EXTENDING TEMPORAL LOGIC TO SUPPORT HIGH-LEVEL SIMULATIONS

Alexander Tuzhilin

Information Systems Department
Stern School of Business
New York University

44 West 4th Street, Room 9-78
New York, NY 10012
atuzhilin@stern.nyu.edu
212-998-0832

Working Paper Series
STERN IS-93-19

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

Abstract

A high-level simulation language based on temporal logic is described. The language combines a large set of temporal tenses and a rich class of high-level modeling primitives. Also an implementation of the language interpreter is presented. Finally, a real-world case study is described that shows how a programmer can develop structured, reliable, and well-maintainable simulation programs using the language.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

1 Introduction

There has been a substantial amount of research done in the field of knowledge-based simulations since the time when the first systems ROSS [KFM80], KBS [FR82], and T-Prolog [FS82] were introduced. The recent developments in the field are presented in the books [FM91, FG90, EÖZ89, WLN89] and in the special issues on knowledge-based simulations of SCS Transactions [kbs90] and ACM Transactions on Modeling and Computer Simulation [tom92]. Many of the knowledge-based simulation systems provide support for rule-based and object-oriented paradigms and for powerful knowledge representation schemes such as frames. Examples of commercial systems of this type are SIMKIT [Int85b], Simulation Craft [SFBB86], and G2 [HSH89].

The rule-based component of these systems is typically based on a logic programming language, e.g. PROLOG, or on a production system, e.g. OPS5 [BFK86]. Therefore, rules used in the knowledge-based simulation methods described above are based on first-order logic since logic programming languages and production systems have their roots in first-order logic.

Since simulation methods deal with processes evolving in time and since first-order logic does not support time directly, knowledge-based simulation methods must provide an explicit support for time. For example, most of the methods explicitly define and manipulate the system clock and provide some form of event scheduling. This means that these systems are quite procedural because the programmer has to specify explicitly how to handle time.

In order to provide a more declarative support for time, [Tuz92] proposed to use *temporal logic* as an alternative to first-order logic in knowledge-based simulations. In particular, [Tuz92] describes a temporal logic programming language SimTL that is specifically designed for simulations. Although SimTL programmers do not have to schedule events or advance the system clock, the language is still low-level in the sense that it does not support important modeling primitives, such as events, activities, structuring constructs of aggregation and generalization [TL82], and the decomposition of activities. The lack of these constructs in SimTL forces programmers to encode them in SimTL programs, thus making the programs longer and more difficult to write and understand. This makes SimTL comparable to a 3GL language, such as C or Fortran, that lacks some of the high-level constructs present in the 4GL languages. In addition, the language contains quite a few technical symbols thus making SimTL programs difficult to understand for a non-technical user.

In this paper, we describe a high-level simulation language Templar that addresses these concerns. The language is also based on temporal logic, but it supports a much richer set of modeling primitives than SimTL does. A Templar program consists of a set of rules and a set of activity

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

specifications. Templar explicitly supports rules, events and activities, time, hierarchical decomposition of activities, sequential and parallel activities, static and dynamic constraints, decisions, data modeling abstractions of aggregation and generalization, and user-defined modelling constructs. To illustrate the use of Templar, consider the following rule:

> If a customer comes to a branch of a bank while the branch is closed, and the branch has ATM machines then he or she should use an ATM machine.

It can be stated in Templar as

when	`arrives(customer,branch)`
while	`close(branch)`
if	`has_atm(branch)`
then-do	`use_atm(customer,branch)`

This rule is interpreted as follows. When an (instantaneous) event `arrives(customer,branch)` occurs, and if it occurs while the activity `close(branch)` is in effect (i.e. the branch was closed in the past but has not reopened yet), and if the condition `has_atm(branch)` holds then perform the activity `use_atm(customer,branch)` (that lasts over some period of time).

The idea to use a rich set of high-level modeling primitives in a simulation language is not new. Some of the existing knowledge-based simulation languages support many high-level modeling constructs. For example, both SIMKIT and Simulation Craft are based on a rich knowledge representation schemes of AI. For instance, SIMKIT is built on top of KEE [Int85a] and therefore takes full advantage of the expressive representational and reasoning tools that KEE provides. As another example, the simulation language ROBS [RS89] supports rules, objects, parallel communicating processes, and actions.

What differentiates Templar from these languages is that it *integrates* a temporal logic that supports many tenses used in a natural language[1] and a rich set of modeling primitives into one language. We believe that this integration will

- allow programmers rapidly produce concise, reliable, and well-maintainable simulation programs;
- allow other members of the development team and experienced users understand these simulation programs with a minimal effort.

To validate these points, we did a case study in which we wrote a program in Templar that

[1] Examples of these tenses are **when**, **while**, **since**, **until**, **before**, **after**, **always**, **sometimes**, etc.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

implements a portion of the Intelligent Adversary system for the Naval Training Systems Center that simulates behavior of navy pilots in combat situations. We describe the results of this case study in Section 6.

In order to make the paper self-contained, we provide some background presentation of temporal logic in the next section before describing Templar.

2 Background: Some Concepts from Temporal Logic and Temporal Logic Programming

We start this section with a review of temporal logic and then describe its multi-sorted extensions. The reader is referred to books by Kroger [Kro87] and by Manna and Pnueli [MP92] for a good introduction to the subject.

Temporal Logic. The syntax of a predicate temporal logic is obtained from the first-order logic by adding various future temporal operators such as **sometimes_in_the_future** ($\diamond$), **always_in_the_future** ($\Box$), **next** ($\circ$), **until** and their past "mirror" images **sometimes_in_the_past** ($\blacklozenge$), **always_in_the_past** ($\blacksquare$), **previous** ($\bullet$), and **since** to its syntax[2]. The meaning of future operators is defined in Fig. 1. The meaning of past "mirror" images of these operators is defined similarly to the future operators except time is referenced only in the past. Besides these eight standard operators, other temporal operators can be defined, such as **before**, **after**, **while** [Kro87], and bounded necessity, **for_time (T)** ($\Box_T$), and possibility, **within_time (T)** ($\diamond_T$), operators [Tuz92]. For example, A **for_time (T)** is true now if A is always true within the next **T** time units, and A **within_time (T)** is true now if A is true at some time within the next **T** time units. Kroger [Kro87] shows how temporal operators **before**, **after**, and **while** can be expressed in terms of the operators **until** and **since** [Kro87]. Furthermore, it easily follows from the expressive completeness of the temporal logic US [Kam68] for the discrete or continuous model of time, that the operators of bounded necessity and possibility can also be expressed in terms of the **until**, **since**, **next**, and **previous** operators.

The following example illustrates the use of temporal logic.

Example 1 The statement

> If an employee has been fired from a company (worked there in the past but not now) then he or she cannot be hired by the same company in the future

[2] Note that the operators $\diamond$ and $\Box$ can be derived from $\circ$ and **until**, and $\blacklozenge$ and $\blacksquare$ from $\bullet$ and **since** [Kro87, MP92].

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

$\diamond A$:	is true now if A is true at some time in the future
$\Box A$:	is true now if A is always true in the future
$\circ A$:	is true now if A is true at the next time moment
A **until** B:	is true now if B is true at some future time t and A is true for all the moments of time from the time interval $[now,\ t)$

Figure 1: Operators of Temporal Logic

can be expressed in temporal logic as

$$\blacklozenge EMPLOY(company, person) \wedge \neg EMPLOY(company, person) \rightarrow$$
$$\Box\neg EMPLOY(company, person)$$

or using a different syntax as

IF sometimes_in_the_past *EMPLOY(company,person)* and not *EMPLOY(company,person)*
THEN always_in_the_future not *EMPLOY(company,person)*

□

The semantics of temporal logic formulas is defined with *temporal interpretations.* A temporal interpretation for some temporal logic language defines the domain of discourse, the model of time (e.g. discrete or continuous, bounded or unbounded, linear or branching), assigns values to constants and function symbols in the language as in classical logic, and specifies a *temporal structure* [Kro87], i.e. the values of all the predicates in the language at *all* the time instances. We assume any arbitrary structure of the domain of discourse and also assume that time is discrete, linear, bounded in the past and unbounded in the future (i.e. time can be modeled with natural numbers)[3]. A temporal structure defines for each predicate P_i in the language a sequence of its instances P_{it} for *all* the moments of time $t = 0, 1, 2, \ldots$. We denote a temporal structure of a temporal logic language at time t as K_t. Then $K_t(P_i) = P_{it}$, since it defines the instance of predicate P_i at time t.

Given a temporal interpretation, we can define the truth value of a temporal logic formula at any moment of time in the standard inductive way [Kro87]. For example, we can define $K_t(A$ **until** $B)$ in terms of $K_t(A)$ and $K_t(B)$ as follows. $K_t(A$ **until** $B)$ is true if there is t' such that $t \leq t'$, $K_{t'}(B)$ is true, and for all t'', such that $t \leq t'' < t'$, $K_{t''}(A)$ is true. Other operators can be defined in a similar way (in fact, Fig. 1 contains some of the informal definitions

[3]Since we consider **next** and **previous** operators in this paper, we have no choice but assume that time is discrete. Alternatively, we could disallow **next** and **previous** operators and make time dense.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

of other temporal operators). An example of an inductive definition of a non-temporal operator would be $K_t(A \wedge B) = K_t(A) \wedge K_t(B)$.

