CHAPTER

8

A Qualitative Biochemistry and Its Application to the Regulation of the Tryptophan Operon

Peter D. Karp

1 Introduction

This article is concerned with the general question of how to represent biological knowledge in computers such that it may be used in multiple problem solving tasks. In particular, I present a model of a bacterial gene regulation system that is used by a program that simulates gene regulation experiments, and by a second program that formulates hypotheses to account for errors in predicted experiment outcomes. This article focuses on the issues of representation and simulation; for more information on the hypothesis formation task see (Karp, 1989; Karp, 1990).

The bacterial gene regulation system of interest is the tryptophan *(trp)* operon of *E. coli* (Yanofsky, 1981). The genes that it contains code for enzymes that synthesize the amino acid tryptophan. My model of the trp oper-

on—called GENSIM (genetic simulator)—describes the biochemical reactions that determine when the genes within the operon are expressed and when they are not, the reactions by which the genes direct the synthesis of the biosynthetic enzymes (transcription and translation), and the reactions catalyzed by these enzymes. Therefore my modeling techniques are specifically designed to represent enzymatically-catalyzed biochemical reactions whose substrates include macromolecules with complex internal structures, such as DNA and RNA. These techniques address such issues as: How might we represent the attributes and the structures of the objects that make up the trp operon? What is a suitable ontology for these objects—an appropriate level of abstraction at which to model them? How might we describe a gene regulation experiment, and how can we maintain a library of known experiments? How might we represent known biochemical reactions? How can we design a simulation program that predicts the outcome of a gene regulation experiment by correctly and efficiently simulating every reaction that occurs in that experiment, and only those reactions?

GENSIM embodies a *qualitative biochemistry* because it provides a framework for representing knowledge of biochemistry, and for performing qualitative simulations of biochemical systems. The specific features of this qualitative biochemistry are as follows.

I employ frames to represent biochemical objects that correspond to homogeneous populations of molecules. This representation describes the decomposition of complex objects into their component parts. I use frame knowledge bases to represent the objects present in the initial conditions of different experiments. Section 2 describes how instance frames represent chemical objects in simulation knowledge bases; Section 3 describes how class frames are used to represent general classes of biochemical objects, and presents a method for automatically instantiating these classes.

I employ frames called processes to represent biochemical reactions; reactions are arranged in an inheritance hierarchy and often inherit portions of their definitions from more general reaction classes. Section 4 discusses the GENSIM process knowledge base.

The GENSIM simulator uses information in the process knowledge base to determine what reactions occur among the objects in an experiment, and to predict what new objects will be present at the end of an experiment. Section 5 presents two different algorithms used by GENSIM to simulate process execution. Because biochemical reactions are probabilistic events that act on populations of molecules, when GENSIM simulates reactions it splits reacting populations of molecules into two subpopulations: those that do react and those that do not react (this operation is called *object forking*). Object forking is necessary to ensure simulation correctness, but it is a computationally expensive operation. Therefore, Section 5.5 presents methods for increasing simulation efficiency. For example, there are times when we can avoid object

forking. Section 5.4 identifies a restriction on the syntax of biochemical reaction descriptions that is necessary to ensure simulation correctness.

Section 6 presents the results of several simulations that have been computed by GENSIM. Section 7 compares my model of the tryptophan operon to models of biochemical systems that have been created by other researchers.

I claim that the methods embodied by GENSIM are sufficient to represent qualitative scientific knowledge about objects and processes in molecular biology and biochemistry, such that the knowledge can be used to predict experimental outcomes, and such that other programs can reason about and modify this knowledge.

The mechanism of *transcription* will be used as an example throughout the remainder of this article. Transcription is a set of processes that are involved in the expression of genes, such as those within the trp operon. Transcription is somewhat analogous to copying a magnetic tape. An enzyme (called *RNA polymerase*) first attaches to the trp operon DNA at a *promoter* site, and then moves along the linear DNA strand, reading the message on the DNA and simultaneously synthesizing another long molecule called RNA that contains what is effectively a copy of the DNA message. When RNA polymerase recognizes a *terminator* DNA site, it releases both the DNA and RNA.

2 Representation of Biochemical Objects and Experiments

In the GENSIM framework, a user defines a gene regulation experiment by describing the objects present at the start of the experiment. I have used IntelliCorp's KEE frame knowledge representation system to represent all of GENSIM's knowledge (Kehler and Clemenson, 1984). To describe an experiment in which a particular strain of *E. coli* is grown in a particular medium, a user creates KEE instance frames that represent the bacterial DNA (wildtype or mutant), the proteins that are present within the cell (such as RNA polymerase and the regulatory trp-repressor protein), and small molecules that are present (such as glucose and tryptophan). Users do not create these frames "out of the blue," but by instantiating existing class frames (see Section 3).

In the ontology for this qualitative biochemistry, each "object" represents not a single molecule, but rather a homogeneous *population* of molecules. For example, all molecules of tryptophan in a experiment that are in a given state (such as those floating free in solution, as opposed to those bound to the trp-repressor protein) are represented by a single KEE frame. All molecules of tryptophan in a different single state (such as bound to the trp-repressor protein) are represented by a different single KEE frame.

These frames reside within a single KEE knowledge base for this particular experiment, called an *SKB* or *simulation knowledge base*. Other experiments could be described within other knowledge bases.[1] In the context of a

GENSIM simulation the SKB corresponds to the working memory of a production system. Since the facts it contains are represented using frames, all facts are literals and contain neither disjunction nor negation.

GENSIM does not represent temporal aspects of objects explicitly; the work of (Simmons and Mohammed, 1987) and (Williams, 1986) is relevant to this problem. GENSIM's task is to simulate the behavior of a biochemical system during a very short interval of time. Within such a short interval, new objects can come into existence because the creation of an arbitrarily small amount of an object is enough to change its concentration from zero to positive. However, we make the simplifying assumption that GENSIM simulations take place in a short enough interval that a population of molecules is never fully consumed; thus objects are never deleted from simulations. When an arbitrarily small amount of an object is destroyed, we cannot assume that its concentration has changed from positive to zero.

This assumption implies that the number of objects in a simulation must increase monotonically. In reality, biologists do perform experiments over intervals of time long enough that objects are completely consumed by reactions. However, this assumption simplifies the implementation of GENSIM significantly. Without it, GENSIM would have to reason about time and quantities. Yet the system is still able to make predictions for an interesting class of experiments. This assumption also simplified the implementation of the HYPGENE hypothesis formation program described in (Karp, 1989).

3. The Class Knowledge Base

GENSIM's *class knowledge base* (CKB) is a taxonomic hierarchy that describes classes of biochemical objects such as genes, proteins, and biochemical binding sites. The KB describes the properties and states of different classes of objects, and the decomposition of objects into their component parts. The CKB can be viewed as a library that records all known types of objects that could be present in experiments on the trp system (in practice, we have omitted many marginally relevant objects because bacteria are incredibly complex biochemical systems). Each object class is represented as a KEE class frame. The CKB is shown in Figures 1 and 2.

A central type of relationship among objects in this domain is the *containment* of one or more objects by a composite object. The example object structure in Figure 3 describes an experiment object that has two components: an enzyme (`RNA-Polymerase`) and a segment of DNA (`Trp.Operon`). The trp operon is in turn divided into a number of component regions. Several issues of interest arose in the representation of objects with complex component structures: how to represent different types of containment relationships, how to define classes of these objects, and how to instantiate these classes. The slots within the `Trp.Operon` class shown in

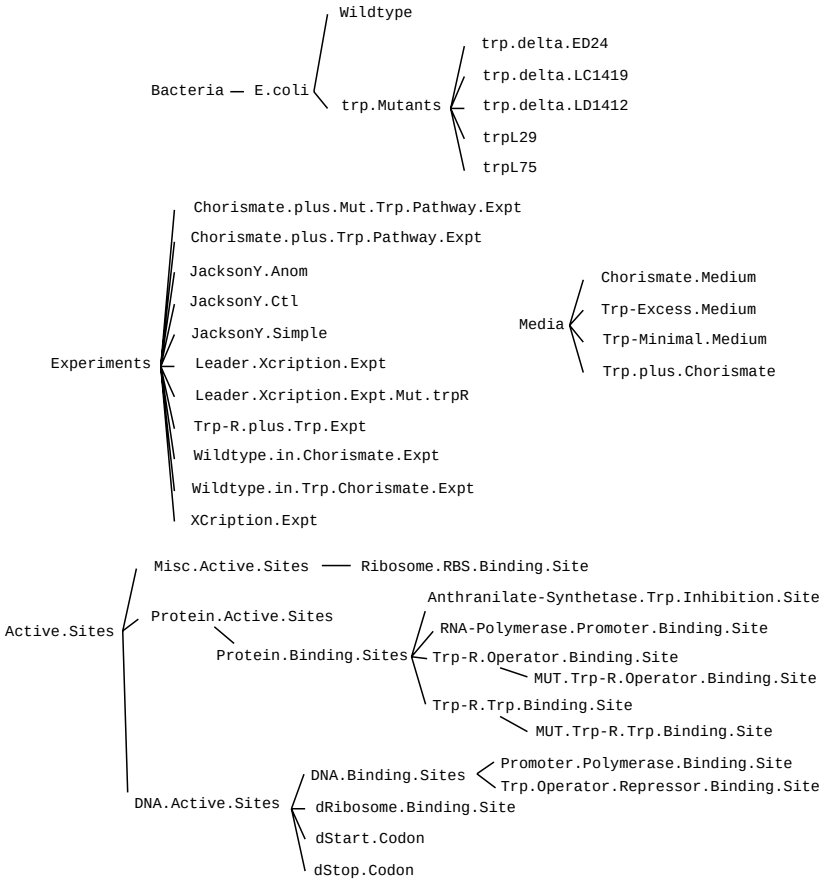

Figure 1. Object classes within the class knowledge base. The lines represent the class–subclass relation and connect object classes with the subclasses to their right. This figure shows objects that describe experiments, media, and bacterial strains, as well as active sites within proteins and DNA, and mutations.

