Polynomial-time Compilation of Self-Explanatory Simulators

Kenneth D. Forbus
Qualitative Reasoning Group, The Institute for the Learning Sciences
Northwestern University
1890 Maple Avenue, Evanston, IL, 60201, USA

Brian Falkenhainer
Xerox Modeling Research & Technology
1350 Jefferson Rd, Henrietta, NY, USA

Abstract: Self-explanatory simulators have many potential applications, including supporting engineering activities, intelligent tutoring systems, and computer-based training systems. Yet compilation methods have been too slow for large-scale systems, interpreter-based strategies are restricted to running on large computers with expensive commercial software, and neither technology has been shown to scale to very large systems. This paper describes an algorithm for compiling self-explanatory simulators that operates in polynomial time. It is capable of constructing self-explanatory simulators with thousands of parameters. This algorithm is fully implemented, and we show empirical evidence that suggests that its performance is quadratic in the size of the system being analyzed. We also analyze the tradeoffs between compilers and interpreters for self-explanatory simulation in terms of application-imposed constraints, and discuss plans for applications.

1. Introduction

Self-explanatory simulators [1 ,2 ,3 ,4] integrate qualitative and quantitative knowledge to produce both detailed descriptions of the behavior of a system and causal explanations of how that behavior comes about. They have many potential applications, such as in intelligent tutoring systems and learning environments [5 , 6] and in supporting the design process [7 ,8]. Realizing this potential requires both developing systems that can operate efficiently on substantial models and understanding the tradeoffs involved in the automatic construction of self-explanatory simulators. This paper makes two contributions towards these goals. First, we describe a polynomial-time method for compiling self-explanatory simulators, and show that it operates successfully and quickly on models larger than most industrial applications require. Second, we analyze how the quality of the simulator produced trades off against the time taken to construct it, and consider how these tradeoffs affect potential applications. Throughout, the discussion is limited to initial-value simulations of lumped-element (ordinary differential-algebraic) systems.

Section 2 reviews the basic idea of self-explanatory simulators and the relevant literature. Section 3 describes our new polynomial-time compilation technique, including both theoretical and empirical complexity analyses. Section 4 identifies tradeoffs in constructing self-explanatory simulators in light of task requirements. Section 5 summarizes and outlines our plans for future work.

2. Self-explanatory simulation: The basics

Traditional numerical simulators generate predictions of behavior via numerical computation using quantitative models of physical phenomena. Most simulators are written by hand, although an increasing number are generated by domain-specific toolkits (e.g., SPICE for electronics). The modeling decisions, such as what phenomena are important to consider, how does the phenomena work, how can it be modeled quantitatively, and how can the quantitative model be implemented for efficient computer solution, are mostly made by hand. Domain-specific toolkits provide fairly good solutions to the last two problems, and by what libraries they do or do not include, can simplify the first problem. However, the choices of how to translate the physical description into the conceptual entities supported by the toolkit, and which quantitative model to use from the library provided by the toolkit to model an entity, are still made by hand. Moreover, no existing toolkit provides the intuitive explanations used by scientists and engineers to describe how a system works. These intuitive, qualitative descriptions serve several important purposes in building and working with simulations. First, they guide the formulation of quantitative models by identifying what aspects of the physical situation are relevant. Second, qualitative descriptions are used to check the results of a simulation, to ensure that it "makes sense." Thus two advantages of self-explanatory simulation are *increased automation* and *better explanations* [1].

Self-explanatory simulators harness the formalisms of qualitative physics to automate the process of creating simulators. Given an initial physical description, a qualitative analysis of the situation reveals what conceptual entities are relevant to the task at hand, and identifies what causal factors affect a parameter under different circumstances. This information is then used, in concert with quantitative information in the domain theory, to construct appropriate numerical programs for simulating the system. By incorporating explicit representations of conceptual entities (such as physical processes) in the simulator, causal explanations can be given for the simulated behavior. Moreover, the qualitative representations provide some of the same opportunities for automatic "reality checks" that an expert would apply in evaluating a simulation. For instance, a simulation of a fluid system which reported a negative amount