Multi-Sorted Temporal Logics. In this paper, we extend single-sorted temporal logic to multi-sorted logic using the approach taken by the ERAE model [DHR91] that differs somewhat from the classical approaches. The reason why ERAE approach is chosen will become apparent in Section 3.7 when we define structuring mechanisms of Templar.

ERAE considers a set of *elementary* sorts – sort names and singletons – and *derived* sorts obtained as a closure of the elementary sorts under the operations of union and intersection. For example, the derived sort `person` is defined as `man` ∪ `woman`. This model differs from the classical model in that it supports derived sorts that can be considered as *types* in programming languages. Each attribute of a temporal predicate and each parameter in an activity specification[4] considered in Templar must belong to a certain sort. For example, in predicate `referees(paper,reviewer)` variable `paper` belongs to the sort `Papers` and variable `reviewer` to the sort `Reviewers`.

3 Overview of Templar

In this section, we briefly describe the language Templar by providing several examples of programs written in it. In Section 4, we formally define the syntax of the language, and in Section 5 describe an interpreter that executes Tempar programs.

Templar features will be introduced with examples based on the description of an IFIP Working Conference [Oll82, Appendix A]. Organization of a working conference involves several activities: sending a call for papers, receiving paper submissions and registering these submissions, sending papers to be refereed, receiving reports back from referees, making acceptance/rejection decisions, and so on.

A Templar program simulating such a conference consists of a set of rules and activities that will be described in turn below. We start with the most basic features of the language in Section 3.1 and introduce additional features in the subsequent sections.

3.1 Basics of Templar Rules

A Templar rule is based on the *Activity-Event-Condition-Activity (AECA)* model. AECA is an extension of the Event-Condition-Action (ECA) model of rules in active databases [dMS88, MD89,

[4] Activity specifications will be defined in Section 3.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

WF90, SJGP90].

The following is an example of a Templar rule. To make an example simple, we consider a rule of the ECA type and describe an AECA rule in Example 4.

Example 2 The user specification

> When a reviewer receives a paper to be refereed, which was sent by the conference program chairperson, he/she evaluates the paper and sends it back to the chair

is expressed with the Templar rule

when	`end.send(paper,chairperson,reviewer)`
if	`referees(paper,reviewer)`
then	**next** `located(paper,reviewer)`
then-do	`review(paper,reviewer); send(paper,reviewer,chairperson)`

□

This rule is interpreted as follows: when an *event* `end.send(paper,chairperson,reviewer)` occurs (reviewer receives a paper) and if the *condition* `referees(paper,reviewer)` is true then set the *post-condition* `located(paper,reviewer)` to be true at the next time moment and start the activities `review(paper,reviewer)` and `send(paper,reviewer,chairperson)` *sequentially* (i.e. when the first activity finishes, start the second one).

This rule illustrates three major modeling primitives in Templar: activities, events, and conditions. *Activity* is a process that occurs *over time*, e.g. a paper is being reviewed by a reviewer for some time. An *event* is a change to the system state that occurs *instantaneously*, e.g. a reviewer receives a paper at some moment in time. Prefix "end" in "end.send" in Example 2 specifies the event "activity `send(paper,chairperson,reviewer)` has finished." A *condition* is a logical formula that describes the state of the system, e.g. predicate `referees(paper,reviewer)` indicates that in the current state of the system, objects `paper` and `reviewer` are engaged in relationship `referees`.

The rule presented above consists of *clauses* **when**, **if**, **then**, and **then-do**. We distinguish between state, temporal, and action types of clauses. A *state* clause describes the state of the system (the working conference in our case). **If** and **then** clauses are examples of a state clause. A *temporal* clause specifies how different events and activities relate to each other in time. **When** and **after** are examples of a temporal clause. Finally, the action clause states imperatively what activities will have to be done. **Then-do** is an example of an action clause.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

Each clause deals with only one type of a modeling primitive. For example, **when** clause pertains to events, **if** and **then** clauses to conditions. and **then-do** clause to activities[5]. This means that in the previous rule `referees` and `located` are predicates, `review` and `send` are activities, and `end.send` is an event (the end of an activity). This relationship between types of clauses and types of modeling primitives that can appear in them forces the user to think more structurally when writing Templar programs.

3.2 Atomic and Composite Activities

Templar distinguishes between atomic and composite activities. A *composite* activity consists of sub-activities. For instance, the activity `review(paper,reviewer)` from Example 2 consists of reading the paper and then evaluating it. This statement can be expressed in Templar with an *activity specification* as illustrated in the following example.

Example 3 A rule for the activity `review` can be stated in Templar as

```
activity review(paper: Papers, reviewer: Reviewers)
    read(paper,reviewer)
    evaluate(paper,reviewer)
end_activity
```

where `Papers` and `Reviewers` are elementary sorts as defined in Section 2 (and in [DHR91]). This means that activities have *types* as temporal predicates do.

□

An activity specification can be compared to a procedure in conventional programming languages or to the body of a method in object-oriented programming, except that it is defined in terms of temporally oriented modeling primitives (activities). We will describe how an activity is "executed" in Section 5.

An activity is *atomic* if it does not consist of several subactivities. It is defined with a *temporal predicate* describing how one of the temporal predicates changes over time[6]. For example, consider the activity specification

```
activity read(paper: Papers, reviewer: Reviewers)
    T = reading_time(paper,reviewer)
```

[5]When we define the syntax of Templar formally and introduce all the clauses in Section 4, we will explain in Figure 4 how clauses correspond to modeling primitives.

[6]Temporal predicates will be described in full in Section 3.5.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

```
    reading(paper,reviewer) for_time T
end_activity
```

where reading_time(paper,reviewer) is a function that specifies how much time it takes a reviewer to read a paper, and reading is a temporal predicate. Then "**reading(paper,reviewer) for_time T**" is an example of an atomic activity. It states that the predicate **reading(paper,reviewer)** will be true for the next T time units.

Templar allows the mixture of composite and atomic activities inside an activity specification. For example, the composite activity review(paper,reviewer) can be rewritten as

```
activity review(paper: Papers, reviewer: Reviewers)
    T = reading_time(paper,reviewer)
    reading(paper,reviewer) for_time T
    evaluate(paper,reviewer)
end_activity
```

Since subactivities in an activity specification can also be composite activities, Templar supports the process of hierarchical decomposition of a complex activity into progressively more and more simple subactivities.

Templar also allows multiple subactivities in the **then-do** clause of a rule. For instance, the **then-do** clause in Example 2 has two subactivities **review(paper,reviewer)** and **send(paper,reviewer,chairperson)**. Alternatively, these two subactivities could be combined into one composite activity, and the **then-do** clause would refer only to this single activity.

The combination of activity specifications and rules makes Templar a powerful simulation language. If Templar programs had only rules then they could contain hundreds of rules, and it would be difficult for the programmer to understand clearly how the rules interact. On the other hand, if Templar programs consisted only of activities, then it could be difficult to describe the control logic with only the *if-then-else* statements for certain applications. With Templar programs, the user has the flexibility of combining rules and activities in such a way that there are much fewer rules than for the strictly rule-based methods, and activity specifications tend to be small, simple and easy to understand, as the case study in Section 6 will demonstrate it.

3.3 Activity-Event-Condition-Activity Rules

The rule from Example 2 has the Event-Condition-Activity (ECA) structure. This structure is extended to the Activity-Event-Condition-Activity (AECA) structure in Templar by supporting

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

while, **before**, and **after** temporal clauses as the following example shows.

Example 4 Assume the organizers of the conference have a rule:

> While the paper is being reviewed, any request to withdraw the paper will be granted by the program chairperson.

This requirement can be expressed in Templar as

```
while    do_reviewing(chairperson,paper)
when     withdrawal_request(paper)
if       submission(paper,author,status)
then-do  withdraw(paper,author)
```

where `do_reviewing(chairperson,paper)` is the activity of sending a paper by the program chairperson for reviewing, `submission(paper,author,status)` is a condition stating that an author submitted a paper to the conference. `withdrawal_request(paper)` is an event indicating that the request to withdraw the paper was received, and `withdraw(paper,author)` is an activity of withdrawing a paper from the conference.

□

This rule says that while a certain activity lasts, and when an event occurs, and if a condition holds, then do a new activity. In this rule, unlike the rule from Example 2, the activities in the **then-do** clause depend not only on some conditions and events but also on some other *activities*. Therefore, we call this type of a rule the Activity-Event-Condition-Activity (AECA) rule because it generalizes the Event-Condition-Activity (ECA) rule as defined in [dMS88, MD89, WF90, SJGP90] by

- allowing activities in the antecedent part of the rule;
- supporting not only **when**, **if**, and **then** clauses of the ECA model but several additional clauses, such as **while**, **before**, **after**, and various other user-defined clauses;
- providing a comprehensive support for time based on temporal logic.

In addition, we assume that Templar rules are *safe* [Ull88] in the sense that all the variables appearing in clauses **then**, **then-do**, **then-cancel**, and **then-dont-do** must appear positively in some other clause in that rule. The rules in all the examples considered so far are safe. An example of a non-safe rule would be **if** `A(x)` **then** `B(x,y)`.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

3.4 Procedures in Templar

In Section 3.3, we considered a rule of an AECA type and in Section 3.1 its restricted ECA version. In general, only the action part of the rule (**then-do** clause) is mandatory in a rule, and all other clauses are optional. For example, the "topmost" activity specifying that a conference has to be organized may not require any preconditions and can be expressed in Templar as

then-do `organize_conference`

or, using **then-do** operator implicitly, as

`organize_conference`

If only the action part of a rule is specified then it is reduced to a procedure. Therefore, in the extreme case, Templar programs may contain no rules at all, and only procedures. This provides the user with the range of options and gives him/her extra flexibility for writing simulation programs based on rules, procedures and the combination of rules and procedures.