Figure 4 are used to represent the component structure of this object (each slot has been truncated for brevity)

A user instantiates `Trp.Operon` by sending a `Create.Instance` message to the `Trp.Operon` frame. The LISP method invoked instantiates `Trp.Operon` and then recurses, sending the same message to each of the class frames listed in the `Component.Object.Classes` slot of `Trp.Operon`. Instantiating `Trp.Operon` itself involves creating a new instance of the class with a unique name, such as `Trp.Operon.1`. The

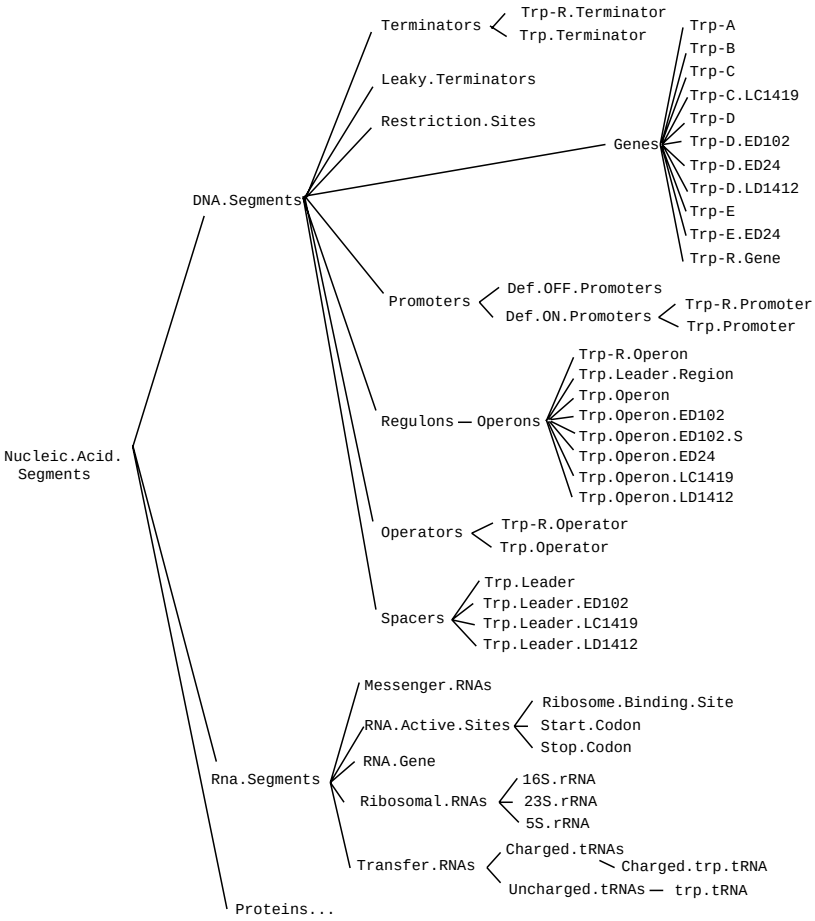

Figure 2. Object classes within the class knowledge base. This figure shows the "nucleic acid segment" objects that are used to describe various regions of DNA and RNA, as well as classes of proteins, which are shown in the continuation of figure 2 on the next page.

names of the created component objects are bound to the variables named in the `Component.Object.Bindings` slot of `Trp.Operon`.

A general problem that occurs when encoding class-level templates that are used to guide the creation of instance frames is how to encode relations among objects at the class level so that those relations can automatically be instantiated at the instance level. For example, every promoter object in an operon records what operator object in that operon controls the promoter. We wish to specify this general relationship in the operon class object, and

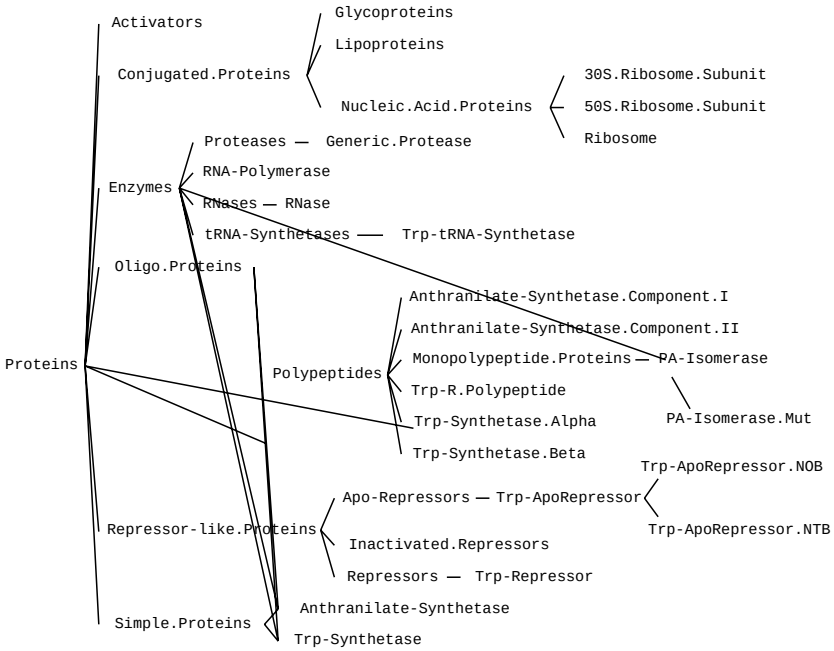

Figure 2 continued: The proteins are classified according to both structural and functional attributes.

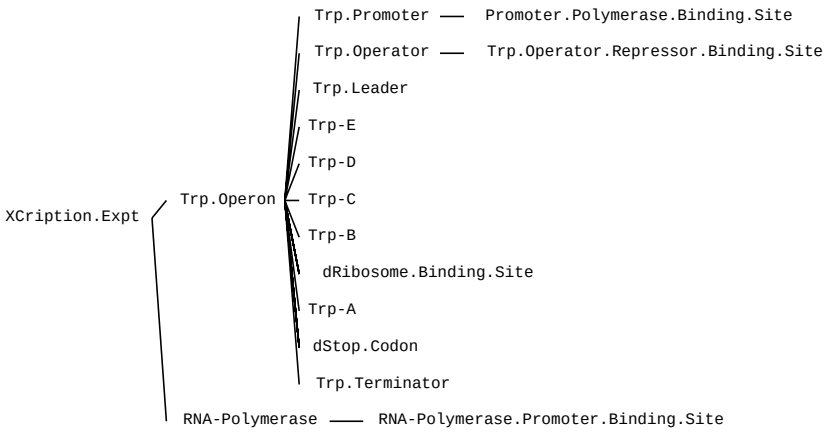

Figure 3. The objects in a transcription experiment. This experiment contains the trp operon and RNA polymerase, both of which have the internal structure shown. The lines in this figure represent the part-of relationship between a containing object and the parts to its right.

```
FRAME: Trp.Operon
        SLOT:                          VALUES
 Component.Object.Classes:    (Trp.Promoter Trp.Promoter
                                 Trp.Operator Trp.Leader
                                 dRibosome.Binding.Site
                                 Trp-E dStop.Codon ... )
  Component.Object.Bindings:  ($pro1 $pro2 $op1 $lead1
                                 $drbs1 $trpe $dsc1 ... )
  Component.Objects:
  Structural.Relations:       ((PUT.VALUE $Object
                                  'Regulated.Promoters $pro1)
                               (PUT.VALUE $pro2
                                  'Regulator $op1))
```

Figure 4. The `Trp.operon` *class.*

whenever a user instantiates the operon class, the promoter within the newly instantiated operon should refer to the operator object within that operon object. GENSIM allows the user to list a set of variablized assertions within the `Structural.Relations` slot of an object class. These assertions are executed by `Create.Instance` using the variable bindings described in the previous paragraph. For example, the first `PUT.VALUE` expression asserts a relationship between a promoter and an object that regulates that promoter.

4. The Process Knowledge Base

The *process knowledge base* (*PKB*) describes the potential behaviors of the objects in the trp system such as biochemical binding, rearrangement, and dissociation events.

The sample process in Figure 5 describes a binding reaction between the activated trp-repressor protein and the trp operator. Table I gives the functions and predicates used in GENSIM processes. The process specifies that for any two molecules of type `Trp.Operator` and `Trp-Repressor`, if these objects contain complementary binding sites, and if these sites are empty and contain no mutations that interfere with this binding reaction, then new instances of these objects should be created and bound together as a new object.

Users represent processes using KEE frames. A process definition specifies actions that will be taken (listed in the `Effects` slot of the process) if certain conditions hold (listed in the `Preconditions` slot). In addition, since processes operate on objects, the `Parameter.Object.Classes` slot specifies the types of objects on which a process acts. Processes are executed by a *process interpreter*. The interpreter activates a process when at least one object is present from each class in the parameter-object classes of the process. The `Parameter.Objects` slot lists variables that are bound to the actual objects with which a process has been activated. In addition, an arbi-

```
Parameter.Object.Classes: Trp.Repressor Trp.Operator
Parameter.Objects:              $A          $B
Preconditions: Check that $B contains an active site that
               interacts with objects of $A's type.
               [EXISTS $Bsite
                 (AND
                   (IS.PART.R $Bsite $B)
                   (OBJECT.EXISTS $Bsite Active.Sites)
                   (EXISTS $site.interaction.class
                     (AND
                       (MEMBER $site.interaction.class
                         (GET.VALUES $Bsite
                           Potential.Interacting.Objects))
                       (OBJECT.EXISTS $A
                         $site.interaction.class]
               Check that $Bsite is not occupied.
               [NOT (EXISTS $obj
                      (AND
                        (MEMBER $obj
                          (GET.VALUES $Bsite
                            Object.Interacting.With.Site))
                        (OBJECT.EXISTS $obj
                          (GET.VALUE $Bsite
                            Potential.Interacting.Objects]
               Check that $Bsite does not contain a
               mutation that disables the current reaction.
               [NOT (EXISTS $mutation
                      (AND (IS.PART $mutation $Bsite)
                           (OBJECT.EXISTS $mutation
                             Mutations)
                           (MEMBER $Current.Process
                             (GET.VALUES $mutation
                               Processes.Disabled]
Effects:       Create a new object that contains $A
               and $B as parts.
               (BINDV $Complex
                 (CREATE.COMPLEX RepOp.Complexes
                   (LIST $A $B) RBOUND))
               Record that $A is interacting with $Bsite.
               (PUT.VALUE $Bsite
                 Interacting.With.Site $A))
               Record that the promoters controlled by $B
               are no longer able to bind RNA Polymerase.
               (PUT.VALUE (GET.VALUE $B
                            Promoters.Controlled)
                 Receptive.To.Polymerase NO)
```

Figure 5. The definition of the process `Trp-Repressor.Binds.Operator`. *This process describes the binding of the activated trp-repressor protein to the trp operator. Comments that explain the process are given in italics.*

Predicate or Function	*Meaning*
(OBJECT.EXISTS X Y)	Object X exists within class Y
(IS.PART X Y)	Object X is part of object Y
(MEMB X Y)	Atom X is an element of list Y
(GET.VALUES X Y)	The value of slot X of object Y
(BINDV $X Y)	Binds variable $X to the value Y
(CREATE.COMPLEX X Y)	Creates an object of type X containing the objects in list Y as parts
(COPY.STRUCTURE X)	Creates a copy of object X
(PUT.VALUE X Y Z)	Stores Z into slot X of object Y
(EXISTS $X Y)	True if expression Y is true for one binding of $X
(FORALL $X Y)	True if expression Y is true for all bindings of $X

Table I. The predicates and functions used within GENSIM process definitions. OBJECT.EXISTS, IS.PART, and MEMB are predicates. The symbols AND, OR, and NOT may also be used, and have their standard logical meanings.

trary list of variable bindings may be given in the Bindings slot.

