal representation records nothing more than the interactions between a system of objects. Also, in terms of "spatial instantiation", ETS temporal representation encodes sufficient information to reconstruct an entire system of objects, from which, a spatial representation may be generated via the process of spatial instantiation.

5.2 Contribution

The main contribution of this thesis work is a conceptual illustration of two difficult to understand concepts, such that, ETS structs in relation to spatial representation are more basic. Secondly, ETS temporal representation may be "spatially instantiated" into a more familiar form of representation.

In more concrete terms, this thesis work extended the 2D illustrative examples of class representation, in [2, Sec. 8], referred to as the class of "2-dimensional bubble men", to set of class representations more suitable for implementation. More specifically, the class representations taken from [2] were incomplete, since they were not intended for implementation. This required a significant amount of effort to correct and modify some of the class representations in order to obtain class generating system specifications that would ultimately generate structs correctly and also interact with each other appropriately. Also, rather than reproduce the 2D shapes of "bubble men" given in [2, Sec. 8], 3D models were generated. The 3D models were generated systematically by implementing a system of 3D models called "bubbles" using the "XNA framework content pipeline" and a system of finite state transducers for spatially instantiating the ETS structs.

In order to obtain structs of bubble men, an initial set of ADT's and data structures for primal classes, primitives, structs, and constraints were implemented. The "bubble man" class generating system specifications obtained by extending the original set in [2, Sec. 8], were also implemented, as struct generating machines. Then, using a sequence of queues, a bubble man structural representations were generated for spatial instantiation. Two structs generated by the software system and their spatial instantiation into 3D models are shown in the following figures, Fig.'s 5.1 and 5.2.

Notice the significant difference in complexity of the structs generated between these figures. The first figure shows that the struct represents a "smaller" Bubble Man instance, and the second struct represents a "bigger" Bubble Man instance, also see Fig. 5.3 for a closer comparision. (The second "Bubble Man" has a larger torso, longer arms, longer feet and larger and longer legs.) The reason the second "Bubble Man" is bigger is apparent from the struct, there are more growth events encoded in the second struct than in the first.

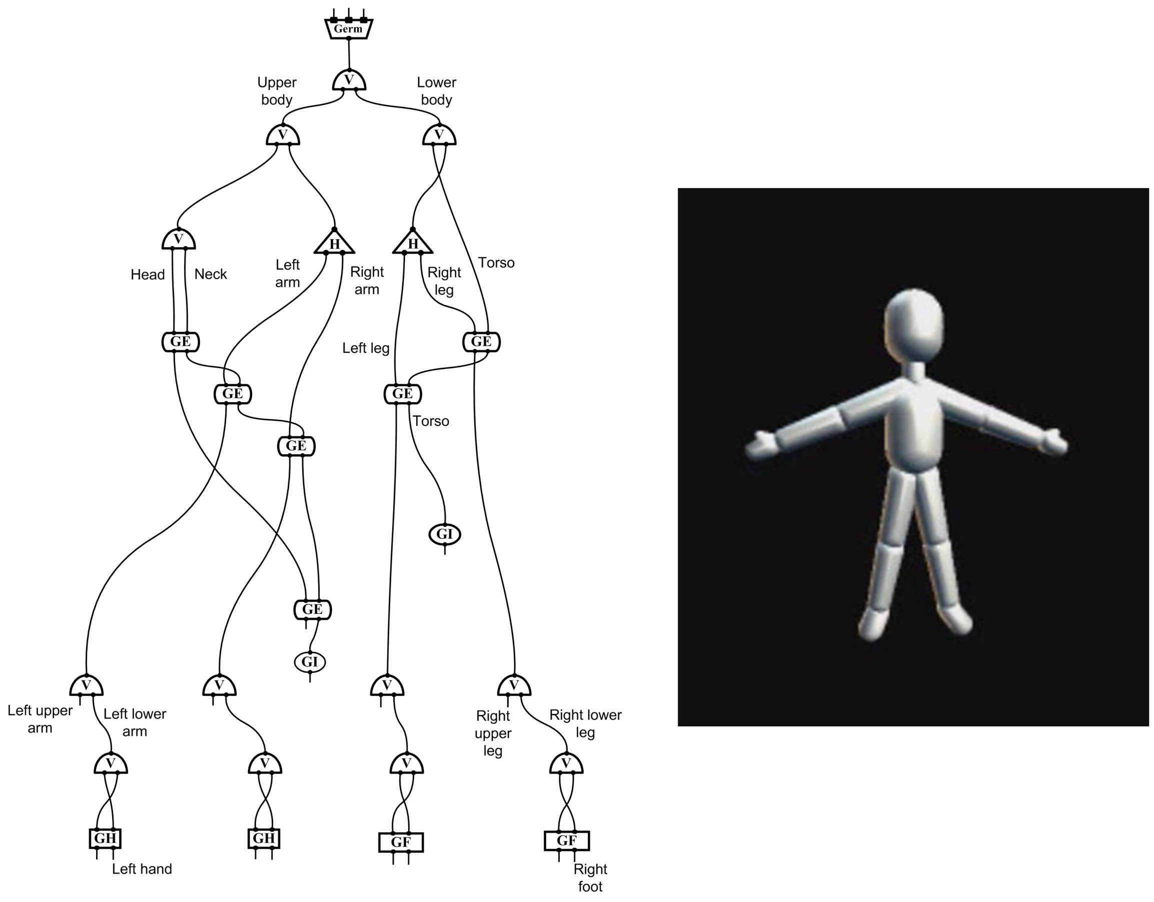
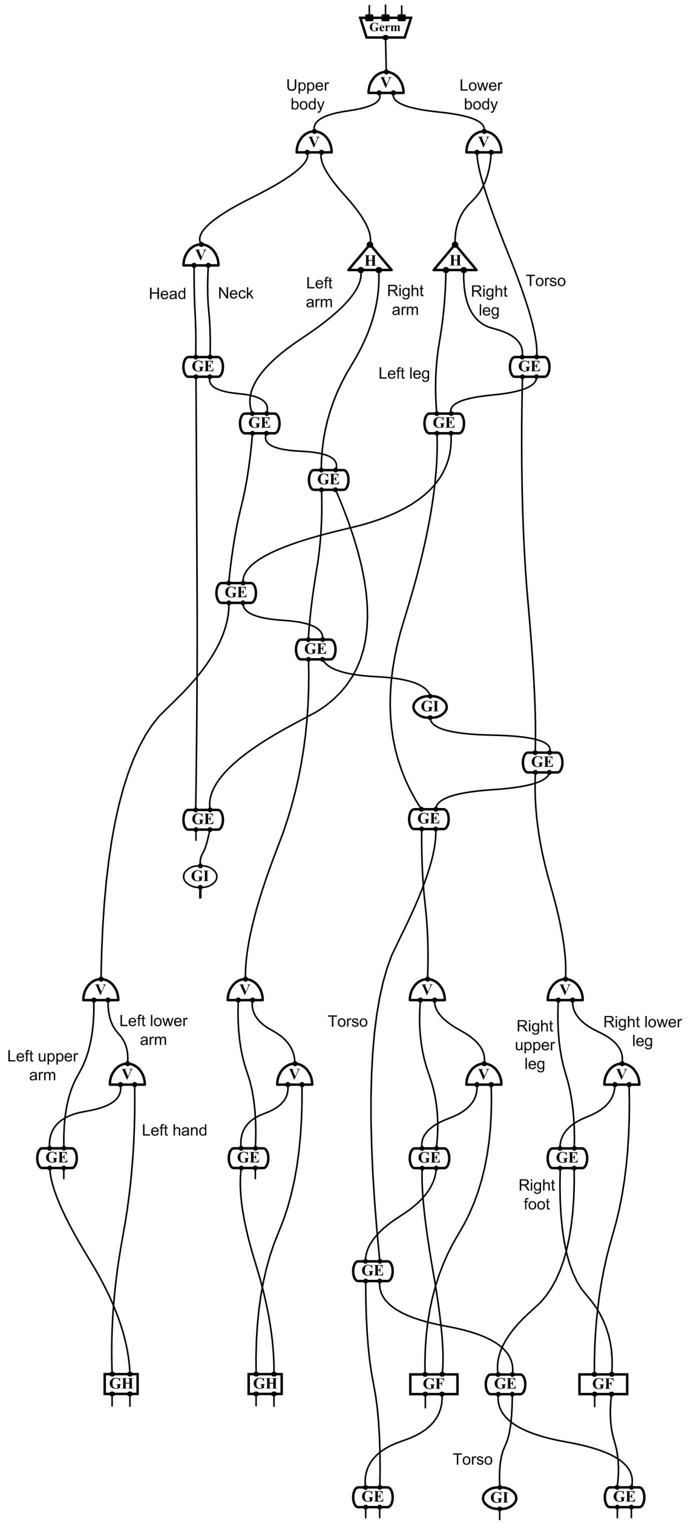


