TEMPLAR: A KNOWLEDGE-BASED LANGUAGE FOR SOFTWARE SPECIFICATIONS USING TEMPORAL LOGIC

by

Alex Tuzhilin
Assistant Professor
Information Systems Department
Leonard N. Stern School of Business
New York University
New York, New York 10003

October 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-27

Templar: A Knowledge-Based Language for Software Specifications Using Temporal Logic

Alexander Tuzhilin

Information Systems Department
Stern School of Business
New York University *

Abstract

A software specification language Templar is defined. The language is based on temporal logic and on the Activity-Event-Condition-Activity model of a rule which is an extension of the Event-Condition-Activity model in active databases. The language supports a rich set of modeling primitives, including rules, procedures, temporal logic operators, events, activities, hierarchical decomposition of activities, and parallelism, combined together in a coherent system. The development of the language was guided by the following objectives: specifications written in Templar should be easy for the non-computer oriented users to understand, should have formal syntax and semantics, and it should be easy to map them into a broad range of design specifications.

1 Introduction

Since the introduction of the "waterfall" model of software development by Royce [Roy70] (Fig. 1), the model has been extended by various researchers [Dav90, HSE90, Tur87, You89]. In particular, the requirements stage of the model was divided into the substages of problem analysis and external behavior definition [Dav90].

In the problem analysis stage, a systems analyst builds a conceptual model of the real-world system that he or she plans to automate. Since this conceptual model emerges as an outcome of discussions with the users, the model must be expressed in the language the users can easily understand. Once the real-world system is understood by the system developer, a decision is required on which part of it, if any, should be automated, i.e. implemented in software. This decision establishes the boundary between the software system and the rest of the non-automated real-world system. In the next substage of the requirements stage, an external behavior of the

*Address: 40 West 4th Street, Room 624, New York, NY 10003; Internet: atuzhilin@stern.nyu.edu.

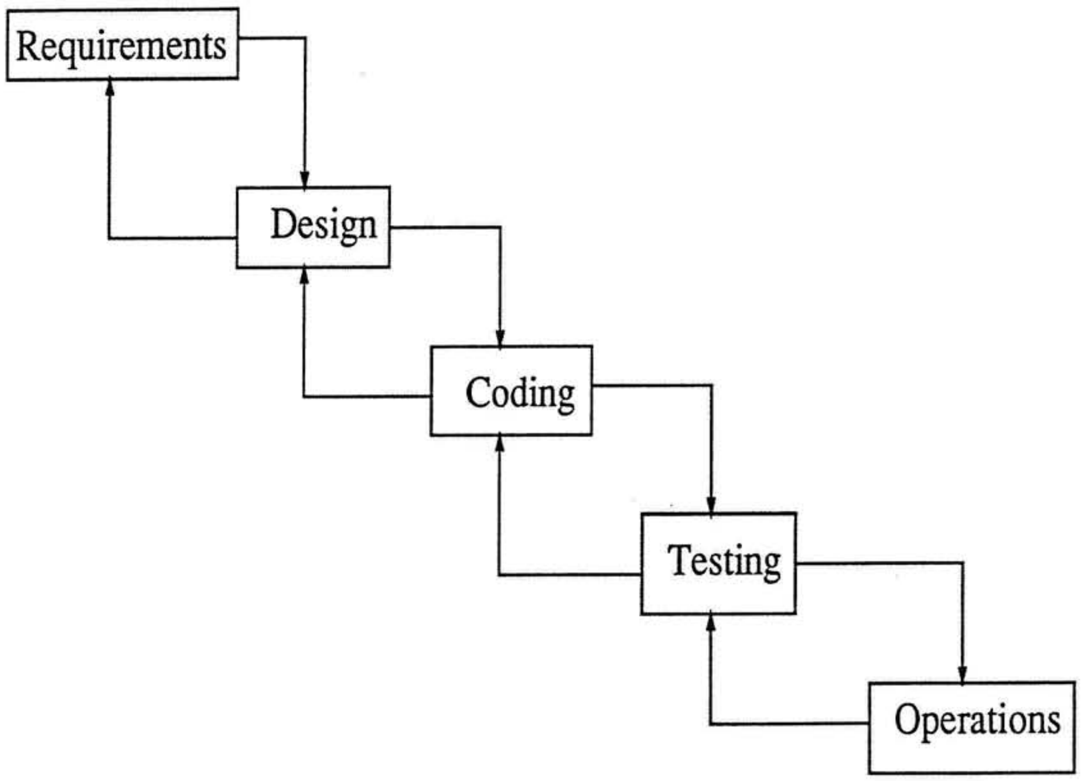

Figure 1: The "Waterfall" Model of the Software Development Life Cycle.

automated system is specified by writing software requirements specifications (SRS) [Dav90], which must also be written in a language that can be easily understood by the users.

In this paper, we propose a software specification language Templar[1]. We have developed the language for use primarily in the requirements specification stage of the life cycle, i.e. for describing a conceptual model of a system in the problem analysis substage [Dav90] and for writing software requirements specifications (SRS) based on this model. However, the language can also be used in the design stage of the life cycle for a certain class of applications that will be described in Section 3.9.

The major challenge in designing Templar is to have a language that "fits well" between the language of the users and the language of system designers as is shown in Fig. 2[2]. On one hand, it should be easy to translate the language of the users into Templar specifications. We believe that a natural language is the most convenient language for the non-computer oriented users. Therefore, we should make Templar specifications to be "close" to some restricted form of a natural language so that the non-computer oriented users could easily understand Templar specifications. This will allow to reduce possible misunderstandings between the systems analyst writing Templar specifications and the users.