Processes possess an additional type of precondition called Efficiency.Preconditions. These preconditions prevent process invocations that, although technically correct, are uninteresting. For example, the trp-repressor protein binds to the operator site at the start of the trp operon. It can bind there during almost any of the 17 intermediate steps of the transcription process that GENSIM generates to model the progression of RNA polymerase along the DNA. These intermediate transcription-elongation complexes, however, have no special functionality when bound to the repressor, and are thus uninteresting. The Efficiency.Preconditions are used to prevent GENSIM from simulating these reactions—thereby increasing simulation speed—and because expert biologists usually ignore these reactions. But by putting these preconditions in a special slot, we make it easy to ignore them during tasks such as hypothesis formation.

The process frames within the PKB are structured as an inheritance hierarchy, part of which is shown in Figure 6. At the top level is a general template for all processes. Its children are templates that provide descriptions of general classes of events, such as chemical-binding and enzymatic-reaction processes. In turn, the children of these templates either describe actual processes (such as the bindings of particular species of molecules), or define important subclasses of processes. An example of such a subclass is Mutually.Exclusive.Binding. This template defines preconditions for a subclass of binding processes: for object A to bind to B, it cannot be the case

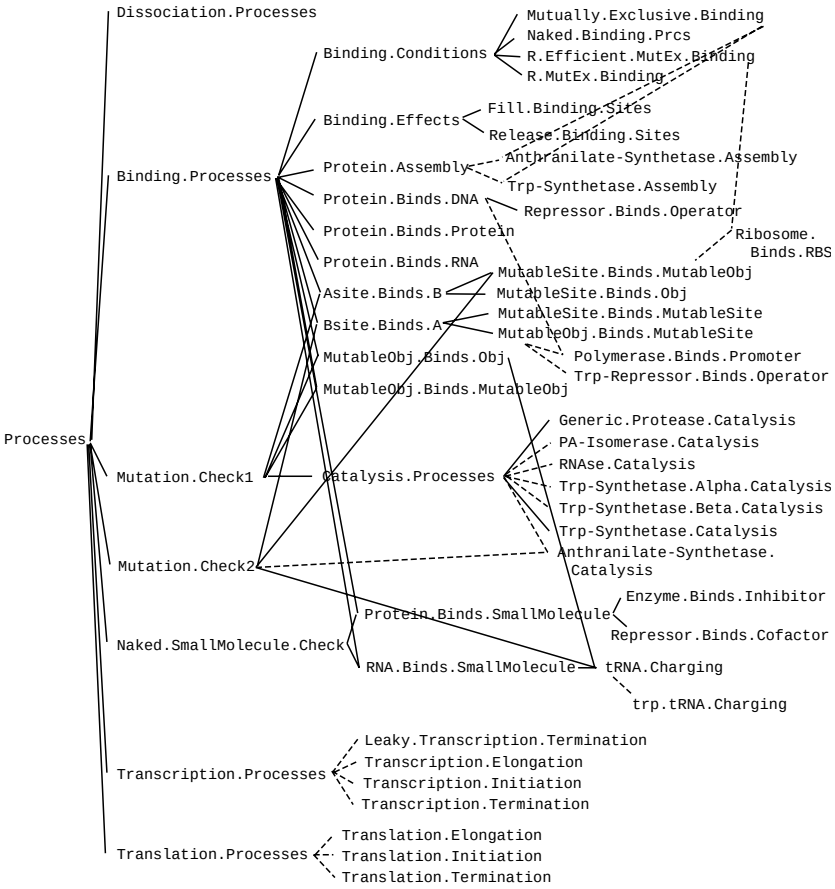

Figure 6. The process knowledge base. Process classes are linked to their superclasses by solid lines; executable process instances are linked to the classes of which they are members by dashed lines.

that A is already bound to an object of class B, or that B is already bound to an object of class A. A particular process instance can inherit slots from one or more template processes. For example, `Trp-Repressor.Binds.Trp` inherits most of its definition from the `Repressor.Binds.Cofactor` template, which specifies the behavior of the class of all repressor proteins. `Trp-Repressor.Binds.Trp` inherits additional preconditions from the process `Mutually.Exclusive.Binding`. Process classes often provide enough of the definition of a process instance that only the `Parameter.Object.Classes` slot must be modified in the instance, to define the actual classes of objects to which the process pertains.

I have created special machinery to facilitate the use of inheritance to modify process templates. Process preconditions, bindings, and effects are actually represented by *pairs* of slots—called for example, `Effects.M` and `Effects.A`. The values of the slot `Effects.M` (*main*) are inherited using KEE's *override* inheritance—new values for this slot override previous values. Values of `Effects.A` (*additional*) are inherited using *union* inheritance—new values for this slot are unioned with the previous values. The process interpreter executes the effects listed in both of these slots. The pairs of slots for preconditions and bindings are defined analogously.

This use of inheritance to define processes is similar to that used in object-oriented programming (OOP) systems (Goldberg and Robson, 1983; Stefik and Bobrow, 1986) (although GENSIM's process interpreter does not employ the message-sending control structure of OOP). Thus, our approach reaps many of the same benefits as OOP systems, such as facilitating the definition of new processes and the maintenance of existing processes. GENSIM's use of inheritance is novel in two ways. First is the use of inheritance in this particular type of language—a production-rule—like process-description language. Because GENSIM processes are so similar to the production rules used in expert systems, inheritance probably could be used in this way as a software-engineering tool in the construction of expert systems. The second novel aspect of this use of inheritance is the manner in which an individual process is dissected into pieces (preconditions, effects, parameter objects, bindings) that can be inherited and modified as required. OOP systems usually treat procedures as indivisible units that can only have additional code wrapped around them, rather than having their internals altered as desired.

The definition of a process inheritance hierarchy should have benefits in addition to those already defined. This approach should facilitate the definition of new processes by hypothesis formation programs such as HYPGENE. These programs could postulate new processes by instantiating process templates (see (Karp, 1989) for more details).

4.1 Alternative Reaction Representations

At least two other approaches could be used to represent biochemical reactions. One approach would eschew the use of processes, and would represent the behaviors of an object using slots within that object. For example, for an object *O*, one slot might list all objects that *O* can react with, and other slots might list other objects that inhibit or activate these reactions. Another slot might list the products of the reactions. This approach is problematic in several ways. First, a given object might participate in several reactions, each of which could involve different other reactants, and produce different products. Thus, the slot values must be encoded in a way that does not lose information about which reactants are associated with which products, activators, and inhibitors. Second, if five objects participate in a given reaction, presum-

ably each object must describe the same reactants, products, activators, inhibitors, and so on, which is highly redundant. Third, as GENSIM processes illustrate, it usually is not sufficient simply to list the activators and inhibitors of a reaction; we usually must test for particular properties of these objects using complicated predicate-calculus formulae. Without the language of predicate calculus we could not express preconditions such as: the trp-repressor protein binds to the trp-operator region only if a specific binding site within the trp repressor is occupied by tryptophan. In summary, when reactions involve several reactants, it is clearer and more efficient to separate process definitions from object definitions. And when reactions involve complex preconditions and effects—as biochemical reactions often do—a special language is required to express this complexity.

A second approach would use processes that are somewhat more general than those most often used by GENSIM (in fact, GENSIM's representation of mutations uses this approach). If we were to model three different repressible operons using GENSIM—say, the trp, lac, and phe operons—we would have to create separate GENSIM processes to describe the binding of the trp-repressor to the trp operator, the lac-repressor to the lac operator, and the phe-repressor to the phe operator. GENSIM allows these processes to be constructed using inheritance from the general `Repressor.Binds.Operator` process, but we still might argue that this approach creates an excessive number of processes. The alternative would be to use a single general process, such as `Repressor.Binds.Operator`, to define a slot within each repressor object that specifies the operator(s) to which the repressor can bind, and to reference this slot from the `Repressor.Binds.Operator` process. Although this approach is considerably more compact, it has two disadvantages. First, it would force a proliferation of the slots that encode these object interactions—every general process (such as `Repressor.Binds.Operator`) would refer to such a slot. Second, this approach blurs the clean separation between structure and behavior that is inherent in the CKB and the PKB. The new specificity slot acts much like a process precondition, yet it resides within an object. If object behaviors are defined using processes only we have much more modular descriptions that will be easier to maintain and extend.

5. The Process Interpreter

GENSIM processes bear significant similarity to production rules, and the program that interprets processes is similar to a production system. The process interpreter uses processes to detect interactions among objects that exist in the current simulation, and computes the effects of these interactions. This section describes how the process interpreter activates and executes processes, and manages the existence of objects during a simulation. Since these is-

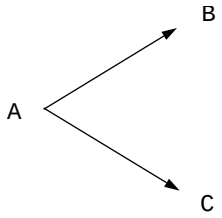

Figure 7. A simple reaction network in which an object A can cause two reactions; one produces B and the other produces C.

sues are so closely intertwined, this section alternates between the two. It begins with a brief description of object management, then presents a detailed description of process execution. It finishes by presenting points related to both issues.

Before proceeding, let us resolve the potential ambiguity of the term *process instance*. I use this term to mean the execution of a process on a given set of objects—an instance of process execution. Because processes are defined hierarchically in a KEE KB, the KB contains both class process frames and instance process frames; I use the term *leaf process* to refer to instance process frames.

5.1 Object Management

At least two possible approaches to the management of objects are conceivable in GENSIM. The term *object management* refers to the manner in which operations on objects (such as creation, deletion, and modification) are implemented. Consider the simple reactions in Figure 7, in which an object A is acted on by two processes: one converts A to object B, the other converts A to object C. A simulator might model the first reaction in one of two ways: it might modify A directly to produce B, or it might copy A to a new object, and then modify that new object to produce B.

