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

ALEXANDER TUZHILIN
New York University

A software specification language Templar is defined in this article. The development of the language was guided by the following objectives: requirements specifications written in Templar should have a clear syntax and formal semantics, should be easy for a systems analyst to develop and for an end-user to understand, and it should be easy to map them into a broad range of design specifications. Templar 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, parallelism, and decisions combined together into a cohesive system.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; *methodologies*; D.2.10 [**Software Engineering**]: Design—*methodologies*; *representation*; H.1.10 [**Models and Principles**]: General; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*representation languages*; *temporal logic*

General Terms: Design, Languages

Additional Key Words and Phrases: Activities, events, rule-based systems, specification languages, temporal logic, time

1. INTRODUCTION

In one of the first steps in the systems development life cycle the systems analyst (SA) interviews the end-user in order to understand how the real-world system to be automated works. Typically, the end-user describes such a system in a natural language. Usually, these descriptions tend to be imprecise, incomplete, and even inconsistent. Therefore, the job of the SA is to understand what the end-user says and to help him or her clarify the

Author's address: Information Systems Department, Stern School of Business, New York University, 44 West 4th Street, New York, NY 10012; email: atuzhilin@stern.nyu.edu.

description of the system. The process of interaction with the end-user consists of the following steps [Dubois et al. 1991]:

—*Elicitation.* In this step the SA collects information about the end-user problems in the form of informal descriptions of the system, often expressed in a natural language.
—*Modeling.* In this step, the SA takes the informal descriptions of the system obtained from the end-user in the previous step and builds a *conceptual model* of the system. This model should "match" the end-user descriptions obtained in the elicitation step.
—*Analysis.* In this step, the SA detects problems in the model developed in the previous step, such as omissions and inconsistencies.
—*Validation.* In this step, the SA resolves the end-user problems detected in the previous step. The analyst also presents to the end-user the model developed in the modeling step to make sure that there are no misunderstandings between the analyst and the end-user regarding the model. If the end-user approves the description of the real-world system presented by the analyst, then the model is complete (we use the term "completeness" in an informal sense here). Otherwise, the SA has to adjust the conceptual model, and the process of interaction between the SA and the end-user enters a new cycle.

These steps are repeatedly applied one after another, starting with the Elicitation step, in the order shown with solid lines in Figure 1. This means that the SA gets the feedback from the end-user *only* in the validation step. In order to facilitate the process of faster development of a conceptual model that matches the end-user needs, it is important to get the feedback from the end-user *as early as possible* in the model development loop in Figure 1. To emphasize this closer interaction between the end-user and the SA in the conceptual model development process, we added two dashed lines to Figure 1. The arrow from Modeling to Elicitation in Figure 1 means that the SA develops parts of the model of the system during the interviewing process and asks the end-user questions based on the partial model developed so far. The arrow from Modeling to Validation in Figure 1 means that the SA explains (and even shows in some cases) the partial model of the system to the end-user and gets the end-user feedback in an interactive fashion.

To achieve this mode of closer interaction between the SA and the end-user, the modeling language that the SA uses should satisfy the following requirements.

(1) The language should be "powerful" and *specifier friendly* so that the SA can develop conceptual models quickly (ideally, during the interviewing process or shortly after it).
(2) The language should be *end-user friendly* so that both the SA can show specifications written in this language to the end-user and the end-user can understand them with minimal help from the SA.

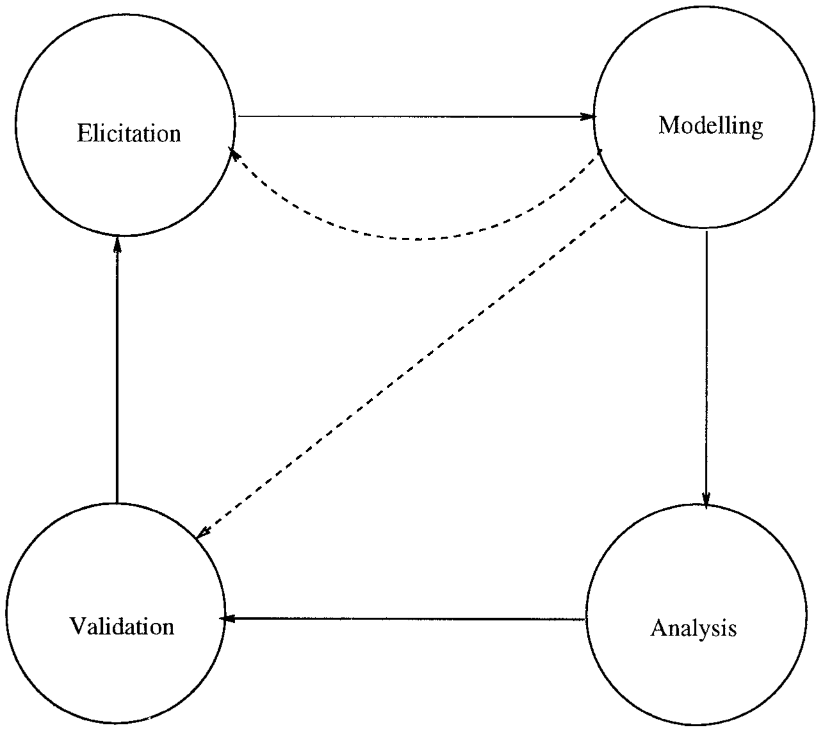

Fig. 1. The model of interactions between the end-user and the systems analyst.

These two requirements will allow the SA to develop conceptual models quickly and explain them to the end-user with fewer problems.

After a conceptual model is developed, and it is understood which part of the system has to be automated [Davis 1990], the system development life cycle proceeds to the design stage. It is generally not clear until the design stage which design specification language is better suited for design specifications. Therefore, the requirements specification language should satisfy the following condition:

(3) The language should be *independent* of specific design specification languages, and it should be equally easy to map specifications written in this language into a broad range of design specification languages. For example, it should be equally easy to map requirements specifications into object-oriented design specifications (e.g., TaxisDL [Borgida et al. 1993]), as well as into set-theoretic specifications language (e.g., Z language [Spivey 1988]), or into a wide-spectrum specifications (e.g., V language [Smith et al. 1985]). This will allow the systems developer to postpone the decision of choosing the design specification language until the design stage.

It is also important that the requirements specification language has a formal semantics because we want these specifications to be formally validated and because it makes it easier to map them into formal design specifications. Therefore, our next requirement states that

(4) The language should have a formally defined semantics.

We propose a specification language Templar that satisfies the four conditions stated above. 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 [Davis 1990] 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 3.10.

A Templar specification consists of a set of rules and a set of activity specifications. It explicitly supports rules, events and activities, time and temporal logic, hierarchical decomposition of activities, sequential and parallel activities, static and dynamic constraints, decisions, data-modeling abstractions of aggregation and generalization [Tsichritzis and Lochovsky 1982], and user-defined modeling constructs. To illustrate the use of Templar, we 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).

Although Templar is a general-purpose specification language, it is especially well suited for the specifications of systems changing over time because the language is based on temporal logic and has extensive features supporting time.

Requirements specification languages for systems evolving over time were developed before by other researchers. In the next section we present the previous work on the subject and describe how this work is related to our language design goals presented above. The rest of the article is organized as follows. In Section 3 we present Templar in an informal way through the series of examples. In Section 4 we describe formally the syntax and the semantics of the language. In Section 5, we present two case studies of using Templar for the specifications of "real-world" systems. Finally, in Section 6 we describe some techniques for validating Templar specifications.

2. RELATED WORK

There have been many requirements specification languages proposed in the literature. Since we are especially interested in the specifications of systems changing over time and in how rules can be used in such specifications, we will primarily consider those specification languages that support time and rules, such as RML [Borgida et al. 1985; Greenspan 1984], Telos [Mylopoulos et al. 1990], Tempora [Loucopoulos et al. 1990], ERAE [Dubois et al. 1991], TRIO [Ghezzi et al. 1990], INFOLOG [Fiadeiro and Sernadas 1986], MAL [Jeremaes et al. 1986], and RDL [Gabbay et al. 1991]. In particular, we want to know how well each of these languages satisfies the four design objectives stated in the introduction—i.e., are they end-user friendly; are they formal; and can they be easily mapped into a broad range of design specification languages?

ERAE is a requirements specification language based on multisorted temporal logic supporting events, partial functions, metric temporal operators, and specification-structuring mechanisms such as *contexts* [Dubois et al. 1991]. ERAE satisfies our objective (4) since it has a rigorously defined semantics. It also satisfies objective (3): ERAE specifications can be mapped without significant problems into a broad range of design specification languages because it has general-purpose modeling primitives such as predicates and events. For example, if we want to write design specifications in TaxisDL (TDL) that satisfy requirements specifications written in ERAE, then it can be done without significant difficulties because predicates and events in ERAE can be simulated within the object-oriented framework of TDL [Borgida et al. 1993].