On the other hand, it should be easy to translate Templar specifications into a broad range of existing software design methods[3]. This can be achieved by making Templar independent of various design specification languages. This will allow the systems developer to postpone the decision of choosing the data and process modeling paradigm until the design stage. This means that the developer has freedom to select those paradigms in the design stage that are the most suitable for the requirements specifications produced in the requirements stage. For example, assume that a requirements specification language has some elaborate extension of the entity-relationship model as part of its data model. Then it might be difficult to map requirements written in this language into some object-oriented design language in case it was decided that the object-oriented design is the most appropriate design method for the application.

As was stated before, we want to design Templar so that it is easy to translate Templar specifications into design specifications. Since many designers use formal methods to describe their designs, Templar should also be a formal specification method. Otherwise, there can be many

[1]Templar stands for *Temp*oral *l*ogic *a*s a *r*equirements specification language. Templar also means, according to the American Heritage Dictionary, "A knight of a religious military order founded at Jerusalem in the 12th century by the Crusaders."

[2]We use the term "language" in an extended sense as some method for communicating ideas. For example, we will think about diagrams and charts as some type of a language.

[3]Here and in the sequel, we mean by *translating* specification S_1 written in language L_1 to a specification written in language L_2 the process of producing specification S_2 in L_2 satisfying the conditions of specification S_1.

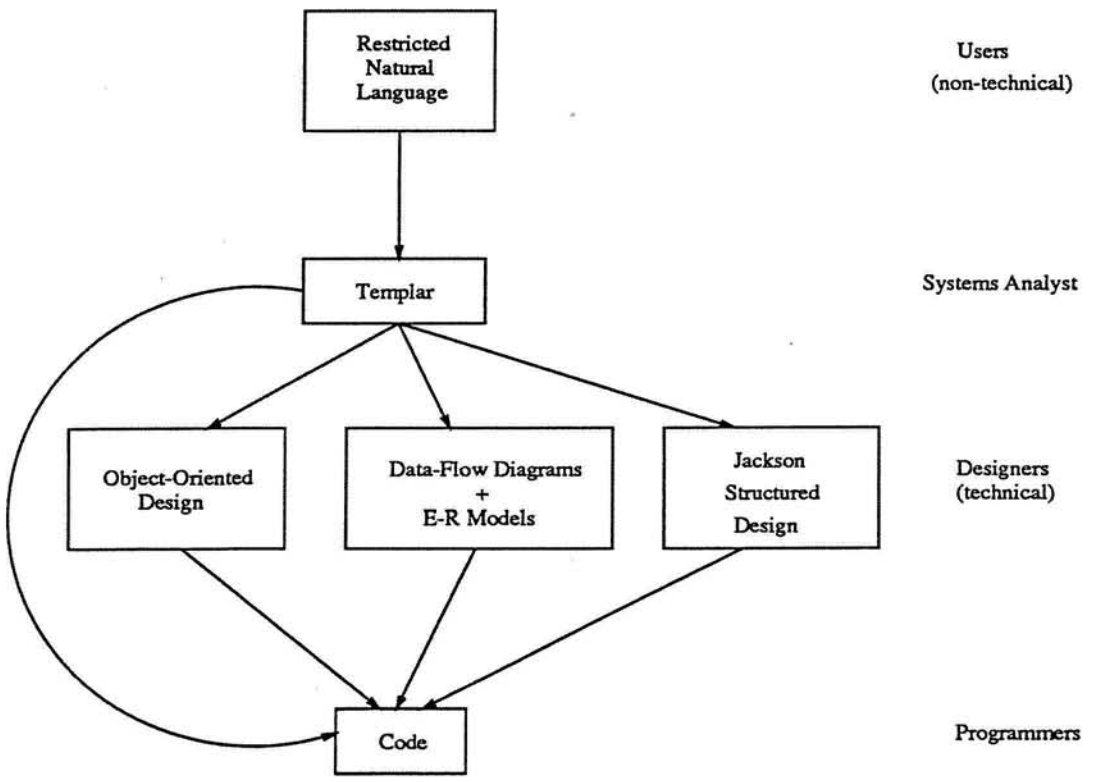

Figure 2: Relationship Between Templar and User and Design Specification Languages.

translation errors from informal requirements to formal design specifications.

To summarize, the development of Templar was guided by the following **design objectives**:

1. Templar specifications should be easily understood by non-computer oriented people, and the requirements specifications stated in some form of a restricted natural language should easily be translated into Templar specifications
2. Templar specifications should be equally easy to translate into a broad range of existing software design methods
3. Templar specifications should be rigorous.

A Templar specification consists of a set of rules and a set of activity specifications. It explicitly supports rules, events and activities, time, hierarchical decomposition of activities, sequential and parallel activities, static and dynamic constraints, and data modeling abstractions of aggregation, generalization, classification, and association [TL82, HK87]. To illustrate the use of Templar, consider the following rule:

If a customer comes to a branch of a bank after its closing time, 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)
after    closing_time(branch)
if       has_atm(branch)
then-do  use_atm(customer,branch)
```

This rule is interpreted as follows. When an event `arrives(customer,branch)` occurs, and if it occurs after the event `closing_time(branch)`, and if the condition `has_atm(branch)` holds then perform the activity `use_atm(customer,branch)`. This rule is based on the *Activity-Event-Condition-Activity (AECA)* model which is an extension of the Event-Condition-Activity (ECA) model[4] of rules in active databases [dMS88, MD89, WF90, SJGP90, sig89].

The rest of the paper is organized as following. Section 2 overviews the basics of temporal logic. Section 3 informally introduces Templar with examples illustrating its various features. Section 4 formally describes the language. Finally, Section 5 compares it with the existing specification languages.

[4]Most of the papers in active databases use the term *action*, and not activity. We will not distinguish between these two concepts and will use the terms "activity" and "action" interchangeably throughout this paper when we refer to a process that occurs over a period of time. In contrast to this, an event occurs instantaneously.

$\diamond A$:	is true now if A is true at some time in the future
$\square A$:	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 3: Operators of Temporal Logic

2 Overview of Temporal Logic

Since Templar is based on temporal logic and since we want the paper to be self-contained, we provide a brief overview of temporal logic in this section. The reader is referred to books by Kroger [Kro87] and Rescher and Urquhart [RU71] for a good introduction to the subject.

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** ($\square$), **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. The meaning of future operators is defined in Fig. 3. 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, when** [Kro87], and bounded necessity, **for_time (T)** ($\square_T$), and possibility, **within_time (T)** ($\diamond_T$), operators [Tuz91]. 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, while**, and **when** can be expressed in terms of the operators **until** and **since** [Kro87]. Furthermore, it easily follows from the completeness of temporal logic *US* [Kam68, Gab89], that the operators of bounded necessity and possibility can also be expressed in terms of **until** and **since** pair.

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.

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). A temporal structure is defined for *each* predicate P_i in the language as 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 structure for temporal logic predicates, we can extend this temporal structure to arbitrary temporal logic formulas in the standard inductive way [Kro87]. For example, we can define $K_t(A \textbf{ until } B)$ in terms of $K_t(A)$ and $K_t(B)$ as follows. $K_t(A \textbf{ until } B)$ 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. Similarly, informal definitions of temporal operators presented in Fig. 3 can be expressed in terms of temporal structures. Furthermore, temporal structures can be extended to arbitrary temporal logic formulas [Kro87]. For example, $K_t(A \wedge B) = K_t(A) \wedge K_t(B)$.

3 Overview of Templar

In this section, we introduce the language Templar by providing several examples of specifications written in it. In Section 4, we formally define the language.

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 specification of 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, WF90, SJGP90, sig89], and of rule-based design methodologies in Information Systems [MNP+91].

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.

Each clause deals with only one type of a modeling primitive: **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 specifications.

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 specification for the activity `review` can be stated in Templar as

```
activity review(paper,reviewer)
    read(paper,reviewer)
    evaluate(paper,reviewer)
end_activity
```

□

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 the structure of an

[5]When we define the syntax of Templar formally and introduce all the clauses in Section 4.1, we will explain how clauses correspond to modeling primitive in Figure 5.

activity specification in detail in Section 4.1.

An *atomic* activity cannot be divided into subactivities. It is defined with a *temporal predicate* describing how one of the relational predicates changes over time[6]. For example, consider the activity specification

```
activity read(paper,reviewer)
    T = reading_time(paper,reviewer)
    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,reviewer)
    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 specification method. If Templar specifications had only rules then they could contain hundreds of rules, and it would be difficult for the user (and often for the developer) to understand clearly how the rules interact. On the other hand, if Templar specifications consisted only of activities, then it could be difficult to describe the control logic with only the *if-then-else* statements for certain

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

applications. With Templar specifications, 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 4.2 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 **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, MNP+91] 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.

It is argued in [SJGP90] that an ECA model of a rule is a powerful model because it can support such diverse database concepts as views, special semantics for updating views, materialized views, partial views, procedures, special procedures, and cashing of procedures. Since ECA is a special type of the AECA model, this means that AECA is a very powerful model of a rule.

The general structure of a Templar rule will be defined in Section 4.

3.4 Procedural Specifications 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 specifications 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 specifications 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. Examples of these temporal operators are **next**, **sometimes_in_the_future**, and **always_in_the_future**.

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)
```

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 [Nic82] and dynamic [CF84, LS87, HS91] 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`.

To simplify the notation, some widely used static constraints can be expressed as macros by the system developer. For example, the last rule can be written as a macro

`paper, author` $\rightarrow$ `status`

for the predicate `submission`.

Other examples of macros the system developer may find useful are one-to-one, one-to-many, and many-to-many relationships, multivalued and inclusion dependencies [Ull88].

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 is accepted to a conference, it cannot be published elsewhere in the future.

can be expressed in Templar as

```
if     submission(paper,author,status) and status = accepted and
       publication ≠ this_conference
then   always_in_the_future not published(publication,paper,author)
```

where `this_conference` is a constant representing the conference being modeled.

3.7 Support for Data Modeling Abstractions and Data Model Independence

The data model in Templar is defined with a set of predicates. For instance, predicates `referees` and `located` from Example 2 constitute a part of the data model. These predicates appear in conditions, i.e. inside the **if** and **then** clauses, and also in atomic activities as was described in Section 3.2.

Furthermore, Templar has two interpreted predicates **isa** and **member-of** that define generalization and association data modeling abstractions [HK87]. For example, we can say **isa**`(invited_paper,paper)`, meaning that `invited_paper` is a special type of a paper. Also, **member-of**`(reviewer,Paper_reviewers)` means that `reviewer` belongs to the *set* of `Paper_reviewers`. These two predicates satisfy the following axiom:

if **member-of**`(B,A)` and `isa(C,B)` then `member-of(C,A)`

We decided to define generalization and association data modeling abstractions with predicates and not make it an integral part of the data model, as is done, for example, in [LMP+90], because it makes Templar data model to be *independent* of any existing data models, such as the entity-relationship model [Che76] and its extensions and various semantic data models [HK87]. This data model independence will allow the designers to map Templar specifications into a broad range of existing *design* data models, such as ER, semantic, and object-oriented data models. This means that the systems designer can postpone the decision about which data model to choose for an application until the subsequent design stage, and that he or she can select *any* design data model and easily map Templar specifications into that data model. For example, if we choose some semantic data model that supports hierarchies and, in particular, hierarchical paths, then the condition