GENSIM should produce correct and complete predictions: it must predict all and only all the correct reactions. Predictions should not depend on the order in which processes are executed; for example, care must be taken that the execution of the first reaction in Figure 7 does not prevent execution of the other by removing the A object, because both reactions would occur to some degree. In general, most chemical processes are probabilistic events that act on populations of molecules. Since objects A, B, and C represent populations of molecules, and reactions occur at some finite *rate*, it is likely that, at some time, members of all populations exist. Thus, when a biochemical reaction converts A to B, all members of population A do not disappear instantaneously.

We conclude that GENSIM should not destroy objects or modify their prop-

erties, because to do so would allow the possibility that the system would overlook some reaction. Rather than modifying A directly to create B, GENSIM copies A and modifies the copy to produce B. I call this operation *object forking*. The assumption that object populations are never fully consumed was discussed in Section 3.

This discussion also implies that, to predict both *what* reactions occur, and the *rates* of these reactions, requires a two-stage computation (GENSIM currently performs only the first stage):

1) Determine the complete set of reactions that will occur—that is, the complete set of objects that will be created and the set of processes that will fire. Forbus calls this task computing the "process and view structures" (Forbus, 1986).

2) Use information about reaction rates and molecular concentrations to compute *how much* of each object population forms. Forbus calls this task "resolving influences" (Forbus, 1986).

The intuition here is that to predict the rate of a reaction R that consumes reactants $I_1,\ldots,I_n$, we must know about all other reactions in which each I_i takes part, so that we can compute the relative rates of these competing reactions.

Most other qualitative-reasoning researchers have not used object forking. One reason for this may be that in their domains, objects do not represent populations of molecules, some of which probabilistically change to another state—they represent individual objects that change completely. Simmons and Mohammed (Simmons and Mohammed, 1987) use an explicit model of time to represent object properties as histories, so in effect, different versions of different objects exist at different points in time due to the different values their properties take on at these different points in time. The biochemistry domain does not allow this approach because the populations of molecules we model often coexist at a single time.

5.2 Reference Patching

One complication that arises during object forking is that the component objects of a complex object may refer to one another, and these references must be altered during the copy operation. This procedure is called *reference patching*. For example, a component of the `Trp.Operon` object—`Promoter.1`—contains a slot `Regulator` with value `Operator.1`. This slot indicates what operator object controls this promoter. The value of this slot must be altered in the object to which `Promoter.1` is copied, to name the object to which `Operator.1` is copied, since a promoter is regulated by only the operator in the same operon object.

Another complication is that there are some object slots for which reference patching should not be performed, such as slots that do not contain the

names of objects. Thus, each slot in the system is described by a special frame that describes whether or not reference patching should be performed for that slot.

5.3 Implementation of the Process Interpreter

I constructed two different implementations of the process interpreter. The first uses a brute-force algorithm and is slow. It iterates through all processes in the PKB, and searches for objects in the simulation that are members of the parameter object classes of the current process. It then determines which combinations of these objects satisfy the preconditions of the process, and executes the effects of such processes, until no new reactions are detected. More precisely, the first interpreter cycles through the following steps until no new processes can be activated in Step 2.

1) **Process selection:** Select a member *P* from the set of all leaf processes that exist in the process knowledge base.

2) **Process activation (binding of processes to objects):** Consider each object class C_k, k=1…N listed in the `Parameter.Object.Classes` of *P*. Let $\Im_k$ be the set of object instances within class C_k. If every set $\Im_k$ is nonempty, then create a new set *A* containing all possible tuples of the elements of each $\Im_k$. Each tuple contains an element of $\Im_1$, an element of $\Im_2$, and so on including an element of $\Im_N$. The set *A* is the set of possible bindings of the parameter-object variables of *P*, and is a list of all possible interactions of objects due to *P*. If any $\Im_k$ is empty, no activations of this process are generated.

3) **Filter process activations:** Remove from *A* every set of variable bindings for which process *P* has already been activated with that set of bindings in this simulation. Because objects never change, and because process preconditions can refer only to objects in the parameter objects of a process, the truth or falsity of the preconditions of a process as applied to a given set of objects will never change. Thus, it is never necessary to activate a process more than once on a given set of parameter objects.

4) **Process execution:** For each A_j remaining in *A:*
   a) Bind the variables in the `Parameter.Objects` slot of *P* to the objects in A_j
   b) Evaluate the variable bindings in the `Bindings` slot of *P*
   c) Evaluate the `Preconditions` of *P*. IF any are false, THEN continue to the next A_j; ELSE
   d) Execute the `Effects` of *P*

This approach is inefficient in two ways:

1) It repeatedly examines every process, even for processes for which no ob-

jects exist in some parameter-object class of the process

2) It repeatedly generates process activations that have been considered previously, which must be filtered out in Step 3 (at a cost) to avoid the larger cost of reevaluating the process preconditions.

The cost of these inefficiencies grows as more objects and processes exist in a simulation.

To improve what is essentially a generate-and-test algorithm, we move part of the test inside the generator. New process activations are generated only when the process interpreter first starts running, and when new objects are created. A given set of object bindings is never generated more than once for a process.

The second algorithm maintains two data structures. The first is the *process-activation queue*, which contains process variable-binding tuples—the variable bindings for which each process has not yet been executed. The interpreter repeatedly pops entries lists off this queue and executes the given process with the given variable bindings.

The second data structure is used to determine what process activations should be created when a new object is created. It is called the *live-objects list* and consists of a list of records of the form

$$(C\ (P_1 \ldots P_n)\ (O_1 \ldots O_n))$$

where

- C is the name of a class of objects that appears in the `Parameter.Object.Classes` slot of at least one process
- $(P_1 \ldots P_n)$ is the set of processes that contain C in their `Parameter.Object.Classes` slot—the processes that describe reactions involving this class of object
- $(O_1 \ldots O_n)$ is the list of objects within class C that exist in the simulation

When a new object O is created by the execution of a process, the following actions are taken:

1. Let C be the class of O.
2. Find the set $\Re$ of all records in the live-objects list such that record R_i describes a class that is equal to or a superclass of C. If none exist, exit.
3. Add O to the object list of each record in $\Re$.
4. For each R_i in $\Re$ do

    For each process P_j in R_i do

        Compute the new variable bindings for P_j. Imagine that P_j operates on two objects—one of class C, the other of class D. The activations of P_j consist of O paired with every object from class D

(as listed in the live-objects record for *D*). Append these activations to the process-activation queue.

Two properties of this approach are worth noting. First, new process activations are generated only when new objects are created. This approach is correct because new process activations must include at least one new object—since old objects are never modified, a group of old objects will never spontaneously activate an existing process that had not been activated previously. Similarly, because objects are forked and not deleted, process activations never have to be removed from the process-activation list.

An additional optimization is possible using a slightly different data structure. It may be the case that the interpreter could prove that an existing object *O* will always prevent process *P* from firing, because *O* will always cause a precondition of *P* to be violated (GENSIM could prove this by *partially evaluating* the preconditions of the process (Hsu, 1988)). In this case, the interpreter should never generate an activation of *P* that includes *O*. This information could be used in a similar approach that stored live objects within a class on a per-process basis, rather than with every process that acts on the class (the latter is done in the current live-objects structure). Objects would be removed from the list for a process when GENSIM proved that they could not fire that process.

5.4 A Restriction on Process Preconditions

The preceding approach to object management and process execution requires that we impose a restriction on the syntax of process preconditions to guarantee the correctness of our simulations. This restriction has an interesting biochemical interpretation.

The restriction is that a precondition of a process *P* may not check for the existence or nonexistence of an object D_1 unless D_1 either is a parameter object of *P*, or is part of an object that is a parameter object of *P*. For example, a process P_1 that describes the binding of object A_1 to B_1 may not check whether no objects exist in the simulation that belong to class *D*. Therefore, the following precondition is forbidden for process *P*:

```
(NOT (EXISTS $X (OBJECT.EXISTS $X D)))
```

But *P* may, however, check whether no objects of class *D* exist in the simulation as parts of B_1:

```
(NOT (EXISTS $X (AND (OBJECT.EXISTS $X D)
(IS.PART $X B1))))
```

Without this restriction the correctness of simulations is no longer guaranteed, because the truth of the shorter precondition will depend on *when* the process interpreter evaluates that precondition (which depends on when process *P* is executed). If evaluation occurs when no objects of type *D* exist, then the precondition will be true. But if it occurs after the execution of an-

other process P_2 that creates an object of type D, then the precondition will be false. Thus, in this example the relative execution times of processes P_1 and P_2 (which times are undefined in GENSIM simulations because GENSIM has no model of time and assumes all reactions occur in a very brief interval) determines the truth of the precondition.

The value of the restriction is that by stipulating that D_1 must be part of B_1 it ensures that D_1 must exist at the time P_1 is activated. The second algorithm activates P_1 when, and only when, all of the parameter objects of P_1 exist. So, if P_2 created B_1, P_2 must have executed before P_1. Furthermore, given the framework of object forking, once created, B_1 cannot be modified. Thus, there is no possible ambiguity in the evaluation of the preconditions of P_1.

An important question to ask is: does this restriction have reasonable semantics in the biochemistry domain? The answer is yes. In general, biochemical reactions occur when a set of reactants is present in solution, and when each of the reactants is in a required state. Process preconditions examine the states of the reactants. In general, the only way one molecule can influence the state of another molecule is by physically attaching to the other molecule and altering its conformation. That is, there is no way for A_1 and B_1 to magically sense the presence or absence of D_1 in solution. To affect the reaction, D_1 must bind to A_1 or B_1 to alter that object's state, in which case D_1 is a reactant in this reaction, and should therefore be listed as a parameter object of the process. Thus, it makes no biochemical sense to write the type of precondition that we prohibit.

5.5 Optimizations

The reason we employ object forking is that this approach to managing simulation objects meets the correctness and completeness criteria described in Section 5.1. A drawback of object forking is that in the trp operon simulations are slower by roughly a factor of 20 than are simulations in which objects are modified directly. Object forking increases the computational resources required for simulation of the trp operon because some processes within this system generate many complex objects. During execution of the transcription process, for example, each movement of RNA polymerase along a DNA strand is accomplished by a different activation of a single process that generates a new copy of the transcription-elongation complex that contains DNA, mRNA, and RNA polymerase (each of which is a complex object). Object forking is costly both because the KEE frames that represent objects are expensive to create, and because the large numbers of created objects can later cause large numbers of additional reactions. Here we discuss methods for increasing the speed of simulations.