Figure 5.1: Pictorial illustration of a “Bubble Man” struct and its spatial instantiation



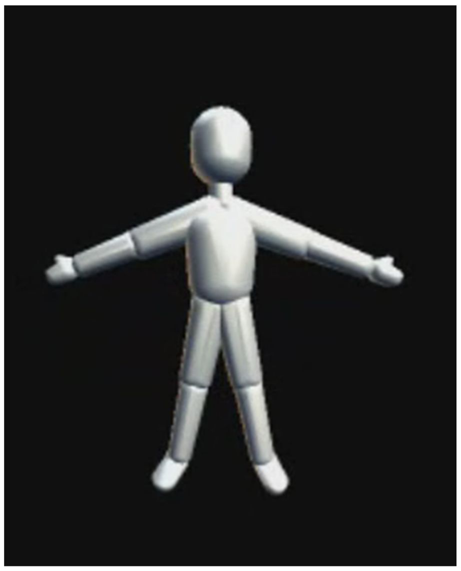


Figure 5.2: Pictorial illustration of another (larger) "Bubble Man" struct and its spatial instantiation.

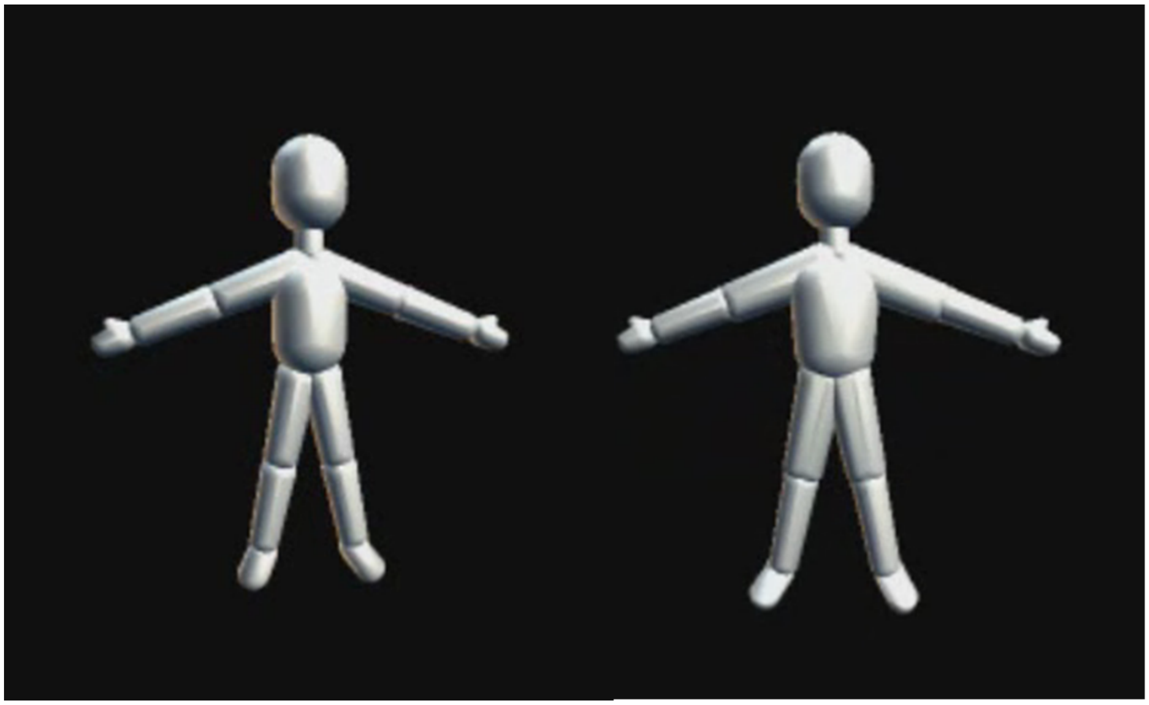


Figure 5.3: Pictorial illustration of the two “Bubble Men” spatial instantiations.

5.3 Future work

In this thesis, the “Bubble Man” class is referred to as a “system of classes”, however, the ETS formalism contains a concept referred to as “higher-level classes” (or k-level classes, $k >= 2$). These classes are recursively defined in terms of lower-level classes, such that, a two-level (level 1) class is defined in terms of single-level (level 0) classes, a three-level (level 2) class is defined in terms of two-level (level 1) classes, and so on. For each level, e.g. k, of class representation there are level k structs, structural constraints, class generating systems and other related level k concepts defined. Future work is required in this area of ETS, and should not only involve the design and implementation of ADTs and data structures but should also involve a study into how and what it means to spatially instantiate higher-level (level k) structs.

Besides further refinement and implementation of the ETS formalism, the area for future work most critical to the dissemination of the ETS framework is (of course) concrete applications. The list of potential application areas contains a broad spectrum of exciting areas in science. More specifically, the application areas for ETS range from machine learning and pattern recognition [7, 8] to decision and risk analysis[9] to developmental biology [6].

Bibliography

[1] H. A. Simon, *The Sciences of the Artificial.* MIT Press, 1988.