```
member-of(paper,Session) and submission(paper,author,status) and
author='Jim' and status = 'accepted'
```

can easily be mapped into

```
Session.paper.author = 'Jim' and Session.paper.status = 'accepted'
```

Furthermore, if an object-oriented model is selected as a design model, then the same condition can be expressed in that model with methods simulating predicates. We will address this issue in Section 6 further.

This shows that the two predicates **isa** and **member-of** enhance data modeling capabilities of Templar and at the same time allow Templar specifications to be independent of any particular design data models.

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 and temporal precedence operators **before** and **after**.

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. Second, 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)
```

□

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.

3.9 Templar as a Design Language

We described Templar as a requirements specification language so far. However, Templar can also be used in the design stage of the software life cycle for certain applications because it has a formally defined semantics (to be defined in Section 4.3) and because it supports *decomposition* of activities into subactivities which is the primary activity during the design stage of an information system.

Templar is especially useful as a design language for those applications in which data is stored in an active database [dMS88, MD89, WF90, SJGP90, sig89] in the implemented system. For example, McCarthy and Dayal [MD89] describe how a stock trading application can be modeled with active databases. Since the rule structure of Templar subsumes the ECA rule structure of active databases, it is clear that Templar is suitable for the *design* of the applications that have data to be stored in an active database.

In this section, we provided an informal overview of the language Templar. In the next section, we formally introduce the syntax of the language and define its semantics.

4 Formal Description of Templar

In this section, we formally define the specification language Templar. Section 4.1 presents the syntax of the language, Section 4.2 describes a small case study illustrating the usage of Templar specifications and Section 4.3 presents its semantics.

4.1 Syntax of Templar

Templar specifications consist of a set of rules and a set of activity specifications. The syntax of a Templar rule is defined with the BNF grammar presented in Fig. 4 (we assume that *name* and *string* are sequences of characters in the figure).

As Fig. 4 shows, a Templar rule consists of a collection of clauses that are divided into body and rule clauses. 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. 5 shows the relationship between clauses and activities, events, and conditions.

Furthermore, a 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 Fig. 4. For example, the user can define operators **until, since, 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. 6. As Fig. 6 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"). Examples of the *for-statement* will be presented in Section 4.2. *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 convenience for the user.

```
rule                    ::=  [body-of-rule] head-of-rule
head-of-rule            ::=  then-clause | do-clause | then-clause do-clause
then-clause             ::=  then conditions
do-clause               ::=  then-do activity { next-activity }
next-activity           ::=  ; activity | || activity
body-of-rule            ::=  { body-clause }
body-clause             ::=  if conditions
                          |  while activities
                          |  when events
                          |  before activities
                          |  before events
                          |  after activities
                          |  after events
                          |  user-defined-operator activities
                          |  user-defined-operator events
user-defined-operator   ::=  string
activities              ::=  activity { logical-op activity }
conditions              ::=  condition { logical-op condition }
events                  ::=  event { logical-op event }
activity                ::=  name ( arguments )
condition               ::=  [not] temporal-predicate
temporal-predicate      ::=  [unary-temp-operator] predicate
                          |  predicate binary-temp-operator predicate
predicate               ::=  name ( arguments ) | isa | member-of | var relop var
event                   ::=  begin-activity | end-activity | temporal-event | external-event
begin-activity          ::=  begin.activity
end-activity            ::=  end.activity
external-event          ::=  name ( arguments )
temporal-event          ::=  temporal-constant | periodic-event
periodic-event          ::=  every period
period                  ::=  hour | day | week | month | year | day-of-week
day-of-week             ::=  Monday | Tuesday | Wednesday | Thursday
                          |  Friday | Saturday | Sunday
temporal-constant       ::=  name
logical-op              ::=  and | or
relop                   ::=  = | ≠ | < | ≤ | > | ≥
unary-temp-operator     ::=  always_in_the_future | sometimes_in_the_future
                          |  always_in_the_past | sometimes_in_the_past | next
                          |  for_time name | within_time name | user-defined-operator
binary-temp-operator    ::=  user-defined-operator
arguments               ::=  name {, name }
var                     ::=  name
```

Figure 4: Syntactic Definition of a Rule.

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

Figure 5: 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
if-statement	::=	**if** condition **then** statement-list **else** statement-list **end_if**
for-statement	::=	**foreach** argument **suchthat** condition **do** statement-list **end_for**
parallel-statement	::=	statement-list || statement-list
composite-activity	::=	name (parameters)
atomic-activity	::=	condition
condition	::=	same as *condition* in Fig. 4
argument	::=	name
parameters	::=	[name {, name }]

Figure 6: Syntactic Definition of Activity Specification .

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 condition, i.e. as a 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.

4.2 Case Study

In this section, we describe a fragment of a *Conference* information system modeling an IFIP Working Conference using Templar specification language. Specifically, we describe the activities of the program committee from the time papers are submitted to the program committee chairperson until the time the acceptance/rejection decisions are made and the conference program is formed. We will use both English and Templar to describe these activities.