From the Proceedings of QR-94, Nara, Japan, July, 1994

1 Create a scenario model via instantiation of model fragments from the domain theory.
2 Analyze the scenario model to define the appropriate notion of state for the simulator:
  2.1 Extract physical and conceptual entities.
  2.2 Define boolean parameters (i.e., logical variables corresponding to conditions such as valves being open or closed, or physical processes being active).
  2.3 Define numerical parameters & relevant comparisons among them.
  2.4 Extract influences to create causal ordering
3 Write simulator code
  3.1 Simplify boolean parameters
  3.2 Compute update order for numerical parameters using the influence graph and for boolean parameters using logical dependencies between them.
  3.3 Write code to evolve state descriptions (*evolver*).
  3.4 Write code to detect state transitions (*transition finder*)
  3.5 Write code to detect inconsistencies (*nogood checker*)
  3.6 Write structured explanation system

Figure 1: The SIMGEN Mk3 Algorithm

of liquid in a container is not producing realistic results. The ability to detect such conditions is called *self-monitoring*.

The first systems to generate self-explanatory simulators were compilers. That is, the creation of a simulator was done off-line, with the goal of producing code whose execution would asymptotically approach the speed of traditional numerical simulators while providing services they did not (i.e., explanations and increased self-monitoring). This goal was met, but only at the cost of high compilation times. For example, SIMGEN Mk1 [1] used envisioning for its qualitative analysis procedure. This provided powerful explanation capabilities (including answering counterfactual questions via comparing the simulated behavior to alternatives in the envisionment) and a high degree of self-monitoring, because the numerical state of the simulator could be checked against a complete qualitative state. Unfortunately, envisioning, like all non-resource limited forms of qualitative simulation, is exponential in the size of the system being analyzed. A closer analysis of what simulation authors do led to the substitution of a simpler qualitative analysis system: Qualitative simulation is simply not necessary for simulation construction. A person writing a numerical simulator never explicitly identifies (with the possible exception of defining initial conditions) even a single global qualitative state of the system being simulated. SIMGEN Mk2 [2] used a qualitative analysis procedure that avoided the obviously exponential steps in qualitative reasoning that are unnecessary given quantitative information, such as branching on all ways to resolve an ambiguous influence on a variable or all possible combinations of state transitions. This compiler could construct simulators of systems larger than any envisioning-based system ever could (i.e., involving dozens to hundreds of parameters), such as a twenty stage distillation column [9]. The tradeoff is that some explanatory capabilities (i.e., efficient counterfactual reasoning) and self-monitoring capabilities (i.e., the guarentee that every numerical simulator state satisfied some legal qualitative state) were lost. However, as Section 3 explains, even this compiler was still subject to combinatorial explosions.

An alternative to compiling self-explanatory simulators is to build simulators that act as *interpreters*, i.e., that interleave model-building, model translation into executable code, and code execution[3,4]. In PIKA [4], for example, Mathematica is used in conjunction with a causal ordering algorithm to produce a decomposition of a set of equations into independent and dependent parameters, along with an order of computation to update dependent parameters. Every state transition which changes the set of applicable model fragments reinvokes Mathematica and causal ordering to produce a new simulator for the new state. (An incremental constraint system was proposed to minimize this cost.) Part of the motivation for such systems was the perceived slowness of compiler techniques: By only building models for behaviors that are known to be relevant, presumably the entire time from formulation of the problem to solution would be reduced, even though any particular execution of a simulator might be slower due to the need to perform reasoning during simulation. Such systems are not themselves immune from combinatorial explosions (see Section 4), but on some examples can exhibit impressive performance.

It should be noted that some of the specific performance claims made for PIKA are problematic, e.g., in [4] it is claimed that PIKA is "5,000 times faster than SIMGEN Mk2." There are several minor problems with this claim, such as the fact that the performance difference is ten times less on the other example for which data is available about both systems (e.g., boiling water), and it is not clear what the relative performance difference between the different computers used is. However, the worst problem is that the domain theories used by each system are substantially different: PIKA uses just two model fragments, with choices for quantitative models "hard wired" into these fragments. By contrast, in keeping with the goal of increased automation, the domain models used in SIMGEN Mk2 were basically the same as used in other qualitative reasoning systems, with quantitative information added in a modular fashion. In the SIMGEN Mk2 quantitative models, for instance, quantitative parameters such as fluid and thermal conductances and container sizes were explicit variables that could be set by the simulation user at run time. In PIKA such information was hard-wired into the model fragments in the form of example-specific numerical constants, which is not very realistic.