[2] L. Goldfarb, D. Gay, O. Golubitsky, D. Korkin, and I. Scrimger, "What is a structural representation? a proposal for an event-based representational formalism," tech. rep., Faculty of Computer Science, UNB, Fredericton, New Brunswick, Canada, 2007. http://www.cs.unb.ca/∼goldfarb/ETSbook/ETS6.pdf.

[3] L. Goldfarb, "On the concept of class and its role in the future of machine learning," tech. rep., Faculty of Computer Science, UNB, Fredericton, New Brunswick, Canada, 2007.

[4] A. N. Whitehead, *Science and the Modern World.* Cambridge University Press, 1945.

[5] M. Sipser, *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[6] L. Goldfarb and B. R. Peter-Paul, "Integrating development, evolution, and cognition via a novel formalism for structural representation." In preparation, 2008.

[7] S. Falconer, D. Gay, and L. Goldfarb, "ETS representation of fairy tales," in *Proc. ICPR 2004 Satellite Workshop on Pattern Representation and the Future of Pattern Recognition* (L. Goldfarb, ed.), (Cambridge, UK), August 2004.

[8] I. Scrimger, "Structural representation of the game of Go," Master's thesis, Faculty of Computer Science, UNB, 2007.

[9] L. Goldfarb, I. Scrimger, and B. R. Peter-Paul, "ETS as a tool for decision modeling and analysis: Planning, anticipation, and monitoring," tech. rep., Faculty of Computer Science, 2007. Awaiting publication in J. Risk and Decision Analysis, http://www.cs.unb.ca/∼goldfarb/conf/DecisionRisk.pdf.

Appendix A

Appendix: Other class representations

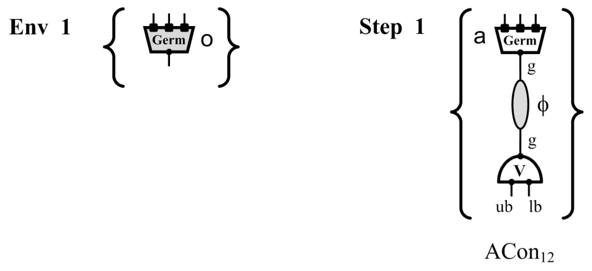


Figure A.1: Pictorial description of $\mathfrak{R}_1$, the level 0 class representation of **Initial Division**, also denoted $\mathfrak{C}_1$.

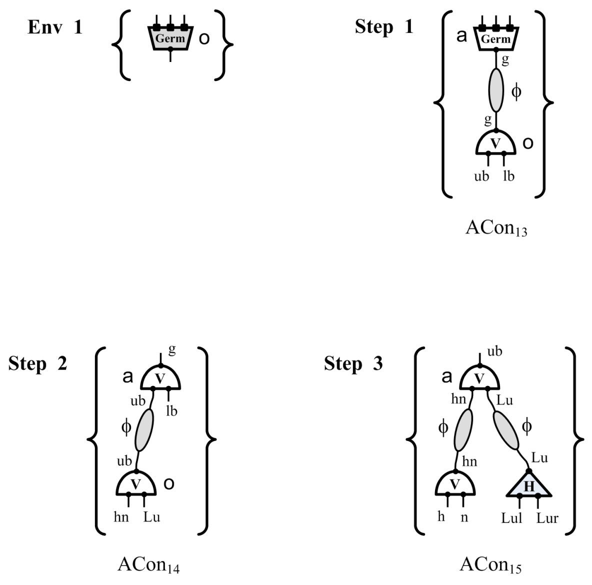


Figure A.2: Pictorial description of $\mathfrak{R}_2$, the level 0 class representation of **Upper Part**, also denoted $\mathfrak{C}_2$.

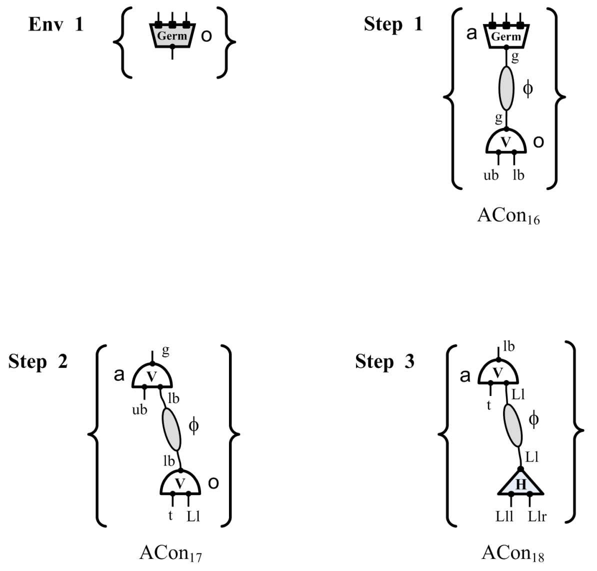


Figure A.3: Pictorial description of $\mathfrak{R}_3$, the level 0 class representation of **Lower Part**, also denoted $\mathfrak{C}_3$.

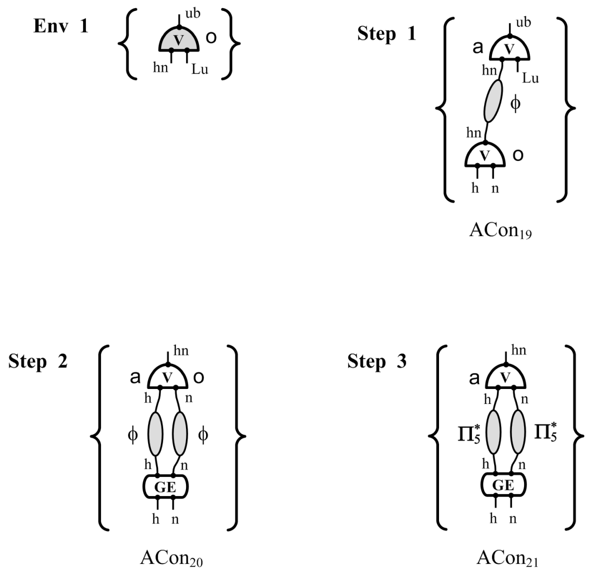


Figure A.4: Pictorial description of $\mathfrak{R}_8$, the level 0 class representation of **Head**, also denoted $\mathfrak{C}_8$.

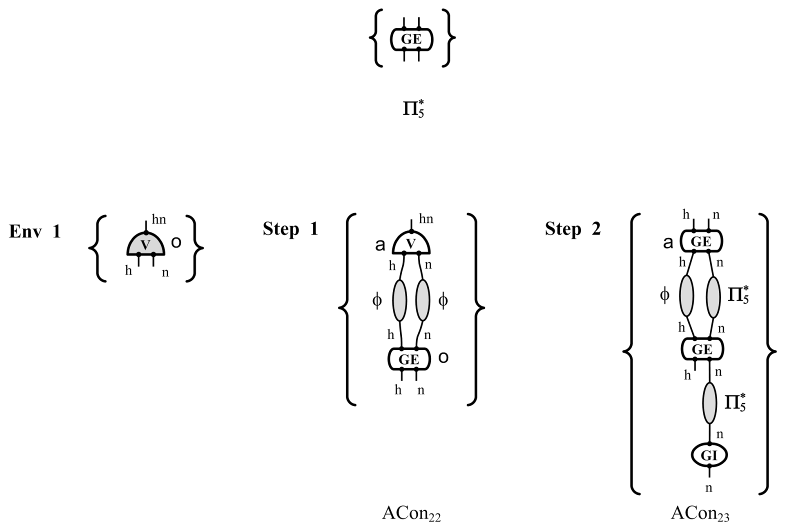


Figure A.5: Pictorial description of $\mathfrak{R}_9$, the level 0 class representation of **Neck**, also denoted $\mathfrak{C}_9$.