The state of the conference system is described with temporal predicates:

`located(paper,person)`:

a paper is located with a person

`submission(paper,author,status)`:

an author submitted a paper to the conference, and the status of the paper is determined by the attribute "status"

`referees(paper,reviewer)`:

a reviewer referees a paper

In addition, the conference system supports the following activities:

`register_paper(paper,author)`:

a paper submitted by an author is registered for the conference

`distribute_paper_to_reviewers(paper,chairperson)`:

a paper is distributed to reviewers by the chairperson

`send_acknowledgement(chairperson,paper,author)`:

an acknowledgment letter is sent by the chairperson to the author that the paper was received `select_reviewers(paper,chairperson):`

the chairperson selects reviewers for a paper

`send(paper,person1,person2):`

a paper is sent from person1 to person2

`review(paper,reviewer):`

a reviewer reviews a paper;

`record_evaluations(paper,referee):`

the chairperson records the evaluation of a paper by a referee

`make_acceptance_decisions:`

decide which papers should be accepted and which rejected

`send_acceptance_letter(author,paper):`

send a letter to an author stating that the paper was accepted

`send_rejection_letter(author,paper):`

send a letter to an author stating that the paper was rejected

`form_conference_program:`

group selected papers into sessions for presentation and select the chairperson for each session

Each activity gives rise to events "the activity begins" and "the activity ends." Besides that, the conference system has an external event

`receives(chairperson,paper,author):`

the chairperson receives a paper submitted by an author to the conference.

Furthermore, we define two temporal constants

`submission_deadline:`

the paper submission deadline (e.g. 1/1/2000)

`pc_meeting_date:`

the date of the program committee meeting (e.g. 3/23/2000)

In the next section, we present rules specifying a fragment of the conference information system informally described at the beginning of this section. In Section 4.2.2, we provide specifications of some of the activities described above.

4.2.1 Rules:

R1: When the program committee chair receives a paper before the submission deadline, the chair registers the paper, sends it to the referees and sends the acknowledgment letter to the author.

```
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))
```

This rule says that the chairperson first registers the paper and then distributes it among the referees and *simultaneously* sends the acknowledgment to the author. `Register_paper` is defined as activity **A3** and `distribute_paper_to_reviewers` as activity **A1** below.

R2: When a reviewer receives a paper, which he/she is supposed to referee, the reviewer evaluates it and sends the results back to the chairperson.

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

R3: When the program committee chair receives the evaluation of a paper from a referee he/she records the evaluations.

```
when      end.send(paper,reviewer,chairperson)
then-do   record_evaluation(paper,reviewer)
```

R4: On the date when the program committee meets, it decides which papers should be accepted for the conference.

```
when      pc_meeting_date
then-do   make_acceptance_decisions
```

R5: After the program committee meeting, the acceptance/rejection letters are sent to the authors.

```
after     pc_meeting_date
if        submission(paper,author,status) and status = accepted
then-do   send_acceptance_letter(author,paper)
```

```
after     pc_meeting_date
if        submission(paper,author,status) and status = rejected
then-do   send_rejection_letter(author,paper)
```

R6: After the program committee meeting, the conference program is formed (selected papers are grouped into sessions for presentation, and session chairs are selected for each session).

```
after     pc_meeting_date
then-do   form_conference_program
```

4.2.2 Activities

In this section, we provide specifications only for several most interesting activities in order to keep the example manageable.

A1:
```
activity distribute_paper_to_reviewers(paper,chairperson)
    select_reviewers(paper,chairperson);
    foreach reviewer suchthat referees(paper,reviewer) do
        send(paper,chairperson,reviewer)
    end_for
end_activity
```

where predicate `referees` determines which reviewers review which papers and is determined as part of the activity `select_reviewers`.

Activity **A1** provides an example of *for-statement.* The main purpose of this statement is to execute a group of activities within the body of the statement in parallel. In our example, the paper is sent by the chairperson to each of the reviewers simultaneously.

Also, notice the relationship between rules **R1**, **R2** and activity **A1**: rule **R1** initiates activity **A1**, and then the end of activity `send` within **A1** initiates rule **R2**.

```
A2: activity send(what,from,to)
        T = transfer_time(what,from,to)
        (next not located(what,from) || transfer(what,to) for_time T)
    end_activity
```

Activity send makes predicate located false at the next time moment, determines time it takes to transfer what from from to to, and does the transfer during that time (predicate transfer is set to be true during that time). Note that the transfer is initiated immediately when the activity send starts. Therefore, the two atomic activities happen in parallel in send.

```
A3: activity register_paper(paper,author)
        next submission(paper,author,status) and status = being_reviewed
    end_activity
```

```
A4: activity form_conference_program(accepted_papers)
        decide_which_sessions_to_have(Sessions)
        group_papers_in_sessions(accepted_papers,Sessions)
        foreach session suchthat member-of(session,Sessions) do
            select_session_chair(session)
    end_activity
```

Sessions is the list of sessions to be held at the conference; it is an output parameter for the activity decide_which_sessions_to_have.

4.3 Semantics of Templar

The meaning of a Templar specification will be based on the concept of a Discrete Event System [VK87, KT89] and will be associated with *all* the possible sequences of events *consistent* with the specification[7].