3.SIMGEN Mk3: A polynomial-time compiler for self-explanatory simulators

Previous work has shown that the advantages of self-explanatory simulation can be achieved in several ways. To better understand the tradeoffs between these methods, we wanted to figure out how fast self-explanatory simulation compilers could be. If such systems were inherently exponential, then the range of applications for them would be strictly limited. If, on the other hand, self-explanatory simulators could be compiled in polynomial time (preferably low-order, of course), then with enough software engineering such compilers could be used in a broad range of applications.

We have succeeded in developing a polynomial-time algorithm for compiling self-explanatory simulators. This has required some important simplifications, which reduce some of the advantages of self-explanatory simulators. However, the ability to quickly generate simulators for systems containing thousands of parameters suggests that these simplifications are worthwhile.

The rest of this section describes our algorithm. As we explain the algorithm we analyze its complexity. Since each step contains so many subprocesses, we only show that each step is polynomial, instead of attempting to theoretically derive a concrete bound. We evaluate its performance, and show empirical data that suggests that its performance is quadratic in the size of the input description.

3.1 The SIMGEN Mk3 Algorithm

The structure of the algorithm is shown in Figure 1. Its overall structure is similar to SIMGEN Mk2 [2], so in the rest of this section we focus mainly on the tradeoffs made to achieve polynomial time performance.

3.1.1 Creation of the scenario model

We assume domain theories are written in a compositional modeling language, using Qualitative Process theory [10] to provide their qualitative aspects. A key tradeoff is how much qualitative reasoning should be performed. More qualitative reasoning provides more constraints on the system's behavior, which can be exploited to generate more compact code and better self-monitoring, but at the price of more inference. In fact, the cost of qualitative reasoning was by far the dominant cost in SIMGEN Mk1 and Mk2. Interpreters like PIKA [4] and DME [3] appear to do no qualitative reasoning beyond instantiating model fragments corresponding to equations, which we suspect is the main factor responsible for their efficiency. Consequently, we minimized the amount of qualitative reasoning in this compiler to see the consequences of that design decision. Specifically, we use a qualitative reasoner to instantiate model fragments and draw certain trivial conclusions (i.e., if $A>B$ then $\neg A=B$). No transitivity inferences are made: In earlier compilers, these inferences were the dominant cost in qualitative reasoning, both because the number of such conclusions rises combinatorially with the size of the systems and the ATMS label update algorithm tended to go exponential in that subsystem. No attempt is made to resolve influences, nor to compute possible state transition conditions (i.e., limit analysis). The consequences of this decision are made clear in subsequent sections.

We use TGIZMO, a publically available QP implementation [11] for our qualitative reasoner. We modified it in two ways. First, the pattern-directed rules that implement many important QP operations were simplifed, to strip out aspects of reasoning not needed by the compiler (e.g., transitivity reasoning over ordinal relations). Second, the modeling language implementation was modified so that logical antecedents for specific kinds of facts (e.g., whether or not a physical process is active) are explicitly asserted in the database as well as being represented via clauses in TGIZMO's LTMS [12]. This information is needed by the compiler in order to write code for updating the truth values of dynamic booleans, i.e., those statements whose truth values may change during simulation. In previous compilers this information was gleaned from the statement's ATMS label, but we chose to use an LTMS [12] instead to avoid the potential exponential growth of labels [13].

The only combinatorial explosions that occurred in previous compilers occurred in the qualitative analysis phase, so the complexity of this step is crucial. Its time complexity is a function of the cost of instantiating model fragments and the cost of drawing conclusions with them. The cost of instantiation can be decomposed into two factors: The cost of pattern matching, and the size of the description produced by the operation of the system's rules. The cost of pattern-matching is polynomial in the number of antecedents [11]. We assume that both the scenario model and the domain theory are finite, and that the number of new entities introduced by the domain theory for any scenario model is a polynomial function of the size of the scenario model. We further assume that at worst the number of clauses instantiated about any particular statement in the scenario model is bounded by a polynomial. (It is easy to construct domain theories which violate these assumptions if one tries [14], but in practice these assumptions are always satisfied.) Since the work of instantiation is the product of the number of instantiations and the work to perform each, the instantiation process is polynomial-time. Furthermore, the dependency network so created is polynomial in size, as a function of the size of the domain theory and scenario model. This means that the cost of inference remains polynomial in these factors, since we use an LTMS, for which the cost of inference is worst-case linear in the size of the dependency network [11]. We thus conclude that the time complexity of this step is polynomial.