3.5 Temporal Predicates

As was explained in Section 2, Templar predicates change over time. For example, the predicate `submission(paper,author,status)` can have different truth values at different moments of time depending on the value of `status` at those moments.

Therefore, temporal operators, described in Section 2, can be applied to these predicates in **if** and **then** clauses. **If** clause takes only the past temporal operators **always_in_the_past**, **sometimes_in_the_past**, **previous**, **for_past_time**, and **within_past_time**. **Then** clause takes only the future temporal operators **always_in_the_future**, **sometimes_in_the_future**, **next**, **for_time**, and **within_time**.

Example 5 The rule

> Only the original papers are accepted for the conference, i.e. if a paper has been published in some journal in the past, it cannot be submitted to the conference.

can be expressed in Templar as

```
if        submission(paper,author,status) and
          sometimes_in_the_past published(paper,author,journal)
then-do   reject(paper,author)
```


Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

where **sometimes_in_the_past** is the temporal possibility operator defined in Section 2 and `reject` is the paper rejection activity.

□

3.6 Static and Dynamic Constraints

Templar supports static and dynamic constraints by specifying rules only with **if** and **then** clauses. The static constraint does not have any temporal operators in either the head nor the body of a rule. For example, the following static constraint

> A paper can have only *one* specific status at a time.

can be expressed in Templar as

```
if    submission(paper,author,status) and submission(paper,author,status')
then  status = status'
```

Note that this constraint specifies that `paper` and `author` functionally determine `status` in predicate `submission`.

A dynamic constraint is defined as an **if-then** rule where some predicates take temporal operators. For example, the following dynamic constraint

> If a paper has been published already, it cannot appear in any other publication in the future.

can be expressed in Templar as

```
if    published(publication,paper,author) and list_of_publications(publication')
      and publication ≠ publication'
then  always_in_the_future not published(publication',paper,author)
```

where `list_of_publications` describes the "universe" of publications in which the paper cannot appear.

3.7 Structuring Mechanisms in Templar

Templar supports structuring mechanisms of aggregation and generalization [TL82] as follows. Generalization is supported exactly as in ERAE [DHR91] by using multi-sorted temporal logic

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

that allows derived sorts (see Section 2). For example, if the sort **Papers** is defined as the union of **Regular_papers** and **Invited_papers** then Papers is the generalization of these two sorts. Assume it is declared that a variable x belongs to a sort and we want to state that it should belong a specialization of this sort. For example, assume that x belongs to **Papers** and we want x to be an invited paper. In this case, we follow the approach of ERAE and make a statement x **in Invited_papers**, where **in** is an interpreted membership predicate.

Aggregation is supported in Templar by the use of x.y notation. For example, an address can be defined by the street address, city, state, and zip. We can say in Templar that a person lives in New York as address.city = 'New York'. Note that the sort of the expression x.y is determined by the sort of variable y. For example, the sort of address.city is Cities.

3.8 Other Properties of Templar

In this section, we consider several additional features of Templar, such as parallel activities, external events, events defined by explicit specifications of time, periodic events, temporal precedence operators **before** and **after**, decisions, cancellations of and constraints on activities.

Example 6 Consider the following rule:

> When the program committee chair receives a paper before the submission deadline, the chair registers the paper, sends it to the reviewers and sends the acknowledgment letter to the author (at the same time as sending it to the reviewers).

It is expressed in Templar as

```
when      receives(chairperson,paper,author)
before    submission_deadline
then      next located(paper,chairperson)
then-do   register_paper(paper,author);
          (distribute_paper_to_reviewers(paper,chairperson)
          || send_acknowledgement(chairperson,paper,author))
```

□

The rule from Example 6 illustrates several important features of Templar. First, it provides an example of the *parallel* operator (||). This operator specifies that the corresponding activities occur simultaneously. For instance, activities distribute_paper_to_reviewers(paper,chairperson) and send_acknowledgement(chairperson,paper,author) occur in parallel in Example 6. Sec-

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

ond, the rule illustrates the use of *temporal precedence* operators **before** and **after**. The clause **before** specifies that the reviewing process can start only if the paper is received by the program chair before the submission deadline (determined by the *temporal constant* `submission_deadline`). Third, the rule shows how time can be referenced explicitly in Templar rules. The temporal constant `submission_deadline` (e.g. 6/22/98) defines the temporal event "the submission deadline is reached," and the rule can be fired only before this event occurs. Fourth, the rule provides an example of an *external* event, `receives(chairperson,paper,author)`. This event did not occur as a result of starting or ending of any internal activity but occurred because of some activity external to the system. Finally, the **then** clause provides an example of using temporal logic operators in post-conditions (e.g. **next**): it says that the predicate `located(paper,chairperson)` will be true at the time moment immediately following the execution time of the rule. In other words, the paper is "physically" located with the chairperson at the next time moment after he or she receives it.

The next example shows how Templar supports *periodic* temporal events.

Example 7 The rule

> Every Monday, the program chair examines review reports sent to him/her by the referees.

can be expressed in Templar as

```
when      every Monday
then-do   examine_reports(chairperson)
```

□

Also, Templar supports *decisions* which are *non-temporal* procedures. For example, when the program committee chair receives a paper, he/she *decides* who should review it, and then sends the paper to the selected reviewers. In this case, `select_reviewers(paper,chairperson,Reviewers)` is a decision, which we assume happens instantaneously in time[7]. Since decisions do not involve time, they can be specified either in Templar with only non-temporal operators or in *any* conventional programming language, e.g. Fortran or C. In the latter case, decision routines are dynamically linked to the main Templar program during the execution.

Templar also allows to refer explicitly to the time of an event. This can be done by using the *time* prefix. The next example illustrates the use of this construct. It also illustrates the use of the

[7] If this decision is made over time, then we treat it as an activity.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

then-dont-do and **then-cancel** clauses that respectively support *cancellations* of and *constraints* on activities.

Example 8 The rule

> If a paper was submitted to a journal and the reviews were not received by the author within 1.5 years, then withdraw the paper from the journal and never submit any papers to it again.

can be expressed in Templar as

```
if             now - time.begin.submission(paper,author,journal) > 18months
then-cancel    submission(paper,author,journal)
then-dont-do   sometimes_in_the_future submission(paper',author,journal)
```

where **now** is the symbol specifying the present time. `submission` is an activity, **begin**.`submission(paper,author,journal)` defines the event when the paper was submitted, and prefix **time** specifies the time when this event occurred. The clause **then-cancel** specifies that the currently scheduled activity `submission(paper,author,journal)` should be canceled, and the clause **then-dont-do** imposes a constraint stating that the activity `submission` should never occur for this author and this journal in the future.

□

Finally, Templar supports namings of the events associated with beginning and ends of activities. For example, the event `end.send` from Example 2 can be called `arrive` by the user.

In the next section, we formally describe the syntax of Templar and in Section 5 how Templar programs are executed.

4 Syntax of Templar

Templar programs consist of a set of predicate declarations, a set of rules and a set of activity specifications. Since Templar is based on multi-sorted temporal logic, all of its predicates must be declared so that it is clear what sorts are involved in their definitions. In order to do so, we have to specify the list of sorts that are used in the program. We adopt the syntax of ERAE for declaring sorts and predicates [DHR91] and will not present it in the paper.

The syntax of a Templar rule is defined with the BNF grammar presented in Fig. 2 and 3[8] (we

[8]We could not fit the BNF syntax on one page, and therefore we put it into two figures.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

```
rule                    ::=  [body-of-rule] head-of-rule
head-of-rule            ::=  head_clause { head_clause }
head_clause             ::=  then-clause | do-clause | dont-do-clause | cancel-clause
then-clause             ::=  then future-conditions
do-clause               ::=  then-do activity { next-activity }
dont-do-clause          ::=  then-dont-do activity { next-activity }
cancel-clause           ::=  then-cancel activity { next-activity }
next-activity           ::=  ; activity { next-activity } | || activity { next-activity }
body-of-rule            ::=  { body-clause }
body-clause             ::=  if past_conditions
                          |  while activities
                          |  when events
                          |  before activities
                          |  before events
                          |  after activities
                          |  after events
                          |  user-defined-operator activities
                          |  user-defined-operator events
user-defined-operator   ::=  name
activities              ::=  activity { logical-op activity }
events                  ::=  event { logical-op event }
activity                ::=  name ( arguments )
event                   ::=  begin-activity | end-activity | temporal-event | external-event
future_conditions       ::=  future_condition { and future_condition }
past_conditions         ::=  past_condition { logical-op past_condition }
future_condition        ::=  [not] future-temp-predicate
past_condition          ::=  [not] past-temp-predicate | expr relop expr | decision
future-temp-predicate   ::=  [unary-future-temp-oper] predicate
                          |  predicate binary-future-temp-oper predicate
past-temp-predicate     ::=  [unary-past-temp-oper] predicate
                          |  predicate binary-past-temp-oper predicate
predicate               ::=  name ( arguments ) | var in name
decision                ::=  name ( arguments )
begin-activity          ::=  begin.activity
end-activity            ::=  end.activity
external-event          ::=  name ( arguments )
temporal-event          ::=  temporal-constant | periodic-event
periodic-event          ::=  every period
```

Figure 2: Syntactic Definition of a Rule (Part I).