To illustrate the notion of consistency of a sequence of events, consider the specification of a working conference as defined in Section 4.2. The sequence of events in Fig. 7 is consistent with the specification presented in Section 4.2. To the contrary, the sequence of events in Fig. 8 is not

[7]Traditionally in temporal logic programming, the meaning of a program is associated with a sequence of *predicates* consistent with that program, i.e. a sequence of predicates that makes all the rules to be true at all the moments of time [AM89, Gab89]. However, we will associate the meaning of a Templar specification with sequences of events and not predicates because we believe that users are more interested in "what is happening" with a real-world system (i.e. interested in the events that occur in such system) rather than interested in the states the system takes over time. We believe that the two approaches are "equivalent" in the sense that knowing a sequence of events, it is possible to construct sequences of states "corresponding" to these events and vice versa. However, the study of this issue is beyond the scope of this paper.

1) a paper is submitted to the conference;
2) the conference program chair registers the paper;
3) the paper is sent to reviewers for evaluations and *simultaneously* an acknowledgment letter is sent to the author;
4) a reviewer receives a paper;
5) a reviewer finishes the evaluation of a paper;
6) a reviewed paper is sent back to the program chair;
7) the program chair records the reviewer's evaluation;
8) the program committee meets to make acceptance/rejection decisions;
9) acceptance/rejection letters are sent;
10) conference program is formed.

Figure 7: Sequence of Events Consistent with the Specification.

1) a paper is submitted to the conference;
2) the conference program chair registers the paper;
3) a reviewer finishes the evaluation of a paper;
4) the paper is sent to reviewers for evaluations and *simultaneously* an acknowledgment letter is sent to the author;
5) a reviewer receives a paper;
6) a reviewed paper is sent back to the program chair;
7) the program chair records the reviewer's evaluation;
8) the program committee meets to make acceptance/rejection decisions;
9) acceptance/rejection letters are sent;
10) conference program is formed.

Figure 8: Sequence of Events Inconsistent with the Specification.

consistent with the specification: according to the specification, the reviewer has to receive the paper to be refereed (Step 5) *before* he or she finishes the evaluation of the paper (Step 3), whereas Fig. 8 states the opposite.

To define consistency, we have to determine if a rule is true at time t for a given sequence of events. If all the rules in a specification are true at all the times t for a certain sequence of events, then such a sequence of events is *consistent* with the given specification. To determine if a rule is true at time t for some sequence of events, the body of the rule is matched against the sequence of events. As a result of this matching process, the activities in the **then-do** clause must be true and must occur in a certain sequence. If the events marking beginnings and ends of these activities do not contradict the events in the sequence then such a rule is true at time t. For example, if rule **R1** matches the given sequence of events (a paper is received before the submission deadline), then the activity `register_paper(paper,author)` must begin immediately,

Activity A	Relationship Between Events
A_1; A_2	begin.A = begin.A_1; end.A = end.A_2; end.A_1 = begin.A_2
$A_1 \parallel A_2$	begin.A = begin.A_1 = begin.A_2; end.A = max { end.A_1, end.A_2 }
foreach x **suchthat member-of**(x, S) **do** $B(x)$ **end_for**	begin.A = begin.$B(x)$; end.A = max { end.$B(x)$ | $S(x)$ }
if c **then** A_1 **else** A_2 **end_if**	if c is true then begin.A = begin.A_1 and end.A = end.A_1; otherwise, begin.A = begin.A_2 and end.A = end.A_2

Figure 9: Definition of Beginnings and Ends of Activities in Terms of Beginnings and Ends of Subactivities.

and when it is finished, the two activities `distribute_paper_to_reviewers(paper,chairperson)` and `send_acknowledgement(chairperson,paper,author)` must begin immediately after that in parallel. Since each activity defines events corresponding to the beginning and the end of the activity, rule **R1** asserts that six new events (two for each of the three activities) must occur in a certain order. If these six events do not contradict the existing sequence of events then rule **R1** is true at the time its evaluation takes place. As another example, rule **R2** says that Event 3 must occur after Event 5 in Fig. 8. Since a paper can be reviewed only once, the sequence of events in Fig. 8 is not consistent with the specification.

The notion of consistency, as defined above, resembles the concept of *satisfiability* of dynamic integrity constraints [CF84, LS87, HS91] by a sequence of database states and satisfiability of a set of temporal logic formulas by a temporal structure [Kro87].

Since activities consist of subactivities, we have to relate beginnings and ends of activities with its subactivities. For example, it is important to know how the events `begin.distribute_paper_to_reviewers(paper,chairperson)` and `end.distribute_paper_to_reviewers(paper,chairperson)` are related to various subactivities of the activity `distribute_paper_to_reviewers`. This relationship can be defined recursively as follows. If activity A consists of subactivities A_1 and A_2 then relationships between events begin.A, end.A and events begin.A_1, end.A_1, begin.A_2, end.A_2 is defined for different statements as in Fig. 9:

The definition of begin.A and end.A for the **foreach** statement, as presented in the table above, is motivated by the fact that activities $B(x)$ in that statement occur *in parallel* for all values of x. Therefore, the definition of end.A for the **foreach** statement follows from the definition of end.A for the parallel operator ($||$).

5 Related Work

There have been many IS specification methods proposed in the literature. Books by Davis [Dav90], Yourdon [You89], Olle et al [OHM$^+$88], Rumbaugh et al [RBPE91] describe some of these methods. A variety of different specification methods exists because different applications, or even different parts of the same application, can best be specified with different methods [Dav90].