3.1.2 Constructing the simulator's state

In this step the results of the qualitative analysis are harvested to create specifications for what information comprises a state of the simulator. The numerical parameters of the state include the quantities mentioned in the qualitative model. The statements for which boolean parameters are introduced are the existence of individuals, the existence of quantities, activation status of processes and views, and any statements mentioned in the antecedents of these `EXIST` and `ACTIVE` statements (except for ordinal relations, which are handled seperately). Each boolean parameter has an associated *antecedents* statement, a necessary and sufficient condition for the truth of the corresponding statement.

Any truth values known in the scenario model are presumed to hold universally over any use of the simulator. If for example a fluid path is assumed to be aligned, every behavior of the simulator will be generated assuming this fact. (The compiler is clever enough to not generate simulator parameters for such statements, although they are still woven into the explanation system appropriately.) Every truth value that is not known is treated as something that must be ascertained at runtime. This technique allows the compiler to produce tighter code by exploiting constraints of the domain and any hints from the user. A simple symbolic evaluation procedure is used to test such constraints. This symbolic evaluator is used for such tasks as ascertaining what statements are universal and simplifying antecedents statements to produce tighter code.

Most of the work in this step consists of fetching information from the TGIZMO database and constructing corresponding internal datastructures in the compiler, which is obviously polynomial time. The only other potentially expensive part of this computation is the symbolic evaluation procedure. This procedure is simply a recursive analysis of propositional statements, checking the (preexisting) LTMS labels of ground terms at the leaves, and so it too is polynomial.

By comparison, when an ATMS is used such simplifications are automatically performed by the label update mechanism, which can take exponential time. The extra information available in ATMS labels was used for several optimizations in SIMGEN Mk2, including merging tests for ordinal relations that could be proven to be equivalent.

3.1.3 Writing the simulator code

The ATMS provided a direct connection between a fact and the assumptions underlying it, irregardless of the structure of the dependency network between them. While the use of explicit antecedents is much cheaper, there is a certain inelegance (and runtime inefficiency) in not being able to collapse long chains of inference. (It seems unfair to penalize domain modelers who use compositional modeling appropriately, i.e., by decomposing knowledge into small fragments which are then woven together inferentially in model formulation.) The symbolic evaluator mentioned above addresses part of this problem. The other technique we use (in step 3.1) is to divide the boolean parameters into equivalence classes with canonical members, according to logical dependency. That is, if a boolean parameter A depends only on B, and B in turn depends only on C, and C either has an empty antecedent or an antecedent with more than one ground term, then A, B, and C would be in the same equivalence class, and C would be its canonical member. Each such equivalence class is represented by a single boolean parameter (although each original statement is still part of the explanation system to preserve clarity). Since dividing a set into equivalence classes is polynomial time, this step is also.

In regard to Step 3.2, the *state space assumption*, common in engineering and satisfied by QP models [15], guarentees we can always divide the set of parameters into dependent and independent parts, with the independent parameters being those which are directly influenced (or uninfluenced) and with the dependent parameters computed from them. To gain a similar guarentee for the boolean parameters we must assume that the domain theory is *condition grounded* [14], which again is reasonable for all the domain theories we have seen in practice.

An independent boolean parameter mentions no other boolean parameters in its antecedents. It could be universally true or false, it could be something whose truth value is ascertained at runtime, either as a consequence of the simulation user's assumptions (e.g., the state of a valve) or ordinal relations (e.g., the existence of a contained liquid when there is a non-zero amount of water in the container). A boolean parameter that is not independent is *dependent*.

The update order is found for both numerical and boolean dependent parameters by sorting them according to their maximum distance from the independent parameters, using the graph of influences in the numerical case and the antecedent relations in the boolean case. This is clearly a polynomial-time process. (For comparison, the computation of a boolean update order was unnecessary when an ATMS was used, because the labels could be processed to find an appropriate set of antecedents. Aside from the cost of label updating, the ATMS method could end up producing less efficient code, by not taking advantage of the caching offered via intermediate parameters to eliminate redundant tests.

