VALIDATING REQUIREMENTS SPECIFICATIONS STATED IN KNOWLEDGE REPRESENTATION LANGUAGE TEMPLAR

by

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

October 1991

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

Working Paper Series

STERN IS-91-28

Validating Requirements Specifications Stated in Knowledge Representation Language Templar

Alexander Tuzhilin

Information Systems Department
Stern School of Business
New York University *

Abstract

Techniques for analysis and validation of software requirements specifications written in the knowledge representation language Templar are presented. Templar specifications are analyzed in terms of ambiguity, non-minimality, contradiction, incompleteness, and redundancy. Since Templar is a powerful knowledge representation language supporting a rich set of modeling primitives, it is difficult to reason directly on Templar specifications. To solve this problem, Templar specifications are mapped into equivalent temporal logic programs which are analyzed in terms the criteria listed above. However, it is hard to reason about Templar specifications because some of the criteria cannot be formally proven, and the verification of other criteria constitute undecidable or intractable problems. To overcome these difficulties, we consider a set of *tractable* conditions for each criteria, which serve as "alarms" for the user. If a condition is violated then it means that the specification either definitely has or potentially can have a problem. Furthermore, the user is notified about the source and the nature of the problem in certain cases.

1 Introduction

The size and complexity of software systems increased dramatically over the past decade [Dav90]. As a result of this, the probability of making errors in specifying and designing these systems has also increased [Dav90]. One way to reduce these errors is to develop suitable techniques for analysis and validation of requirements specifications.

Meyer [Mey85] described seven problems commonly found in requirements specification documents. He called them seven "sins" of the specifier. Subsequently, Dubois and Hagelstein [DH87] consolidated them into five problems:

- *ambiguity*: a specification admits multiple interpretations

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

- *non-minimality*: the presence of an element in the specification that corresponds not to a feature of the problem but features of a possible solution (also called over-specification)
- *incompleteness*: omission of relevant aspects of the system being specified (also called under-specification)
- *contradiction*: a specification has two or more incompatible features (also called inconsistency)
- *redundancy*: repetition of information.

In this paper, we present some formal methods for validating requirements specifications written in the knowledge representation language Templar[1] in terms of these five criteria. Templar is a software specification language based on knowledge representation methods designed to meet the following objectives: specifications written in Templar should be easy for the non-computer oriented users to understand, should have formal syntax and semantics, and it should be easy to map them into a broad range of design methods. A Templar specification consists of a set of rules and a set of activity specifications that describe composite activities in terms of its subactivities. It explicitly supports rules, events and activities, time, hierarchical decomposition of activities, sequential and parallel activities, static and dynamic integrity constraints, and data modeling abstractions of aggregation, generalization, classification, and association. The relationship of Templar to other knowledge representation languages for requirements specifications will be discussed in Section 3 after we introduce the features of the language.

It is difficult to reason about arbitrary Templar specifications because Templar supports a rich set of modeling primitives and some of these primitives are not based on logic and, therefore, not amenable to reasoning. For this reason, we map Templar specifications into a certain type of *equivalent* temporal logic programs [AM89, BFG+89, KKN+90, Tuz91b] and then try to validate these programs, thus validating Templar specifications. Since temporal logic programming is based on sound theories of logic programming [Llo87] and temporal logic [Kro87], it is much easier to reason on temporal logic programs than on Templar specifications.

Any attempt to determine algorithmically if a software specification has any of the five problems listed above, encounters the following difficulties:

- *Informality.* Some of the five problems described above cannot be detected with any formal method even if the specification itself is stated in formal terms. These problems cannot be

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

detected because they are not formally defined. For example consider the problem of incompleteness. It says that a specification is complete if it captures all the *relevant* knowledge about the real-world system the user has in mind. Clearly, it is impossible to define formally what knowledge the user perceives as "relevant" and what as "irrelevant." Therefore, incompleteness of a requirements specification cannot be formally validated in general. For the same reason, non-minimality condition cannot be formally validated unless there exists a formal definition of a problem space, a solution space and of the boundary between these two spaces.

- *Undecidability and Intractability.* Even if some of the five problems can be formally defined, such as contradiction, redundancy, and ambiguity, it may turn out that validation of these problems can be an undecidable problem. For example, the problem of showing that one of the logical statements follows from the set of other statements is undecidable in general [Chu36]. Furthermore, although the implication problem can be made decidable by restricting formulas to some smaller classes [DG79], it can still be intractable, i.e. cannot be decided in polynomial time.