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

```
period                     ::=  hour | day | week | month | year | day-of-week
day-of-week                ::=  Monday | Tuesday | Wednesday | Thursday
                            |   Friday | Saturday | Sunday
temporal-constant          ::=  name
expr                       ::=  term + term | term - term
term                       ::=  factor * factor | factor / factor
factor                     ::=  var | const | time.event
logical-op                 ::=  and | or
relop                      ::=  = | ≠ | < | ≤ | > | ≥
unary-future-temp-oper     ::=  always_in_the_future | sometimes_in_the_future | next
                            |   for_time name | within_time name | user-defined-operator
binary-future-temp-oper    ::=  until | user-defined-operator
unary-past-temp-oper        |   always_in_the_past | sometimes_in_the_past | previous
                            |   for_past_time name | within_past_time name
                            |   user-defined-operator
binary-past-temp-oper      ::=  since | user-defined-operator
arguments                  ::=  name {. name }
var                        ::=  name.var | name
const                      ::=  now | name
```

Figure 3: Syntactic Definition of a Rule (Part II: continuation).

assume that *name* is a sequence of characters in Fig. 2 and 3). As Fig. 2 shows, a Templar rule consists of a collection of clauses that are divided into body and head clauses. There can be more than one clause of the same type in a rule (e.g. one **before** clause refers to activities and another to events). However, each clause deals only with an entity of one type: either with an activity, or an event, or a condition. Therefore, clauses provide a natural way to separate activities from events and from conditions and force the user of Templar language to think in these terms. Fig. 4 shows the relationship between clauses and activities, events, and conditions.

Furthermore, the user can define his or her own clause operators as long as the semantics of these operators is defined precisely. These operators are denoted as "user-defined-operator" in Figures 2 and 3. For example, the user can define such operators as **unless**, **atnext** [Kro87], or any other temporal operator he or she needs. This provides an extra flexibility in describing real-world systems in more natural terms.

The syntax of activity specifications is defined with the BNF rules presented in Fig. 5. As Fig. 5 shows, an activity specification consists of a list of statements. The *for-statement* is needed for iterations (to be able to express statements of the form "for each element ... perform some activity"). *If-statement* is not strictly necessary because the activity containing this statement can be expressed in terms of rules and activities without *if-statement*. However, it was added as a

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

	clauses
conditions	**if, then**
events	**when, before, after**
activities	**then-do, then-dont-do, then-cancel, while, before, after**

Figure 4: Types of Clauses

activity-spec	::=	**activity** name [(parameters)] statement-list **end_activity**
statement-list	::=	statement { ; statement }
statement	::=	composite-activity
	|	atomic-activity
	|	if-statement
	|	for-statement
	|	parallel-statement
	|	decision-statement
if-statement	::=	**if** condition **then** statement-list **else** statement-list **end_if**
for-statement	::=	**foreach** variable **suchthat** condition **do** statement-list **end_for**
parallel-statement	::=	statement-list || statement-list
decision-statement	::=	[variable =] name (parameters)
composite-activity	::=	name (parameters)
atomic-activity	::=	future-temp-predicate
future-temp-predicate	::=	same as *future-temporal-predicate* in Fig. 2
variable	::=	name.variable | name
parameters	::=	name: type {, name: type }
type	::=	name

Figure 5: Syntactic Definition of Activity Specification .

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

1. advance the system clock to the next event that is scheduled in the future;
2. match the antecedents of the rules against the current and the past states of temporal predicates and against the previous events and activities; as a result of this matching process, a set of tuples is instantiated; form a set of future activities and future values of predicates from this set of instantiated tuples;
3. resolve conflicts among conflicting activities and among conflicting predicates;
4. schedule the activities and the predicates that passed the conflict resolution step for the future executions;
5. execute the previously scheduled activities and predicates whose execution time has come.

Figure 6: Temporal Recognize-Act Cycle for Templar Rules.

convenience for the user. Activities occur either sequentially or in parallel. Semicolon (;) is the operator delineating sequential activities, and parallel bars (||) is the operator delineating parallel activities.

As was pointed out in Section 3.2. we distinguish between atomic and composite activities. An atomic activity is defined as a future temporal predicate. For example, `deliver(paper,referee)` **for_time** `T`, where `deliver` is a *predicate* indicating that the paper is being delivered to the referee for `T` time units, is an atomic activity. A composite activity consists of several subactivities and requires an activity specification that describes the decomposition of the composite activity into several subactivities.

As Fig. 5 shows, each element in the list of parameters belongs to a certain type.

5 Executing Templar Programs

In this section, we describe how Templar rules are executed in a recognize-act cycle. As in the case of production systems, such as OPS5 [BFK86], the cycle consists of the matching, conflict resolution and execution steps. The sequence of these steps is presented in Fig. 6. Steps 1 and 2 in this sequence correspond to the matching part of the cycle, Step 3 to the conflict resolution part, and Steps 4 and 5 to the execution part of the cycle. We will describe each step in Fig. 6 in detail in the remainder of this section. However, before we describe these steps, we present the data structures being used.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

FIELD	TYPE	DESCRIPTION
FROM:	time	beginning of a temporal interval when the tuple belonged to the predicate
TO:	time	end of the temporal interval when the tuple did not belong to the predicate
NEXT:	pointer	pointer to the next node (in the decreasing order of time)

Figure 7: The Structure of a Node in the Past List of a Dynamic Predicate Table.

FIELD	TYPE	DESCRIPTION
I_D:	boolean	is equal to 1 if the tuple will be added to the predicate and 0 if deleted
A_S:	boolean	is equal to 1 if the temporal operator associated with the tuple is *always* and is 0 if it is *sometimes*
FROM:	time	beginning of a temporal interval when the tuple will be added to the predicate
TO:	time	end of a temporal interval when the tuple will be added to the predicate
NEXT:	pointer	pointer to the next node

Figure 8: The Structure of a Node in the Future List of a Dynamic Predicate Table.

We use two separate data structures for activities and temporal predicates. We first describe predicate structures. Schemas of all the predicates being used in the application are stored in a *static predicate table*. For each predicate, there is one record in the table describing how many arguments the predicate has, types of arguments, and also containing a pointer to the *dynamic predicate table*.

The dynamic predicate table for predicate P contains all the time-dependent information about P. Specifically, it contains the list of all the tuples $l(P)$ that were ever inserted into this predicate. It also contains two linked lists for each tuple t in $l(P)$. The first list $p(t)$ is the list of all the *past* time intervals when $P(t)$ was true. Each node in this list has the structure presented in Fig. 7. Nodes in the past list are organized in the decreasing order of time (from the most recent to the more distant in time). The second linked list $f(t)$ is the set of all the time intervals when tuple t is *scheduled* to be either inserted into or deleted from the predicate in the *future*. Each node in the list $f(t)$ consists of the fields presented in Fig. 8. Nodes in the future list are organized as follows. If node n_1 has FROM and TO fields equal to FROM(n_1) and TO(n_1) respectively, and node n_2 has fields FROM(n_2) and TO(n_2), and if TO(n_1) $<$ FROM(n_2) then node n_1 must precede node n_2 in $f(t)$. If time intervals of the nodes in $f(t)$ intersect then these nodes can have an arbitrary

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

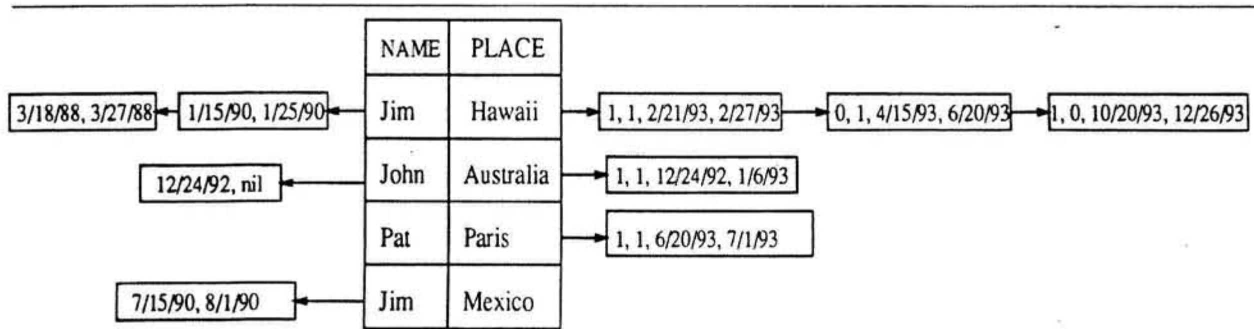

Figure 9: An Instance of the Temporal Predicate VACATION.

precedence.

Example 9 Consider the predicate VACATION(NAME.PLACE) that specifies where a person spends vacations. An instance of this predicate may look as the one shown in Fig. 9. Assume that the current time is 1/1/93. Then the tuple (Jim, Hawaii) in Fig. 9 has three nodes in the future list and two nodes in the past list associated with it. The future list says that Jim has a planned vacation in Hawaii from 2/21/93 to 2/27/93, that he will not go to Hawaii from 4/15/93 to 6/20/93, and he will go to Hawaii at some point between 10/20/93 and 12/26/93 (but does not know when and for how long yet). The past list says that Jim had vacations in Hawaii from 3/18/88 to 3/27/88 and also from 1/15/90 to 1/25/90.

□

For each activity, we maintain two data structures. The first one, called the *static activity table*, contains all the time-invariant information about the activity, e.g. the name of the activity, descriptions of the arguments of the activity, what the subactivities of the activity are, definition of the activity if it is an atomic one, etc. The static activity table can be thought of as a schema of the activity.