The overall structure of the code produced by the compiler in steps 3.3 through 3.5 is the same as that produced by SIMGEN Mk2. In evolvers, the effects of direct influences are calculated first to estimate derivatives, the dependent numerical parameters are then updated, followed by the boolean parameters.[1] In transition finders, the limit points of the system are tested to see if any state transitions have occurred, and rollback signals are generated to allow the simulator to modulate its step size to ensure that state transition points are included in the simulated behavior. In nogood checkers, logical constraints are tested against the state of a simulator to warn if the numerical model has diverged from what is qualitatively legal. We summarize the important changes in how they are generated that are caused by restricted inferencing.

The main impact of restricted inferencing in generating evolvers is in the selection of quantitative models for updating dependent numerical parameters. For instance, a domain theory might have two quantitative models for how the level of liquid in a container depends on its amount, one for cylindrical containers and one for rectangular containers.[2] If the compiler knows the shape of the container it can install the appropriate model, otherwise it must write a runtime conditional and provide both models in the simulator. In SIMGEN Mk2 influence resolution was performed to see what combinations of qualitative proportionalities might co-occur, so that appropriate quantitative models could be constructed for each combination. For efficiency this compiler eschews influence resolution, using instead the assumption that the domain modeler has supplied, for each class of dependent parameter, an appropriate set of quantitative models (specified in a manner similar to [1]). The antecedents for these models are used to construct a runtime conditional, ensuring that each model is executed as appropriate. Here compile-time error checking has been sacrificed to efficiency: previous compilers would detect when a quantitative model was not available for a logically possible condition.

---

[1] The default output mode uses Euler integration, since that is often the method of choice for training simulators and minimizes runtime costs, so that we can produce code which runs well on very small machines. However, we have decomposed evolvers into subroutines that can be used with more complex integration methods when needed.

[2] One of our domain theories in fact includes these models.

Example	SIMGEN Mk3	SIMGEN Mk2
Two containers	6.42 seconds	22.8 seconds
Boiling water	5.32 seconds	25.2 seconds
Spring/Block	1.85 seconds	6.4 seconds
3×3 grid of containers	54 seconds	16286 seconds

Table 1: Compilation times for SIMGEN Mk3 vs Mk2 on standard test examples (IBM RS/6000 Model 530, 128MB RAM, Lucid Common Lisp 4.01)

In generating transition finders, restricted inference can result in "dead code," i.e., runtime tests that are moot because they will never occur. Given how cheap inequality tests are, this is not a serious drawback, and many of them are avoided by using the symbolic evaluation procedure to exploit any information that is available about a comparison. Generating nogood checkers is also greatly simplified: Previous versions filtered the ATMS nogood database to find contradictory combinations of assumptions that, if detected in the runtime system, indicated that something is amiss. Empirically, those the filter conditions had to be quite strong, since most of the nogoods would never arise, given the definition of qualitative state in terms of known numerical parameters. SIMGEN Mk3 simply uses the symbolic evaluation procedure to see what ordinal relations are known to be impossible and test for those. In principle this could result in reduced self-monitoring, but in practice this appears to be negligible.

Each of these computations involves simple polynomial-time operations (see [2] and Section 3.1.2) over structures whose size is polynomial in the initial scenario description, so they are polynomial time as well.

3.1.4 Writing the explanation system

The final step in generating a self-explanatory simulator is creating a runtime system that will provide the same causal explanations that were available in the original qualitative analysis, as well as links to the quantitative models used. For this purpose we use a *structured explanation system* that concisely summarizes the qualitative and quantitative analyses. Such systems provide an abstraction layer between a reasoning system and an interface that allows each to be optimized independently. Every conceptual entity, every boolean parameter, and every numerical parameter has an associated element in the explanation system, as well as every influence and every mathematical model. These explanation elements have associated procedures that enable them to evaluate whether or not they hold at any state of the simulation, so that a user can get explanations either while the simulator is operating or as a post-mortem. These explanations are more detailed than those generated by equation-based systems such as PIKA, since they can respond both in qualitative and quantitative terms.

Generating a structured explanation system requires a case analysis of the elements used in earlier steps of the compiler construction. This analysis selects the appropriate class of explanation element and creates the necessary pointers between it and other such elements to provide coherent causal explanations. This translation procedure is is linear time, in the size of the results of the qualitative analysis and the number of quantitative models used by the system. Importantly, restricted inferencing has little effect on the quality of the explanation system: Models are still completely instantiated, so full ontological and causal information remains available.

3.2 Empirical Results

SIMGEN Mk3 is fully implemented, and has been tested successfully on the suite of examples described in [2]. In all cases it is substantially faster than SIMGEN Mk2, as Table 1 shows.

