Jitter in Self-Explanatory Simulation

Brian A. Kyckelhahn **Kenneth D. Forbus**
Qualitative Reasoning Group
Department of Computer Science
Northwestern University
1890 Maple Avenue
Evanston, IL 60201
{kyckelhahn, forbus}@northwestern.edu
http://www.qrg.northwestern.edu/

Abstract

Self-explanatory simulators combine qualitative and quantitative models to produce results that have both numerical behaviors and explicit qualitative causal structure. A problem that can arise in such simulations is *jitter*, a back and forth changing of a comparison that at first glance appears to be an unnatural artifact of the simulation. Jitter can be a serious problem because it slows simulators down and can even cause crashes due to memory exhaustion. We discuss a technique that uses a data structure to dynamically detect and eliminate jitter.

1. Introduction

Self-explanatory simulators [Forbus & Falkenhainer, 1990; Amador *etal* 1993; Iwasaki & Low, 1993; Erignac, 2000] combine the explanatory power of qualitative representations with the power of numerical simulation. This makes them especially useful in education, where the causal explanations they provide can facilitate a student understanding how the principles of a domain lead to the behavior that they are observing [Forbus, 1996, 1997]. In making self-explanatory simulators for middle-school curricula, we discovered an interesting problem that can arise with such simulators that we call *jitter*. Jitter manifests itself as a rapid switching back and forth between partial qualitative states. Jitter drastically increases the size of the qualitative behavior description, in the worst case leading to a state transition every two simulator steps. This is problematic for two reasons. First, the conciseness of qualitative behavior summaries is lost. Second, the extra memory load imposed by constructing such large qualitative descriptions can, and has, caused simulators to crash in classroom settings.

This paper describes the jitter problem and a solution we have developed for it. We begin by reviewing the relevant ideas of self-explanatory simulators. Section 3 examines the causes and meaning of jitter. Section 4 discusses solutions to problems similar to jitter. Section 5 describes an approach for dynamically removing jitter during simulation. Section 6 reflects on the issues involved.

2. Review of Self-Explanatory Simulation

Self-explanatory simulator compilers are members of a class of simulators and analysis tools that combine qualitative and quantitative information (cf. Kay, 1998 or Biswas *et al* 1997). Self-explanatory simulator compilers exploit qualitative reasoning to automatically build simulations. That is, the conceptual analysis provided by qualitative reasoning identifies the relevant phenomena for the system being simulated. For each entity and process, mathematical models are drawn from the domain theory to describe it quantitatively. The conditions under which the qualitative models are applicable are translated into conditions that govern which aspects of the potentially possible mathematical models are used at any particular time. A major tradeoff in creating self-explanatory simulators is when reasoning should occur. The compilation strategy [Forbus & Falkenhainer, 1994] does as much reasoning as possible when the simulator is being created, to enable the runtime systems to be as compact and efficient as possible. The interpreter strategy [Amador *etal* 1993; Iwasaki & Low, 1993; Erignac, 2000] dynamically formulates new models during simulation, to minimize the time from the creation of the model to running a simulation. We believe that the jitter problem can occur with either strategy, and indeed can potentially occur with any mixed qualitative/quantitative simulation strategy, although we focus in the rest of this paper on compiled self-explanatory simulators for concreteness.

SIMGEN Mk3 is a self-explanatory simulator compiler that can produce simulators containing thousands of parameters in polynomial time [Forbus & Falkenhainer, 1994]. The run-time system it produces is based on a static analysis of the scenario model, which has been instantiated from a domain theory containing both qualitative and quantitative model fragments. The modeling language is essentially qualitative process theory [Forbus, 1984] with quantitative extensions. Potential limit hypotheses are identified during the qualitative analysis, without qualitative simulation, by identifying the quantity conditions of instantiated model fragments. The qualitative description of runtime behavior is generated from a set of concise histories for Boolean parameters that represent the status of the model fragments in the scenario model. At any specific time during the simulation the set of active model fragments can thus be identified. The simulator includes a *structured explanation system* [Forbus, 1997] generated from the qualitative analysis of the scenario model that encodes the dependencies between causal relationships and model

fragments. Thus at any simulated time the runtime system can ascertain the exact causal structure that held during that time.

To ensure that this qualitative description is accurate, the simulator run-time checks for transitions (using the limit hypothesis information) at every simulation clock tick. The odds of exactly hitting a transition to equality are nearly zero, of course. Consequently, when the runtime system detects that it has stepped over a transition point, it does a binary search in simulated time to find the exact time of the transition, and makes that the next time-step. This guarantees that the qualitative explanation is complete, without any gaps.