However, ERAE rules have the **if-then** structure and do not support activities, decomposition of activities into subactivities, and the combination of activities and events with temporal clauses **when**, **while**, **before**, and **after** (as was demonstrated in the example in the introduction). For this reason, ERAE specifications require various techniques to encode certain end-user statements. For example, the statement "when a package arrives in the source station" [Dubois et al. 1991, p. 360] is expressed in ERAE as $\mathsf{location(p) = SourceStation} \wedge \bullet \neg \mathsf{location(p) = SourceStation}$ [Dubois et al. 1991, p. 425], i.e., that at present the package p is at the source station and that at the previous time moment (●) it was not, which is an *indirect* definition of the event arrival. Note that if Duboise et al. introduced the event "arrival" then this still would not solve encoding problems because, most likely, we would have to identify location of the package and would need rules describing how event "arrival of a package" is related to the predicate identifying its location. We believe that if ERAE supported various temporal clauses, such as **where**, **while**, **before**, and **after**, and supported activities (happening over time) directly in the language, this would have eliminated some of the encoding problems ERAE users have to face. In addition to the encoding problem, ERAE has a mathematical syntax which might be difficult to read and understand by a nontechnical end-user. Therefore, the ERAE specification method does not fully satisfy requirements (1) and (2) stated in the introduction.

INFOLOG [Fiadeiro and Sernadas 1986] is another specification language based on many-sorted predicate temporal logic supporting temporal triggers and events. INFOLOG triggers have the form ⟨trigger⟩::⟨transition pattern⟩, where trigger is an event variable and where transition pattern is an event structure consisting of individual events (atomic transitions) combined together using sequencing, alternative, and concurrency operators. As ERAE, INFOLOG satisfies our objective (4) since it has a rigorously defined semantics. Furthermore, it also satisfies objective (3): INFOLOG specifications can be mapped relatively easily into a broad range of design specification languages for the same reasons as ERAE specifications can.

However, INFOLOG has the same limitations as ERAE: it only supports events and does not support activities; it also does not support the combination of activities and events with temporal clauses **while**, **before**, and **after**. This means that INFOLOG specifications should use similar encoding techniques to model end-user requirements as EREA does. For example, INFOLOG has to use some encoding methods to model the statement presented in the introduction (if a customer comes to a branch of a bank *while* the branch is closed...). Furthermore, INFOLOG has also a rigorous mathematical syntax that might be difficult to read and understand by a nontechnical end-user. Therefore, INFOLOG, as ERAE, does not fully satisfy requirements (1) and (2).

TRIO [Ghezzi et al. 1990] is still another specification language based on temporal logic. TRIO uses the linear predicate temporal logic with operators $Futr(A, t)$ and $Past(A, t)$ that have the following meaning. $Futr(A, t)$ is true now if A will be true t time units from now. Also, $Past(A, t)$ is true now if A was true t time units before. It is shown by Ghezzi et al. how the standard operators of temporal logic (necessity, possibility, etc.) can be expressed in terms of *Futr* and *Past*. A TRIO specification is just a closed TRIO formula, i.e., any formula being temporally and classically closed.

As in the cases of INFOLOG and ERAE, TRIO is a rigorously defined and powerful specification language that can be mapped into a broad range of design specification languages. In fact, it is easier to map TRIO specifications into various design languages than ERAE or INFOLOG because it is based only on temporal predicates and does not support events. However, it does not fully satisfy requirements (1) and (2) for the same reasons as for INFOLOG and ERAE: TRIO does not directly support such important concepts as events, activities, and the interaction between events and activities and various temporal clauses; it also has a rigorous mathematical syntax that is hard to understand for a nontechnical end-user.

RDL [Gabbay et al. 1991] is still another specification language based on the intuitionistic propositional temporal logic. RDL specifications consist of a set of rules of the form

antecedent about the past → consequent about the future.

RDL satisfies our third and fourth requirements for the same reasons as TRIO does. However, RDL is less powerful than TRIO because it is based on propositional logic, while TRIO is based on predicate logic. Furthermore, RDL does not satisfy our first and second requirements for the same reasons as for TRIO.

RML [Borgida et al. 1985; Greenspan 1984] is a requirements specification language based on the object-oriented framework and multisorted first-order logic. An RML specification consists of a set of interrelated object definitions. RML distinguishes three types of objects, i.e., entity, activity, and assertion. Also, RML supports time, but unlike TRIO, ERAE, INFOLOG, and RDL, it is based on first-order rather than on temporal logic. Moreover, RML has a formal semantics, as described by Greenspan [1984].

However, RML does not fully satisfy some of the objectives stated in the introduction. Greenspan tries to design RML so that specifiers could organize knowledge in a natural and convenient fashion and make RML specifications easily understood by end-users. Although he achieves his objective in many respects, RML specifiers still have to use some encoding techniques in their specifications. For example, the statement "a new patient's location after he has been admitted is the ward to which he is being admitted" [Borgida et al. 1985, p. 87] has to be rephrased as "at the end of an ADMIT event, the value of the *toWard* property of the ADMIT event equals the value of the location property of the patient being admitted" [Borgida et al. 1985, p. 87, footnote] and is expressed in RML as:

```
toWard  of ADMIT at end(ADMIT) =
            location of (newPatient of ADMIT at end(ADMIT)) at end(ADMIT)
```

We believe that this kind of statement would require less encoding if RML were based on temporal logic, especially since the English statement has conjunction "after" in it.[1] Also, RML does not fully satisfy our third objective because it is more difficult to map requirements specifications written in RML into a broad range of design specification languages than for some of the previously considered specification languages, such as ERAE or TRIO, mainly because RML supports a wide range of knowledge representation primitives. For example, it would be more difficult to map RML requirements specifications into the design specification language Z [Spivey 1988] than it would be to map TRIO specifications into Z. The reason for that is that object-oriented constructs of RML are mapped into set-theoretic constructs of Z, and this requires a paradigm shift from the rich knowledge representation world of RML objects to the simpler world of Z values.

Telos [Myloupoulos et al. 1990] is an extension of RML and, therefore, is also based on the object-oriented framework. Telos extends RML by improving RML facilities for representing and reasoning about temporal knowledge, provides more general forms of generalization and classification abstractions than RML does, supports linguistic extensions through the definition of

[1]See Prior [1967] for an argument as to why temporal logic provides a more user-friendly approach to time than the first-order logic.

metaattributes, and provides support for deductive rules and integrity constraints. All these features added to RML make Telos a powerful requirements specification language that is relatively easy for the specifier to use. Also, Telos has a rigorously defined semantics.

However, Telos does not fully satisfy some of the objectives stated in the introduction. First, it is more difficult to map requirements specifications written in Telos into a broad range of design specification languages than for some of the previously considered specification languages, such as ERAE or TRIO, for the same reasons as it is for RML. Second, Telos does not fully satisfy requirements (1) and (2) because the SA still has to do a certain amount of encoding. Although Telos supports Allen's [1984] time interval temporal logic, it does not support point-based operators of temporal logic, as INFOLOG, ERAE, TRIO, and RDL do. For instance, the example in Mylopoulos et al. [1990, p. 333] saying that an author cannot referee his own paper is stated in Telos as:

$$(\forall\, y/\text{Person})(y \in \text{paper.author} \Rightarrow \neg(\exists\, t/\text{Time})y \in \text{paper.referee [at t]})$$

This expression is contrasted with an equivalent temporal logic expression that provides a more user-friendly treatment of time:

$$(\forall\, y/\text{Person})(y \in \text{paper.author} \Rightarrow \textbf{always\_in\_the\_future}\ y \notin \text{paper.referee})$$

Furthermore, Telos rules have the **if-then** structure and do not support **when**, **while**, **before**, and **after** clauses. Without these clauses and without the full support of temporal logic, Telos specifications require various encoding techniques to specify end-user requirements involving time.