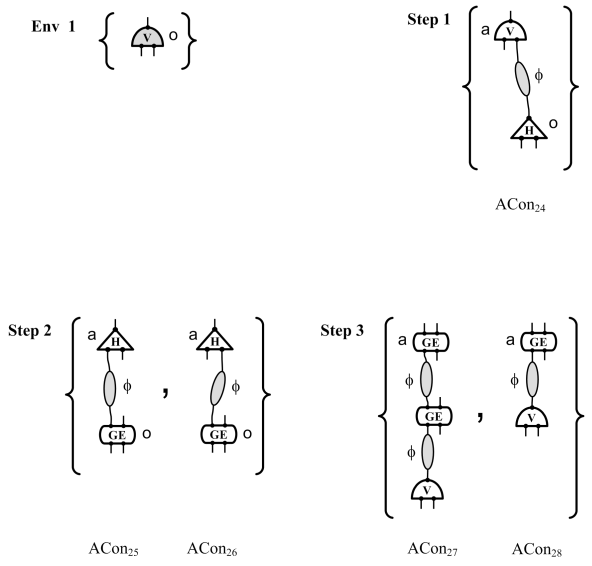


Figure A.6: Pictorial description of $\mathfrak{R}_5$, the level 0 class representation of **Proto Limb**, also denoted $\mathfrak{C}_5$.

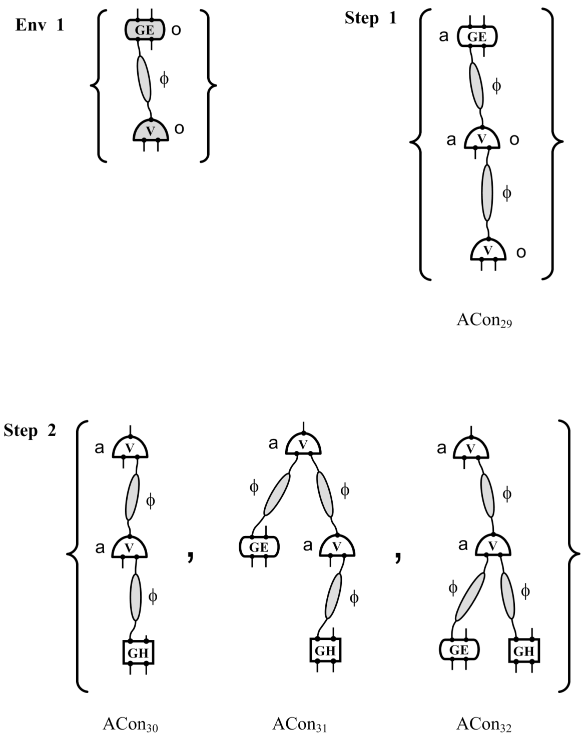


Figure A.7: Pictorial description of $\mathfrak{R}_6$, the level 0 class representation of **Final Arm**, also denoted $\mathfrak{C}_6$.

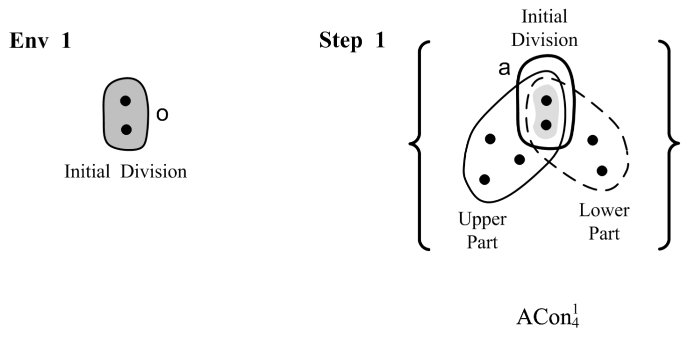


Figure A.8: Pictorial description of $\mathfrak{R}_1^1$, the level 1 class representation of **Proto Body**, also denoted $\mathfrak{C}_1^1$.

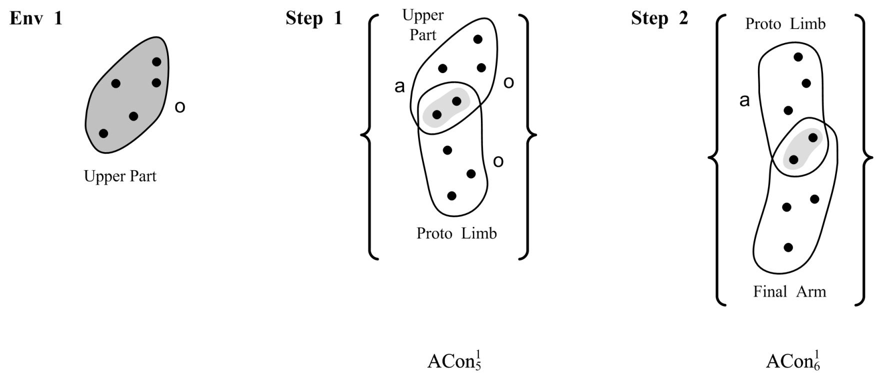


Figure A.9: Pictorial description of $\mathfrak{R}_3^1$, the level 1 class representation of **Arm**, also denoted $\mathfrak{C}_3^1$.

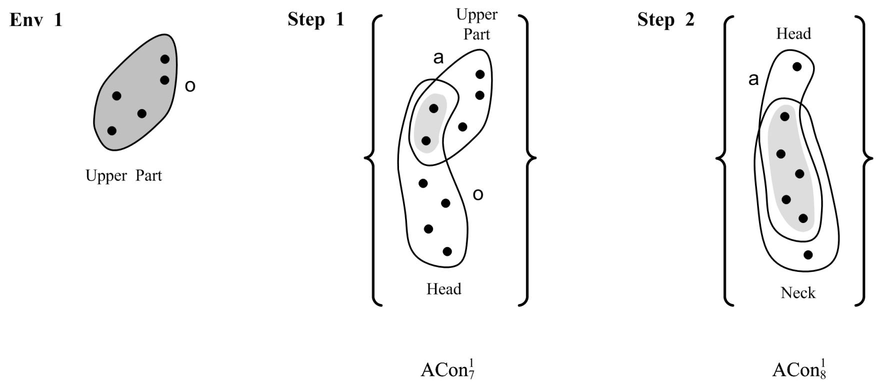


Figure A.10: Pictorial description of $\mathfrak{R}_6^1$, the level 1 class representation of **Head-Neck**, also denoted $\mathfrak{C}_6^1$.