The simulators it produces, like those of SIMGEN Mk2, operate at basically the speed of a traditional numerical simulator, with the only extra runtime overhead being the maintenance of a concise history [1] for explanation generation. Currently the compiler's output is Common Lisp, and even with this performance handicap, the simulators it produces run quite well on even small machines (i.e., Macintosh Powerbooks).

To demonstrate that SIMGEN Mk3's performance is in fact polynomial time, we generated a set of test examples similar to those used in [2]. That is, a scenario description of size n consists of an n by n grid of containers, connected in Manhattan fashion by fluid paths. We generated a sequence of scenario descriptions, with n ranging from 2 to 10. (The reason we chose 10 as an upper bound is that the simulator which results contains just over 2,400 parameters, which is roughly three times the size of the STEAMER engine room mathematical model [16]) Extending the domain theory in [11], contained liquids include mass, volume, level, pressure, internal energy, and temperature as dynamical parameters, as well as other static parameters (e.g., boiling temperature, specific heat, density, etc.). Containers can be either cylindrical or rectangular, with appropriate numerical dimensions in each case. The liquid flow process affects both mass and internal energy. We then ran the compiler to produce simulators for each scenario, to see how its performance scaled.
The results are show in Table 2. In an $n \times n$ grid scenario, there are n^2 containers and $2[n^2\text{-}n]$ fluid paths, so the numbers of parts in these examples ranges from *8* to *280*. The count for quantities includes both static and dynamic parameters, and the count for booleans includes both conditions controllable by the user (e.g., the state of valves) and qualitative state parameters, such as whether or not a particular physical process is occurring. The proposition count is the number of statements in the simulator's explanation system.

Grid Size	# parts	# quantities	# booleans	# propositions	Compile time (seconds)
2	8	83	24	456	6
3	21	198	63	1131	19
4	40	363	120	2108	49
5	65	578	195	3387	105
6	96	843	288	4968	202
7	133	958	399	6851	356
8	176	1523	528	9036	586
9	225	1938	675	11523	927
10	280	2403	840	14312	1429

Table 2: Results of SIMGEN Mk3 on $n \times n$ Manhattan grid

(IBM RS/6000 Model 350, 64MB RAM, Lucid Common Lisp 4.01)

The theoretical analysis in previous sections suggests that the compile time should be polynomial in the number of parts in the system. A least-squares analysis indicates that this is correct: A quadratic model ($0.017P^2 + 0.399P + 4.586$, where P is the number of containers and paths) fits this data nicely, with $X^2 = 0.03$. Additional evidence for quadratic performance is found in Table 3, which shows the compiler's performance on examples constructed out of chains of containers. A chain of length N has $2N-1$ parts, i.e., N containers and $N-1$ fluid paths. A least-squares analysis indicates again that a quadratic model ($0.018P^2 + 0.554P + 0.228$, where P is the number of containers and paths) fits this data well, with $X^2 = 0.004$.

Additional tests are in progress. For example, we plan to translate Sgouros' distillation theory [9] into the simpler format used by SIMGEN Mk3 to measure the performance improvement on it. Since this example was larger than the 3 × 3 container grid, and yet was compiled by SIMGEN Mk2 in less time than that example (1.5 hours versus four hours), we expect substantial speedup on this problem as well.

4.Tradeoffs in self-explanatory simulators

Different applications entail different tradeoffs: In some cases potential users have powerful workstations and can afford the best commercial software (e.g., many engineering organizations), and in some cases potential users have only hand-me-down computers and publically available software (e.g., most US schools). Here we examine the tradeoffs in self-explanatory simulation methods with respect to potential applications.

Broadly speaking, the computations associated with self-explanatory simulations can be divided into three types: (1) *model instantiation*, in which the first-order domain theory is applied to the ground scenario description, (2) *model translation*, in which the equations associated with a state are identified, analyzed, and converted into an executable form, and (3) *model execution*, i.e., using numeric integration to derive descriptions of behavior from a given set of initial values. The choice of compiler versus interpreter is mainly a choice of how to apportion these computations, and the tradeoffs are analogous to those of programming language interpreters and compilers. Interpreters are more suited for highly interactive circumstances, where a substantial fraction of effort is spent changing models compared to running them. Scientists and engineers formulating and testing models of new phenomena and highly interactive, exploratory simulation environments for education are two such applications. Compilers are more suitable for circumstances where the additional cost of compilation is offset by repeated use of the model, or when the environment for model execution cannot support the resources required by the development environment. Engineering analysis and design, where a small number of models are used many times (e.g., in numerical optimization of system properties), and most educational software and training simulators, where maximum performance must be squeezed out of available hardware, are applications where compilers have the edge.