Tempora [Loucopoulos et al. 1990] is still another specification language supporting time, complex objects, an extended entity-relationship (E-R) data model, and deductive rules. As in Telos, it also represents a rich modeling language. However, it also does not fully satisfy some of the requirements stated in the introduction. The rule structure of Tempora supports temporal logic, events, the **when** clause, and is based on the Event-Condition-Action model of a rule [McBrien et al. 1991]. Therefore, it provides a more user-friendly rule structure than other languages considered so far. However, Tempora supports only events and conditions and does not support activities. This means that activities occurring over time require some type of encoding in Tempora. For example, Tempora rules have to use some encoding techniques to model statement "while an activity lasts...," such as the one presented in the introduction. Therefore, Tempora violates, to a certain extent, our second objective because the end-user has to understand encoding techniques used by the SA. Moreover, Tempora depends heavily on the E-R data model and complex objects. This makes it more difficult to map requirements specifications written in Tempora into design specifications that use other paradigms, such as the object-oriented paradigm (e.g., language TDL [Borgida et al. 1993]), than for such language as TRIO (because TRIO is based only on temporal predicates that can be easily simulated in most of the modeling paradigms). Therefore, Tempora specifications do not fully satisfy our third objective to provide a specification language that can be mapped into a broad range of design specification languages.

MAL (Modal Action Logic) [Jeremaes et al. 1986] is still another requirements specification language that forms the basis of the FOREST project [Goldsack and Finkelstein 1991]. At the heart of MAL is the first-order logic which is extended with agent/action modalities, deontic expressions, temporal operators similar to Allen's [1984] interval operators, and action combinators. MAL is a rigorously defined and powerful requirements specification language. Furthermore, MAL specifications can be mapped into certain design specification languages. For example, a restricted set of MAL specifications can be mapped into Prolog [Costa et al. 1990]. However, it requires more effort to map MAL specifications into various design specification languages than such languages as TRIO [Ghezzi et al. 1990] or RDL [Gabbay et al. 1991], because MAL supports various additional constructs, such as agents, actions, permission and obligation operators, temporal operators, and action combinators, that make such mapping more complicated than in the cases of TRIO and RDL. Although MAL is a relatively specifier-friendly language, as the case study of a real-time operating system kernel in Goldsack and Finkelstein [1991] shows, its end-user (and to some extent specifier) friendliness can be improved. For example, the statement "if the kernel performs a dispatch operation, and if the clock ticktocks n times and the occurrence of p *executing* still exists, then time overrun has happened" is stated in MAL [Goldsack and Finkelstein 1991, Axiom 10, p. 112] as:

$$[\mathit{kernel},\ \mathit{dispatch}(p)]\mathit{occurrence}(p,\mathit{execute}) \supset (\mathit{occurrence}(\mathit{clock},\mathit{tick};\mathit{tock}^{n}) \rightarrow [\mathit{clock},\mathit{tick};\mathit{tock}^{n}]\mathit{time\_expired})$$

Since MAL's temporal model is exclusively interval based and therefore does not support instantaneous *events*, it *encodes* the English statement "the process p has been executing for n time units since the kernel performed a dispatch operation" with the statement saying that the time interval defined by n ticks of the clock is contained in the time interval defined by the execution of process p. Also, MAL has a technical syntax that is difficult for a nontechnical end-user to understand. Therefore, MAL does not fully satisfy requirements (1) and (2).

Statecharts [Harel 1988] is still another specification language for modeling complex reactive systems. It is based on the visual formalism of structured state diagrams (statecharts) and is successfully used in modeling various complex reactive systems. The language has a rigorously defined semantics, is relatively specifier and end-user friendly, and we believe that Statecharts specifications can easily be mapped into various specification languages since they are based on such a fundamental concept as finite-state automata. However, statecharts are based exclusively on the visual approach to specifications and therefore inherit some of the known limitations of the visual approach. As Harel [1992, p. 13] admits "... the job [of development of a "perfect" specification language] is far from complete. Some aspects of the modeling process have not been as forthcoming as others in lending themselves to good visualization. Algorithmic operations on variables and data structures, for example, will probably remain textual. In addition... some of the less obvious connections between the various parts of the system models are not easily visualized." Therefore, in this article we deal with an alternative (textual) approach to user-friendly specifications that is based on rules and temporal logic.

The work of Lansky and Georgeff [1986] on representing procedural knowledge and of Allen [1984] on the theory of action and time is also related to Templar, although not as directly as the other languages discussed in this section. The discussion of how this work is related to Templar can be found in Tuzhilin [1993].

In summary, we examined several requirements specification languages that support time and rules. Some of these languages, such as INFOLOG, ERAE, TRIO, and RDL, are based on temporal logic. Most of these languages have a rigorous semantics and have a high expressive power. However, they are not designed in such a way that the systems analyst can write requirements specifications in these languages quickly (either during the interviewing process or shortly after it) and can show them to the end-user and expect him or her understand these specifications with minimal help. The reason for that is that these languages require encoding to specify certain end-user requirements and that some of them have the syntax that is difficult for an end-user to understand. To solve these problems, we developed a specification language Templar that we will describe now.

3. OVERVIEW OF TEMPLAR

Templar features will be introduced with examples based on the description of an IFIP Working Conference [Olle 1982, 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 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 [de Maindreville and Simon 1988; McCarthy and Dayal 1989; Stonebraker et al. 1990; Widom and Finkelstein 1990] and of rule-based design methodologies in information systems [McBrien et al. 1991].

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 3.3.1.

Example 3.1.1. 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	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 (1) set the *postcondition* located(paper,reviewer) to be true and (2) 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 3.1.1 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 among 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.[2] 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.

We also impose a *safety* restriction [Ullman 1988] on Templar rules: a variable appearing in an action clause of a rule (e.g., **then**, **then-do**, etc.) must also appear positively in a state clause (e.g., **when**, **if**, etc.) of the rule.

[2]When we define the syntax of Templar formally and introduce all the clauses in Section 4.1, we will explain in Figure 3 how clauses correspond to modeling primitives.

For example, the previous rule was safe, whereas the rule **when** receives(paper,chairperson,author) **then-do** send(paper,chairperson,reviewer) is not safe (because the variable reviewer does not appear in the **when** clause).

3.2 Atomic and Composite Activities

Templar distinguishes between atomic and composite activities. A *composite* activity consists of subactivities. For instance, the activity review(paper,reviewer) from Example 3.1.1 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 3.2.1. A specification 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 in the multisorted model of Templar that we adopt from the ERAE model [Dubois et al. 1991].

Following the ERAE model, we define multiple sorts as follows. We start with a set of *elementary* sorts, i.e., sort names and singletons. Then the set of *derived* sorts is 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 $\cup$ woman. Sorts can be considered as *types* in programming languages. Each attribute of a temporal predicate and each parameter in an activity specification considered in Templar must belong to a certain sort. For instance in the previous example the variable paper belongs to the sort Papers and variable reviewer to the sort Reviewers.

An activity specification can be compared to a procedure in conventional programming languages or to 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 activity specification in detail in Section 4.1.

An *atomic* activity cannot be divided into subactivities. It is defined with a *future temporal predicate* which specifies how a predicate changes over time. We will define temporal predicates in detail in Section 3.5. For example, consider the activity specification

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

where reading_time(paper,reviewer) is a decision function that specifies how much time it takes a reviewer to read a paper (we will define decision functions in Section 3.9), and reading is a predicate that changes over time.

Then "reading(paper,reviewer) **for_time** T" is a future temporal predicate stating that the predicate reading(paper,reviewer) will be true for the next T time units. This expression is based on the bounded temporal operator **for_time** [Tuzhilin 1993] (also called *metric operator* by Koymans [1990]). The temporal predicate "reading(paper,reviewer) **for_time** T" defines an atomic activity.

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 simple subactivities.

Templar also allows multiple subactivities in the **then-do** clause of a rule. For instance, the **then-do** clause in Example 3.1.1 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 end-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 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 studies in Section 5 will demonstrate.

3.3 Activity-Event-Condition-Activity Rules

The rule from Example 3.1.1 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 3.3.1. Assume the organizers of the conference have a rule:

While a submitted 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 3.1.1, 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 rule the Activity-Event-Condition-Activity (AECA) rule because it generalizes the Event-Condition-Activity (ECA) model of a rule 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 summary, AECA rules can be viewed as an extension of the ECA model of a rule to support the temporal domain.

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 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 the **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 or her extra flexibility for writing specifications based on rules, procedures, and the combination of rules and procedures.

3.5 Temporal Predicates

Templar predicates can change over time. For example, if a paper is submitted to a journal today, then the predicate submit(paper,journal) is true today, and was not true yesterday or a week ago. Similarly, it may not be true in two years from now assuming that the submission process will be over by that time. Therefore, these types of predicates are called *temporal* [Kroger 1987], and their semantics is defined with a *temporal structure* [Kroger 1987] that describes how their instances change over time. The reader is referred to Kroger [1987] and Manna and Pneuli [1992] for in-depth descriptions of temporal structures and temporal logic in general.