Appendix B

Appendix: Algorithms

B.1 Struct Link algorithm

The first operation Struct Link, or SL, is a modification of CL, such that given the same four-tuple, e.g. $\langle \pi_{ia}, u_i, \pi_{jb}, v_j \rangle$, SL will only store a class link between the primitives if it does not invalidate the struct, see ADT and Alg. B.1. In this algorithm *Ibid.*, besides disallowing the multiple connections from a single primal process, temporal order is considered, such that if the second primitive, of the four-tuple, is an event which occurred earlier (or in parallel to) the first primitive, then the struct link is invalid. Note that labels are assumed to be unique[1] in a given struct (each primitive in the struct has a unique label), also the operation SL does not add primitives to a struct rather it states their interconnections.

```
SL(LinkedPrimitiveADT P, int u, LinkedPrimitiveADT Q, int v,
StructADT S)
LinkedPrimitiveADT p, q ⇐ Null
for $i$ ⇐ 1 to Size(S.Queue) do
```

[1]This is a reasonable assumption since labeling is accomplished during the sensing process, if the same (exact) event is observed at two different times then there is a problem with the sensor.

```
    List list $\Leftarrow$ Front(S.Queue)
    Dequeue(S.Queue)
    Enqueue(list, S.Queue)   //S.Queue is never emptied!
    bool ll $\Leftarrow$ $false$   // flag for parallel events
    **if** p = Null **then**
      p $\Leftarrow$ Retrieve(P.Label, list)
      **if** p $\neq$ Null **then**
        ll $\Leftarrow$ $true$
    **if** p $\neq$ Null, ll $= true$, **and** q = Null **then**
      q $\Leftarrow$ Retrieve(Q.Label, list)
  **if** p $\neq$ Null, **and** q $\neq$ Null **then**
    **if** p.Output[u][1] = Null **and** q.Input[v][1] = Null **then**
      CL(p, u, q, v)
```

B.2 Primitive attachment algorithm

The second operation, also shown in Fig. 4.6, is the attachment of primitives to a struct. The following algorithm, given two primitives, e.g. P and Q, first generates the struct links[2] between the given primitives:

```
  int[ ][ ] links
  int indx $\Leftarrow$ 1
  **for** $u$ $\Leftarrow$ 1 to Length(P.Term) **do**
    **for** $v$ $\Leftarrow$ 1 to Length(Q.Init) **do**
      **if** P.Term[u] $\equiv$ Q.Init[v] **then**
```

[2]Plurality of struct links is used here, but it ultimately depends on the specification of the "sensor" (software system) applying this operation, one sensor might be designed to observe all possible connections between two primitives, or it may be restricted to observing particular kinds of "connections" (i.e. primal class elements).

```
        links[indx] $\Leftarrow$ { u, v }
        indx++
```

Then, if the queue is empty the first primitive is added to a list and pushed onto the queue, followed by the second primitive:

```
for $i$ $\Leftarrow$ 0 to Length(links) do
  SL(P, links[i][1], Q, links[i][2])
List l, ll $\Leftarrow$ new lists
Insert(P, P.Label, l)   //Insert P at position P.Label
Insert(Q, Q.Label, ll)   //Insert Q at position Q.Label
Enqueue(l, S.Queue)
Enqueue(ll, S.Queue)
```

In the case where the first primitive is already present in the structure, then this primitive is located. The struct links generated[3] between the first and the second primitive, still apply for the primitive already present in the struct.

```
LinkedPrimitiveCDT p, q $\Leftarrow$ Null
for $i$ $\Leftarrow$ 1 to Size(S.Queue) do
  List list $\Leftarrow$ Front(S.Queue)
  Dequeue(S.Queue)
  Enqueue(list, S.Queue)   //S.Queue is never emptied!
  if p = Null then
    p $\Leftarrow$ Retrieve(P.Label, list)
  if q = Null then
    q $\Leftarrow$ Retrieve(Q.Label, list)
if p $\neq$ Null and q = Null then
```

[3]It is not necessary to check if a struct link may be added between two primitives since the design of (relevant) sensors should specify when and where connections are added between primitives. In other words, struct links are not incidental, but appear by specification of the sensor.

```
    for $j \Leftarrow$ 1 to Length(links) do
      SL(p, links[j][1], Q, links[j][2])
    List l $\Leftarrow$ new list
    Insert(Q, Q.Label, l)   //Insert Q at position Q.Label
    Enqueue(l, S.Queue)
```

In the case where both primitives are already present in the structure, then attachment is reduced to a series of SL operations.

```
  for $j \Leftarrow$ 1 to Length(links) do
    SL(p, links[j][1], q, links[j][2])
```

In the case where both primitives are not present in the struct and the struct is not a null struct, then the primitives are not attached, since the result of the operation would lead to an inconsistent view of reality.

B.3 Single-level substruct algorithm

Given two structs σ_1 and σ_2, implementation of the substruct operation between these to structs, e.g. $\sigma_1 \Subset \sigma_2$, involves iteration of the first struct data structure *at most once* and of the other *at least once*, such that each primitive of the "substruct" (σ_1) is checked against those of "superstruct" (σ_2). If a matching primitive is found in σ_2, then the struct links stored in the primitive of the first struct are tested against those stored in that primitive of the second struct (see Alg. B.3.) If there is at most one primitive or one struct link, from the first struct, not found in the second struct, then σ_1 is not a substruct of σ_2 and the algorithm immediately terminates and returns false. Otherwise the first struct is a substruct of the second struct, and the substruct operation returns true.

```
  Substruct(StructADT S, StructADT T):bool
```

```
for $i \Leftarrow 1$ to Size(S.Queue) do
  List list$_1$ $\Leftarrow$ Front(S.Queue)
  Dequeue(S.Queue)
  Enqueue(list$_1$, S.Queue)   //S.Queue is never emptied!
  for all LinkedPrimitiveADT p $\in$ list do
    for $j \Leftarrow 1$ to Size(T.Queue) do
      List list$_2$ $\Leftarrow$ Front(S.Queue)
      Dequeue(S.Queue)
      Enqueue(list$_2$, S.Queue)   //S.Queue is never emptied!
      LinkedPrimitiveADT q $\Leftarrow$ Retrieve(p.Label, list$_2$)
      if q $\neq$ Null then
        for $k \Leftarrow 1$ to Length(p.Input) do
          if p.Input[$k$][1] $\neq$ Null and (p.Input[$k$][1] $\neq$ q.Input[$k$][1]
          or p.Input[$k$][2] $\neq$ q.Input[$k$][2] ) then
            return $false$
        for $k' \Leftarrow 1$ to Length(p.Output) do
          if p.Output[$k'$][1] $\neq$ Null and (p.Output[$k'$][1] $\neq$
          q.Output[$k'$][1] or p.Output[$k'$][2] $\neq$ q.Output[$k'$][2] ) then
            return $false$
    if q = Null then
      return $false$
return $true$
```

B.4 Single-level assembly algorithm

Finally, the last operation identified over structs is that of struct assembly. Struct assembly is an important operation as it is how class elements are built from class representation, i.e. class elements are, generally, not built from a series of simple attachments. The idea for this algorithm originated in using an analog watch, or wall clock, as a guiding metaphor for the struct data structure, see Fig. B.1. The reason for such a metaphor, is that the time intervals, especially on analog watches, are not the same for all such watches, e.g. some watches represent intervals[4] of five minutes, some minutes, and some may only have a single marking for the twelfth hour for representing intervals of an hour.