The cost of model generation is dominated by the expressiveness of the representation language for models and the amount of simulator optimization that is performed. In SIMGEN Mk3, the order of computation is specified as an inherent part of the domain theory due to the causal ordering imposed by qualitative process theory influences. Thus, no algebraic manipulation is required at model generation time. Other systems allow a domain theory to contain equations in an arbitrary form. Thus, the equations must be sorted (using a causal ordering algorithm [7]) and symbolically reformulated to match that sort. This technique provides the ease of using arbitrarily-ordered arithmetic expressions, but can lead to exponential behavior for some classes of equations.

Chain Length	# quantities	# booleans	# propositions	Compile Time (sec)
2	38	9	194	2.05
3	58	15	305	3.43
4	78	21	416	5.02
5	98	27	527	6.66
6	118	33	638	8.72
7	138	39	749	10.5
8	158	45	860	12.5
9	178	51	971	14.8
10	198	57	1082	17.8
11	218	63	1193	19.9
12	238	69	1304	22.6
13	258	75	1415	25.6
14	278	81	1526	28.4
15	298	87	1637	31.9
16	318	93	1748	35.1

Table 3: SIMGEN Mk 3 data, linear chain of containers

(IBM RS/6000, 64MB RAM, Lucid Common Lisp 4.01)

Another way in which the representation language for models affects potential applications is in the kinds of explanations that can be generated. Domain theories that explictly represent conceptual entities as well as equations can provide better explanations than those which do not. While in a few domains (e.g., electronics) expert causal intuitions are not strongly directional, in many domains (e.g., fluids, mechanics, thermodynamics, chemistry, etc.) expert causal intuitions are strongly directed [17], and there is no a priori guarentee that the causal accounts produced by causal ordering will match expert intuitions [18]. Using equation-based models reduces the overhead of uncovering and formalizing expert intuitions, but at the cost of reducing explanation quality. Using explicit qualitative representations provides an additional layer of explanations, but at the cost of increased domain theory development time. Interestingly, TGIZMO accounts for less than 15% of SIMGEN Mk3's time, so the penalty for using rich, compositional domain theories appears to be quite small. How these design choices fare in real applications is, of course, an empirical question, and characteristics of task environments often prove suprising. For instance, in [4] it is suggested that PIKA "...isn't quite fast enough to drive a truely interactive simulation [for an embedded multimedia system]" This assumes that the user requires instant feedback on arbitrary model changes. We doubt that this assumption is correct in practice; for example, precompiling a set of simulations for common variations of particular examples would probably cover the majority of interactions with a user community, and our experience with other educational software suggests that users who wanted to try something novel wouldn't mind waiting a minute or two for their simulation. On the other hand, our working assumption that self-explanatory simulation via interpreters is too resource-intensive for most eduational applications could be proven wrong by the combination of advances in computer technology coupled with domain-specific algebraic manipulation systems. Interestingly, even our current implementation of SIMGEN Mk3 can, running on a PowerBook, compile new simulators for small systems reasonably quickly. Both kinds of systems may end up on students' desks and in their homes in the near future.

5.Discussion

Previous work on self-explanatory simulation has produced systems that can handle medium-sized systems (e.g., a few dozen to a few hundred parameters). In this paper we describe a new
algorithm for compiling self-explanatory simulators that extends the range of the technology to systems involving thousands of parameters. We have shown, both theoretically and empirically, that self-explanatory simulators can be compiled in polynomial time, as a function of the size of the input description and the domain theory. This advance was made possible by the observation that minimzing inference could substantially improve performance [4]. These gains are not without costs: SIMGEN Mk3 does less self-monitoring and less compile-time error detection than previous versions. Algebraic manipulation is neither performed at compile time nor at run time, for example, and the simulators produced can contain code that will never actually be executed. On the other hand, no explanatory capability is lost over SIMGEN Mk2, and the ability to run rapidly on small examples, and to scale up to very large systems, outweighs these drawbacks for most applications.