The only costs of executing a self-explanatory simulator compared to a traditional numerical simulator are (a) the cost of transition-testing and the search process to find the exact transition time and (b) the concise history it builds up to enable the complete reconstruction of the causal account for any simulated time. The transition-testing and searching are comparable to what is required for any simulator where the equations governing the system change over time. The concise history requires allocating a new record for each Boolean that changes state. In most simulations this is not an issue, since the number of state changes is proportional to the qualitative complexity of the behavior, independent of the particular time-step chosen for the simulation. Jitter, as the next section indicates, changes things.

3. Jitter: Its cause and meaning

Jitter occurs when a quantity rapidly changes back and forth at a limit point. Consider a quantity Q with a limit point L in its quantity space. Suppose further that there are two processes P1 and P2 directly influencing Q. Let us say that P1 is always active, and providing a negative influence on Q. P2, on the other hand, is active exactly when Q < L, and supplies a positive influence on Q. Consider the case where P1's effect on Q is always less than P2's effect, and Q starts below L. Under the dominating influence of P2, Q will rise until it reaches L, at which point Q will no longer be influenced by P2. Q will then begin to drop. But once it drops, P2 is again active, driving Q back to L. And the cycle continues, with a new transition every few simulation time-steps. This is an abstract description of jitter.

We encountered jitter periodically in building self-explanatory simulators, but generally only when particularly extreme ranges of parameters were used in a small handful of models. To our knowledge, it never caused problems in our fielded simulators[1] until we attempted to field an ecosystem simulator for our Mars Survival Station middle-school curriculum [Forbus *et al* 2004]. The idea of the Mars Survival Station is that students have to create an ecosystem that can sustain a crew of astronauts for two years, in case they are stranded. They could choose from a variety of plants and animals, which were modeled as populations with particular predation relationships between them, caloric values of prey species, etc. in a reasonably general manner.[2] This simulator exhibited jitter with a vengeance. The Java-based runtime would routinely crash by running out of memory, providing an intensely frustrating experience for students.

The ecosystem simulation provides an excellent concrete example of the abstract pattern described above. In this simulation, all the plants and animals live in a dome. Consider the case where the only animals in the dome are chickens, the only plants are wheat, and the chickens eat the wheat. Say that, initially, the number of chickens is small enough so that they don't consume many wheat plants, and the wheat plants soon reproduce enough to occupy the entire floor space of the dome. At this point, the wheat cannot continue reproducing at the same rate, because they have no place to grow. Those wheat plants that die do so because they are eaten by chickens. As is often the case, the domain modeler did not know the rate at which the wheat would, in reality, reproduce under these conditions. Instead, the equation for the reproduction rate of plants specifies that when there is no more space available in the dome for them to reproduce into, they stop reproducing. That is, there is a quantity condition on reproduction that the amount of available area in the dome is greater than zero. Therefore, the number of wheat plants will decline during the next time step of the simulation because they are not reproducing. Moreover, the negative influence of predation remains in effect. This frees up floor space in the dome. The wheat's reproduction process then becomes active again, and soon the floor space will be entirely occupied by the wheat. Figure 1 shows the SIMGEN plot of the number of wheat in a simulation exhibiting this behavior, and Figure 2 shows a close-up view of a small portion of the jittering region.

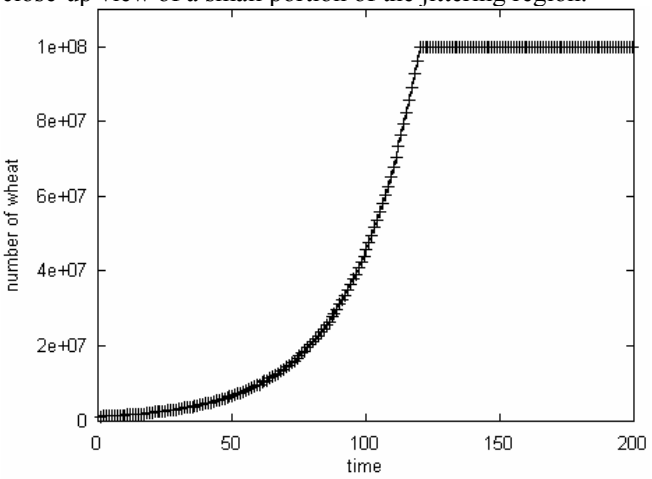

Figure 1: Number of wheat in a jittering simulation.

[1] We give them away on our web site, so we have no means of tracking what users do with them, except for volunteered reports.

[2] Since students had the option of including large predators such as tigers in their ecosystem, we modeled the caloric value of humans. This had an unfortunate interaction with the modeling assumption that humans are omnivores, which required some special-purpose modeling to work around.