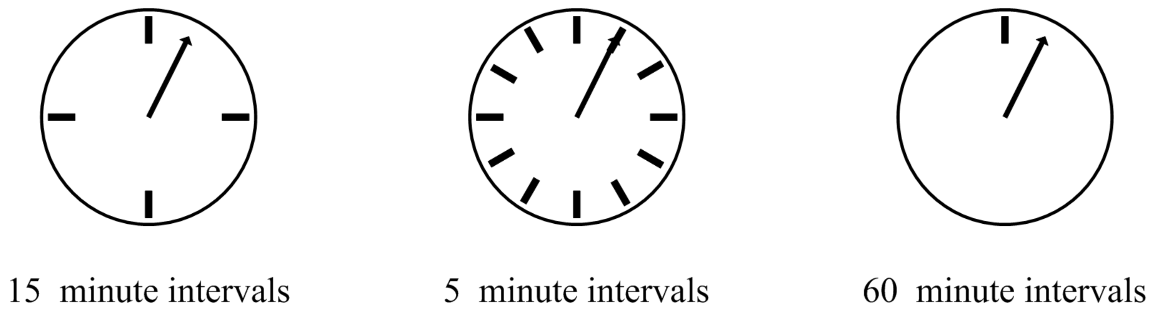


Figure B.1: Pictorial illustration of three (analog) clock faces.

Visually, the assembly of, say, two structs is accomplished by superimposing one onto another usually requiring a slight adjusting of a few primitives to make the whole thing fit, see Fig. B.2. Notice how the primitive π_{2a} in the first struct is represented in parallel to π_{3n}, and in the second struct, β is represented as a later event. To assemble these two structs, visually, is to superimpose one on top of the other, in order to do this π_{2a} must be adjusted slightly (according to the temporal information implicit in struct β). Interestingly, such adjustments do not invalidate the struct but rather increase its informational content. Technically assembly is the unions of both the sets of primitives and the sets of struct links, so long as the result of such unions is a valid struct, however, there is another sort of union achieved as a result, and it is the union of temporal content (or the partial orderings of primitives in both structs).

[4]Intervals are evaluated relative to the minute hand.

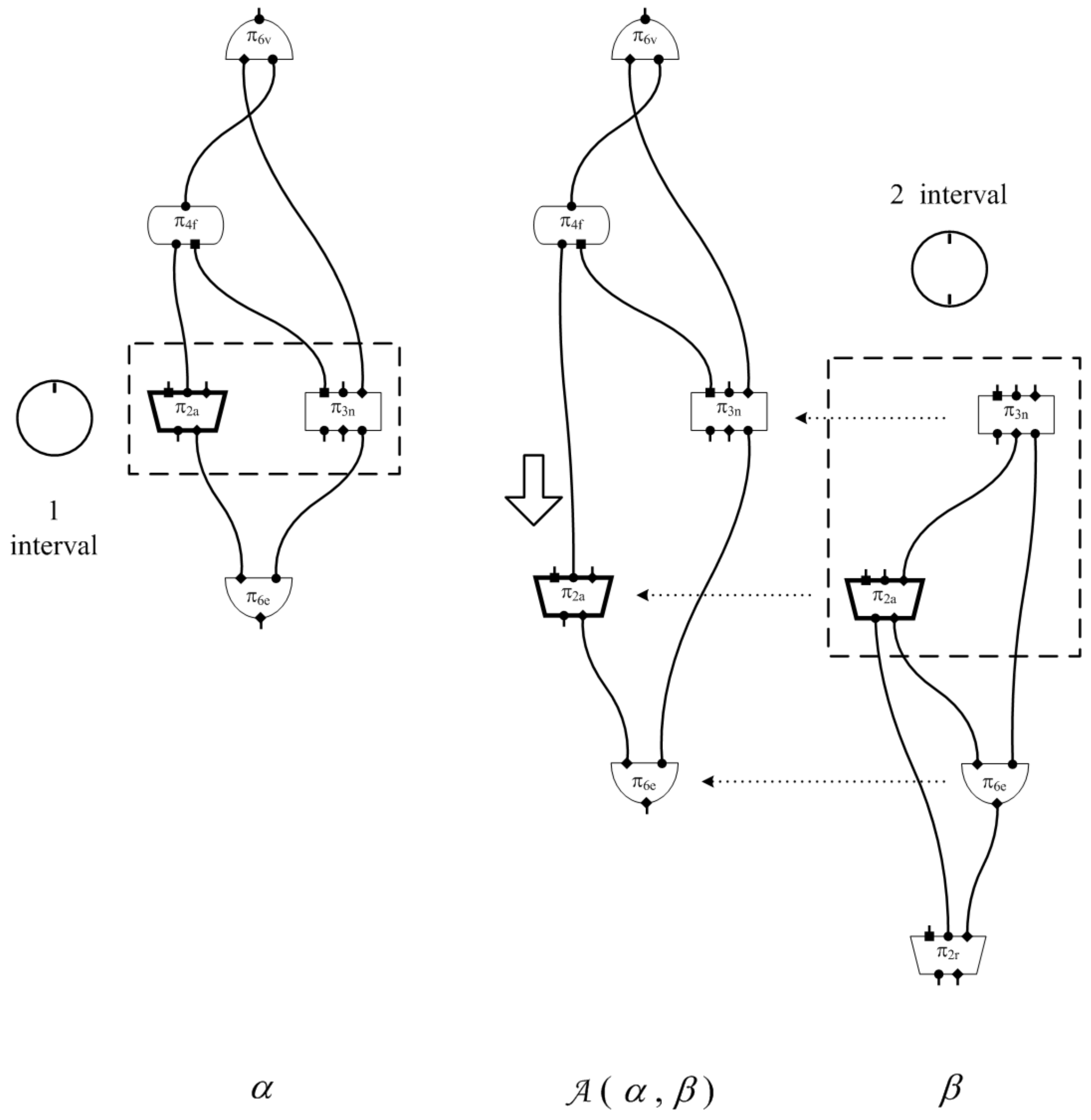


Figure B.2: Pictorial illustration of two structs α and β and their assembly.