The specification language ERAE [DHL+86] addresses these problems by providing a combination of manual deductive reasoning techniques and interactions with the user. Also, the specification language Tempora [LMP+90] addresses these problems by using a validation technique called *semantic prototyping* [TWL90]. This technique also involves an active participation of the user in the validation process.

In this paper, we have chosen another approach to solving these two problems. We formulate a set of *tractable* conditions for some of the five problems in software specifications listed above. These conditions serve as "alarms" for the user and are divided into two types. If the condition of the first type holds, then the software specification definitely has a problem, and the user is notified about it. If the condition of the second type holds, then the software specification *may* have a problem. In this case, the system only warns the user about the potential problem and shows the source of the problem. Then the user has to decide if he or she wants to change the specification or leave it unchanged.

The rest of the paper is organized as follows. In Section 2, we overview temporal logic and temporal logic programming since they will be extensively used throughout the paper. In Section 3, we present the language Templar. In Section 4, we show how Templar specifications can be mapped into temporal logic programs of a certain type. Finally, in Section 5, we address the issue of validation of Templar specifications and their corresponding temporal logic programs.

2 Preliminaries: Overview of Temporal Logic and Temporal Logic Programming

Since Templar is based on temporal logic (TL) and since we will map Templar specifications into temporal logic programs (TLPs) in Section 4, we briefly review temporal logic and temporal logic programming in this section. Books by Kroger [Kro87] and Rescher and Urquhart [RU71] provide a good introduction to temporal logic. Also, some of the TLP systems are described in [AM89, BFG+89, KKN+90, Mos86, Tuz91b].

The syntax of a predicate temporal logic is obtained from first-order logic by adding various future temporal operators such as **sometimes_in_the_future** ($\diamond$), **always_in_the_future** ($\Box$), **next** ($\circ$), **until** and their past "mirror" images **sometimes_in_the_past** ($\blacklozenge$), **always_in_the_past** ($\blacksquare$), **previous** ($\bullet$), and **since** to its syntax. Note that function symbols are allowed in temporal logic formulas since they are based on first-order logic.

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

A temporal structure can be extended from predicates to arbitrary temporal logic formulas in the standard inductive way [Kro87]. For example, $K_t(\Box A)$ is true if for all t' such that $t' > t$, $K_{t'}(A)$ is true. The meanings of the four standard future temporal operators are defined in Fig. 1. The meanings of past "mirror" images of these operators are defined similarly to the future operators except that time is referenced only in the past. A temporal interpretation is a *model* for a set of temporal logic formulas if all the formulas are true at *all* the times in this interpretation.

Besides these eight standard operators, other temporal operators can be defined, such as **before, after, while, when** [Kro87], and bounded necessity, **for_time (T)** ($\Box_T$), and possibility, **within_time (T)** ($\diamond_T$), operators [Tuz91b]. For example, A **for_time (T)** is true now if A is always true within the next **T** time units. These additional temporal operators have the same

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

Figure 1: Operators of Temporal Logic

expressive power as the **until, since** pair [Gab89] and are introduced only for the ease of use.

After reviewing temporal logic, we consider temporal logic programming. There are different types of temporal logic programming systems described in the literature. We will briefly review the system based on [Tuz91b].

A *present temporal literal* or just a *literal* is either a predicate or a negated predicate. A *past (future) temporal literal* is a temporal literal with the past (future) temporal operator associated with it.

A *temporal logic program (TLP)* is a set of temporal clauses. A temporal clause has the form $BODY \rightarrow HEAD$, where $BODY$ is a conjunction of *present* and *past* temporal literals and $HEAD$ contains a single *present* or *future* temporal literal. It follows from this definition that the body of a rule refers to the present and/or to the past, whereas the head of a rule refers to the present or to the future. It also follows from this definition that negations are allowed both in the head and the body of a rule.

Example 1 The statement

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

can be expressed as a temporal logic programming clause

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

or using a different syntax as

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

□

The meaning of a temporal logic program is associated, as in the case of a logic program, with a certain *model* of that program. As it follows from the previous definition, a model of a program is a temporal interpretation, the temporal structure K of which satisfies the condition that $K_t(body_i \rightarrow head_i)$ is true for all the rules i in the program at *all* the moments of time t. Additional discussion of the semantics of this specific temporal logic programming system can be found in [Tuz91a].