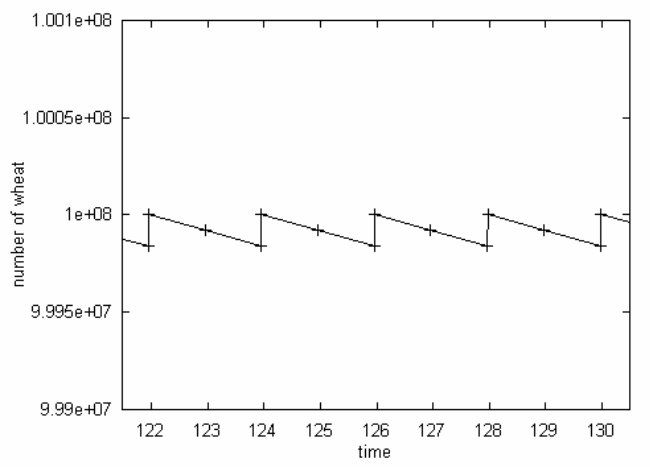

Figure 2: Close-up view of a portion of Figure 1 illustrating the jittering of wheat.

Each simulator state in Figures 1 and 2 is marked with a '+'. The states between local minima and maxima exist because, although the dominant process has become active at each of them, the derivative of the number of wheat is calculated using the value of reproduction rate from the previous time step due to the causal structure of the domain model. A simplified diagram of the causal structure in the ecosystem domain model pertinent to the jittering wheat and chicken system is shown in Figure 3. Dashed arrows point from processes to those quantities that appear in the quantity conditions of those processes, while solid arrows indicate influences between two quantities. The influenced quantity is at the head of the arrow.

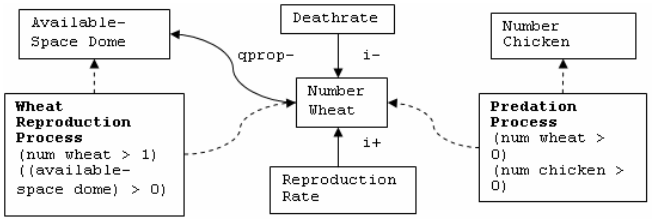

Figure 3: Causal diagram of jittering ecosystem simulation.

Is jitter an oscillation? Yes and no. There is a sense in which the description above is perfectly reasonable as an explanation of what is happening in the system. However, that kind of explanation seems to belong to a lower level of abstraction that is normally implicit in the qualitative description of behavior since it relies on quite fine-grained distinctions in time.

Comparing jitter to normal oscillation provides some interesting insights. Consider a normal second-order system, such as a spring-block oscillator. This is easily distinguishable from jitter because of the pattern of excursions on both sides of the limit points of the system. A closer comparison is a system such as a neon bulb oscillator, made up of a battery, capacitor, resistor, and neon bulb. The neon bulb is connected across the terminals of a capacitor, which in turn is hooked up in series to a battery via a resistor. Neon bulbs have a voltage at which they conduct, V_f, and a lower voltage (the "quenching voltage") at which they cease conducting, V_q. Upon connecting the battery to the circuit, the capacitor conducts and begins collecting charge. The bulb does not start conducting until the voltage across it has reached V_f. Upon reaching that voltage, the bulb acts as a resistor until the capacitor is discharged to the point where the voltage across it is V_q. The bulb stops conducting, and the cycle begins again as the capacitor builds up charge. The waveform of voltage across the capacitor, V_{CAP}, of such a system are shown in Figure 4. Sawtooth oscillators such as this more closely resemble jitter because of their abrupt changes. However, unlike jitter, an oscillator will take a reasonable number of time-steps for each transition.

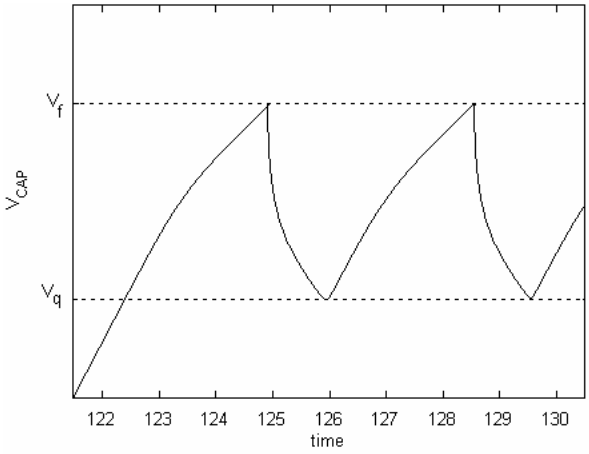

Figure 4: Neon bulb oscillator waveform.