Thus, like the face of an analog watch, a struct represents time intervals[5]. The relevance of the watch metaphor, now, has to do with the reconciliation of temporal information between structs before they can be assembled, i.e. the intervals in which primitives are located relative to one another is important. This reconciliation can be understood by example: given two watches, they both represent intervals of fifteen minutes, however, the second watch has special purpose such that the second fifteen minute interval is divided further into five minute intervals. This refinement clearly represents those fifteen minutes of an hour more precisely, this is the same situation with structs. Structs with more primitives in parallel have less temporal information then the struct composed of the same primitives but represents them in sequence implying a partial order not present in the first. Reconciling the differences between the watches only requires the same interval to also be divided into five minute intervals, with the important benefit of not interfering with the rest of the watch, i.e. time can still be read consistently. The same goes for the structs such that if one struct contains a pair of primitives in parallel and another contains the same primitives but has recorded one before another, then the primitives may be adjusted like-wise, resulting in a refinement of time representation, i.e. where the was only one interval now there is two and the struct remains consistent with reality (*cf.* Fig. B.2).

B.4.1 Reconciling partial orders between two structs

```
Reconcile(StructADT S, StructADT T)
Enqueue(Null, S), Enqueue(Null, T)
while Front(S.Queue) $\neq$ Null do
  List list $\Leftarrow$ Front(S.Queue)
```

[5]Of course, the time intervals represented in a struct are not necessarily uniform like those of a watch.

```
  Queue q $\leftarrow$ new queue
  while Front(T.Queue) $\neq$ Null do
    List list' $\Leftarrow$ new list
    for all LinkedPrimitiveADT p $\in$ Front(S.Queue) do
      LinkedPrimitiveADT p' $\Leftarrow$ Retrieve(p.Label, Front(T.Queue))
      if p' $\neq$ Null then
        Insert(p, p.Label, list')
        Delete(p.Label, list
    Enqueue(list', q)
    Enqueue(Front(T.Queue), T.Queue)  //``spinning the queue''
    Dequeue(T.Queue)
  for all LinkedPrimitiveADT p $\in$ list do
    Insert(p, p.Label, Front(q))  //maintain temporal order
    of ``other'' primitives
  Dequeue(S.Queue)
  while Front(q) $\neq$ Null do
    Enqueue(Front(q), S.Queue)  //``slight adjustments''
    Dequeue(q)
```

B.4.2 Assembly of two structs

```
Assembly(StructADT S, StructADT T):StructADT
StructADT S' $\Leftarrow$ S, StructADT T' $\Leftarrow$ T
StructADT U $\Leftarrow$ new struct
while Front(S'.Queue) $\neq$ Null do
  List $list_1$ $\Leftarrow$ Front(S'.Queue)
  Queue $q_1$ $\leftarrow$ new queue
  while Front(T'.Queue) $\neq$ Null do
    Queue $q_2$ $\leftarrow$ new queue
    bool sync $\Leftarrow$ $false$
    for all LinkedPrimitiveADT p $\in$ $list_1$ do
      LinkedPrimitiveADT p' $\Leftarrow$ Retrieve(p.Label, Front(T'.Queue))
      if p' $\neq$ Null and Superimpose(p, p') $\neq$ false then
        Delete(p.Label, Front(T'.Queue))
        sync $\Leftarrow$ $true$
    Enqueue(Front(T'.Queue))
    Dequeue(T'.Queue)
  if sync $= false$ then
    Enqueue($list_1$, $q_1$)
    while Front($q_2$) $\neq$ Null do
      Enqueue(Front($q_2$), T.Queue)
      Dequeue($q_2$)
  else
    Reverse($q_1$), Reverse($q_2$)
    $q_3$ $\Leftarrow$ new queue
    while Front($q_1$) $\neq$ Null or Front($q_2$) $\neq$ Null do
      list $\Leftarrow$ new list
      if Front($q_1$) $\neq$ Null then
        for all LinkedPrimitiveADT p $\in$ Front($q_1$) do
          Insert(p, p.Label, list)
        Dequeue($q_1$)
      if Front($q_2$) $\neq$ Null then
        for all LinkedPrimitiveADT p $\in$ Front($q_2$) do
          Insert(p, p.Label, list)
        Dequeue($q_2$)
      Enqueue(list, $q_3$)
    Reverse($q_3$)
    while Front($q_3$) $\neq$ Null do
      Enqueue(Front($q_3$), U.Queue)
      Dequeue($q_3$)
return U
```

B.5 Active extension algorithm: generating structure to specification

The algorithm for generating an active extension, given an active struct and an active constraint, e.g. `Ext(ActiveStructADT S, ActiveConstraintADT ACon)`, is described as follows.

B.5.1 Iterating through the constraint data structure

To generate an active extension of the given struct satisfying a particular constraint, is simply a matter of generating simpler structs satisfying each unit-constraint while incorporating the given struct's primitives as noise.

```
while Front(ACon.Queue) $\neq$ Null do
  for all UConPrimitiveADT p $\in$ Front(ACon.Queue) do
    ...
```

Remember that the unit-constraints are not stored individually, but in an array of unit-constraints which, itself is stored within the shared primary primitive. Thus, to access each unit constraint, one must iterate through the unit-constraint array, `p.UCon`:

```
for $i \Leftarrow$ 1 to Length(p.UCon) do
    ...
```

Then, one must iterate through each unit-constraint `p.UCon[i]`, to first find the reference to FR in memory, then to find the sets of admissible noise primitives and then (at the second last step of iteration) the secondary primitive in tandem with its index.

```
LinkedPrimitiveADT[ ][ ] p.FR$_i$ $\Leftarrow$ p.UCon[$i$][1]
...
for $j \Leftarrow$ 2 to Length(p.UCon[$i$])$-2$ do
    ...
```

B.5.2 Assembling pivots to existing structure

Before generating structure, there are two special cases in the assembly of the sub-structures, specified by `p.UCon[`i`]`, to (active) struct T. Unit-constraints specify which terminal and initial primal process of its pivot elements the resulting structure is connected to, as can be seen in Fig. 4.7, this is handled next. More specifically, the primary primitive must superimpose with an equivalent primitive from the given active struct `S` if it is marked as an anchor, furthermore the equivalent primitive found in `S` must have an open marking for superimposition to be possible. If the primary pivot is not an anchor, it may still be superimposed on the given active struct (only if the equivalent primitive is marked open) or on a pivot primitive from the extension (also only if the equivalent primitive is marked open), otherwise the algorithm is flagged to fail since the pivot primitive can not be assembled to the existing structures.

```
LinkedPrimitiveADT p' $\Leftarrow$ new linked primitive $\langle$ p.Name, p.Init, p.Term,
p.Label $\rangle$
```

```
LinkedPrimitiveADT a  $\Leftarrow$  Find(p', S)
LinkedPrimitiveADT b  $\Leftarrow$  Find(p', T)
**if** p.Marks[p.Label][1] $= 1$ **then**
  **if** a $\neq$ Null and S.Marks[a.Label][2] $= 1$ **then**
    FAIL  $\Leftarrow$  Superimpose(p', a)
    T.Marks[p'.Label][2]  $\Leftarrow$  p.Marks[p.Label][2]
  **else**
    FAIL  $\Leftarrow$  *true*  //anchor not found, or open for superimposition
**else**
  **if** a $\neq$ Null **then**
    FAIL  $\Leftarrow$  Superimpose(p', a)
    T.Marks[p'.Label][2]  $\Leftarrow$  p.Marks[p.Label][2]
  **else if** b $\neq$ Null **then**
    FAIL  $\Leftarrow$  Superimpose(p', b)
    T.Marks[p'.Label][2]  $\Leftarrow$  p.Marks[p.Label][2]
  **else**
    FAIL  $\Leftarrow$  *true*  //pivot not assembled to existing structure
```