5.5.1 Avoidance of Object Forking. In the biochemical domain there is a specific case in which objects can be modified directly to avoid the cost of

forking the object, without sacrificing the correctness of the simulation. The need to copy-then-modify objects rather than to modify them directly arose from the possibility that multiple processes might act on the original object. Modifying the original object could cause some of its behaviors to go undetected. However, if inspection of the PKB reveals that only a single process acts on a given object class (in which case the object class would be named in the `Parameter.Object.Classes` slot of a single process), the preceding consideration would appear to be nullified. Unfortunately, it is not completely nullified, because multiple activations of the process could act on the same object. For example, if we found that the only process acting on the class `RNA-Polymerase` is the `Polymerase.Binds.Promoter` process, the object `RNA-Polymerase.1` still could bind to two different instances of `Trp.Promoter`, such as `Trp.Promoter.1` and `Trp.Promoter.2`. Thus, we cannot avoid forking `RNA-Polymerase.1`. We can, however, avoid copy-then-modify when only a *single* known process acts on a given object class, and that process has only *one* parameter object. For example, the transcription-elongation process acts on a single `Transcription.Elongation.Complex` object (whose components are the DNA, enzyme, and RNA described earlier). This optimization has not been implemented within GENSIM.

Note that this optimization may cause problems for a program that must analyze a simulation-dependency trace, such as a hypothesis formation program. Such a program might try to understand why a final set of objects are predicted to be present in a given experiment by inspecting the intermediate objects that reacted to form those final objects. It is just those intermediates that this optimization destroys.

5.5.2 Object Merging. A procedure called *object merging* is used to detect when two different processes independently create the same object. When this event is detected, only one of the object descriptions is retained. This procedure produces a small economy by eliminating redundant storage of the merged objects. It produces a much larger savings by preventing processes from being invoked by the redundant objects, and by preventing the creation of the additional redundant objects that these duplicate reactions would produce.

5.5.3 Sharing Object Descriptions with an ATMS. It is possible to use an ATMS (DeKleer, 1986) to reduce the storage consumed by object forking. This approach was inspired by the ATMS property of allowing efficient exploration of alternative decisions through storing rather than recomputing elements of the problem-solving state that the alternatives share. For example, the ATMS has been used previously in qualitative physics to represent envisionments more efficiently (Forbus, 1984; deKleer and Brown, 1984). My hope was that an ATMS could be used in a similar way to provide efficient storage of different objects that have a large amount of common structure.

This use of the ATMS is novel because I propose using the ATMS to represent more efficiently common aspects of similar objects that coexist within a *single context* of the simulation. Previously, it has been used to represent more efficiently common aspects of similar objects that exist in *alternative* predictions of the state of the physical system (envisionments).

IntelliCorp's KEE contains an ATMS implementation (Intellicorp, 1986; Morris and Nado, 1986). In the following paragraphs, I describe this ATMS implementation, sketch how it might be used to solve this problem, discuss why this approach will fail, and then examine additional ATMS functionality that will solve the problem.

KEE's ATMS forms the basis for a facility called *KEEworlds*, which is well integrated with KEE's frame representation system. By default, any modification to a KEE frame (e.g. of a slot value) affects the *background* (the root world), and all slot-value queries access the background. New worlds are defined as children of the background and/or existing worlds. Users may explicitly direct assertions and queries to the background or to any existing world. By default, any fact true in the parent of a world *W* is also true in *W.* But *W* may override *some* of the facts defined in its parents: the values of existing slots in existing frames may be modified arbitrarily, however, it is possible to create, delete, or rename both frames and slots in the background only.

Figure 8 shows how the KEEworlds facility could be used to implement object forking. The background and the worlds W_1 and W_2 represent three consecutive states of the transcription process discussed earlier. In this process, a transcription-elongation complex object contains two other objects: an RNA whose length grows as the process executes repeatedly, and a DNA. Rather than create new versions of the elongation complex object for every step, the KEEworlds facility allows the core descriptions of each object to be inherited by the KEEworlds facility with only the changes to each object recorded explicitly, as shown in Figure 8. The substructure of the `RNA.1` object changes the length of the RNA increases, and new objects `A.1` and `B.1` are created.[2]

The limitation of this mechanism arises in the following situation. Imagine the existence of a biological process that specifies that two RNA objects may bind together when they are components of a transcription-elongation complex. If this reaction were to occur between the two versions of the `RNA.1` object in worlds W_1 and W_2, we would create a new world W_3 with parents W_1 and W_2. W_2 would inherit descriptions of `RNA.1` from both W_1 and W_2. The problem is that the new world W_3 would contain only a single version of `RNA.1`, whose properties would result from merging the `RNA.1` objects from W_1 and W_2. Chemically, two distinct RNA objects must exist in W_3, but KEE's approach to world inheritance causes the descriptions of

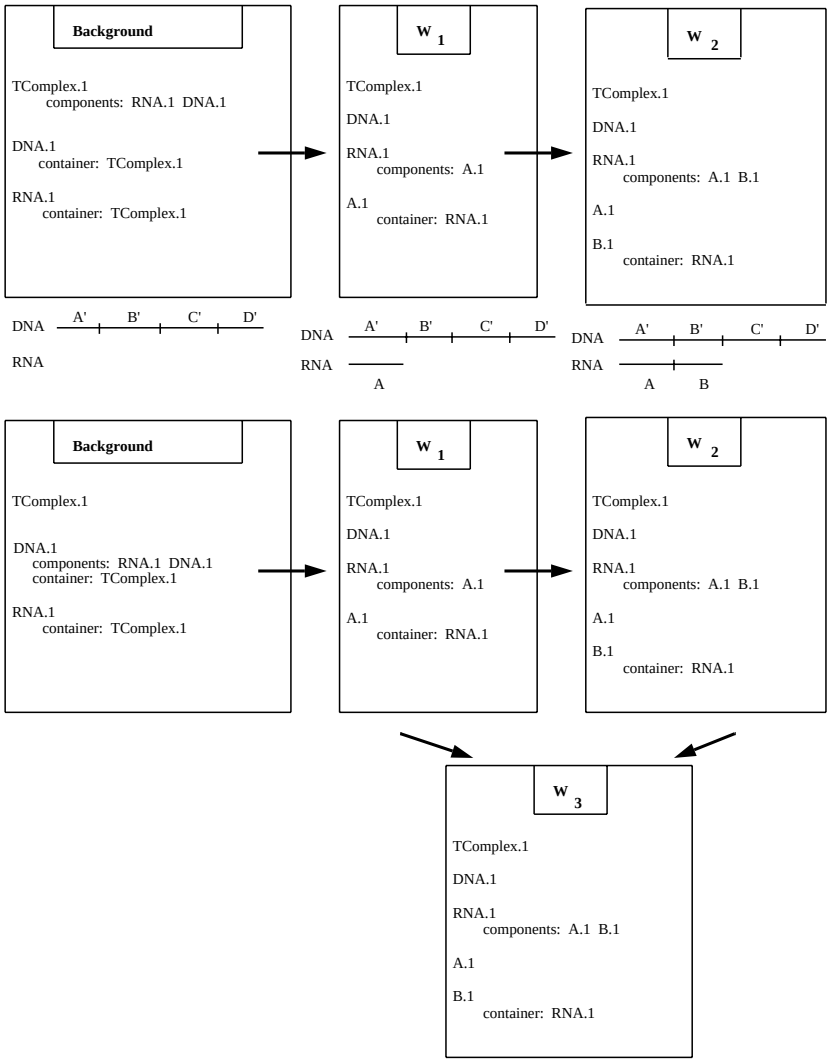

Figure 8. Use of an ATMS to share the descriptions of similar objects during a simulation.

`RNA.1` from W_1 and W_2 to be merged, because both objects have the same name. Because object names are both the basis for sharing information between worlds and the source of the merging problem, the KEEworlds implementation is not able to reduce the space required to represent similar objects.

This limitation would not exist in a worlds mechanism that had the additional functionality of being able to rename objects from one world to anoth-

er. That is, if we were able to rename `RNA.1` to `RNA.2` within W_2, no merging of the two RNA objects would occur in the child world W_3. GENSIM does not use this technique because we lack such an ATMS. (In KEEworlds, frames can be renamed only in the background.)

Other researchers employ an ATMS, but for different purposes: Forbus uses it to represent alternative envisionments more efficiently (Forbus, 1986), and Simmons and Mohammed use it to represent causal dependencies for later analysis by their diagnostic system (Simmons and Mohammed, 1987).

6. GENSIM Trials

I tested the GENSIM program in a number of trial runs. In each trial I used the program to predict the outcome of a different biological experiment. This section describes each trial by stating what objects were present in the initial conditions of the experiment whose outcome GENSIM predicts, and what reactions and new objects were predicted by GENSIM. For some trials, I show the internal structures of objects in the initial conditions or the prediction.

6.1 The Trp Biosynthetic Pathway

This simple trial models the trp biosynthetic pathway, in which a set of enzymes convert chorismate to tryptophan (the current model ignores the reactant serine). The initial conditions of the experiment are shown in Figure 9, the predicted outcome in Figure 10. Figure 11 shows the internal structure of every object in the prediction. GENSIM's prediction is correct in that it omits no reactions that should occur, it includes all reactions that do occur, and the objects produced by each reaction have the predicted parts and properties.

6.2 The Trp Biosynthetic Pathway with a Mutant Enzyme

This trial is a variation of the previous trial. In this trial, the enzyme tryptophan synthetase contains a mutation that prevents it from catalyzing the reaction that converts InGP to tryptophan. The mutation is represented as an object that is part of the tryptophan-synthetase object. GENSIM correctly predicts that the last two (rightmost) reactions in Figure 10 do not occur.