The second data structure associated with an activity, called *dynamic activity table*, contains all the time-dependent information about the activity. For an activity A, the dynamic activity table contains the instances of the activity, $Ia(A)$, that ever occurred or are scheduled to occur for A. For each tuple t in $Ia(A)$, we also maintain two linked lists as for temporal predicates. The first list $pa(t)$ is the list of all the *past* time intervals when activity $A(t)$ occurred in the past for tuple t. Each node in this list has the structure presented in Fig. 10. Nodes in the past list are also organized in the decreasing order of time (from the most recent to the more distant in time).

The second linked list $fa(t)$ is the set of all the *future* time intervals when activity $A(t)$ is

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

FIELD	TYPE	DESCRIPTION
FROM:	time	starting time of an activity
TO:	time	ending time of an activity
NEXT:	pointer	pointer to the next node (in the decreasing order of time)

Figure 10: The Structure of a Node in the Past List of a Dynamic Activity Table.

FIELD	TYPE	DESCRIPTION
TYPE:	integer	is equal to 0 if it is an activity scheduled by the **then-do** clause, 1 if it is a constraint scheduled by the **then-dont-do** clause, and 2 if it is a cancellation of an activity
FROM:	time	beginning time of the scheduled activity
TO:	time	ending time of the scheduled activity
NEXT:	pointer	pointer to the next node

Figure 11: The Structure of a Node in the Future List of a Dynamic Activity Table.

scheduled at some time in the future for tuple t. Each node in this list has the structure presented in Fig. 11. Nodes in the future list of the dynamic activity table are organized as in the future list of the dynamic predicate table. In particular, if node n_1 has FROM and TO fields equal to FROM(n_1) and TO(n_1) respectively, and node n_2 has fields FROM(n_2) and TO(n_2), and if TO(n_1) < FROM(n_2) then node n_1 must precede node n_2 in $fa(t)$. If time intervals of the nodes in $fa(t)$ intersect then these nodes can have an arbitrary precedence.

Example 10 Consider an activity STUDY(NAME,SCHOOL) that specifies the past studying history and future studying plans of a person. An instance of the dynamic activity table for STUDY is shown in Fig. 12. Assume that the current time is 1/1/93. Then the tuple (John,NYU) has two nodes in its past activity list. The first node says that John attended NYU from 9/1/89 until 6/1/91 and from 9/1/92 until the present time. The same tuple (John,NYU) has also three nodes in the future activity table. The first node says that John resumed his attendance of NYU on 9/1/92 and will continue to attend it until 6/1/93. Also, he will take a year off and will not attend NYU from 9/1/93 until 9/1/94, and then will resume attendance from 9/1/94 until 6/1/95.

□

As was stated before, the recognize-act cycle consists of the matching, conflict resolution, and execution parts. We describe each part in turn now.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

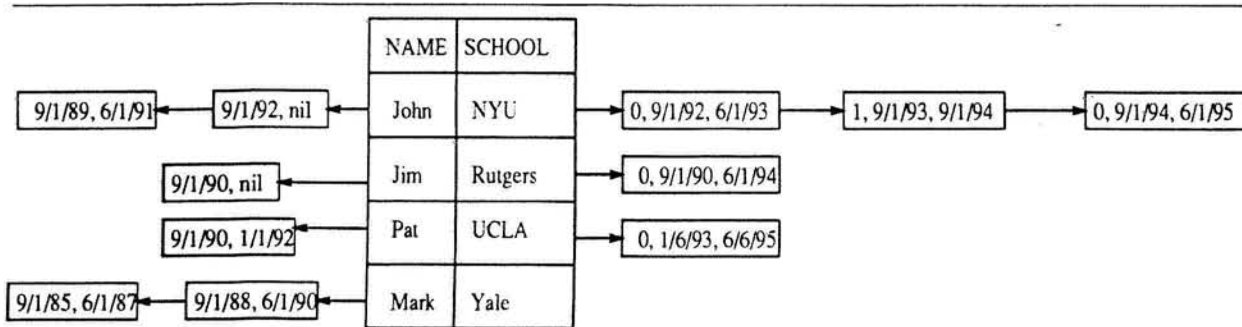

Figure 12: An Instance of the Dynamic Activity Table for Activity STUDY.

5.1 Matching Part of the Recognize-Act Cycle

The matching part of the cycle starts with the selection of the smallest time t_{next} associated with *any* future event that is scheduled in the system. This time is determined by selecting the smallest time $t_{next} > now$ from the nodes of the future lists of all the predicates and all the activities appearing in all the rules. For example, assume that there is only one predicate VACATION and only one activity STUDY in the program. Also assume that the current time is 1/1/93. Then the smallest time t_{next} based on the data in Fig. 9 and Fig. 12 is 1/6/93. This time is associated with the end of John's vacation in Australia and the beginning of Pat's studies at UCLA. If any of the external events occurred between *now* and t_{next} then set t_{next} to the value of the smallest time among these external events. Also, if any temporal constants in **when** clauses of any of the rules happen to be between the current moment of time and t_{next} then set t_{next} to the value of the smallest temporal constant. After t_{next} is determined as just described, make the current time *now* equal to t_{next}.

Once the time clock is advanced to t_{next}, the matching process starts. Matching is done on a clause-by-clause basis within a rule based on the following ordering of its clauses. The highest order is associated with the **when** clause, then the **if** clause, then the **while** clause, and finally, the **before** and **after** clauses. For example, if a rule has **when**, **if**, and **before** clauses then first the **when** clause is matched against the data, then the **if** clause, and finally, the **before** clause.

The **when** clause is matched against the data as follows. We first find all the events (beginnings and endings of activities, external and temporal events) that occur at (new) time t_{next}. For example, if t_{next} = 1/6/93 and if a rule contains the clause **when begin**.STUDY(Pat,UCLA), then there is only one event selected for the data from Fig. 12, i.e. **begin**.STUDY(Pat,UCLA). However, there can be more than one event selected in general since more than one event can occur at the same time. The matching of the **when** clause against the data produces the relation R_w (it is

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

{(Pat,UCLA)} in our example). We expect the size of R_w to be small on average in comparison to the size of the data because there should be a small number of events that occur exactly at the same time on average. For this reason, we started the matching process with the **when** clause so that the size of the instantiated relation be reduced at the early stage of the matching process.

After the matching of the **when** clause is finished, we match the **if** clause against the past data and relation R_w as follows. Without loss of generality, we assume that the **if** clause has the form P_1 *and* ... *and* P_n, where P_i is a temporal literal (otherwise, we convert the clause into the disjunctive normal form and split the rule into several rules, each rule containing one disjunct). We replace each P_i with the semijoin [Ull88] $P_i' = P_i \ltimes R_w$ and evaluate P_i' against the past data in the dynamic predicate table for P_i as follows. If $P_i'(t)$ is **sometimes_in_the_past** $P(t)$ then check for each tuple t in P if the past list $p(t)$ is not empty. If $P_i'(t)$ is **always_in_the_past** $P(t)$ then check for each tuple t in P if $p(t)$ has only one node and if it covers all the time points. If $P_i'(t)$ is **within_past_time** T $P(t)$ then for each tuple t in P go over the nodes in $p(t)$ to see if some node has times that fall between *now* and T. The case for **for_past_time** T P is handled similarly. As a result of matching the **if** clause against the data and relation R_w, we obtain the relation R_{wi}.

After that, we match the **while** clause against the data and relation R_{wi} as follows. Also without loss of generality we assume that the **while** clause has the form A_1 *and* ... *and* A_n, where A_i is an activity. We replace each A_i with the semijoin $A_i' = A_i \ltimes R_{wi}$. For each activity A_i and for each tuple t, such that $A_i'(t)$ is true, check if the first node in the past list $pa(t)$ of the dynamic activity table for A_i has a non-*nil* time in the FROM field and *nil* in its TO field. All the tuples t that satisfy this condition form a relation P_i''. Then all the relations P_i'', $i = 1, \ldots, n$ are joined together to form the relation R_{wiw}. For example, consider the clause **while** `STUDY(name,school)`, and let the tuple (Pat, UCLA) belong to the semijoin of STUDY and R_{wi}. Since the first node in *pa*(Pat,UCLA) is (9/1/90, 1/1/92), the tuple does not pass the **while** test and does not belong to R_{wiw}. However, if the first node in *pa*(Pat,UCLA) had *nil* in the TO field (e.g. was (9/1/90, *nil*)) then the tuple (Pat, UCLA) would have passed the **while** test and should have been added to relation R_{wiw}.