Intuitively, the distinction between jitter and true oscillation comes down to whether or not the changes, if considered as real, constitute activity at a time-scale below the focus of attention for the simulation. A strong case for jitter not being real can be made on the basis of violation of continuity implied, but unfortunately, such violations are standard fare for systems with discontinuous effects.

Given that jitter is a problem, how can it reliably be detected? To be sure, there are particular categories of causal structures that result in jitter. However, we believe that predicting when during a simulation jitter will occur is impossible.

To see this, consider, for example, an ecosystem simulation in which there are relatively few wheat plants initially, and the wheat is soon eaten to extinction by the chickens. At the start of this simulation, the state is qualitatively the same as the example whose results are shown in Figure 1. Numerical information about the rates of reproduction and death for wheat and the number of

chickens and wheat plants are needed to determine whether the number of wheat plants will jitter. Even with this information, scenarios can be imagined that allow jitter to be predicted only for a very small length of time in the future. Say, for example, that we added carnivores, such as wolves, to the simulation where the initial number of wheat was small. The initial rate of wheat reproduction might still be small compared to its death rate, but the wolves could be great enough in number to eat the chickens to a large degree. Consequently, the wheat could grow to occupy the entire floor space of the dome and then begin to jitter. In other words, the initial states of two simulations may be qualitatively identical – the same processes and influences are active – and yet one will produce jitter and the other won't.

In some cases, it may be possible to rewrite small portions of either or both of the qualitative and quantitative models and at least lessen the frequency with which jitter occurs, while still remaining faithful to the system's true behavior. However, while in some simulations jitter may not represent a desired behavior of the system, it is a product of the domain theory. Therefore, any technique for lessening the drawbacks of jitter should be careful to retain this fidelity.

4. Related solutions

While we know of no previous attempts to analyze or solve the jitter problem, solutions to related problems exist.

Some researchers of hybrid systems have described a problem closely related to related to jitter called the Zeno phenomenon, in which an infinite number of transitions between qualitative states can occur in a finite amount of time [Johansson, 1999]. Johansson *etal* proposed a regularization technique, which involves adding a small perturbation to the hybrid automata representing the system, that makes the system non-Zeno. This perturbation can take spatial, temporal, or dynamical forms. Qualitative simulators such as SIMGEN are alleviated from having to deal with the problem of Zeno phenomena, which may be present in the domain models they simulate, because they impose a finite time step on the calculation of the values of model variables. Only when a limit point is skipped in the evolution of the system does the simulator roll back the clock and create a new simulator state at a time earlier than that dictated by the step size.

Another related problem is that of *chatter*, which is intractable branching of qualitative states. Chatter arises because a variable is constrained only by continuity of its derivative [Fouché and Kuipers, 1991]; when the qualitative value is unknown, chatter occurs. Solutions to the chatter problem have included chatter box abstraction [Clancy and Kuipers, 1993] and dynamic chatter abstraction [Clancy and Kuipers, 1997], among others. Both chatter box abstraction and dynamic chatter abstraction involved the creation of a new qualitative state, which would be an attractive solution to the jitter problem. However, jitter seems to be harder to detect, as discussed in Section 5, as the problem with jitter is not a lack of constraints on variables; both the qualitative and quantitative states are known when jitter occurs. Therefore, determining when the system would enter and exit such a state is problematic.

The next section describes a solution to the problem of jitter.

5. Dynamic correction of jitter

The most direct solution to jitter is to detect it during a simulation and filter it out. The algorithm we present here monitors all quantities involved in limit hypotheses, i.e. those quantities that may exhibit jitter. It does this by instantiating a *jitter filter* data structure, call it J_Q, for each quantity Q involved in a comparison in the simulation. For example, in the ecosystem simulation there will be a J_Q for tracking the number of chickens, since it is compared to zero. Even if a parameter participates in multiple comparisons, only one J_Q will be instantiated for it. After observing the quantity for the time necessary to detect jitter, it updates the elements of the concise history for the time period it has observed. By postponing the creation of the concise history until the jitter detector has finished analyzing a given time period, we preclude ourselves from having to modify the same period later because of jitter.

As the simulation runs, the transition finder detects new limit hypotheses. These limit hypotheses are passed as input to the appropriate J_Q, which stores the new limit hypothesis it has just received from the transition finder, the current limit hypothesis, and the limit hypothesis it previously held. The J_Q's determine that a comparison is jittering after recording a pattern of a comparison changing values between two limit hypotheses for several times. The new, current, and previous limit hypotheses of a J_Q themselves form, conceptually, a queue; a new limit hypothesis is pushed onto the new slot of this queue only if it is different from the current one. Therefore, the current limit hypothesis will always be different from the new and previous ones.