6.3 Transcription of the Trp Leader Region

The leader-region transcription trial focuses on another subset of the overall trp system: the transcription of DNA. Figure 12 shows the objects in the initial conditions of this experiment, which include a truncated version of the trp operon called `Trp.Leader.Region.1` (I removed all the genes in the operon to make this trial easier to describe), the enzyme RNA polymerase, the trp-aporepressor protein, and tryptophan. The prediction generated by

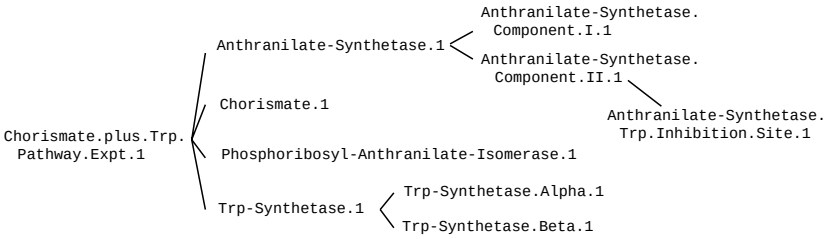

Figure 9. The initial conditions of the trp biosynthetic-pathway experiment. Every object in this figure contains the objects to its right as parts. For example, the `Trp-Synthetase.1` *enzyme has two parts: the alpha and beta subunits of the protein. The experiment as a whole is represented by the object* `Chorismate.plus.Trp.-Pathway.Expt.1`*, which contains all the objects in the experiment as parts.*

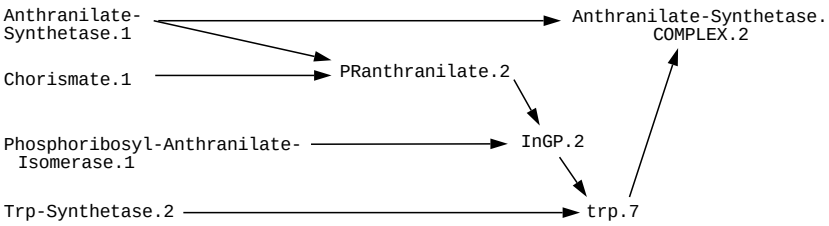

Figure 10. The predicted outcome of the trp biosynthetic-pathway experiment. The lines in this figure indicate the process firings whereby objects react to create the objects to their right. For example, `Trp-Synthetase.2` *reacts with* `InGP.2` *to form* `trp.7`*.*

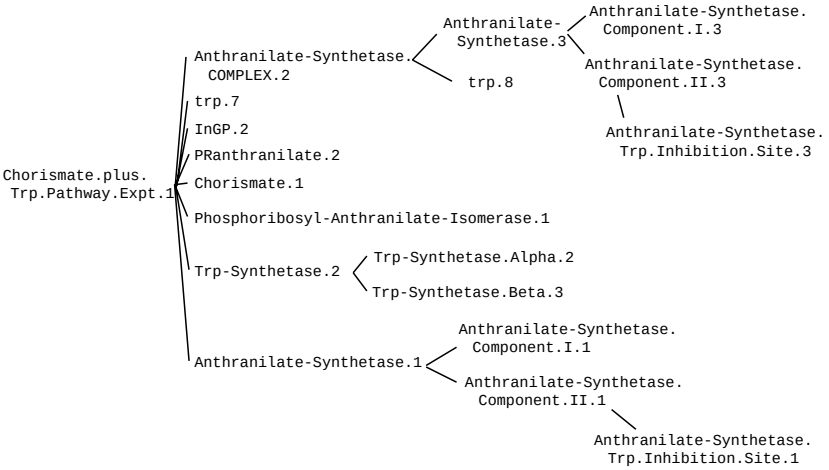

Figure 11. The internal structures of the objects predicted to be present at the end of the trp biosynthetic-pathway experiment.

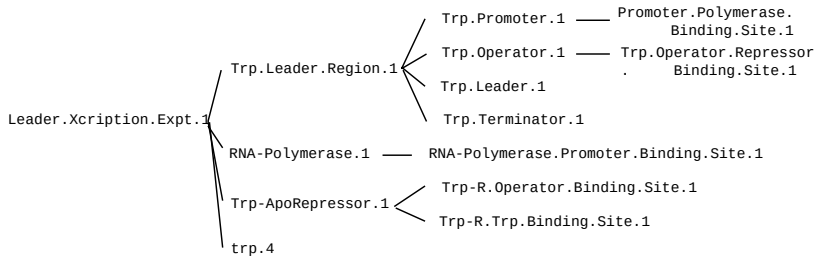

Figure 12. The objects in the initial conditions of the leader-region—transcription experiment.

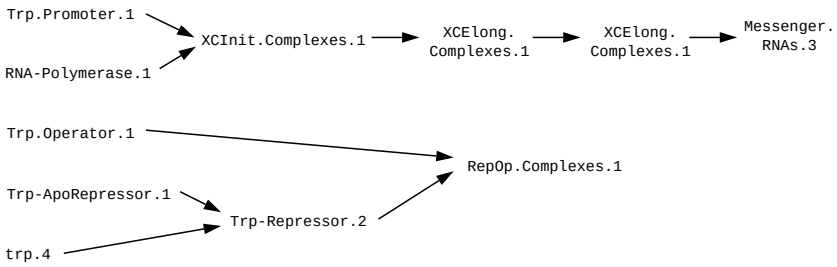

Figure 13. The outcome of the leader-region transcription experiment that was predicted by GENSIM.

GENSIM is shown in Figure 13. This prediction is correct. The two sequences of reactions in this experiment fork the population of trp operon leader-region DNA into two classes: those whose operator regions bind to the activated repressor protein `Trp-Repressor.2`, and those whose promoters bind to `RNA-Polymerase.1` and undergo transcription to produce a messenger RNA.[3] The figures do not name `Trp.Leader.1` as participating in these reactions, but rather name the components of the operon that react: the promoter `Trp.Promoter.1` and the operator `Trp.Operator.1`.

The internal structure of one of the transcription-elongation complexes is shown in Figure 14. A transcription-elongation complex contains RNA polymerase, the DNA that RNA polymerase is transcribing, and the mRNA that RNA polymerase has synthesized thus far. The number of segments (parts) within the mRNA reflects the length of DNA that RNA polymerase has traversed. Since the mRNA contains two segments, we can infer that RNA polymerase traveled two segments along the DNA to produce this transcription-elongation complex. Figure 15 shows the internal structures of every object in this experiment.

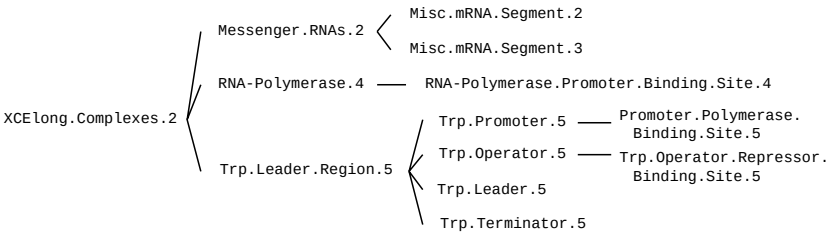

Figure 14. The internal structure of a transcription-elongation complex.

6.4 The Full Trp System

This trial simulates the entire trp system as it was known in the late 1960s. Figure 16 shows the initial conditions of this experiment. GENSIM's prediction is shown in Figures 17 and 18. Figure 17 shows the transcription of the trp operon by RNA polymerase, which yields a free mRNA (`Messenger.RNAs.18`). Some of this mRNA is degraded into its constituent bases by the enzyme `RNase.1`. The mRNA also reacts with ribosomes, as shown in Figure 18. `Messenger.RNAs.18` contains five ribosome-binding sites, including `Ribosome.Binding.Site.47`.[4] Each binding site attracts a ribosome, which translates the five different regions of mRNA into polypeptides such as `Trp-Synthetase.Beta.1` Some of these polypeptides bind together to form larger, functional proteins, such as `Trp-Synthetase.1`.

The enzymes produced from translation of the trp-operon mRNA react with chorismate to carry out the steps in the trp pathway. The trp thus produced enters into several reactions: It binds to and inhibits anthranilate synthetase, and it activates the trp aporepressor protein (the latter complex then binds to the trp operator). Finally, the trp-tRNA-synthetase enzyme catalyzes the binding of $tRNA^{trp}$ and trp to form charged $tRNA^{trp}$ (which is used in all protein synthesis).

Generation of this prediction required approximately 70 minutes of Dorado (Xerox 1132 LISP machine) CPU time. The prediction contained a total of 1050 objects (including components).

7. Related Work

Here I review the work of other AI researchers who have created models of biochemical systems, comparing and contrasting their techniques with those described in this article:

• Meyers' model of the life cycle of Lambda phage (Meyers, 1984)

• Round's model of the E. coli trp operon (Round, 1987)

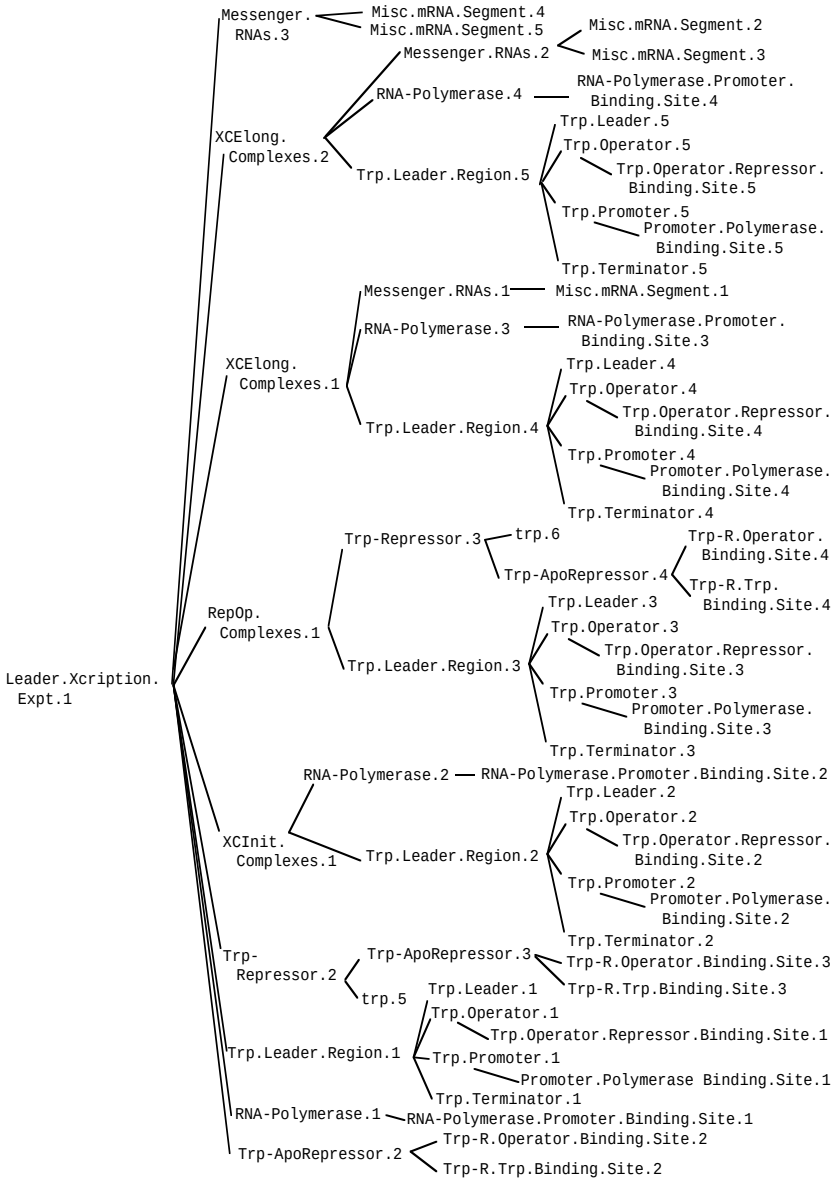

Figure 15. The objects in the predicted outcome of the leader-region-transcription experiment.