A model of a TL program is *finite* if all the program predicates have finite instances in its temporal structure[2]. We will consider only temporal logic programs with finite models in this paper. For example, the following program consisting of two rules $q(x) \rightarrow \circ p(x)$ and $p(x) \rightarrow p(x+1)$ and a fact $q(0)$ is not a valid program because at time 1 predicate p has an infinite instance.

3 Description of Templar

In this section, we briefly describe the software specification language Templar. For the complete presentation of the language refer to [Tuz91a]. The development of Templar was guided by the following design objectives [Tuz91a]:

1. Templar specifications should be easily understood by non-computer oriented people, and the requirements specifications stated in some form of a restricted natural language should easily be translated into Templar specifications.

2. Requirements specifications written in Templar should be easy to map into a broad range of existing software design methods, such as object-oriented design methods [RBPE91], combination of data flow and ER diagrams, and other process and data modeling languages, such as Telos [MBJK90] and Tempora [LMP+90]. This can be achieved by making Templar independent of various design specification languages. This will allow the systems developer to postpone decisions about which data and process modeling paradigm to choose until the design stage. Therefore, he or she has a freedom to select those paradigms in the design stage that are the most suitable for the requirements specifications produced in the requirements stage.

3. Templar specifications should be rigorous. Otherwise, there can be many translation errors from informal requirements into formal design specifications.

[2]We assume that the Universe of Discourse (UoD) is infinite. In case of the finite UoD, we can replace the requirement of a finite model with the requirement of a *safe* model, where safety is defined as in the case of safe queries in databases [Ull88].

Templar is designed so that it can be used at different stages of the software development life cycle. In particular, it can be used in two substages of the software requirements specification stage. In the problem analysis substage [Dav90], it can be used for the purpose of conceptual modeling. In the substage of actual writing of software requirements specifications [Dav90], it can be used as a language used in these specifications. Furthermore, Templar can be used in the design stage of the software life cycle, especially for the applications in which the data is stored in an active database [dMS88, MD89, WF90, SJGP90] in the implemented system.

A Templar specification consists of a set of rules and a set of activity specifications. It explicitly supports rules, events and activities, time, hierarchical decomposition of activities, sequential and parallel activities, static and dynamic integrity constraints, and data modeling abstractions of aggregation, generalization, classification and association. We describe Templar informally in this paper with a set of examples. Formal definition of the language can be found in [Tuz91a].

Examples of Templar specifications will be based on the description of an IFIP Working Conference [Oll82, Appendix A]. Organization of a working conference involves several activities: sending a call for papers, receiving paper submissions and registering these submissions, sending papers to be refereed, receiving reports back from referees, making acceptance/rejection decisions and so on. A Templar specification of such a conference consists of a set of rules and activities that will be described in turn below.

Rules. A Templar rule is based on *temporal logic* and on the *Activity-Event-Condition-Activity (AECA)* model. AECA is an extension of the Event-Condition-Action (ECA) model of rules in active databases [dMS88, MD89, WF90, SJGP90], and of rule-based design methodologies in Information Systems [MNP+91] that provide a more comprehensive support for time.

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.

Example 2 The user specification

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

is expressed with the Templar rule

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

□

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

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

The rule presented above consists of *clauses* **when**, **if**, **then**, and **then-do**. 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. 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 clauses and types of modeling primitives that can be used in clauses forces the user to think more structurally when writing specifications.

Besides the clauses described above, Templar supports other types of clauses, such as **while**, **before**, **after**, and user-defined clauses. Figure 2 shows the relationship between clauses and activities, events, and conditions. For example, events can occur only in **when**, **before**, and **after** clauses, and the **if** clause can take only conditions.

In general, a Templar rule has not only events and conditions in its antecedent, as rules in the ECA model have, but also activities, as the following example shows.

Example 3 Assume the organizers of a conference have a rule:

While the paper is being reviewed, any request to withdraw the paper will be granted

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

Figure 2: Types of Clauses

by the program chairperson.

This requirement can be expressed in Templar as

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

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

□

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

- allowing activities in the antecedent part of the rule;
- supporting not only **when**, **if**, and **then** clauses of the ECA model but several additional clauses, including the clauses shown in Fig. 2;
- providing a comprehensive support for time, as will be described below.

It is argued in [SJGP90] that an ECA model of a rule is a powerful model because it can support such diverse database concepts as views, special semantics for updating views, materialized views,

partial views, procedures, special procedures, and cashing of procedures. Since ECA is a special type of the AECA model, this means that AECA is a very powerful model of a rule.