The **after** clause is matched against the data similarly to the **while** clause. First, all the activities in the clause are semi-joined with R_{wiw}. Then for each conjunct in the **after** clause and each tuple t in the newly created semijoin in that conjunct, check if the TO field in the first node in $pa(t)$ is not empty. All the tuples that passed this test form the relation R_{wiwa}. The matching process for the **before** clause is done similarly to the **after** clause, except the check is done against the future list $fa(t)$. The resulting relation R_{wiwab} forms the set of instantiated tuples for the rule.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

The relation R_{wiwab} of instantiated tuples is used to schedule new instances of temporal predicates to be true or false in the future. It is also used to schedule new instances of activities and their subactivities. For example, assume that the **then-do** clause of a rule is **then-do** `review(paper, reviewer)`, where activity `review(paper, reviewer)` was defined in Example 3, and assume that R_{wiwab} contains two tuples { `(paper_29,Jack)`, `(paper_43,Susan)` }. Then this predicate instance R_{wiwab} gives rise to the instances of activities `read(paper_29,Jack)`, `read(paper_43,Susan)`, `evaluate(paper_29,Jack)`, `evaluate(paper_43,Susan)` that will have to be scheduled in the future. The scheduling is based on the computations of the time intervals of individual subactivities and on the composition of these subactivities into activities. For example to schedule activities mentioned before, we first compute durations of atomic activities `read` and `evaluate` at the scheduling time (since an atomic activity is defined as a temporal predicate, and it is known for how long it will be true at the scheduling time). Assume that `read(paper_29,Jack)` will be true for 30 days, `read(paper_43,Susan)` for 40 days, `evaluate(paper_29,Jack)` for 1 day, `evaluate(paper_43,Susan)` for 2 days. Then `read(paper_29,Jack)` will be scheduled from time *now* until *now* + 30, `evaluate(paper_29,Jack)` from *now* + 30 until *now* + 31, and `review(paper_29,Jack)` from *now* until *now*+31. Similarly, `read(paper_43,Susan)` will be scheduled from *now* until *now* + 40, `evaluate(paper_43,Susan)` from *now* + 40 until *now* + 42, and `review(paper_43,Susan)` from *now* until *now* + 42.

Note that during the execution of the recognize-act cycle, the past activity and predicate lists grow longer with time. Therefore, a special care was taken for the Templar interpreter not to deteriorate its performance with time. To illustrate how it is done, consider the operator **sometimes_in_the_past** $P(x)$. To check if it is true, we have to see if the past list in the dynamic predicate table for $P(x)$ is not empty, and this can be accomplished in constant time. Similarly, to check if the operator **for_time**$(T)P(x)$ is true, we have to check the past portion of the dynamic table for at most T time units, and this can be done in the amount of time proportional to T. Therefore, the performance of the matching part of the cycle does not deteriorate with time in these two cases. We employ similar techniques for other temporal operators in the Templar interpreter. Therefore, the performance of the matching part of the cycle does not deteriorate with time for other cases as well.

As a result of the matching step of the recognize-act cycle, we obtain a set of new predicates and activities to be scheduled in the future and times at which these operations begin and end. In the next step of the cycle, we have to resolve conflicts among these operations and *also* the conflicts with the previously scheduled operations.

There have been two conflict resolution approaches proposed in the past. The first is the logic

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

based approach. It says that if we conclude that P and **not**P are true at the same time then this is a contradiction and the program execution should stop. Doubly negated Datalog, Datalog$^{\neg *}$ [AV91], follows this approach. In similar situation, MetateM [BFG+89] stops the current execution, backtracks and tries to find another model in which the conflict does not appear. The second approach is used in production systems. It says that the conflict should be resolved according to some conflict resolution strategy [BFKS86]. We will follow the production system approach in this paper and describe how conflicting actions are resolved in Templar in the next section.

5.2 Conflict Resolution Part of the Recognize-Act Cycle

In (non-temporal) production systems, such as OPS5, conflicts between adding to and removing elements from a working memory occur between operations generated within the *same* recognize-act cycle. In the temporal case the situation becomes more complex because conflicts can also occur between activities and between predicates scheduled at *different moments of time*. For example, assume that the current time is 80 and some rule schedules predicate P to be true from time 100 to 120. Suppose that another rule scheduled predicate P to be false from time 110 to 130, and this scheduling occurred at time 60. Clearly, these two rules conflict, even though they scheduled predicates to be true at different moments of time.

According to Ioannidis and Sellis [IS89], conflicts in rules can occur either at the rule, or the antecedent, or the consequent levels. For example, OPS5 resolves conflicts at the antecedent level. In the temporal case, conflicts *must* be resolved at the consequent level because activities and temporal predicates can conflict with the *previously scheduled* activities and predicates. For this reason we consider conflict resolutions at the consequent level. One consequence of this choice is that rules can be fired in parallel in the temporal case (unlike OPS5, which can fire only one rule at a time) since conflicts are resolved at the consequent part.

We describe conflict resolution strategies separately for temporal predicates and activities because they are handled somewhat differently.

5.2.1 Conflicts Between Temporal Predicates

Conflicts between two temporal predicates scheduled in the future *can* occur if one predicate is scheduled to be true over the time interval $[T_1, T_2]$ and another is scheduled to be false over the time interval $[T_3, T_4]$, and the time intervals $[T_1, T_2]$ and $[T_3, T_4]$ intersect.

Since the scheduled operations come in two "flavors" *always* and *sometimes* (based on the value of the A_S field in Figure 8), we have to consider three types of conflicts: between two *always*

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

operations, between *always* and *sometimes* operations. and between two *sometimes* operations.

If two potentially conflicting actions are of the type *always* then the conflict occurs when their time intervals intersect. Formally. P_1 **for_time** T_1 scheduled at time T_3 conflicts with **not** P_2 **for_time** T_2 scheduled at time T_4 if the time intervals $[T_3, T_3 + T_1]$ and $[T_4, T_4 + T_2]$ intersect.

We consider two types of conflicts between *always* and *sometimes* operations. Let the first operation be P_1 **for_time** T_1 and let it be scheduled at time T_3. Let the second operation be **not** P_2 **within_time** T_2 and let it be scheduled at time T_4. Then the *intersection semantics* of conflicts says that the two operations conflict when intervals $[T_3, T_3 + T_1]$ and $[T_4, T_4 + T_2]$ *intersect.* Intuitively, it says that if an *always* operation overlaps with a *sometimes* operation then the *sometimes* operation cannot be scheduled at any arbitrary time in the interval $[T_4, T_4 + T_2]$ and must be restricted to some smaller time domain. which may not be what the programmer had in mind when he or she had written the program. The *containment semantics* of conflicts says that the two operations conflict when interval $[T_3, T_3 + T_1]$ *contains* interval $[T_4, T_4 + T_2]$. Intuitively, it says that if *always* operation is scheduled during the whole time interval of *sometimes* operation, then the *sometimes* operation cannot occur at *any* point in this time interval. Clearly, this means that *sometimes* operation is invalid. and the two operations conflict.

The last type of conflict occurs between two *sometimes* operations. In this case, we also consider two types of semantics for conflicts. As in the previous case. if two *sometimes* operations occur at time intervals $[T_3, T_3 + T_1]$ and $[T_4, T_4 + T_2]$ then the *intersection* type of conflict occurs when these intervals intersect. The *containment* type of conflict occurs when $T_1 = T_2 = 0$.

Once we identified when conflicts between temporal predicates occur, we are ready to describe how they can be resolved. As was pointed out before. we distinguish between two types of conflicts: conflicts between operations scheduled at the same time. and conflicts between operations scheduled at different moments of time. We start with the conflicts between operations scheduled at the same moment of time.

Conflicting operations scheduled at the same moment of time can be resolved with any of the conflict resolution strategies proposed for production systems and active databases [MD89, WF90. SJGP90, GJ91]. One such strategy orders rules (either partially or totally) according to their precedence. Then the qualifying rules with the highest precedence are selected. This is the conflict resolution strategy adopted in active databases Starburst [WF90] and POSTGRES [SJGP90]. The conflict resolution strategy of OPS5 is based on several tuple selection criteria that take into account structural properties of rules and recency of tuple insertions into the working

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

memory [BFK86]. If all these criteria fail to resolve the conflict, a single instantiation is chosen at random. Still another conflict resolution strategy initially proposed in [KT89] and later extended in [TK91] operates on the consequent part of a rule. It assumes that the insertion of a tuple has a precedence over its deletion if the database does not contain the tuple and the deletion has a precedence over the insertion if the tuple exists in the database. The intuitive justification for this strategy is presented in [TK91]. Furthermore, de Maindreville and Simon [dMS88] describe a conflict resolution strategy (within a rule), such that if an insert operation conflicts with a delete operation, then both operations are canceled. In conclusion, any of these strategies can be used to resolve conflicts between the operations scheduled at the same time.

If the operations are scheduled at different moments of time, we propose the following *temporal* conflict resolution strategy:

> If the operations of two rules conflict, then select the operation of the rule that *fired first.* If both rules are fired at the same time then apply any conflict resolution strategy for the non-temporal case described above, e.g. cancel the conflicting operations or select the conflicting operation from the rule with the higher precedence.

For example, if rule R_1 scheduled predicate $P(a_1, \ldots, a_n)$ to be always true from time 40 to time 60 at time $t = 20$, and rule R_2 scheduled predicate $P(a_1, \ldots, a_n)$ to be always false from time 50 to 80 at time $t = 30$, then the first operation has a precedence over the second operation because rule R_1 was fired before rule R_2.

Intuitively, this conflict resolution strategy says that once an operation is scheduled for a future execution, then the commitment is made to execute it at some later time, and the scheduled operation cannot be canceled[9].

5.2.2 Conflicts Between Activities

Activities can be divided into three groups: activities that appear in the **then-do**, **then-dont-do**, and **then-cancel** clauses. Correspondingly, we consider three types of conflicts between activities: in two **then-do** clauses, in the **then-do** and the **then-dont-do** clauses, and in the **then-do** and the **then-cancel** clauses.

We define conflicts between activities recursively. Since atomic activities are defined in terms of temporal predicates then two atomic activities conflict if their corresponding temporal predicates

[9]If there is a need to cancel the previously scheduled operation, the user has to use the **then-cancel** clause in a rule. The semantics of this clause will be described in the next section.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

conflict. Two non-atomic activities conflict if some of their subactivities conflict.