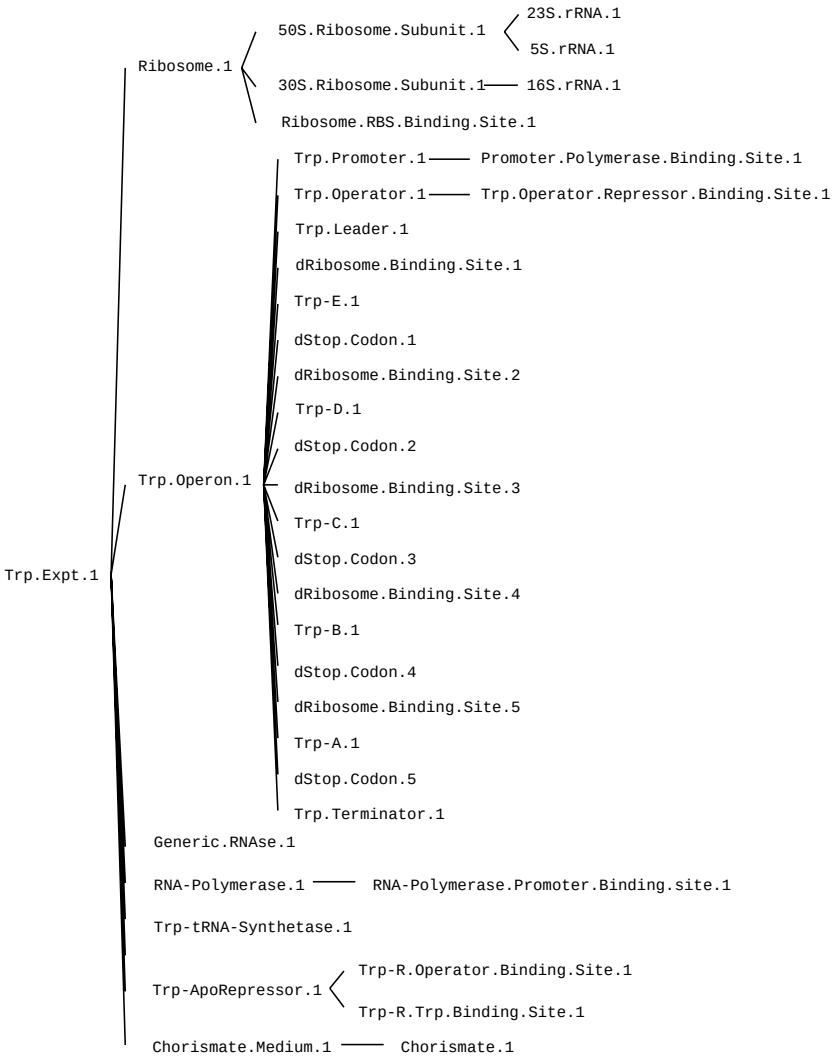

Figure 16. The initial conditions of the experiment involving the full trp system.

- Weld's PEPTIDE model of biochemical reactions (Weld, 1984; Weld, 1986)
- Koton's GENEX model of gene expression (Koton, 1985)
- Karp's earlier model of the trp operon (Karp and Friedland, 1989)

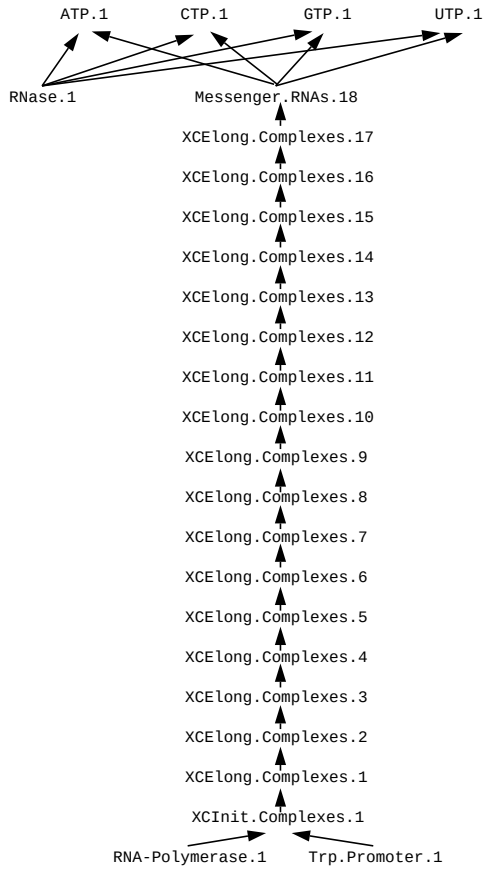

Figure 17. Simulation of the trp system, part 1. This figure shows a simulation of transcription of DNA and degradation of the resulting mRNA. The nodes of the graph are the names of objects (many objects in this figure contain 10 to 30 component objects). The links in the figure connect the reactants and products of each reaction; for example, `Trp.Promoter.1` *and* `RNA-Polymerase.1` *reacted to yield the transcription-initiation complex* `XCInit.Complexes.1`.

- Koile and Overton's model of the life cycle of the HIV virus (Koile and Overton, 1989)
- Brutlag *et al*'s model of DNA metabolism (Brutlag *et al*., 1991)
- Mavrovouniotis' model of intermediary metabolism (Mavrovouniotis, 1990; see also Mavrovouniotis, this volume)

Virtually all of these researchers represent objects as frames that describe

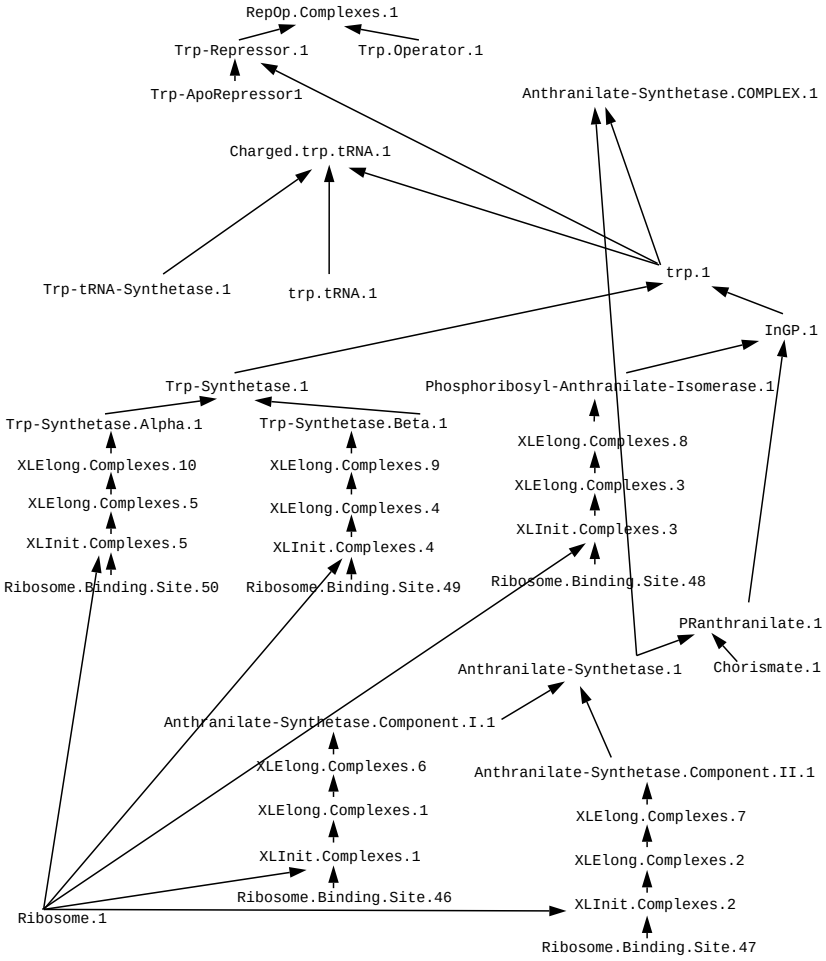

Figure 18. Simulation of the trp system, part 2. This figure shows the translation of the trp operon mRNA to produce the enzymes in the trp biosynthetic pathway, the reactions catalyzed by these enzymes, and the reactions involved in repression of the trp operon. Each ribosome-binding site named in this figure is a component of the `Messenger.RNAs.18` *object in the previous figure.*

object properties and object part-whole structures. Exceptions are Koton, who used PROLOG clauses to represent this information, and Mavrovonioutis, who used Flavors. Although most workers represented object parts such as binding sites, none approached the complexity used for example in the GENSIM representation of a transcription-elongation complex (see Figure 14). In

addition, none of the other programs represented classes of biological objects that were automatically instantiated to describe particular experiments, as described in Section 3. Koile and Overton represented little or no information about object properties or structures; their approach focuses on object quantities such as concentrations.

Only GENSIM and Weld's PEPTIDE perform object forking (in Weld's terminology he "splits the histories" of the "nodes" that represent populations of molecules—he also merges histories when identical nodes are produced by different reactions, which is identical to GENSIM's object merging). Koton states (p. 33) that her approach is to focus only on those binding reactions with the highest affinities, and not to simulate all possible competing reactions. Section 5 of this article provides the most thorough analysis of the reasons to perform object forking, and of ways to optimize this procedure.

The different researchers took varying approaches to representing quantitative information about the biological systems they modeled. Like GENSIM, Koton represented no quantitative information at all. Mavrovonioutis, Weld, Koile, and Karp (Karp and Friedland, 1989) applied techniques developed for qualitative physics (and developed new techniques) to their biological systems. They could represent very coarse quantitative information such as "the concentration of the HIV primary transcript is greater than zero" (Koile and Overton, 1989), or somewhat more precise order-of-magnitude information such as "the concentration of *X* is much greater than the concentration of *Y*" (Mavrovouniotis, 1990). Mavrovonioutis also used chemical theory to estimate values for kinetic and thermodynamic parameters of his system. Meyers represented reaction rates and protein concentrations as real numbers, but these numbers were not solidly grounded in experimental data. Brutlag is the only researcher to accurately represent and reason with experimental conditions such as temperature and pH.