Temporal predicates can take *temporal operators*, such as *possibility*—**sometimes_in_the_future** ($\Diamond$), *necessity*—**always_in_the_future** ($\Box$) [Manna and Pneuli 1992], *bounded necessity*—**for_time** T, *bounded possibility*—**within_time** T [Koymans 1990; Tuzhilin 1992], and their past mirror images can be applied to temporal predicates. Examples of future temporal operators are send(paper,A,B) **for_time** 3days (send a paper from person A to person B, and let it travel for 3 days), **always_in_the_future not** submit(paper,journal) (never submit paper to journal in the future). Examples of past temporal operators are **within_past_time** 6months vacation(person) (a person had a vacation sometime within the past 6 months), **always_in_the_past not** visited(person,Australia) (never before, a person visited Australia).

Temporal predicates and temporal operators can appear in the clauses **if** and **then**. **If** clause takes past temporal operators, and **then** takes future operators. The following example shows how temporal predicates can be used in Templar rules.

Example 3.5.1. The rule

Only the original papers can be submitted to the conference, i.e., if a paper has been published in some journal in the past, it has to be rejected.

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)
```

3.6 Static and Dynamic Constraints

Templar supports static [Nicolas 1982] and dynamic [Casanova and Furtado 1984; Hulsmann and Saake 1991; Lipeck and Saake 1987] constraints by specifying rules only with **if** and **then** clauses. The static constraint, also called *invariant*, does not have any temporal operators in neither 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'
```

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 predicate list_of_publications guarantees safety of the rule by restricting the universe of all possible publication outlets to a finite set.

3.7 Structuring Mechanisms in Templar

Templar supports structuring mechanisms of aggregation and generalization as follows. Generalization is supported exactly as in ERAE by using multi-sorted temporal logic that allows derived sorts [Dubois et al. 1991]. 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 assume that we want to state that it should belong to 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 User-Defined Modeling Constructs

Templar allows the SA to define his or her own language constructs, *assuming* that an appropriate semantics is specified for these constructs. For example, assume that the SA wants to define the temporal predicate **since** [Manna and Pneuli 1992] as a user-defined operator (assuming it is not the part of the language). To do this, the SA can define B **since** C in terms of the temporal variable x (not appearing anywhere else in the specification), the temporal structure of which is defined with the following rules:

if C **then** X
if B and not C and **previous** X **then** X
if B and not C and not **previous** X **then** not X
if not B and not C **then** not X

Additionally, the SA can state some of the properties of the user-defined construct if he or she feels that the definition of the construct is somewhat cryptic. In the previous example, the SA could define **since** in the standard way, as in Manna and Pnueli [1992], in addition to the rules listed above.

These user-defined constructs are *macros* in the sense that the semantics of these constructs is specified in terms of the *substitution* of their definitions into Templar programs. Thus, user-defined constructs do not extend the expressive power of Templar; they only make Templar specifications easier to read and write.

The user-defined modeling constructs are needed because Templar supports various modeling primitives that make the language easy to use for the system analyst and easy to understand for the end-user. However, different applications may require additional modeling constructs, not defined in Templar, that vary across these applications. If all of these modeling constructs are added to Templar, then the language will be overburdened with many modeling primitives, and quite a few of them will not be needed in many applications. Therefore, Templar supports a "core" of modeling primitives, and the modeling primitives not included in Templar and definable in terms of the "core" primitives can be included as user-defined constructs.

3.9 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, and cancellations of and constraints on activities.

Example 3.9.1. 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     located(paper,chairperson)
then-do  register_paper(paper,author);
         (distribute_paper_to_reviewers(paper,chairperson)
         ||send_acknowledgment(chairperson,paper,author))
```

The rule from Example 3.9.1 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_acknowledgment(chairperson,paper,author) occur in parallel in Example 3.9.1. 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.

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

Example 3.9.2. 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 *nontemporal* specifications. For example, when the program committee chair receives a paper, he or 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. Decisions are specified by the systems analyst and are needed to model atemporal phenomena in Templar, such as selection of reviewers, decisions which papers to accept and which to reject, how to group accepted papers into sessions, etc. Since decisions do not involve time, they can be specified in *any* temporal or nontemporal specification language (not necessarily Templar). Alternatively, if the SA does not think that a formal description of a decision is important to the specification of systems requirements, then the decision can be specified informally (e.g., in a natural language) because it does not affect the temporal part of Templar specifications. For example, it may not matter for the overall specification of an IFIP Working Conference how reviewers of a paper are selected by the conference chair (as long as there is an effective procedure of doing this).

As we stated, Templar is based on temporal logic. However, it is important sometimes to refer to time explicitly, as the next example will show. Therefore, we allow explicit reference to the time of an event in Templar using the *time* prefix. The next example illustrates the use of this construct and the **then-dont-do** and **then-cancel** clauses that respectively support *cancellations* of and *constraints* on activities.

Example 3.9.3. 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 it to the journal 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, paper, and 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 3.3.1 can be called arrive by the user.

3.10 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 presented in Section 4.2) 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 are stored in an active database [Maindreville and Simon 1988; McCarthy and Dayal 1989; Stonebraker et al. 1990; Widom and Finkelstein 1990] in the implemented system. For example, McCarthy and Dayal 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 and Section 4.2 its semantics.

4.1 Syntax of Templar

Templar specifications consist of a set of predicate declarations, a set of rules, and a set of activity specifications. Since Templar is based on multisorted 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 specification. We adopt the syntax of ERAE for declaring sorts and predicates [Dubois et al. 1991] and will not present it in the article.

The syntax of a Templar rule is defined with the BNF grammar, the topmost portion of which is presented in Figure 2. The complete description of this grammar can be found in Tuzhilin [1993]. As Figure 2 shows, a Templar

```
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
```

Fig. 2. Topmost part of the syntactic definition of a rule.

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

Fig. 3. Types of clauses.

rule consists of a collection of clauses that are divided into the body and the 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 Templar user to think in these terms. Figure 3 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 Figure 2. For example, the user can define such operators as **unless**, **atnext** [Kroger 1987], and so on. As was explained in Section 3.8, Templar treats user-defined operators as macros. These user-defined operators provide extra flexibility in describing real-world systems in terms that are more natural.

The syntax of activity specifications is defined with the BNF rules, the topmost portion of which is presented in Figure 4. The complete description of the grammar of Templar activities can be found in Tuzhilin [1993]. As Figure 4 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

```
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)
```

Fig. 4. Topmost part of the syntactic definition of activity specification.

terms of rules and activities without *if-statement*. However, it was added as a 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.

4.2 Semantics of Templar

In this section we define the semantics of Templar by mapping Templar specifications into some intermediate representation and then defining the semantics of the resulting specifications. In the first step of the conversion process, we map a Templar specification into an equivalent specification without composite activities. After that, we replace atomic activities by the corresponding temporal predicates and then provide the semantics of the resulting specification. We start with the process of removal of composite activities.

4.2.1 *Removal of Composite Activities.* Composite activities occur in **then-do**, **then-dont-do**, **then-cancel**, **when**, **before**, **after**, and **while** clauses. We will show how composite activities in these clauses can be replaced with subactivities that comprise them. We assume, without loss of generality, that the head of a rule contains only a *single* activity because multiple activities in the head of a rule can be grouped into a single composite activity containing these subactivities.

We will recursively consider various decompositions of activities into subactivities. We start with the sequential composition.

Sequential Composition of Activities. Let the composite activity comp-activity consist of subactivities activity1; activity2.

Assume that comp-activity occurs in the **then-do** clause of a rule, i.e., the rule has the form

```
⟨rest-of-rule⟩
then-do      comp-activity
```

where ⟨rest-of-rule⟩ consists of all the clauses of the rule except the clause **then-do** comp-activity. Then this rule is replaced with the following two rules:

```
⟨rest-of-rule⟩
then-do      activity1
then         flag

when         end.activity1
if           flag
then-do      activity2
then         not flag
```

where flag is a predicate, containing all the variables from ⟨rest-of-rule⟩, that does not occur anywhere else in the program.

Assume that comp-activity occurs in the **then-dont-do** clause of a rule. Let the rule have the form

```
⟨body-of-rule⟩
then-dont-do comp-activity
⟨rest-of-rule⟩
```

Then the rule is replaced with the following rules

```
⟨body-of-rule⟩
⟨rest-of-rule⟩

⟨body-of-rule⟩
if           begin.activity1
then         flag

when         end.activity1
if           flag
then-dont-do activity2
then         not flag
```

where flag is a predicate, containing all the variables from ⟨body-of-rule⟩, that does not occur anywhere else in the program.

If comp-activity occurs in the **then-cancel** clause of the rule ⟨rest-of-rule⟩ **then-cancel** activity1; activity2 then the rule is replaced with the rules