As was stated in the introduction, the major design objectives for the development of Templar were 1) its closeness to natural language specifications, 2) independence of specific design data models and 3) existence of a formal semantics. Since in this paper we are interested in the knowledge-based methods describing evolution of information systems in time, we will compare our work to the existing specification methods dealing with rules and with time, such as RML [BGM85], Telos [MBJK90], Tempora [LMP$^+$90], ERAE [DHL$^+$86], and RDL [GHH91][8]. In particular, we are interested to know how well each of these languages satisfies the design objectives stated above.

RML [BGM85] is a requirements specification language based on the object-oriented framework with some support for time. Telos [MBJK90] is an extension of RML that supports deductive rules and data modeling abstractions of aggregation, classification and generalization in addition to the object-oriented framework and time. All these features make Telos a powerful requirements specification language. However, Telos does not satisfy our second objective: it depends heavily on the object-oriented framework, and the data modeling abstractions (aggregation, classification and generalization) constitute an integral part of Telos data model. This means that software designers would find it difficult not to use these paradigms in the life cycle stages following the software requirements specifications stage. For example, it may be difficult to map Telos specifications into some conventional programming language (e.g. Pascal or C) coupled with a relational database in case there is a need to do so.

Furthermore, Telos and Templar rules have a different structure. The rules in Telos have the **if-then** structure and are based on many-sorted first-order logic. Since Templar rules are based on temporal logic and since temporal logic with **until** and **since** operators has the same expressive power as first-order logic with explicit references to time for the discrete bounded model of time [Kam68, Gab89], this means that Templar rules are as expressive as Telos rules. Furthermore,

[8]We do not make any claims about the completeness of this list.

Templar rules support various temporal clauses in addition to **if** and **then** clauses of Telos and have a richer rule structure based on a powerful AECA model.

Tempora [LMP$^+$90] is another specification language supporting time, complex objects, an extended entity-relationship data model, and deductive rules. As Telos, it also represents a rich modeling language. However, it does not satisfy our second objective for the same reason as Telos does not: it depends on the specific design specification methods, i.e. complex objects and entity-relationship diagrams.

The rule structure of Tempora is based on ECA model [MNP$^+$91] and on temporal logic and is closer to the rule structure of Templar than that of Telos. However, Tempora distinguishes only between events and conditions, and does not treat activities on the equal footing with events. For example, it does not allow activities in the antecedent part of the rule (e.g. in the **while** clause).

ERAE is still another specification language supporting time, entities and relationships among them, events, deductive reasoning system based on first-order logic, and some data modeling abstractions, such as association (*is-in* predicate) [DHL$^+$86]. It can support a broader range of design methods than Telos and Tempora because it is less dependent on specific modeling constructs, such as complex objects and ERT diagrams of Tempora and object-oriented features of Telos. For example, association is modeled with predicate *is-in*, and is not built into the data model, as is done in Telos. However, the rule structure of ERAE is based on the **if-then** model, as in Telos, and does not support the AECA rule model and temporal logic operators in rules.

Finally, RDL [GHH91] is a specification language for the requirements and design of time-dependent systems based on the intuitionistic temporal logic. RDL specifications consists of a set of rules of the form

```
antecedent about the past → consequent about the future
```

RDL satisfies our second and third language design objectives: it is a rigorous and very general specification language and, as a result of this, its specifications can be easily mapped into most of the design specification languages and also can be formally verified. However, it may be difficult to map user requirements specifications, written in natural language, into RDL specifications; or alternatively, it may be difficult for a non-computer oriented user to understand RDL specifications. The reason for that is that RDL does not support some of the features that make a specification language "close" to a natural language, such as an explicit support for events and activities, hierarchical decomposition of activities, and the support for the parallel and sequential composition of activities in the sense defined in this paper (although the authors consider some of these issues as a topic of future research).

Templar is also related to the work in the database community on active databases [dMS88, MD89, WF90, SJGP90, sig89]. As was stated before, active databases integrate rules and databases by providing an ECA model of a rule. However, active databases do not incorporate time in the rules. Furthermore, they do not fully distinguish between events and activities (because they do not support time), and their clause structure is limited only to three clauses **if**, **then**, and **when**.

As a conclusion, none of the software specification methods considered in this section satisfies *all* the three design goals: closeness to natural language specifications, independence of specific design data models and existence of a formal semantics. Furthermore, the rule structures of these methods are not as universal and powerful as the AECA rule model of Templar.

6 Conclusion and Future Work

In this paper, we defined the syntax and the semantics of a software specification language Templar. The language is based on the Activity-Event-Condition-Activity (AECA) model that supports rules, temporal logic, and such modeling primitives as events, conditions, and activities. Furthermore, Templar supports procedures, hierarchical decomposition of activities, and parallelism.

Templar has the following properties desirable in a software requirements specification language. First, Templar specifications follow closely user requirements specifications written in natural language. As was illustrated in the case study in Section 4.2, user requirements specifications are translated sentence by sentence into Templar specifications. Since Templar is so close to the language of the users, developers can translate informal user specifications into formal software specifications with the minimal amount of errors.

Second, Templar requirements specifications can easily be translated into a broad range of design specifications. This allows the software developers to not have to be concerned about appropriateness of different data and process modeling paradigms for an application in the requirements specification stage. The decision which modeling paradigm to choose can be postponed until the design stage and can be based on the specifications produced in the requirements stage.

Third, Templar has a formally defined syntax and semantics. Therefore, Templar specifications can be mapped into design specifications so that it may even be possible to verify formally that the design specifications satisfy the requirements specifications.

Since Templar satisfies the three properties described above and since these properties are desirable in a software requirements specification language, Templar will primarily be used as a requirements specification language. However, Templar can also be used as a design specification language because it has formal semantics and because it supports the process of decomposition of activities into subactivities.