Each J_Q also records the amount of simulated time it has held a particular limit hypothesis. If the J_Q has the same limit hypothesis for more than some amount of time, the jitter detection algorithm declares that the comparison is not jittering. This amount of time is a parmeter, which we shall call $t_{LH\text{-}max}$, that can be set. In practice, we think that tLH-max should be some small multiple of the time step. When the dominant influence on a jittering parameter is relatively large, the parameter will quickly re-establish the quantity condition that de-activated the dominant process. The dominant process will then, in these cases, be inactive for only one or two time steps before the parameter returns. The number of times a comparison must switch from one limit hypothesis to another before being declared jittering is dictated by another system parameter, which we shall refer to as N_{Jmin}. A sequence of Limit Hypothesis A, Limit Hypothesis B, Limit Hypothesis A can be seen as a jitter wave, and N_{Jmin} is the minimum number of transitions, or half waves.

The jitter detecting algorithm is run on each J_Q at every iteration of the simulator, including those inserted by the transition finder. The psuedocode for the algorithm is given in Listing 1. The algorithm works as follows. First, it checks that the J_Q has not gone from equality with one limit point to equality with another, regardless of whether the parameter spent time in the space inbetween. It is possible for the parameter to skip the space inbetween two limit points, because, from the perspective of any individual comparison, no transition has been skipped. For example, say that we have a J_Q for the voltage across the capacitor in a simulator for the neon bulb oscillator. The limit hypotheses found by the transition finder as the voltage across the bulb reaches V_f, and then drops to V_q, will be: ($voltage_{CAP}$, <, =, V_q), ($voltage_{CAP}$, =, <, V_q), ($voltage_{CAP}$, >, =, V_f). The first of these indicates, for example, that the voltage across the capacitor was less than but is now equal to the quenching voltage. Note that the second and third of these could occur at the same instant, or, if the rate of voltage drop were small enough, at different time steps. If they occur at the same instant, the function calling `Jitter-Detector` will set the new limit hypothesis of the J_Q to the one in which a transition to equality occurs, so that equality-to-equality transitions can be detected. Regardless, the jitter detecting algorithm sees that $voltage_{CAP}$ has gone from equality with V_q to equality with V_f, and that these limit points are not the same. At that point it determines that the parameter is not jittering. The $t_{correct}$ field of the JQ is set here so that a record of when the JQ started jittering is kept; this variable is set in anticipation of jittering. It would be more difficult to set it after it were known that the parameter were jittering.

If the test for an equality-to-equality transition fails, the jitter detector algorithm then tests that the new limit hypothesis is equal to the previous limit hypothesis, i.e. that before the current one, and different from the current one. These three limit hypotheses constitute one wavelength of the jitter cycle. The time-at-switch field of the J_Q indicates when the J_Q switched to its current limit hypothesis, and is used to verify that the J_Q has not had that same limit hypothesis for greater than the maximum amount of time, $t_{LH\text{-}max}$. The J_Q records, in N_J, the number of transitions of limit hypotheses consistent with a jittering pattern it has seen. Once N_{Jmin} have been seen, the comparison is said to be jittering. Note that N_J is compared with $N_{Jmin} - 1$ in this first "else if" block. This is because N_{Jmin} is updated after, rather than before, the conditions are checked. Finally, the return variable LH is set to the new limit hypothesis of the J_Q to indicate to the calling function what the jittering limit hypothesis is.

The second "else if" block corresponds to the case where the pattern of limit hypothesis transitions seen so far is consistent with a jittering comparison, but fewer than N_{Jmin} transitions have been seen. This includes the case where the first limit hypothesis "wave" has just been seen. In this case, it may be true that there is no previous limit hypothesis in the J_Q, in which case the N_J of the J_Q will be 0. For this condition, the function returning-LH checks that the current and new limit hypotheses of the J_Q are symmetric (as are ($voltage_{CAP}$, <, =, V_q) and ($voltage_{CAP}$, =, <, V_q), for example). The jitter detector is able to determine which quantity condition causes the dominant process(es) to be deactivated for any given jitter cycle; it is simply the one that is true at the center of the first wavelength. We will call this the dominant quantity condition. Upon seeing the first wave, we know that the dominant quantity condition is the one that is true in the "after" statement of the current limit hypothesis of the J_Q. In the chicken and wheat example, the dominant process, wheat reproduction, is deactivated when the available space in the dome equals 0, therefore, equality of the available space in the dome with 0 is the dominant quantity condition.