```
⟨rest-of-rule⟩
then-cancel  activity1

⟨rest-of-rule⟩
then-cancel  activity2
```

In other words, we assume that the cancellation of a composite activity and of all of its subactivities happens at once.

If comp-activity occurs in the **while** clause of a rule, then the clause **while** comp-activity is replaced with **while** activity1 or activity2. If comp-activity occurs in the **before** clause then **before** comp-activity is replaced with **before** activity1. If comp-activity occurs in the **after** clause then **after** comp-activity is replaced with **after** activity2.

If comp-activity occurs as part of the end.comp-activity event then this event is replaced with end.activity2, and if it occurs as part of the begin.comp-activity event, then this event is replaced with begin.activity1.

Parallel Composition of Activities. Assume that a composite activity comp-activity consists of subactivities activity1 || activity2. If comp-activity occurs in the **then-do** clause of a rule ⟨rest-of-rule⟩ **then-do** comp-activity then this rule is replaced with the following rules:

⟨rest-of-rule⟩ **then-do** activity1
⟨rest-of-rule⟩ **then-do** activity2

If comp-activity occurs in the **then-dont-do** clause of a rule then the rule is replaced with:

⟨rest-of-rule⟩ **then-dont-do** activity1
⟨rest-of-rule⟩ **then-dont-do** activity2

If comp-activity occurs in the **then-cancel** clause of a rule then the rule is replaced with:

⟨rest-of-rule⟩ **then-cancel** activity1
⟨rest-of-rule⟩ **then-cancel** activity2

If comp-activity occurs in the **while** clause of a rule, then the clause **while** comp-activity is replaced with **while** activity1 or activity2. If comp-activity occurs in the **before** clause then **before** comp-activity is replaced with **before** activity1 (or equivalently with **before** activity2). If comp-activity occurs in the **after** clause then **after** comp-activity is replaced with **after** activity1 or activity2.

If comp-activity occurs as part of the begin.comp-activity event then this event is replaced with begin.activity1 (or equivalently with begin.activity2), and if it occurs as part of the end.comp-activity event that this event is replaced with max{end.activity1, end.activity2}.

IF-Statement. Assume that a composite activity comp-activity consists of the IF-statement **if** cond **then** stat-list-1 **else** stat-list-2 **end_if**.

If comp-activity occurs in the **then-do** clause of a rule ⟨rest-of-rule⟩ **then-do** comp-activity then this rule is replaced with the following equivalent rules:

```
⟨rest-of-rule⟩
if          cond
then-do     stat-list-1

⟨rest-of-rule⟩
if          not cond
then-do     stat-list-2
```

Very similar rules replace comp-activity appearing in the **then-dont-do** and **then-cancel** clauses, and therefore, we omit their conversion here.

If comp-activity occurs in the **while** clause of a rule, then the rule containing the clause **while** comp-activity is replaced with two rules. In the first rule, the **while** clause is replaced with two clauses (**if** cond **while** stat-list-1) and in the second rule with the clauses (**if not** cond **while** stat-list-2). **Before** and **after** clauses are handled similarly, and therefore we omit their conversion here.

Finally, if comp-activity occurs as part of the begin.comp-activity event in a rule, then this rule is split into two rules. The first rule is obtained from the original rule (1) by adding the clause **if** cond to it and (2) by replacing all the events begin.comp-activity in the original rule with the events begin.stat-list-1. The second rule is obtained from the original rule in a similar way by adding the clause if not cond to it and replacing all the events begin.comp-activity with the events begin.stat-list-2. Since the event end.comp-activity is replaced in a way very similar to the event begin.comp-activity, we omit its description here.

FOREACH Statement. Assume that a composite activity comp-activity consists of the statement **foreach** arg **suchthat** condit **do** stat-list **end_for**. If comp-activity occurs in the **then-do** clause of the rule ⟨rest-of-rule⟩ **then-do** comp-activity then this rule is replaced with:

```
⟨rest-of-rule⟩
if          condit
then-do     stat-list
```

Similar transformations are applicable to **then-dont-do** and **then-cancel** clauses.

If comp-activity occurs in the **while** clause of the rule **while** ⟨comp-activity⟩ ⟨rest-of-rule⟩, then the rule is replaced with the rule

```
while       stat-list
if          condit
⟨rest-of-rule⟩
```

where condit contains the variable that *does not* occur anywhere else in the rule except the stat-list. For example, if comp-activity is **foreach** x **suchthat** S(x) **do** B(x,y) **end_for** then the rule is replaced with the following rule (assuming x′ does not occur anywhere else in the rule):

```
while       B(x′,y)
if          S(x′)
⟨rest-of-rule⟩
```

If comp-activity occurs as part of the begin.comp-activity event of a rule, then this rule is modified by adding the clause **if** condit to the rule (where condit contains the variable that does not occur anywhere else in the rule) and by replacing begin.comp-activity with begin.stat-list (this is the case because all the instances of activities in the stat-list begin at the same time). Finally, if comp-activity occurs as part of the end.comp-activity event of a rule, then this rule is modified as follows. Intuitively the end of activity comp-activity occurs when all of the activities in the stat-list are finished. We will illustrate the replacement strategy for end.comp-activity using the following example that can easily be extended to the general case. Assume we have a rule

```
when      end.comp-activity
then-do      activity
```

where comp-activity is

```
foreach x suchthat S(x) do B(x) end_for
```

Then this rule is replaced with the rules:

```
when      end.B(x)
if        S(x)
then      S'(x)

if        S ≡ S'
then-do   activity
then      not S'(x)
```

The first rule adds tuples to the new predicate S′. As time passes and activities for different values of x finish, the size of S′ grows. The second "rule" checks whether S′ becomes equal to S. It is a pseudorule since its syntax is not supported by Templar (we represented it this way for clarity); but it can be easily replaced by an equivalent valid Templar rule. If S′ becomes equal to S, this means that all activities B(x) have finished for the values of x satisfying S(x), and thus the end of comp-activity occurred.

We considered all the composite statements in activity specifications by now and, therefore, completed the process of recursive replacements of composite activities with its subactivities. This process can be continued recursively until only atomic activities are left in the rules of the program.

Example 4.2.1.2. Consider the rule from Example 3.1.1:

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

where activity review, as defined in Example 3.2.1, is

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

and activity send is

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

As a first step in the conversion process, we replace the sequential composition of activities in the **then-do** clause of the rule. As a result, we obtain the following rules:

```
when      end.send(paper,chairperson,reviewer)
if        referees(paper,reviewer)
then      located(paper,reviewer)                        (R1)
then-do   review(paper,reviewer)
then      flag1(paper,chairperson,reviewer)

when      end.review(paper,reviewer)
if        flag1(paper,chairperson,reviewer)              (R2)
then-do   send(paper,reviewer,chairperson)
then      not flag1(paper,chairperson,reviewer)
```

After that, we break the composite activity review into its subactivities. Rule (R1) produces the following two rules:

```
when      end.send(paper,chairperson,reviewer)
if        referees(paper,reviewer)
then      located(paper,reviewer)
then-do   read(paper,reviewer)                           (R11)
then      flag1(paper,chairperson,reviewer) and
          flag2(paper,chairperson,reviewer)

when      end.read(paper,reviewer)
if        flag2(paper,chairperson,reviewer)              (R12)
then-do   evaluate(paper,reviewer)
then      not flag2(paper,chairperson,reviewer)
```

Since end-review coincides with end.evaluate, rule (R2) is converted to rule (R3):

```
when      end.evaluate(paper,reviewer)
if        flag1(paper,chairperson,reviewer)              (R3)
then-do   send(paper,reviewer,chairperson)
then      not flag1(paper,chairperson,reviewer)
```

Finally, we have to eliminate activity send from rules (R11) and (R3). According to our conversion rules, end.send(paper,chairperson,reviewer) equals *max*{end(**not** located(paper,chairperson)), end(transfer(paper,reviewer) **for_time** transfer_time(paper,chairperson,reviewer))}.

If we assume that sending papers in the mail never happens instantaneously, i.e., transfer_time(paper,chairperson,reviewer) is never equal to 0, then

```
end.send(paper,chairperson,reviewer) = end(transfer(paper,reviewer)
  for_time transfer_time(paper,chairperson,reviewer))
```

Making this substitution, we obtain the following final set of rules out of rules (R11), (R12), and (R3) (in the process, we also split rule (R3) into two rules because send in (R3) consists of two parallel activities):