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

Example 4

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

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

□

An activity specification can be compared to a procedure in conventional programming languages or to the body of a method in object-oriented programming, except it is defined in terms of temporally oriented modeling primitives (activities).

An *atomic* activity cannot be divided into subactivities. It is defined with an (optionally negated) *temporal predicate* describing how one of the relational predicates changes over time. For example, consider the activity specification

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

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

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

```
activity review(paper,reviewer)
    T = reading_time(paper,reviewer)
```

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

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

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

The combination of activity specifications and rules makes Templar a powerful specification method. If Templar specifications had only rules then they could contain hundreds of rules, and it would be difficult for the user (and often for the developer) to understand clearly how the rules interact. On the other hand, if Templar specifications consisted only of activities, then it could be difficult to describe the control logic with only the *if-then-else* statements for certain applications. With Templar specifications, the user has the flexibility of combining rules and activities in such a way that there are much fewer rules then for the strictly rule-based methods, and activity specifications tend to be small, simple and easy to understand, as the case study in [Tuz91a] shows.

Temporal predicates. Templar predicates can change over time. For example, the predicate `submission(paper,author,status)` can have different truth values at different moments of time depending on the value of `status` at those moments. Therefore, temporal operators, described in Section 2, can be applied to these predicates in **if** and **then** clauses.

Example 5 The rule

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

can be expressed in Templar as

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

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

□

Constraints. Templar also supports static [Nic82] and dynamic [CF84, LS87, HS91] constraints by specifying rules only with **if** and **then** clauses. The static constraint does not have any temporal operators neither in the head nor in the body of a rule. For example, the following static constraint

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

can be expressed in Templar as

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

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

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

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

can be expressed in Templar as

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

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

Other features. Furthermore, Templar supports data modeling abstractions of classification, aggregation, generalization, and association [TL82, HK87], parallel activities, external events, events defined by explicit specifications of time, periodic events and temporal precedence operators **before** and **after**. These features of Templar are described in [Tuz91a].

In summary, Templar supports a rich set of modeling primitives, including a powerful AECA model of a rule, that are integrated into one coherent specification language.

1) the paper is with the chairperson, and the reviewer does not have the paper
2) the reviewer has the paper and the chairperson does not have the paper
3) the paper is with the chairperson, and the reviewer does not have the paper
4) the paper is accepted for the conference

Figure 3: Sequence of Conditions Consistent with the Specification.

1) the paper is with the chairperson, and the reviewer does not have the paper
2) the paper is accepted for the conference
3) the reviewer has the paper and the chairperson does not have the paper
4) the paper is with the chairperson, and the reviewer does not have the paper

Figure 4: Sequence of Conditions Inconsistent with the Specification.

The meaning of Templar specifications is defined in terms of sequences of predicates (conditions) over time that are *consistent* with the specification, i.e. in terms of *models* of specifications. A sequence of predicates over time is consistent with a specification if it makes all the rules in the specification to be true at all the moments of time. For example, in the IFIP case, the sequence of conditions ("fragments" of predicates) shown in Fig. 3 is consistent with the specification. On the other hand, the sequence of conditions shown in Fig. 4 is not consistent with the specification because condition (2) ("paper accepted for the publication") should follow conditions (3) and (4). More detailed description of semantics of Templar specifications can be found in [Tuz91a].

Related Work There have been many IS specification methods proposed in the literature. Books by Davis [Dav90], Yourdon [You89], Olle et al [OHM+88], Rumbaugh et al [RBPE91] describe some of these methods. A variety of different specification methods exist because different applications, or even different parts of the same application, can best be specified with different methods [Dav90]. Since in this paper we are interested in the knowledge-based methods describing evolution of information systems in time, we will compare our work to existing knowledge-based specification methods dealing with rules and with time, such as RML [BGM85], Telos [MBJK90], Tempora [LMP+90], ERAE [DHL+86], and RDL [GHH91][3].