It is a similar situation for the secondary pivot, while the first pivot primitive from the unit-constraint primitive *must* either superimpose with equivalent primitives from the given struct or the extension under construction, the secondary pivot *may* be assembled to existing structure or be attached. Therefore, the algorithm is not flagged to fail, if the secondary element is not present in the existing structure.

```
UConPrimitiveADT q  $\Leftarrow$  p.UCon[$i$][Length(p.UCon[$i$])$-1$]
int v  $\Leftarrow$  p.UCon[$i$][Length(p.UCon[$i$])]
...
q'  $\Leftarrow$  new linked primitive $\langle$ q.Name, q.Init, q.Term, p.Label $\rangle$
```

```
LinkedPrimitiveADT a $\Leftarrow$ Find(q', S)
LinkedPrimitiveADT b $\Leftarrow$ Find(q', T)
if p.Marks[q.Label][1] $= 1$ then
  if a $\neq$ Null and S.Marks[a.Label][2] $= 1$ then
    FAIL $\Leftarrow$ Superimpose(q', a)
    T.Marks[q'.Label][2] $\Leftarrow$ p.Marks[q.Label][2]
  else
    FAIL $\Leftarrow$ $true$ //anchor not found, or open for superimposition
else
  if a $\neq$ Null then
    FAIL $\Leftarrow$ Superimpose(q', a)
    T.Marks[q'.Label][2] $\Leftarrow$ p.Marks[q.Label][2]
  else if b $\neq$ Null then
    FAIL $\Leftarrow$ Superimpose(q', b)
    T.Marks[q'.Label][2] $\Leftarrow$ p.Marks[q.Label][2]
```

B.5.3 Connecting primary pivot primitives to FR-specified substructure

With regard to regular structural constraints and extension, the above discussion regarding primitive markings can be ignored. However, in both cases structural constraint pivot primitives must have specific connections to the structures being constructed per each unit-constraint.

```
for $i$ $\Leftarrow$ 1 to Length(p.UCon) do
  for $j$ $\Leftarrow$ 2 to Length(p.UCon[$i$])$-2$ do
    ...
```

```
LinkedPrimitiveADT p' $\Leftarrow$ new linked primitive
$\langle$ p.Name, p.Init, p.Term, p.Label $\rangle$
...
$indx \Leftarrow$ 1
if p'.Label $\neq$ p.Label then
  repeat
    r $\Leftarrow$ p.FR$_i$(p.UCon[$i$][$j$])[$indx$]
    r' $\Leftarrow$ Find(r, S)
    if r' $\neq$ Null then
      int[ ][ ] links $\Leftarrow$ Links(p', r')
      if Length(links) $> 0$ then
        $u \Leftarrow -1$
        for all int[ ] link $\in$ links do
          if link[1] = i then
            SL(p, i, r', link[2])
          else
            FAIL $\Leftarrow$ true
      else
        FAIL $\Leftarrow$ true
      Attach(p', r', T)   //Build active extension T
      p' $\Leftarrow$ r'
    $indx{+}{+}$
  until p'.Label $\neq$ p.Label
else
  ...   //General algorithm specified above
...
```

B.5.4 Generating substructure according to formation rules

For each step of iteration, $j (2 \leq j <$ `Length(p.UCon[`i`])`$-2)$, a set of admissible concrete (noise) primitives may be generated using $\mathtt{FR}_i$. These must also be iterated and matched against the given active struct `S`, since if any of these concrete noise primitives are found in S, then they will compose (with possibly other noise primitives) the simple structure formed between two pivot primitives—such structures are implied by the shaded ellipses shown in Fig.'s 4.7 and 4.9.

In general, the structures specified between each pivot primitive of a constraint, is generated as follows:

```
for $i \Leftarrow$ 1 to Length(p.UCon) do
  for $j \Leftarrow$ 2 to Length(p.UCon[$i$])$-2$ do
    ...
    for all LinkedPrimitiveADT r $\in$ p.FR$_i$(p.UCon[$i$][$j$]) do
      LinkedPrimitiveADT r' $\Leftarrow$ Find(r, S)
      if r' $\neq$ Null then
        Attach(p', r', T)  //Build active extension T
        p' $\Leftarrow$ r'
      ...
```

B.5.5 Connecting secondary pivot primitives to FR-specified substructure

Finally, the unit-constraint structure is completed when the secondary pivot primitive `LinkedPrimitiveADT q` is attached:

```
for $i \Leftarrow$ 1 to Length(p.UCon) do
  for $j \Leftarrow$ 2 to Length(p.UCon[$i$])$-2$ do
```

```
...
UConPrimitiveADT q  $\Leftarrow$  p.UCon[$i$][Length(p.UCon[$i$])$-1$]
int v  $\Leftarrow$  p.UCon[$i$][Length(p.UCon[$i$])]
...
q'  $\Leftarrow$  new linked primitive $\langle$ q.Name, q.Init, q.Term, p.Label $\rangle$
FAIL  $\Leftarrow$  Superimpose(q', Find(q', S))
FAIL  $\Leftarrow$  Superimpose(q', Find(q', T))
**if** p'.Label $=$ p.Label **then**
  SL(p', i, q', v)
  **if** p'.Output[i][2] $\neq$ v **then**
    FAIL  $\Leftarrow$ $true$
  **else**
    Attach(p', q', T)
**else**
  int[ ][ ] links  $\Leftarrow$  Links(p', q')
  **if** Length(links) $> 0$ **then**
    $u \Leftarrow -1$
    **for all** int[ ] link $\in$ links **do**
      **if** link[1] $=$ i **then**
        SL(p, i, r', link[2])
      **else**
        FAIL  $\Leftarrow$ $true$
  **else**
    FAIL  $\Leftarrow$ $true$
  Attach(p', r', T)
```

Vita

Candidate's full name:	Benjamin Reuben Peter-Paul
University attended:	University of New Brunswick, 2000–2008
	New Brunswick, Canada
	BCS (Computer Science) 2005

Publications:

L. Goldfarb, I. Scrimger, B. R. Peter-Paul, "ETS as a tool for decision modeling and analysis: planning, anticipation, and monitoring", paper prepared for the 2007 Decision and Risk Analysis Conference, May 21–22.

L. Goldfarb, B. R. Peter-Paul, "Integrating Development, Evolution, and Cognition via a Novel Formalism for Structural Representation", in preparation, 2008

Conference Presentations:

B. R. Peter-Paul, "Why is the String an Unreliable Data Structure?", Acadia University, APICS 29th Annual Conference, October 21–23, 2005

Poster Sessions: L. Goldfarb, B. R. Peter-Paul, I. Scrimger, "ETS Representation of Human Movement", UNB Computer Science Research Expo, April 4, 2007