```
when      end.(transfer(paper,reviewer) for_time
            transfer_time(paper,chairperson,reviewer))
if        referees(paper,reviewer)
then      located(paper,reviewer) and flag1(paper,chairperson,reviewer)
            and flag2(paper,chairperson,reviewer)
then-do   read(paper,reviewer)

when      end.read(paper,reviewer)
if        flag2(paper,chairperson,reviewer)
then-do   evaluate(paper,reviewer)
then      not flag2(paper,chairperson,reviewer)

when      end.evaluate(paper,reviewer)
if        flag1(paper,chairperson,reviewer)
then      not located(paper,chairperson) and
            not flag1(paper,chairperson,reviewer)

when      end.evaluate(paper,reviewer)
if        flag1(paper,chairperson,reviewer)
then      transfer(paper,reviewer) for_time
            transfer_time(paper,chairperson,reviewer) and
          not flag1(paper,chairperson,reviewer)
```

4.2.2 *Semantics of the Intermediate Specifications.* In the previous section we replaced composite activities with atomic activities. Since an atomic activity is defined with a temporal predicate, in the next step we replace atomic activities with such predicates.

We describe the conversion process clause-by-clause. Atomic activities in the **then-do** clause are replaced with the corresponding temporal predicates (since an atomic activity is defined in terms of a temporal predicate), and the **then-do** clause is replaced with the **then** clause. If an atomic activity appears in the **then-dont-do** clause then this clause is replaced with the clause **then not** temp-predicate, where temp-predicate is the temporal predicate defining that atomic activity. Furthermore, if an atomic activity appears in the **then-cancel** clause then this clause is replaced with the clause **then not-prec** temp-predicate, where temp-predicate is defined as for the **then-dont-do** case, and **not-prec** is a negation operator that has a special meaning to be defined below in this section.

Activities can also appear in **while**, **before**, and **after** clauses. The only changes in these clauses result from the replacement of atomic activities with the corresponding temporal predicates. Finally, activities can appear as parts of events specifying beginnings and ends of activities. In these cases, atomic activities are also replaced by the corresponding temporal predicates.

As a result of this change, the **then-do**, **then-dont-do**, and **then-cancel** clauses are replaced by the **then** clause. Furthermore, the **while**, **before**, and **after** clauses are integrated into the **if** clause so that **while**, **before**, and **after** become corresponding temporal operators (since they refer now only to temporal predicates). Therefore, the Templar clauses are reduced now to **if**, **then**, and **when** clauses. Furthermore, these clauses contain only temporal predicates and events specifying when temporal predicates change over time. Such a system was studied by Tuzhilin [1991], where its semantics was defined in terms of the temporal recognize-act cycle.

However unlike the system described by Tuzhilin [1991], we consider two different types of the **not** operator in this article: regular **not** and **not-prec**. The semantics of **not** is that a predicate and its negation cannot contradict at the same moment of time (if they do, then the specification is invalid). The semantics of **not-prec** is that if *p* and **not-prec** *p* are true at the same time then **not-prec** *p* has precedence over *p*, and therefore *p* is canceled. This semantics is motivated by the fact that **not-prec** is obtained by converting activities in the **then-cancel** clause of Templar into temporal predicates preceded by the **not-prec** operator. Since cancellation assumes that activities in progress are terminated, so for the same reason we assume that negation has precedence for the corresponding predicates.

This completes the description of the Templar semantics. In the next section we describe two case studies that show how Templar can be used in real-world applications.

5. CASE STUDIES

To demonstrate suitability of Templar for the specification of real-world problems, we have undertaken two case studies. The first case describes the data transfer component of the TCP communication protocol [Stallings et al. 1988] that is responsible for sending pieces of data (segments) between sending and receiving nodes. The main body of the data transfer component of the protocol deals with ensuring that segments are delivered error free, in sequence, and with no loss and duplication. This component has a rich temporal semantics since messages are sent and received between communication nodes over periods of time; timers are set on and off; nodes wait for messages; and so on.

Since one of the objectives of this case study was to demonstrate user-friendliness of Templar, we took the English description of the TCP protocol, as described by Stallings et al., and went through it paragraph-by-paragraph creating Templar rules out of the English text. The Templar specification of the data transfer component of the TCP protocol consists of 11 rules, 5 activities, 5 predicates, 4 decisions, and 3 external events. Because of the space limitation, we cannot present this case study in the article, and the interested reader is referred to Tuzhilin [1993]. Our experience was quite positive: most of the Templar rules followed fairly closely the corresponding English sentences. For example, the English statement "if the receiving node receives a duplicate segment before the connection is closed, it must acknowledge the duplicate" can be stated in Templar as