The third "else if" block corresponds to those cases where a limit hypothesis transition has not occurred, but the pattern of limit hypotheses seen so far may or may not be consistent with a jittering parameter. In these cases, no action should be taken. The final "else" block corresponds to the case where the parameter is definitely not jittering. Here, we reset the N_J, previous limit hypothesis, and $t_{correct}$ fields. Finally, a separate "if" block updates the time-at-switch, previous-LH, and current-LH fields of the J_Q if necessary.

```
Jitter-Detector(JQ, time)
LH := none

if equality-to-equality(JQ)
    previous-LH(JQ) := none
    NJ(JQ) := 0
    tcorrect(JQ) := time
else if (new-LH(JQ) ≠ current-LH(JQ) and
          new-LH(JQ) = previous-LH(JQ) and
          time - time-at-switch(JQ) ≤ tLH-max and
          NJ(JQ) ≥ NJmin - 1)
    LH := new-LH(JQ)
else if (new-LH(JQ) ≠ current-LH(JQ) and
           time - time-at-switch(JQ) ≤ tLH-max  and
           (new-LH(JQ) = previous-LH(JQ) or
                (NJ(JQ) = 0 and returning-LH(JQ)))
    if NJ(JQ) = 0
        dominant-LH(JQ) := current-LH
        NJ(JQ) := 2
    else
        NJ(JQ) := NJ(JQ) + 1
else if (new-LH(JQ) = current-LH(JQ) and
    time - time-at-switch(JQ) ≤ tLH-max)
else
    NJ(JQ) := 0
    previous-LH(JQ) := none
    tcorrect(JQ) := time

if new-LH(JQ) ≠ current-LH(JQ)
     time-at-switch(JQ) := time
     previous-LH(JQ) := current-LH(JQ)
     current-LH(JQ) := new-LH(JQ)

return(LH)
```

Listing 1: Jitter-Detector psuedocode.

Each boolean episode is predicated on zero or more quantity conditions. In our implementation, there is a queue of values for each of these quantity conditions. At each

iteration of the simulation, the values of the quantity conditions upon which boolean episodes are predicated are calculated and put onto queues, independently of the operation of the jitter-detector. Each value in a queue represents the value of the quantity condition at a particular time during the simulation. These queues allow the jitter detector to observe a jittering comparison for the longest amount of time it might take to detect a jittering comparison, ($t_{LH\text{-}max} \cdot N_{Jmin}$), before calculating the final value of the boolean episode. When the jitter detector indicates that a comparison is jittering, the system finds all of those queues for quantity conditions that are derived from that comparison. The jitter detector has, in $t_{correct}(J_Q)$, the time of the oldest value in the queue that was calculated while the parameter jittered. Another queue contains the simulated times at which all of the values in all of the queues were calculated. Since a new value is pushed onto every queue at every step of the simulator, and only then, all queues are the same length, and every element of any queue was calculated at the same time as any other element in the same position of any other queue. Therefore, the system has all of the information it needs to re-calculate the appropriate values in the queues of quantity conditions whose comparisons are jittering. The re-calculation is simple; those quantity conditions equal to the dominant quantity condition are given the value true, and the others are given the value false.

After the amount of time governed by $t_{LH\text{-}max}$ and N_{Jmin} has passed, the front of each queue of values is removed and the value of the episode is calculated. The most recent process episode of the history is extended if the two values are the same, or a new episode is created if they are different.

Consider, for example, the wheat and chicken ecosystem simulation. The wheat reproduction process is represented by a boolean episode, and associated with that episode is a queue representing the value of the quantity condition "available space in the dome is greater than 0", as well as a queue for the quantity condition "number of wheat is greater than 1". Say that, after new values are added onto the queue, the jitter detector indicates that the comparison between available space in the dome and 0 is jittering. Figure 5 illustrates the steps of the calculation of the value of the boolean episode for this example. In this case, N_{Jmin} of the J_Q is 5, in other words, once 5 appropriately-formed half wavelengths have been seen, the jitter detector declares the comparison to be jittering. Remember that, in the case that the wheat has grown to the full capacity of the dome and the chicken are eating it, the dominant condition is "available space in the dome is equal to 0". Therefore, the new value for this "available space" quantity condition is "false". The jitter filter replaces the existing values in the queue with this new value for the time over which the jittering pattern was observed, which happens to include the entire queue. Finally, the value of the episode at time 10.0 is calculated using the first items in the two queues and stored in the history.

Note that it may be more intuitive to the user to give the boolean episode a special value, rather than true or false, indicating that an equilibrium had been reached. This change could be quite easily implemented.