If the conflict occurs between two activities from the **then-do** clauses then the conflict is resolved in the same manner as for temporal predicates as described in Section 5.2.1, i.e., one activity has a precedence over another conflicting activity, if it was scheduled before the other one. If two activities occur at the same time then we can also use any of the conflict resolution strategies described for the predicates in the same case. If the conflict occurs between activities from the **then-do** and **then-dont-do** clauses then it means that the program is incorrect because the explicit "don't do" constraint imposed by the user is violated, and the execution of the program terminates with an error message. If the conflict occurs between activities from the **then-do** and **then-cancel** clauses and if the activity in the **then-do** clause was scheduled before, then it is terminated (i.e. is removed from the future list in the dynamic activity table).

As a result of the conflict resolution step, we have the list of non-conflicting activities that are ready to be scheduled for the execution in the future and the list of predicates that are ready to be set true or false at certain times in the future. The next step in the recognize-act cycle is to execute scheduled activities and operations.

5.3 Execution Part of the Recognize-Act Cycle

The execution part of the cycle consists of two subparts. In the first part, the activities and operations that survived the elimination process in the conflict resolution part of the cycle are scheduled for the future execution. In the second part, previously scheduled activities and operations are actually executed. We start our description with the scheduling part first.

To schedule a new activity $A(t)$ that takes place from time T to T', we have to go to the future list $fa(t)$ of the dynamic activity table A for tuple t. Let the beginning and end times of the node NODE$_i$ in the list $fa(t)$ be T_{FROMi}, T_{TOi}, for $i = 1, \ldots, k$. Then the new activity should be placed in the list $fa(t)$ so that if it turns out that $T_{TOi} < T$ then the new activity should be placed *after* the node NODE$_i$. Similarly, if $T' < T_{FROMi}$ then the new activity should be placed *before* NODE$_i$. If these two conditions are not satisfied then the new activity can overlap the previously scheduled one. However, this does not cause any problems because the activities do not conflict with each other (conflicts have been resolved in the previous step). Future predicate values are scheduled similarly to the future activities, and we omit the description of how it is done.

Once all the new activities and predicate values are scheduled, we are ready to execute previously scheduled activities and predicates whose execution times have arrived. In the matching part of the cycle described in Section 5.1, the system clock was advanced forward to the closest

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

event(s) scheduled in the future. In the execution part of the cycle, we go over all these events and, depending on their types, do the following things. We first start with the events associated with beginnings and endings of the time intervals when *predicates* are true or false. Then we consider beginnings and endings of activities.

If the event is associated with the *beginning* of the time interval $[T_1, T_2]$ during which predicate $P(t)$ is ***always true*** for tuple t (note that T_1 must be the current moment of time) then we check the TO field of the first node in the past list $p(t)$ for $P(t)$. If the value is not *nil* then it means that $P(t)$ is false now, and we have to make it true again. In this case, we create a new node in the past list of time intervals $p(t)$ and put it in front of the first node of $p(t)$. We also place the current time into the FROM field and *nil* into the TO field of that node.

If the event is associated with the *beginning* of the time interval $[T_1, T_2]$ during which predicate $P(t)$ is ***always false*** for tuple t (again, T_1 must be the current moment of time) and if the past list $p(t)$ has a node with the TO field being *nil*, then this means that $P(t)$ is true now, and we have to make it false. In this case, we insert the value of the current time, T_1, in that node.

If the event is associated with the *beginning* of the time interval $[T_1, T_2]$ during which predicate $P(t)$ is ***sometimes true***, then we check the TO field of the first node in the past list $p(t)$ for $P(t)$. If the value is *nil* then it means that $P(t)$ is true now, and the commitment to have $P(t)$ true between times T_1 and T_2 is fulfilled. Therefore, we don't have to do anything in this case. If the value is not *nil* then it means that $P(t)$ is false now, and we attempt to make it true now (at time T_1) for one time instance. To do this, we create a new TRUE node and make its FROM and TO fields equal to "now" (T_1). Then we check if this new node conflicts with the previously scheduled future nodes. If it does, we do nothing. If it does not conflict, we put it in front of the first node of the past list of time intervals $p(t)$.

If the event is associated with the *beginning* of the time interval $[T_1, T_2]$ during which predicate $P(t)$ is ***sometimes false***, then we check the TO field of the first node in the past list $p(t)$ for $P(t)$. If the value is not *nil* then it means that $P(t)$ is false now, and the commitment to have $P(t)$ false between times T_1 and T_2 is fulfilled. Therefore, we don't have to do anything in this case. If the value is *nil* then it means that $P(t)$ is true now, and we attempt to make it false now (at time T_1) for one time instance. To do this, we create a new FALSE node and make its FROM and TO fields equal to "now" (T_1). Then we check if this new node conflicts with the previously scheduled future nodes. If it does, we do nothing. If it does not conflict, we incorporate it into the past list of time intervals $p(t)$.

If the event is associated with the *end* of the time interval $[T_1, T_2]$ during which predicate $P(t)$

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

is *sometimes true* then we check whether the TO field in the first node in the past-list $p(t)$ is either *nil* or is greater than T_1. If this is the case, it means that at some point between the times T_1 and T_2 $P(t)$ was made to be true. In this case, the commitment to make $P(t)$ true sometime between times T_1 and T_2 is satisfied, and there is no need to do anything. However, if the TO field has the value less than T_1 then the commitment was not satisfied, and we have to fulfill it at the last point of the time interval $[T_1, T_2]$ by making $P(t)$ true now (at time T_2). In this case, we create a new TRUE node with the FROM and TO fields equal to "now" (T_2) and see if it conflicts with the previously scheduled future nodes. If it does, we *cancel* the execution of Templar program (because we did not satisfy the commitment to make $P(t)$ to be true sometimes between T_1 and T_2). If it does not conflict, we put it in front of the first node in $p(t)$.

If the event is associated with the *end* of the time interval $[T_1, T_2]$ during which predicate $P(t)$ is *sometimes false* then we check whether there is a node in the past list $p(t)$ with begin/end times (T_1', T_2') such that $T_1 < T_2' < T_2$. If this condition is true, this means that $P(t)$ was false sometime after T_1 and before T_2, and therefore the commitment to make $P(t)$ false sometime between times T_1 and T_2 was fulfilled. In this case, we don't have to do anything. However, if the condition does not hold, this means that we have to fulfill the commitment at the very last point of the interval $[T_1, T_2]$, as in the previous (true) case. In this case, we take steps similar to the previous case, i.e., create a FALSE node with FROM and TO fields equal to T_2 and see if it conflicts with the previously scheduled future nodes. If it does conflict, we abort the execution of the program. If it does not, we put T_2 in the TO field of the first node in the past list of $p(t)$.

If the event is associated with the *end* of the time interval $[T_1, T_2]$ during which predicate $P(t)$ is either *always true* or *always false* then we do nothing. We do nothing because after the predicate stops to be true, it can take any value, i.e. either true or false. Therefore, unless stated otherwise, the predicate remains to be true. The same argument applies to the case when the predicate is false.

Also, in case of the events associated with endings of the time interval $[T_1, T_2]$ for some $P(t)$, the corresponding node is removed from the future list for $P(t)$.

This completes the consideration of events associated with the beginnings and endings of the time intervals when predicates are true or false. We next consider activities.

If the event is associated with the beginning of activity $A(t)$, then check the past list $pa(t)$ of tuple t in the dynamic activity table $A(t)$. If the TO field of the first node in this table has the value that is different from *nil* then create a new node and place it at the beginning of the $pa(t)$ list. Set the FROM field in this node to the current value of time, and the TO field to *nil*.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

If the event is associated with the end of activity $A(t)$, then again check the past list $pa(t)$ of tuple t in the dynamic activity table $A(t)$. If the TO field of the first node in this table has *nil* in it then replace it with the current value of time.

This completes the description of the recognize-act cycle and the semantics of Templar.

5.4 Implementation of the Templar Interpreter

The Templar interpreter, based on the description of the temporal recognize-act cycle presented in this section (Section 5), was implemented on a Sun Workstation in C.

The interpreter works as follows. It takes a Templar program and parses it using the parsing tables generated by YACC parser generator. The result of the parsing process is a set of internal data structures, including the static predicate table, the static activity table, and an internal representation of Templar rules. After that, the interpreter initializes the dynamic tables, including the dynamic activity, predicate and external event tables[10], by reading the initial state of the system specified by the user (e.g. the list of the papers being initially submitted, the list of the initially selected reviewers, etc.). Following this stage the interpreter executes the temporal recognize-act cycle, as described above, either until no rules can be fired or until the time limit specified by the user is reached.

As was pointed out in Section 5.2, different strategies can be used by the interpreter to resolve conflicts between activities and between predicates. In our implementation, we selected the following strategy (that was described in Section 5.2). If action A_1 conflicts with action A_2 and action A_1 was scheduled first, then action A_1 has precedence over action A_2. If two conflicting actions are planned to be scheduled at the same time, then we cancel both actions[11].

It took 10 man-months to develop the interpreter, and the program contains over 5000 lines of C code. In the next section, we describe a case study that will be used for testing the interpreter.

6 Case Study

To test the Templar interpreter described in the previous section, and to test the language in terms of ease of development, reliability, and maintainability of its programs, we did a case study. In

[10] We assume that the user specifies the future occurrences of external events before the execution starts by placing all of them into the external event file. However, we plan to extend this part of the interpreter in the future by modeling external events with some Poisson arrival processes, as is usually done in simulation systems.