```
when      end.send(segment,TCP_sender,TCP_receiver)
before    connection_closed(TCP_sender,TCP_receiver)
if        duplicate(segment,TCP_receiver)
then-do   send_acknowledgment(segment,TCP_receiver,TCP_sender)
```

where send(segment,TCP_sender,TCP_receiver) is the activity of sending a segment from a sending to a receiving TCP node; send_acknowledgment is the activity of acknowledging of the receipt of the segment; duplicate(segment,TCP_receiver) is a predicate specifying that segment is a duplicate for TCP_receiver; and connection_closed is an external event indicating that the connection between TCP_sender and TCP_receiver is closed.

The second case describes a portion of the Intelligent Adversary (IA) system developed by a company specializing in military simulations. The IA system simulates behavior of adversary pilots in combat situations so that the US Navy pilots can be trained for air battles using a computerized training system (this pilot training system 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 is implemented in OPS5, and it took two man-years to develop it.

In this case study, we implemented in Templar a module of the IA system that selects an appropriate radar mode and then designates the target. The specification is based on extensive discussions with the IA Project Leader who acquired the knowledge of the system as a result of many hours of discussions with the US Navy pilots regarding their air combat tactics. The specification of the module contains 27 Templar rules, 13 activities, 21 predicates, and 5 external events.

6. VALIDATING TEMPLAR SPECIFICATIONS

As Figure 1 indicates, analysis and validation are crucial to the development of correct requirements specifications. As part of the development process, the systems analyst (SA) converts informal natural language specifications provided by the end-user during the interviewing process into formal Templar specifications, validates them using validation tools, and then shows inconsistencies and omissions in the specifications to the end-user so that the end-user can correct them.

Templar specifications can have two types of mistakes. The first type of mistake is made *by the end-user*. For example, the end-user might say that if activity A finishes, then start activity B; but he or she might forget to tell what happens when activity B finishes. The second type of mistake is made by the SA, e.g., forgetting to declare predicates, or accidentally switching arguments in predicates or activities. For example, in one rule, the SA can say send(paper,chairperson,reviewer) and in another send(chairperson,reviewer,paper). In this article, we will study only the mistakes made by the end-user because one of the major objectives in writing Templar specifications is to elicit knowledge from the end-user and validate it (as shown in Figure 1).

Among the mistakes the end-user can make, the most important are *inconsistencies* (contradictions) and *incompleteness* of specifications since they make specifications invalid. Contradictions in specifications seldomly arise because the end-user makes wrong statements (it is assumed that the end-user has a considerable experience and knows the application well). For example, in the IA project, the navy pilots practically never made statements that were plain wrong (personal communication, D. Bodoff, Dec. 1992). Most of the contradictions happen because the end-user fails to provide additional specification details, and this leads to inconsistencies. For example, by far the most common mistake the US Navy pilots made describing their combat activities occurred when they made statements of the form "if A then B," "if C then D," such that B and D could contradict each other; in this case, the pilots failed to specify what happened when A and C occurred simultaneously which lead to a contradiction. This kind of inconsistency occurred because the specification was incomplete. For this reason, we concentrate on the issue of incompleteness of specifications in this section.

According to Dubois et al. [1991] and Myer [1985], a specification is incomplete if it omits relevant facts about the real-world system. Since only the user knows what facts are relevant and what are irrelevant, it is impossible for the system developer to determine formally if a Templar (or any other) specification captures all the relevant facts the user has in mind. Therefore, Templar specifications (or any other specifications) cannot be formally proven to be complete in general.

However, in certain cases, it may be possible to determine if a specification is incomplete by detecting certain types of omissions made by the user. In this section, we describe some types of omissions and present methods for their detection. These omissions can be divided into the following three categories.

The first category consists of *certain* omissions. For example, the end-user can say that if activity *A* finishes, then start activity *B* but does not mention activity *B* anywhere else in the rules. This means that we do not know what happens when activity *B* finishes, and that the end-user *certainly* omitted this fact. In this case, the validation system must state such an omission to the end-user in no uncertain terms.

The second category consists of most-*likely* omissions. For example, a specification can have a rule **if** A **then-do** B saying that activity B is triggered by condition A. Although it is quite possible to have a situation like this, we can expect that activities are most likely triggered by beginnings and endings of other activities. Therefore, it is *likely* that the user omitted something in this rule (e.g., **when** clause), and the validation system should issue a *warning* message to the user.

The third category consists of *hard-to-tell* type of omissions. For example, if a rule triggers an activity, and no other rule says what happens while the activity lasts, then there is *some chance* that the end-user made an omission.

However, if a validation system starts issuing a warning message in each such case, then the user will receive too many false warning messages because in many cases nothing should happen while an activity lasts. In case of such an omission, no warning messages should be issued by the validation system; instead, the SA should consider such an omission as a *methodological guideline*. This means that the SA has to keep such type of omission in mind and use his or her own judgment when to ask the end-user questions during the interviewing process if the SA suspects such type of an omission.

All three types of omissions can be either *temporal* or *nontemporal*. For example, a temporal omission occurs if the end-user does not specify what happens while an activity lasts. Since Templar deals mostly with specifications of systems evolving in time, we will concentrate on temporal omissions in this section.

We compiled a list of temporal omissions as a result of interviewing the Project Leader of the Intelligent Adversary project that was described in Section 5. As part of developing the Intelligent Adversary system, he conducted extensive interviews of U.S. Navy pilots in order to understand their patterns of behavior and reactions in combat situations. As a result of discussions with the Project Leader, we compiled the list of important omissions that these pilots make typically in their attempts to describe their behavior. As it follows from the description of these omissions, they are typical for a wide range of systems with a rich temporal component and not limited just to this specific system. We describe each type of temporal omission now.

Contradictions in Rules. The user can say "if A then B" and "if C then D." If B and D contradict each other then it should be specified what happens when A and C occur simultaneously. This was, by far, the most common type of omission in pilot descriptions (personal communication, D. Bodoff, Dec. 1992). It comes in two "flavors"—temporal and nontemporal. We consider the more general type of a temporal omission and treat the nontemporal type as a special case of the temporal omission.

To detect the temporal omission of this type, we proceed as follows. First, we map Templar specifications into the intermediate representation as described in Section 4.2. For each Templar rule, keep track of all the intermediate subrules into which the Templar rule is decomposed as a result of this mapping. The intermediate rules produced in Section 4.2 have the structure $BODY \rightarrow HEAD$, where $HEAD$ is a conjunction of temporal literals (a temporal literal is a predicate preceded optionally by negation and by one of the unary future temporal operators (such as "sometimes-in-the-future," etc.). To detect whether or not two Templar rules have a conflict, consider pairs of the intermediate rules obtained from these Templar rules during the conversion process. Then check if there is a pair of intermediate rules, such that one has predicate Q in its head and another predicate $\neg Q$ in its head. After that, we have to check whether or not the bodies of the rules can conflict. The checking procedure for this type of conflict is described by Tuzhilin [1991]. If any of the intermediate rules can conflict this means that the corresponding Templar rules can also conflict. In this case, the system issues a warning message to the user specifying that the corresponding Templar rules can conflict.

Interaction between a Rule and Activities It Fires. If a rule initiates some actions or makes some temporal predicates true, and these actions or predicates do not invalidate preconditions of the rule, then it is not clear if the rule has to be fired again *while* these activities last or predicates hold. In particular, the user may say "when A then-do B," and it may turn out that "when A while B" can be true. If this is the case, then ask the user if he or she really wants the rule to be fired again. For example, consider one of Templar rules describing pilot combat activities: "**when** an enemy fires a missile at a plane **then-do** *beam* that plane," where *beam* is a pilot jargon meaning that the plane has to be turned away so that it disappears from the enemy's radar screen in order to evade the missile (and keep doing so for, say, 10 seconds). However, it is not clear what happens when the enemy fires the second missile while the plane is beaming, i.e., it is not clear what to do when the condition "**when** an enemy fires a missile at a plane **while** the plane is beaming" holds. The most disastrous solution is to fire the rule the second time (and probably get hit by the first missile). However, it is also not clear if the rule should not be fired at all. The most appropriate solution in this situation is to detect this type of temporal omission and ask the pilot what to do (and maybe replace this rule with some other rule(s)). This was another very common type of temporal omission that pilots made (personal communication, D. Bodoff, Dec. 1992).

To detect this type of omission in the rule presented above, it should be tested whether or not the precondition "when A while B" of the Templar rule is satisfiable. Satisfiability problem for the general case of an arbitrary predicate temporal logic formula is undecidable [Harel 1985]. Even in the "best-case" scenario, the satisfiability problem is NP-complete and thus intractable.

This means that we have to use heuristics to detect sufficient conditions for the unsatisfiability of the precondition of a Templar rule. One such heuristics can work as follows. We can convert the part of precondition (containing only IF, WHILE, BEFORE, and AFTER clauses) to the temporal formula as is done in Section 4.2. The resulting expression is an IF clause with some temporal logic formula in it. Then convert this formula further to the conjunctive normal form, and test if any of the conjuncts is tautologically false. Clearly, this is a sufficient (but not necessary) test for unsatisfiability. If it turns out that none of the sufficient tests for unsatisfiability is passed, then the end-user must be warned about a potential problem by the system. The system should issue a warning message, and the end-user should be asked what happens in case "when A while B" is true.

Unspecified Terminations of Activities. If there is a statement "when A then-do B," and no rule says what to do when B is finished, then this means that the specification is incomplete, and the "omission" message should be issued.