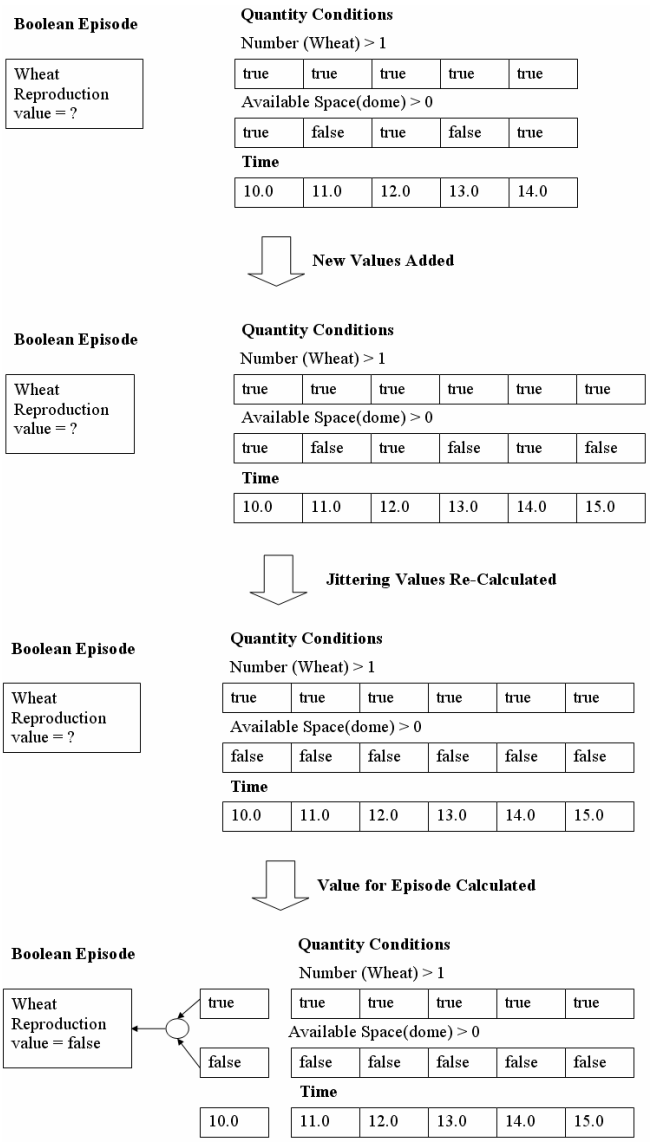

Figure 5: Example sequence of steps taken in calculating values for the boolean episode of the wheat reproduction process.

Empirically, we have found that this jitter filter does indeed produce a more accurate record of events during a simulation. In simulators where jitter occurs, jitter filtering improves performance as well as explanation clarity, since many fewer Boolean episodes are created. For example, in the chicken and wheat simulation, the jitter filter successfully replaces all of the boolean episodes for the wheat reproduction process in the plateau region of Figure 1, of which there were originally 80, with two episodes. The latter of these two is simply an "edge effect" and exists

only for the final time step. In simulators where jitter does not occur, the jitter filter incurs a slight runtime penalty and an extra storage cost, dependent on the size and number of the queues. These, in turn, depend on the $t_{LH\text{-}max}$ and N_{Jmin} parameters, and on the number of times new states are introduced before regularly scheduled time steps by the transition finder.

Another drawback of our solution is the fact that it relies on two parameters to identify jitter. Recall that one of these, $t_{LH\text{-}max}$, indicates the maximum amount of time a jittering comparison can stay at a given limit hypothesis. We have seen that $t_{LH\text{-}max}$ can vary across simulations. If these parameters are set incorrectly, our system will incorrectly identify a jittering system as not jittering, or a system that is not jittering as jittering.

Jitter is such a slippery problem that it can at times be difficult to judge when a parameter is jittering and when it is not. If the influence exerted by the subordinate process increases, it may cause the jittering parameter to take longer to return to the limit point around which it jitters. For example, we ran a simulation similar to the one having only chicken and wheat, in which the number of wheat jitters after fully occupying the floor space of the dome. We increased the maximum allowable density of chickens per square meter so that the chickens are able to eat the wheat until there are no more of them, at which point the chickens rapidly die off. A close-up view showing the region where the number of wheat was dislodged from the jittering state is shown in Figure 6. Notice that the amount of time during which the number of wheat is less than the maximum allowed by the available space in the dome increases with time, thereby making it difficult to say at what point the jitter stops. Because the time a jittering comparison stays at any limit hypothesis can vary, $t_{LH\text{-}max}$ is valuable as a maximum threshold beyond which the person running the simulation is willing to allow the simulation to run unaltered by the jitter filter.