[11] As was stated in Section 5.2, the non-temporal component of this conflict resolution strategy was proposed in [dMS88]. However, our interpreter can easily incorporate any other conflict resolution strategy described in Section 5.2. This is in the spirit of OPS5 that can use either LEX or MEA strategies [BFK86]

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

this case study, we implemented a portion of the Intelligent Adversary (IA) system for the Naval Training Systems Center that simulates behavior of navy pilots in combat situations. This system helps to train navy pilots for air battles and can be thought of as a very sophisticated version of a flight simulator video game, where the IA subsystem simulates the behavior of the "bad guys." The IA system has been implemented in OPS5 before, and it took two man-years to develop it. In our case study, a portion of it was rewritten in Templar.

We selected this case study because it has a very rich temporal component since navy pilots have to react to adversary actions in time. For example, the following statement is a part of an informal English description of pilot's behavior demonstrating the richness of the temporal domain in this application [Bod92]:

> If the enemy flies on an intercept course for at least 3 seconds and then he flies with at least 30° of aspect for 5 seconds and if the elapsed time between the end of his flying intercept and the beginning of his flying aspect is less than 3 seconds, then this means that he may have fired a missile at you and is doing an F-pole now.

In this case study, we implemented a module of the IA system that selects an appropriate radar mode and then designates the target. "Designation" is a technical term meaning that a pilot presses a special "designate" button on his radar that locks the radar on a particular target and displays vital information about that target. The designation process continues until the pilot makes the final decision which target to pursue. This module constitutes about 10% of the total IA system [Bod92].

The description of the Templar program that simulates the selection of a radar mode and the designation process is presented in [Bod93] and is based on the extensive practical experience of interviewing navy pilots. It contains 30 Templar rules, 11 activities, 21 predicates, and 5 external events. We present examples of three rules from this program in order to show its "flavor."

The first rule says that if a pilot is waiting for a radar return, while designating a target, and when he actually gets the return, then he should stop waiting for the results of the designation and check the returned results. This rule is expressed in Templar as

```
when     new_radar_return(pilot)
while    designate(pilot,target)
if       waiting_for_radar_return(pilot)
then-do  check_results(pilot,target)
then     not waiting_for_radar_return(pilot)
```


Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

where new_radar_return is an external event specifying that the radar gets a new return, designate is an activity designating a target, waiting_for_radar_return is a predicate specifying that the pilot is waiting for the radar return. and check_results is an activity that checks the results of the new radar return.

The second rule says that when the targets on the radar screen change (the set of targets becomes different) while the pilot tries to choose the desired target, then terminate the process of choosing, and start it all over again. It is expressed in Templar as

```
when         change_in_targets(pilot)
while        choose_desired_target(pilot)
then-cancel  choose_desired_target(pilot)
then-do      choose_desired_target(pilot)
```

where change_in_targets is an external event, and choose_desired_target is an activity. This rule cancels the old selection activity and starts a new one when the set of targets changes on the radar screen.

The third rule says that every 10 seconds, if the radar has been in the RWS mode continuously for the last 10 seconds, and if the pilot is not in the process of choosing a desired target then set the radar to the TWS mode (for a quick look at the airplanes). This rule is expressed in Templar as

```
when     every 10seconds
while    not_choosing_desired_target(pilot)
if       RWS_mode(pilot,radar) for_past_time(10seconds)
then-do  check_TWS_data(pilot,radar)
```

7 Conclusions

In this paper we described a high-level simulation language Templar based on temporal logic. We also described an interpreter for the language that executes Templar programs.

Templar combines a large set of temporal logic operators and a rich set of high-level modeling primitives, such as events, activities, predicates, rules, hierarchical decomposition of activities, sequential and parallel activities, static and dynamic constraints, decisions, and data modeling abstractions of aggregation and generalization. As our experience with a real-world case study shows, this combination can help a programmer rapidly develop structured, reliable, and well-

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

maintainable simulation programs.

Acknowledgments

The author wishes to thank David Bodoff for numerous discussions of some of the issues in this paper and Ira Minsky for writing an interpreter for Templar.

References

[AV91] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.

[BFG+89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A framework for programming in temporal logic. In *Stepwise Refinement of Distributed Systems*, pages 94–129. Springer-Verlag, 1989. LNCS 430.

[BFK86] L. Brownston, R. Farrell, and E. Kant. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming*. Addison-Wesley, 1986.

[Bod92] D. Bodoff, November-December 1992. Personal communications.

[Bod93] D. Bodoff. Templar specification of the intelligent adversary system. Unpublished Manuscript, March 1993.

[DHR91] E. Dubois, J. Hagelstein, and A. Rifaut. A formal language for the requirements engineering of computer systems. In A. Thayse, editor, *From Natural Language Processing to Logic for Expert Systems*. John Wiley and Sons, 1991.

[dMS88] C. de Maindreville and E. Simon. Modelling non deterministic queries and updates in deductive databases. In *International Conference on Very Large Databases*, pages 395–406, 1988.

[EÖZ89] M.S. Elzas, T.I. Ören, and B.P. Zeigler, editors. *Modelling and Simulation Methodology: Knowledge Systems' Paradigms*. North-Holland, 1989.

[FG90] I. Futo and T. Gergely. *Artificial Intelligence in Simulation*. Ellis Horwood/Wiley, 1990.

[FM91] P. A. Fishwick and R. B. Modjeski, editors. *Knowledge-Based Simulation: Methodology and Application*, volume 4 of *Advances in Simulation*. Springer-Verlag, 1991.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

[FR82] M. S. Fox and Y. V. Reddy. Knowledge representation in organizational modeling and simulation: Definition and interpretation. In *Proceedings of the 13th Annual Pittsburgh Conference on Modeling and Simulation*, 1982.

[FS82] I. Futo and J. Szeredi. A discrete simulation system based on artificial intelligence methods. In A. Javor, editor, *Discrete Simulation and Related Fields*, pages 135–150. North-Holland, 1982.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *International Conference on Very Large Databases*, 1991.

[HSH89] A. G. Hofmann, G. M. Stanley, and L. B. Hawkinson. Object-oriented models and their application in real-time expert systems. In W. Webster, editor, *Simulation and AI*, volume 20. SCS Simulation Series, 1989.

[Int85a] IntelliCorp, Mountain View, Calif. *IntelliCorp. KEE Software Development System User's Manual*, 1985.

[Int85b] IntelliCorp, Mountain View, Calif. *The SIMKIT System: Knowledge-Based Simulation Tools in KEE*, 1985.

[IS89] Y.E. Ioannidis and T.K. Sellis. Conflict resolution of rules assigning values to virtual attributes. In *Proceedings of ACM SIGMOD Conference*, pages 205–214, 1989.

[Kam68] H. Kamp. *On the Tense Logic and the Theory of Order*. PhD thesis, UCLA, 1968.

[kbs90] Transactions of the Society for Computer Simulation, September 1990. Special Issue on Knowledge Based Simulation.

[KFM80] P. Klahr, W. S. Faught, and G. R. Martins. Rule-oriented simulation. In *Proceedings of 1980 IEEE International Conference on Cybernetics and Society*, pages 350–354, Cambridge, MA, 1980.

[Kro87] F. Kroger. *Temporal Logic of Programs*. Springer-Verlag, 1987. EATCS Monographs on Theoretical Computer Science.

[KT89] Z. M. Kedem and A. Tuzhilin. Relational database behavior: Utilizing relational discrete event systems and models. In *Proceedings of PODS Symposium*, pages 336–346, 1989.

[MD89] D. McCarthy and U. Dayal. The architecture of an active, object-oriented database system. In *Proceedings of ACM SIGMOD Conference*, 1989.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[Oll82] T. W. Olle. Comparative review of information systems design methodologies, stage 1: Taking stock. In T. W. Olle, H. G. Sol, and A. A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: A Comparative Review*, pages 1 – 14. North-Holland, 1982.

[RS89] A. Radiya and R. G. Sargent. ROBS: Rules and objects based simulation. In M. S. Elzas, T.I. Ören, and B.P. Zeigler, editors, *Modelling and Simulation Methodology: Knowledge Systems' Paradigms*, chapter III.4. North-Holland, 1989.

[SFBB86] N. Sathi, M. Fox, V. Baskaran, and J. Bouer. Simulation Craft: An artificial intelligence approach to the simulation life cycle. In *Proceedings of the SCS Summer Simulation Conference*, 1986.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, cashing and views in database systems. In *Proceedings of ACM SIGMOD Conference*, pages 281 – 290, 1990.

[TK91] A. Tuzhilin and Z. M. Kedem. Modeling dynamics of databases with relational discrete event systems and models. Working Paper IS-91-5, Stern School of Business, NYU, 1991.

[TL82] D. C. Tsichritzis and F. H. Lochovsky. *Data Models.* Prentice-Hall, 1982.

[tom92] ACM Transactions on Modeling and Computer Simulation, October 1992. Special Issue on AI and Simulations.

[Tuz92] A. Tuzhilin. SimTL: A simulation language based on temporal logic. *Transactions of the Society for Computer Simulation*, 9(2):87–100, 1992.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of ACM SIGMOD Conference*, pages 259 – 270, 1990.

[WLN89] L.E. Widman, K.A. Loparo, and N.R. Nielsen. *Artificial Intelligence, Simulation and Modeling.* Wiley, 1989.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-93-19