However, there is a caveat to this problem. For example, consider a rule "when A then-do B" and assume that activity B consists of subactivities C1 followed by C2. Also, assume that the specification has the rule "when end.C2 then-do D" but does not contain any references to B among the preconditions of any rule. In this case the end of activity B is recognized *implicitly* by the end of its last subactivity C2. This example motivates the following strategy.

For each activity *A* appearing in the **then-do** part of a rule, consider its last subactivity in the specification of this activity (or consider the set of last subactivities if some of the subactivities occur in parallel). Starting with this subactivity, build the set of activities recursively by considering last subactivities in the activities added to this set. Then issue the "omission" message if none of the activities in this set appears in any of the preconditions of any of the rules. For example, if neither B *nor* C2 appears in the precondition of any of the rules of the specification in the previous example, then issue the "omission" message.

Failure to Specify What Happens While an Activity Lasts. One of the common types of temporal omissions comes from the failure to ask what happens while an activity lasts. For instance, in the "beaming" example the SA can ask the pilot what happens *while* the beam operation is performed.

It may turn out that nothing significant happens while an activity lasts. For this reason we feel that this condition should not be checked by the system. However, the SA should keep this condition in mind during the interviewing process as a methodological guideline.[3]

Dual Temporal Operators. If a rule contains a **before** clause, and there is no rule with the same preconditions but with an **after** instead of the **before** clause, then the check for this omission might be in order. For example, the user may say "when A before B then-do C" but does not specify what happens in case "when A after B." However, automation of this type of a check may produce many false alarms, and we believe that it is better to provide this check as a methodological guideline for the SA.

Only IF-Clause in Precondition. If a rule has only the IF clause as its precondition, and the postcondition triggers some activity or activities, then it is quite possible that the user failed to specify some events or activities in the precondition. The reason for this is that in many applications new activities start when old activities finish or when external events occur. Both of these situations require events or activities in preconditions and, therefore, other types of clauses. However, this is not a certain type of omission because the SA could use predicates for the encoding purposes (following the OPS5 style of programming and thus convoluting the logic of a specification in many cases). Nevertheless, a warning message should be issued to the SA. In case the SA uses OPS5 "mentality" in writing Templar specifications, the purpose of the warning message is to reprimand him or her for that.

[3]As a personal experience, the author detected a few omissions of this type (by asking "what happens while a certain activity lasts") during the process of interviewing the IA Project Leader.

In summary, we provided a list of temporal omissions that can happen in writing Templar specifications and described methods to check some of them, while listing others as methodological guidelines.

7. CONCLUSIONS

We defined the syntax and the semantics of the 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 satisfies the language design requirements stated in the introduction for the following reasons. First, Templar specifications are end-user- and specifier-friendly because Templar supports a powerful set of features that are integrated into one system. For this reason, it took only 11 rules, 5 activities, and 5 predicates to develop a sizable part of the TCP communications protocol discussed in Section 5. Furthermore, as our experience with the case studies demonstrates, well-formulated Templar rules are naturally expressed with English sentences that are meaningful to the end-user. Because of its end-user- and specifier-friendliness, the language facilitates closer interaction and greater feedback between the systems analyst and the end-user. In particular, the systems analyst can show Templar specifications to sophisticated end-users (such as communications engineers in the TCP case) or explain them with fewer problems to unsophisticated end-users in order to get their feedback.

Second, Templar requirements specifications can be translated into a broad range of design specifications for the following reason. The data model of Templar is based on predicates, and Templar predicates can be mapped into appropriate modeling constructs of most of the data models. Furthermore, activities and events are also two fundamental components of any model dealing with time and therefore should be either directly supported or easily simulated in a design specification language that supports time. This means that it should be easier to map Templar specifications into a broad range of design specification languages supporting time than to map requirements specifications written in a highly specialized language into the same range of design specification languages. This independence from the design specifications 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. Furthermore, formal semantics allows validation tools to be used in order to validate specifications written in Templar.

Since Templar satisfies the 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.

ACKNOWLEDGMENTS

The author wishes to thank Vasant Dhar, Matthias Jarke, Jon Turner, Bruce Weber, P. R. Balasubramanian, and anonymous reviewers for valuable comments on the earlier drafts of the article and David Bodoff for extensive discussions of some of the issues presented in this article.

REFERENCES

ALLEN, J. F. 1984. Towards a general theory of action and time. *Artif. Intell. 23*, 123–154.

BORGIDA, A., GREENSPAN, S., AND MYLOPOULOS, J. 1985. Knowledge representation as the basis for requirements specifications. *IEEE Comput. 18*, 4 (Apr.), 82–91.

BORGIDA, A., MYLOPOULOS, J., AND SCHMIDT, J. W. 1993. The TaxisDL software description language. In *Database Application Engineering with DAIDA*. Springer-Verlag, Berlin.

CASANOVA, M. A. AND FURTADO, A. L. 1984. On the description of database transition constraints using temporal languages. In *Advances in Database Theory*. Vol. 2. Plenum Press, New York, 211–236.

COSTA, M. C., CUNNINGHAM, R. J., AND BOOTH, J. 1990. Logical animation. In *Proceedings of the 12th International Conference on Software Engineering* (Nice, France). IEEE Computer Society Press, Los Alamitos, Calif., 144–149.

DAVIS, A. M. 1990. *Software Requirements: Analysis and Specification*. Prentice-Hall, Englewood Cliffs, N.J.

DE MAINDREVILLE, C. AND SIMON, E. 1988. Modelling non deterministic queries and updates in deductive databases. In *Proceedings of the International Conference on Very Large Databases*. VLDB Endowment, 395–406.

DUBOIS, E., HAGELSTEIN, J., AND RIFAUT, A. 1991. A formal language for the requirements engineering of computer systems. In *From Natural Language Processing to Logic for Expert Systems*, A. Thayse, Ed. John Wiley and Sons, New York.

FIADEIRO, J. AND SERNADAS, A. 1986. The INFOLOG linear tense propositional logic of events and transactions. *Inf. Syst. 11*, 61–85.

GABBAY, D., HODKINSON, I., AND HUNTER, A. 1991. Using the temporal logic RDL for design specifications. In *Concurrency: Theory, Language, and Architecture*. Lecture Notes in Computer Science, vol. 491. Springer-Verlag, New York, 64–78.

GEORGEFF, M. P. AND LANSKY, A. L. 1986. Procedural knowledge. *Proc. IEEE 74*, 10, 1383–1398.

GHEZZI, C., MANDRIOLI, D., AND MORZENTI, A. 1990. TRIO: A logic language for executable specifications of real-time systems. *J. Syst. Softw. 12*, 107–123.

GOLDSACK, S. J. AND FINKELSTEIN, A. C. W. 1991. Requirements engineering for real-time systems. *IEE Softw. Eng. J. 6*, 3.

GREENSPAN, S. J. 1984. Requirements modeling: A knowledge representation approach to software requirements definition. Ph.D. thesis, Dept. of Computer Science, Univ. of Toronto, Toronto, Ontario.

HAREL, D. 1985. Recurring dominoes: Making the highly undecidable highly understandable. *Ann. Discr. Math. 24*, 51–71.

HAREL, D. 1988. On visual formalisms. *Commun. ACM 31*, 5, 514–530.

HAREL, D. 1992. Biting the silver bullet: Toward a brighter future for system development. *IEEE Comput. 25*, 1, 8–20.

HULSMANN, K. AND SAAKE, G. 1991. Theoretical foundations of handling large substitution sets in temporal integrity monitoring. *Acta Informatica 28*, 4.

JEREMAES, P., KHOSLA, S., AND MAIBAUM, T. S. E. 1986. A modal (action) logic for requirements specifications. In *Software Engineering '86*, P. J. Brown and D. J. Barnes, Eds. Peter Peregrinus, 278–294.

KOYMANS, R. 1990. Specifying real-time properties with metric temporal logic. *J. Real-Time Syst. 2*.

KROGER, F. 1987. *Temporal Logic of Programs*. EATCS Monographs on Theoretical Computer Science, vol. 8. Springer-Verlag, New York.

LIPECK, U. W. AND SAAKE, G. 1987. Monitoring dynamic integrity constraints based on temporal logic. *Inf. Syst. 12*, 3, 255–269.

LOUCOPOULOS, P., MCBRIEN, P., PERSSON, U., SCHUMACKER, F., AND VASEY, P. 1990. TEMPORA—Integrating database technology, rule based systems and temporal reasoning for effective software. In *Esprit '90 Conference Proceedings*. Kluwer Academic, Dordrecht, Holland.

MANNA, Z. AND PNUELI, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York.

MCBRIEN, P., NIEZETTE, M., PANTAZIS, D., SELTVEIT, A. H., SUNDIN, U., THEODOULIDIS, B., TZIALLAS, G., AND WOHED, R. 1991. A rule language to capture and model business policy specifications. In *Proceedings of the 3rd Conference on Advanced Information Systems Engineering* (Trondheim, Norway, May).

MCCARTHY, D. AND DAYAL, U. 1989. The architecture of an active, object-oriented database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York.

MEYER, B. 1985. On formalism in specification. *IEEE Softw.* (Jan.), 6–26.

MYLOPOULOS, J., BORGIDA, A., JARKE, M., AND KOUBARAKIS, M. 1990. Telos: Representing knowledge about information systems. *ACM Trans. Inf. Syst. 8*, 4, 325–362.

NICOLAS, J.-M. 1982. Logic for improving integrity checking in relational data bases. *Acta Informatica 18*, 227–253.

OLLE, T. W. 1982. Comparative review of information systems design methodologies, Stage 1: Taking stock. In *Information Systems Design Methodologies: A Comparative Review*, T. W. Olle, H. G. Sol, and A. A. Verrijn-Stuart, Eds. North-Holland, Amsterdam, 1–14.

PRIOR, A. 1967. *Past, Present, and Future*. Clarendon Press, Oxford.

SMITH, D. R., KOTIK, G. B., AND WESTFOLD, S. J. 1985. Research on knowledge-based software environments at Kestrel institute. *IEEE Trans. Softw. Eng. SE-11*, 11.

SPIVEY, J. M. 1988. *Understanding Z*. Cambridge Tracts in Theoretical Computer Science, vol. 3. Cambridge University Press, Cambridge, U.K.

STALLINGS, W., MOCKAPETRIS, P., MCLEOD, S., AND MICHEL, T. 1988. *Handbook of Computer-Communications Standards*. Vol. 3. Macmillan, New York.

STONEBRAKER, M., JHINGRAN, A., GOH, J., AND POTAMIANOS, S. 1990. On rules, procedures, cashing and views in database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Atlantic City, N.J., May). ACM, New York, 281–290.

TSICHRITZIS, D. C. AND LOCHOVSKY, F. H. 1982. *Data Models*. Prentice-Hall, Englewood Cliffs, N.J.

TUZHILIN, A. 1991. Temporally active databases = Active databases + Time. Working Paper IS-91-43, Stern School of Business, New York, Univ., New York.

TUZHILIN, A. 1992. SimTL: A simulation language based on temporal logic. *Trans. Soc. Comput. Simul. 9*, 2, 87–100.

TUZHILIN, A. 1993. Templar: A knowledge-based language for software specifications using temporal logic. Working Paper IS-93-33, Stern School of Business, New York Univ., New York.

ULLMAN, J. 1988. *Principles of Database and Knowledge-Base Systems*. Vol. 1. Computer Science Press, Rockville, Md.

WIDOM, J. AND FINKELSTEIN, S. J. 1990. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Atlantic City, N.J., May). ACM, New York, 259–270.

Received October 1991; revised August 1993; accepted June 1994