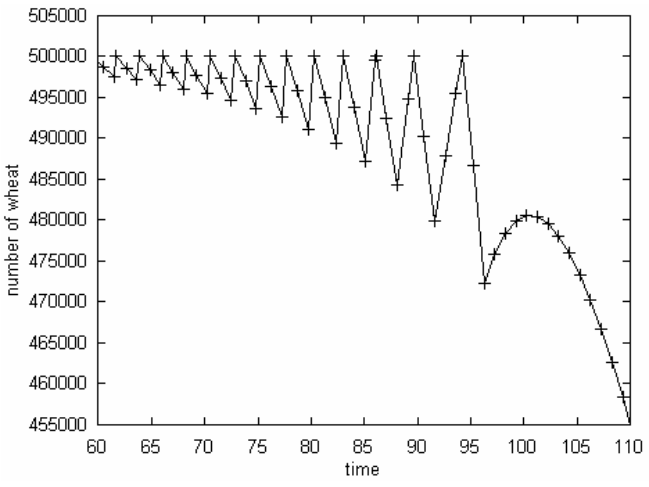

Figure 6: Number of wheat dislodged from the jittering state.

6. Discussion

Ultimately, jitter seems to be an undesirable consequence of making simulation models that are more explicit that usual. The ability to make intricate distinctions unfortunately sometimes leads to them being made inappropriately. In some cases, the jitter filtering algorithm we developed can eliminate it dynamically. This work suggests three open questions:

1. Is there some general, well-grounded guidance for how to set the jitter filter parameters?
2. Is there a formal method for always identifying via static analysis or by a parameter-free method at runtime, when jitter will occur?
3. While jitter arises in the context of self-explanatory simulators, we suspect that the problem is more general than that, and can affect any hybrid qualitative/quantitative simulation scheme that has conditionally applicable models. This is of course an empirical question.

Acknowledgments

This research was supported by the National Science Foundation under the REPP program.

References

Amador, F., Finkelstein, A. and Weld, D. Real-time self-explanatory simulation. *Proceedings of AAAI-93*.

Biswas, G., Kapadia, R., and Xudong W. Combined Qualitative-Quantitative Steady State Diagnosis of Continuous-valued Systems. *IEEE Transactions on Systems, Man and Cybernetics*, vol. 27, PART A, no. 2, pp. 167-185, 1997.

Clancy, D. and B. Kuipers. Behavior abstraction for tractable simulation. *Proceedings from the Seventh International Workshop on Qualitative Reasoning,* May 1993.

Clancy, D. and B. Kuipers. Dynamic chatter abstraction: a scalable technique for avoiding irrelevant distinctions during qualitative simulation. *Proceedings from the 11th International Workshop on Qualitative Reasoning about Physical Systems (QR-97)*, June 1997.

Erignac, C. 2000. Interactive semi-qualitative simulation. *Proceedings of the 14th international workshop on qualitative reasoning (QR2000),* Morelia, Mexico. June, 2000.

Forbus, K. D. (1984). Qualitative Process Theory. *Journal of Artificial Intelligence, 24,* 85-168.

Forbus, K. Self-Explanatory Simulators for Middle-School Science Education: A Progress Report. *Proceedings of QR96*.

Forbus, K. Using qualitative physics to create articulate educational software. IEEE Expert, 12(3), May/June 1997.

Forbus, K., Carney, K., Sherin, B. and Ureel, L. To appear. VModel: A visual qualitative modeling environment for middle-school students. To appear in *Proceedings of the 16th Innovative Applications of Artificial Intelligence Conference*, San Jose, July 2004.

Forbus, K. and Falkenhainer, B. Self-explanatory simulations: An integration of qualitative and quantitative knowledge, *Proceedings of AAAI-90*.

Forbus, K. D., & Falkenhainer, B. (1994). Polynomial-time Compilation of Self-Explanatory Simulators. *Proceedings of the Eighth International Workshop on Qualitative Reasoning,* Nara, Japan.

Fouché, P. and Kuipers, B. Towards a Unified Framework for Qualitative Simulation. In *Proceedings of the Fifth International Workshop on Qualitative Reasoning about Physical Systems*, 295-301, 1991.

Iwasaki, Y. & Low, C. Model generation and simulation of device behavior with continuous and discrete changes. *Intelligent Systems Engineering*, **1**(2), 1993.

Johansson, K., Egerstedt, M., Lygeros, J., and Sastry, S. On the regularization of Zeno hybrid automata. *Systems and Control Letters*, 38:141-150, 1999.

Kay, H. SQSIM: a simulator for imprecise ODE models. *Computers and Chemical Engineering*, 23(1):27-46, 1998.