All of the researchers represented reactions using some type of condition—action formalism. Meyers employed production rules. Round developed a process-description language that allows a user to decompose processes into their component subprocesses. Brutlag used KEE production rules. Koton used PROLOG rules containing *preconditions* and *postconditions* to represent reactions; Weld used a similar precondition-postcondition representation, but in addition each reaction had a *pattern* that was equivalent to GENSIM's parameter object classes. Section 5 shows that separating the parameter object classes from the rest of the preconditions facilitates the indexing of processes according to the objects whose behaviors they describe and leads to a more efficient simulation algorithm. Koile and Overton encode reactions using Forbus' qualitative process theory (see next paragraph). None of the systems except for GENSIM arrange reactions in an inheritance hierarchy, which is useful for describing complex reactions in a modular fashion. This approach may also be useful for hypothesis formation because by sum-

marizing knowledge of known classes of reactions the taxonomy provides expectations for the types of unknown reactions we are likely to discover. Only this article and Weld's work describe simulation algorithms in any detail; Weld's algorithm is equivalent to the first GENSIM algorithm presented in Section 5. Only GENSIM and PEPTIDE allow quantification within reaction preconditions and postconditions, which provides the ability to express complex reactions.

Forbus (Forbus, 1984; Forbus, 1986) and Simmons and Mohammed (Simmons and Mohammed, 1987) use a notion of process in their process-modeling systems that is similar to that used by GENSIM; all include parameters, preconditions, and effects. GENSIM processes are most similar to what Forbus terms *individual views*, because both are concerned with creating and deleting objects, and with altering relations between objects. Those things that Forbus calls processes describe how quantities change in a physical system, whereas GENSIM has no notion of quantity. Forbus briefly discusses the use of an abstraction hierarchy to define processes, although he did not implement this idea (Forbus, 1984, p. 44). The main difference in our approaches is that for a process P_1 to be a specialization of another process P_2, Forbus requires that the parameter-object classes, preconditions, and quantity conditions of P_1 must be a subset of those of P_2. Since our `Preconditions.M` slot allows parent preconditions to be removed in a child process, we do not use this restriction. In addition, Forbus does not discuss inheritance from multiple parents. There are also differences between our process-definition languages. The preconditions of Forbus' processes must contain a conjunctive list of (possibly negated) predicates; the preconditions of GENSIM processes can include arbitrary predicate-calculus formulae, including disjunction and quantification. In addition, GENSIM and the system built by Simmons and Mohammed allow process effects to be conditionalized and universally quantified; Forbus does not allow this.

Weld addressed the problem of using a model of a given chemical reaction to predict the aggregate behavior of many molecules that undergo the same reaction. His technique is similar to mathematical induction. The *aggregation* technique allows him to predict the transcription of an entire gene by reasoning about an individual transcription-elongation event that advances RNA polymerase one base along a DNA strand. This technique reduces the execution time of simulations and produces simpler and more understandable causal explanations, but it has an important limitation: Weld's program cannot predict the final sequence of the transcribed RNA because the aggregation technique does not copy every base from the DNA to the RNA; in fact, PEPTIDE does not even know that the length of the mRNA is the same as the length as the DNA from which it was transcribed. For similar reasons, his program would be unable to predict what proteins would be translated from a given mRNA (as Weld notes on p. 52 of (Weld, 1984)).

8 Conclusions

This article has presented a qualitative biochemistry: an ontology for biochemical objects and reactions. It has also presented and analyzed methods for simulating biochemical systems. The ontology and the simulation methods provide a framework for representing theories of molecular biology, and for using these theories to predict outcomes of biological experiments. A user describes a theory by building a knowledge base of classes of objects that are relevant to the biochemical system (the CKB), and a knowledge base of processes that describe potential interactions among these objects (the PKB). A user describes the initial conditions of an experiment by constructing a knowledge base that describes the objects present at the start of the experiment. Predictions are computed by a process interpreter that uses the knowledge of reactions in the PKB to detect and simulate the interactions among the objects in the experiment.

The limitations of the GENSIM framework are that it does not allow us to represent quantitative aspects of the trp operon, nor does it allow us to represent temporal or complex spatial information. Although many interesting problems in this domain do not involve time, space, or quantities, many do. In addition, GENSIM incorporates the assumption that its predictions span a sufficiently short time interval that no population of objects within a simulation will be fully depleted. If we required GENSIM to reason about temporal aspects of the regulation of the trp operon, this assumption would be violated.

The contributions of this research include methods for representing the decomposition of objects into their component parts, and for instantiating structured objects from class descriptions. GENSIM processes are more expressive than are many production-rule languages because process preconditions can include negation, disjunction, and quantification. The PKB describes both particular chemical reactions and general classes of chemical reactions; it also uses KEE's frame inheritance to define processes in a novel way. This use of inheritance is applicable to traditional production-rule languages as well as to GENSIM's process-description language.

To produce correct simulations, GENSIM uses object forking to manage objects during a simulation, and restricts the syntax of process preconditions to eliminate process descriptions that have no valid chemical interpretation. I identified several optimizations to the simulator: under certain conditions we can modify objects directly rather than copying and then modifying (forking) them, we can merge the descriptions of identical objects that are created during a simulation, and we can share the descriptions of similar objects in a simulation using an ATMS. I also presented a novel algorithm that GENSIM uses to activate processes efficiently.

I tested GENSIM by predicting the outcomes of several experiments in the

bacterial tryptophan operon from the history of attenuation. The program produced flawless predictions of the outcomes of experiments involving the trp biosynthetic pathway, the transcription of the trp operon, and the entire trp operon gene regulation system.

Acknowledgements

This work was supported by funding from NSF grant MCS83-10236, NIH grant RR-00785, DARPA contract N00039-83-C-0136, and by the National Library of Medicine. This work benefited greatly from many discussions with Peter Friedland, Charles Yanofsky, Bruce Buchanan, and Edward Feigenbaum. Dan Weld contributed comments on a draft of this article.

Notes

1. It is important to note that this is a simplified conception of experiments. Often experimenters both establish initial experimental conditions, and perturb the system at later times (often using a complex protocol) by adding reagents, applying heat or cold, etc.
2. For simplicity, we show the new objects `A.1` and `B.1` in W_1 and W_2 only, but as noted earlier, new objects must be created in the background.
3. Most bacteria contain only a single copy of the trp operon. Thus, when I refer to the population of operons, I assume that the experiment involves many cells, each of which has a trp operon.
4. I have modeled the trp operon as it was known before Platt's experiments revealed the presence of a ribosome-binding site in the leader region of the operon (Platt and Yanofsky, 1975).

References

D. G. Brutlag, A. R. Galper, and D. H. Millis. Knowledge-based Simulation of DNA Metabolism: Prediction of Enzyme Action. *Computer Applications in the Biosciences*, **7**(1):9-19, 1991.

J. DeKleer. An Assumption-based TMS. *Artificial Intelligence*, 28(1):127-162, 1986.

J. De Kleer and J. S. Brown. A Qualitative Physics Based on Confluences. *Artificial Intelligence*, 24(1-3):7-84, 1984.

K. Forbus. Qualitative Process Theory. Technical Report TR-789, Massachusetts Institute of Technology AI Laboratory, 1984.

K. Forbus. The Qualitative Process Engine. Technical Report UIUCDCS-R-86-1288, University of Illinois Computer Science Department, 1986.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

J. Y. Hsu. On the Relationship Between Partial Evaluation and Explanation-based Learning. Technical Report Logic-88-10, Stanford University, 1988.

IntelliCorp. *KEEworlds Reference Manual*, 1986.

P.D. Karp. *Hypothesis Formation and Qualitative Reasoning in Molecular Biology.* PhD thesis, Stanford University Computer Science Department, June 1989. Technical reports STAN-CS-89-1263, KSL-89-52.

P.D. Karp. Hypothesis Formation as Design. In *Computational Models of Discovery and Theory Formation*. Morgan Kaufmann Publishers, 1990. (See also Stanford Knowledge Systems Laboratory report KSL-89-11.)

P. D. Karp and P. E. Friedland. Coordinating the Use of Qualitative and Quantitative Knowledge in Declarative Device Modeling. In *Artificial Intelligence, Modeling and Simulation*, pages 189-206. John Wiley and Sons, 1989. See also Stanford Knowledge Systems Laboratory report KSL-87-09.

T. P. Kehler and G. D. Clemenson. KEE, the Knowledge Engineering Environment for Industry. *Systems And Software*, 3(1):212-224, 1984.

K. Koile and G. C. Overton. A Qualitative Model for Gene Expression. In *Proceedings of the 1989 Summer Computer Simulation Conference*, 1989.

P. Koton. Towards a Problem Solving System for Molecular Genetics. Technical Report 338, Massachusetts Institute of Technology Laboratory for Computer Science, 1985.

M. Mavrovouniotis. Group Contributions for Estimating Standard Gibbs Energies of Formation of Biochemical Compounds in Aqueous Solution. *Biotechnology and Bioengineering*, 36:1070-1082, 1990.

S. Meyers. A Simulator for Regulatory Genetics and its Application to Bacteriophage Lambda. *Nucleic Acids Research*, 12(1):1-9, 1984. Also available as Stanford Heuristic Programming Project report HPP-83-12.

P. H. Morris and R. A. Nado. Representing Actions with an Assumption-based Truth Maintenance System. In *Proceedings of the 1986 National Conference on Artificial Intelligence*, pages 13-17. Morgan Kaufmann Publishers, 1986.

T. Platt and C. Yanofsky. An Intercistronic Region and Ribosome-binding Site in Bacterial Messenger RNA. *Proceedings of the National Academy of Sciences, USA*, 72(6):2399—2403, 1975.

A. D. Round. QSOPS: A Workbench Environment for the Qualitative Simulation of Physical Processes. Technical Report KSL-87-37, Stanford Knowledge Systems Laboratory, 1987. Also appears in Proceedings of the European Simulation Multiconference.

R. Simmons and J. Mohammed. Causal Modeling of Semiconductor Fabrication. Technical Report 65, Schlumberger Palo Alto Research, 1987.

M. Stefik and D. G. Bobrow. Object-oriented Programming: Themes and Variations. *AI Magazine*, 6(4):40-62, 1986.

D. S. Weld. Switching between Discrete and Continuous Process Models to Predict Molecular Genetic Activity. Technical Report TR-793, Massachusetts Institute of Technology AI Laboratory, 1984.

D. S. Weld. The Use of Aggregation in Causal Simulation. *Artificial Intelligence*, 30:1-34, 1986.

B. C. Williams. Doing Time: Putting Qualitative Reasoning on Firmer Ground. In *Proceed-*

ings of the 1986 National Conference on Artificial Intelligence, pages 105-112. Morgan Kaufmann Publishers, 1986.

C. Yanofsky. Attenuation in the Control of Expression of Bacterial Operons. *Nature*, 289:751-758, 1981.