One open question concerns the possibilty of recovering most, if not all, of the self-monitoring and error checking of previous compilers by the judicious use of hints. Many programming language compilers accept advice from programmers, in the form of declarations. Qualitative representations can be viewed as declarations, providing advice to self-explanatory simulators at the level of physics and mathematics rather than code. Most qualitative reasoning systems infer as much as possible from limited information, such as inferring that a particular flow rate must always be positive. It would be interesting to see how well domain-specific and example-specific hints could replace the functionality provided by inference in earlier compilers.

At this point, we believe self-explanatory simulators are ready for applications. We believe that the major remaining hurdles are building domain theories plus software engineering. The only way to prove this is to attempt some applications. Consequently, we are building a *virtual laboratory* for engineering thermodynamics, containing the kinds of components used in building power plants, refrigerators, and

heat pumps, using a domain theory developed in collaboration with an expert in thermodynamics. We are also building a shell to support the construction of training simulators, such as a self-explanatory simulator for a shipboard propulsion plant, to finally fulfill one of the early goals of qualitative physics [19].

6.Acknowledgements

This research was supported by grants from NASA Langley Research Center and from the Office of Naval Research. We thank Franz Amador for supplying us with a sample PIKA domain theory.

7.Bibliography

1 Forbus, K. and Falkenhainer, B. Self-explanatory simulations: An integration of qualitative and quantitative knowledge, *Proceedings of AAAI-90.*
2 Forbus, K. and Falkenhainer, B. Self-Explanatory Simulations: Scaling up to large models, *Proceedings of AAAI-92.*
3 Iwasaki, Y. & Low, C. Model generation and simulation of device behavior with continuous and discrete changes. *Intelligent Systems Engineering*, **1**(2), 1993.
4 Amador, F., Finkelstein, A. and Weld, D. Real-time self-explanatory simulation. *Proceedings of AAAI-93.*
5 Forbus, K. Towards Tutor Compilers: Self-explanatory simulations as an enabling technology, *Proceedings of the Third International Conference on the Learning Sciences*, August, 1991.
6 Neville, D., Notkin, D., Salesin, D., Salisbury, M., Sherman, J., Sun, Y., Weld, D. and Winkenbach, G. Electronic `How Things Work' Articles: A Preliminary Report. *IEEE Transactions on Knowledge and Data Engineering*, August 1993.
7 Gautier, P. and Gruber, T. Generating explanations of device behavior using compositional modeling and causal ordering. *Proceedings of AAAI-93.*
8 Forbus, K. Self-Explanatory Simulators: Making computers partners in the modeling process. In Carrete, N. P. & Singh, M.G. (Eds.), *Qualitative Reasoning and Decision Technologies,* CIMNE, Barcelona, Spain, 1993.
9 Sgouros, N. Integrating qualitative and numerical models in binary distillation column design, *Proceedings of the 1992 AAAI Fall Symposium on Design of Physical Systems*, October, 1992.
10 Forbus, K. Qualitative Process theory. *Artificial Intelligence*, **24**, 1984
11 Forbus, K. and de Kleer, J. *Building Problem Solvers*, MIT Press, 1993.
12 McAllester, D. An outlook on truth maintenance. MIT AI Lab memo AIM-551, 1980.
13 DeCoste, D. and Collins, J. CATMS: An ATMS which avoids label explosions. *Proceedings of AAAI91.*
14 Forbus, K. Pushing the edge of the (QP) envelope. In *Recent Progress in Qualitative Physics*, Faltings, B. and Struss, P. (Eds.), MIT Press, 1992.
15 Woods, E. The Hybrid Phenomena theory. In *Proceedings of IJCAI-91*, Sydney, Austrailia.
16 Roberts, B. and Forbus, K. The STEAMER mathematical simulation. BBN Technical Report No. 4625, 1981.
17 Forbus, K. and Gentner, D. Causal reasoning about quantities. *Proceedings of the Eighth annual conference of the Cgnitive Science Society*, Amherst, Mass., August, 1986
18 Skorstad, G. Finding stable causal interpretations of equations. In Faltings, B. and Struss, P. (Eds.), *Recent advances in qualitative physics*, MIT Press, 1992.
19 Hollan, J., Hutchins, E., & Weitzman, L. STEAMER: An interactive inspectable simulation-based training system. *AI Magazine*, **5**(2), 15-27.