As a future work, we plan to study how Templar specifications can be mapped into different design specifications including object-oriented design specifications and active databases. In mapping Templar specifications into object-oriented specifications, activities can be converted into methods, events into some boolean flags (modeled as objects) indicating whether or not activities have begun or finished, and temporal predicates into special types of methods simulating these predicates (i.e. such a method should specify when a predicate is true and when it is false). Furthermore, such a mapping can support time by some object "time manager" that can schedule various activities and events. However, this mapping has the following problem requiring some additional considerations. If a certain activity is based on several objects, it is not clear with which object it has to be associated. For example, the activity `send(paper,chairperson,reviewer)` can be mapped into the method `send1(chairperson,reviewer)` associated with the object `paper`, or into the method `send2(paper,chairperson)` associated with the object `reviewer`, or into the method `send3(paper,reviewer)` associated with the object `chairperson`, or into all three methods. Alternatively, a new class can be created that has `paper`, `reviewer` and `chairperson` as its subclasses, and the method `send` can be associated with this class.

Another topic of interest is the mapping of Templar specifications into active databases that also support user-defined procedures. Examples of such systems include HiPAC [MD89], STARBURST [HCL+90], POSTGRES [SJGP90], and Ode [GJ91]. Since active databases are based on the Event-Condition-Action model and Templar specifications on the Action-Event-Condition-Action model, the two models are sufficiently similar. The main problem in this mapping would be to determine how to incorporate time in active databases.

Acknowledgments

The author wishes to thank Jon Turner and P.R. Balasubramanian for several valuable comments on earlier drafts of the paper and Al Zimmerman for the discussions of some of the issues in this paper.

References

[AM89] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.

[BGM85] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specifications. *IEEE Computer*, pages 82 – 91, April 1985.

[CF84] M. A. Casanova and A. L. Furtado. On the description of database transition constraints using temporal languages. In *Advances in Database Theory*, pages 211–236. Plenum Press, 1984. vol. 2.

[Che76] P.P. Chen. The entity-relationship model: Towards a unified view of data. *TODS*, 1(1):9–36, 1976.

[Dav90] A. M. Davis. *Software Requirements: Analysis and Specification.* Prentice Hall, 1990.

[DHL$^+$86] E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert, and A. Rifaut. A knowledge representation language for requirements engineering. *Proceedings of the IEEE*, 74(10):1431–1444, 1986.

[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.

[Gab89] D. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification*, pages 402–450. Springer-Verlag, 1989. LNCS 398.

[GHH91] D. Gabbay, I. Hodkinson, and A. Hunter. Using the temporal logic RDL for design specifications. In *Concurrency: Theory, Language, and Architecture*, pages 64 – 78. Springer-Verlag, 1991. LNCS 491.

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

[HCL$^+$90] L. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, 1990.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[HS91] K. Hulsmann and G. Saake. Theoretical foundations of handling large substitution sets in temporal integrity monitoring. *Acta Informatica*, 28(4), 1991.

[HSE90] B. Henderson-Sellers and J. M. Edwards. The object-oriented systems life cycle. *Communications of the ACM*, 33(9):143 – 159, 1990.

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

[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*, 1989.

[LMP+90] P. Loucopoulos, P. McBrien, U. Persson, F. Schumacker, and P. Vasey. TEMPORA - integrating database technology, rule based systems and temporal reasoning for effective software. In *Esprit'90 Conference Proceedings.* Kluwer Academic Publishers, 1990.

[LS87] U. W. Lipeck and G. Saake. Monitoring dynamic integrity constraints based on temporal logic. *Information Systems*, 12(3):255–269, 1987.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325 – 362, 1990.

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

[MNP+91] P. McBrien, M. Niezette, D. Pantazis, A. H. Seltveit, U. Sundin, B. Theodoulidis, G. Tziallas, and R. Wohed. A rule language to capture and model business policy specifications. In *Proceedings of the Third Conference on Advanced Information Systems Engineering*, Trondheim, Norway, May 1991.

[Nic82] J.-M. Nicolas. Logic for impoving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

[OHM+88] T. W. Olle, J. Hagelstein, I. G. MacDonald, C. Rolland, H. G. Sol, F. J. M. Van Assche, and A. A. Verrijn-Stuart. *Information Systems Methodologies: A Framework for Understanding.* Addison-Wesley, 1988.

[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.

[RBPE91] J. Rumbaugh, M. Blaha, W. Premerlani, and F. Eddy. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[Roy70] W. Royce. Managing the development of large software systems: Concepts and techniques. In *IEEE WESCON*, pages 1–9, 1970. Reprinted in Nineth IEEE International Conference on Software Engineering, 1987.

[RU71] N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, 1971.

[sig89] SIGMOD Record, September 1989. Special issue on rule management and processing in expert database systems.

[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.

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

[Tur87] J. Turner. Understanding the elements of system design. In R. J. Boland and R. A. Hirschheim, editors, *Critical Issues in Infirmation Systems Research*, pages 97–111. John Wiley, 1987.

[Tuz91] A. Tuzhilin. Temporal logic as a simulation language. In *Proceedings of the International Conference on Artificial Intelligence and Simulation*, New Orleans, Louisiana, April 1991.

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

[VK87] P. Varaiya and A.B. Kurzhanski, editors. *Discrete Event Systems: Models and Applications.* Springer-Verlag, 1987. Lecture Notes in Control and Information Sciences, 103.

[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.

[You89] E. Yourdon. *Modern Structured Analysis.* Yourdon Press, 1989.