RML, Telos and Tempora are powerful knowledge representation languages supporting a rich set of modeling primitives. Among other features, RML [BGM85] and Telos [MBJK90] support deductive rules, object-oriented specifications, time, and data modeling abstractions of aggregation, classification and generalization. Tempora [LMP+90] supports time, complex objects, an extended entity-relationship data model, and deductive rules. However, all the three languages do not satisfy

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

our design objective of being independent of specific design data models. They depend heavily on specific design specification methods, such as object-oriented design, complex objects and entity-relationship diagrams. Furthermore, the rule structure of RML, Telos and Tempora do not support the powerful AECA rule model of Templar. The rules in Telos have the **if-then** structure and are based on some variant of many-sorted first-order logic. The rule structure of Tempora is based on ECA model [MNP+91] and on temporal logic and is closer to the rule structure of Templar than that of Telos. However, Tempora mainly supports events and conditions, and does not treat activities on the equal footing with events. For example, it does not allow activities in the antecedent part of the rule (e.g. in the **while** clause).

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

Finally, RDL [GHH91] is a specification language for the requirements and design of time-dependent systems based on the intuitionistic temporal logic. RDL has a rigorous and very general specification language and, as a result of this, its specifications can be easily mapped into most of the design specification languages and also can be formally verified. However, RDL does not support many of the modeling primitives described in this paper, such as an explicit support for events and activities, hierarchical decomposition of activities, and the support for the parallel and sequential composition of activities. As a result of this, RDL specifications may be difficult to understand by non-computer oriented users.

In summary, none of the software specification methods considered in this section satisfies all the three design goals stated above. Furthermore, the rule structures of these methods are not as universal and powerful as the AECA rule model of Templar.

In this paper, we are interested in the problem of validation of Templar specifications that were described in this section. However, Templar specifications support a rich set of modeling primitives, including such procedural features as activity specifications. Therefore, they are less suited for reasoning about software specifications. To solve this problem, we have chosen the following strategy. We will map Templar specifications into equivalent specifications expressed as

temporal logic programs (TL programs). Then we will validate these TL programs in terms of the five criteria listed in the introduction. Since the mapping from Templar specifications always produces an equivalent TLP specification, the validation results obtained for TLP specifications will be applicable to the original Templar specifications.

In the next section, we describe the mapping of Templar specifications into equivalent temporal logic programs, and in Section 5, we describe how to reason about these programs.

4 Mapping Templar Specifications into Temporal Logic Programs

In Section 2, we briefly described temporal logic (TL) programs. In this section, we show how Templar specifications can be mapped into these programs. We will not describe the precise algorithm that converts Templar specifications into TL programs but illustrate this process with an example.

Example 6 Consider the rule from Example 2:

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

where activity `review`, as defined in Example 4, 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)
    next not located(what,from) || transfer(what,to) for_time T
end_activity
```

The first step in the conversion process replaces all composite activities with atomic activities. To show how this can be done, notice that the end of the composite activity `send(paper,chairperson,reviewer)` coincides with the end of atomic activity `transfer(paper,reviewer) for_time T`, i.e.

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

Therefore, if we replace all the composite activities in our rule with the corresponding atomic activities, we obtain the rule:

```
when      end(transfer(paper,reviewer) for_time
          transfer_time(paper,chairperson,reviewer))
if        referees(paper,reviewer)
then      next located(paper,reviewer)
then-do   read(paper,reviewer); evaluate(paper,reviewer);
          (next not located(paper,reviewer) || transfer(paper,chairperson)
          for_time transfer_time(paper,reviewer,chairperson))
```

We will refer to this rule as the "main" rule subsequently.

In the second step, the rule is split into several rules so that each rule contains one atomic activity from the **then-do** clause. In our example, the main rule is split into rules:

```
when      end(transfer(paper,reviewer) for_time
          transfer_time(paper,chairperson,reviewer))
if        referees(paper,reviewer)
then      next located(paper,reviewer)
then-do   read(paper,reviewer)
```

```
when      end.read(paper,reviewer)
then-do   evaluate(paper,reviewer)
```

```
when   end.evaluate(paper,reviewer)
then   next not located(paper,reviewer)
```

```
when      end.evaluate(paper,reviewer)
then-do   transfer(paper,chairperson) for_time
          transfer_time(paper,reviewer,chairperson))
```

Notice that these rules follow the sequence of operators in the **then-do** clause. For example, if the activity `read(paper,reviewer)` is in the **then-do** clause of the first rule and the activity `evaluate(paper,reviewer)` sequentially follows it in the main rule, then the second rule contains the event `end.read(paper,reviewer)` in the **when** clause and `evaluate(paper,reviewer)` in the **then-do** clause. In case two activities occur in parallel, they have the same action in the **when** clause (as in the case of the last two rules).

To summarize, after the second step, a rule has only atomic activities in its clauses and the **then-do** clause contains only a single activity.

In the third step, we convert all the (atomic) activities and events to temporal logic formulas. To illustrate this, consider the first rule in the set of rules produced in step two:

```
when      end(transfer(paper,reviewer) for_time
          transfer_time(paper,chairperson,reviewer))
if        referees(paper,reviewer)
then      next located(paper,reviewer)
then-do   read(paper,reviewer)
```

The event `end(transfer(paper,reviewer) for_time transfer_time(paper,chairperson,reviewer))` can be expressed as the condition **previous** `transfer(paper,reviewer) and not transfer(paper,reviewer)`, which says that the transfer process was true at the previous time moment and is completed now. Also, the atomic activity `read` is replaced with the corresponding temporal expression (as defined in Example 4): `reading(paper,reviewer) for_time reading_time(paper,reviewer)`. Furthermore, all the clauses are mapped into the **if** and **then** clauses. **Then** and **then-do** clauses are mapped into the **then**, and the rest into the **if** clause. Therefore, the previous AECA rule becomes:

```
if      previous transfer(paper,reviewer) and not transfer(paper,reviewer) and
        referees(paper,reviewer)
then    next located(paper,reviewer) and
        reading(paper,reviewer) for_time reading_time(paper,reviewer)
```

This completes the conversion process from Templar specifications to TL programs.

However, we want to simplify the structure of TL programs even further. We want to convert TL programs with necessity, possibility and other temporal operators to equivalent TL programs with only **previous** ($\bullet$) and **sometimes_in_the_future** ($\diamond$) temporal operators. In other words, we want the rules to have the form

$$body \rightarrow (\neg)p$$

$$body \rightarrow (\neg) \diamond p$$

where *body* has only **previous** ($\bullet$) as a temporal operator, and the negation sign is optional.

The solution to this problem for some of the temporal operators was presented in [KKN+90]. For example, the TLP rule $p \rightarrow \Box q$ can be replaced by two rules $p \rightarrow q$ and $\bullet q \rightarrow q$. Similarly, the rule $p \rightarrow \Box_T q$ can be replaced with rules $p \rightarrow q$, $p \rightarrow t = T$, $\bullet q \wedge t > 0 \rightarrow q$, and $\bullet q \wedge t > 0 \rightarrow t = t-1$.

Also, the rule $\bullet p \wedge q \rightarrow \circ r$ can be converted to $\bullet^2 p \wedge \bullet q \rightarrow r$. Notice that the conversion process requires the use of function symbols, such as subtraction, in some of the rules.

Finally, the TL programs can be simplified even further as follows. Assume we have a rule $\bullet p \wedge q \rightarrow r$. Then this rule can be converted to the following three rules $\bullet p \rightarrow p'$, $\bullet p' \wedge \neg \bullet p \rightarrow \neg p'$, and $p' \wedge q \rightarrow r$. Therefore, a TL program can be converted to an equivalent program with two types of rules. The first type does not have any temporal operators in them, and the second type of the rule (with temporal operators) is in one of the forms

$$\begin{aligned} \bullet p &\rightarrow p' \\ \bullet p' \wedge \neg \bullet p &\rightarrow \neg p' \\ body &\rightarrow \diamond p \end{aligned} \tag{1}$$

where *body* does not contain any temporal operators. If a TL program is simplified to this form, i.e. temporal operators in such a program can appear only in rules of the form (1), then we say that this program is in the *canonical form.*

In this section, we presented a method that converts Templar specifications into temporal logic programs in the canonical form. TL programs have a simpler form making them more suitable for reasoning than Templar specifications. Therefore, we will reason about them and not about their equivalent Templar specifications.

5 Validation of Templar Specifications

As was described in the introduction, Dubois and Hagelstein [DH87] present five problems occurring in software specifications:

- *ambiguity*: a specification admits multiple interpretations
- *non-minimality*: the presence in the problem of an element that corresponds not to a feature of the problem but features of a possible solution (over-specification)
- *incompleteness*: omission of relevant aspects of the system being specified (under-specification)
- *contradiction*: a specification has two or more incompatible features
- *redundancy*: repetition of information.

As was also described in the introduction, the validation process cannot be fully automated because of the following reasons. First, some of these problems cannot be detected with any formal

method because they are not formally defined. For example, completeness of a specification cannot be formally defined because it is impossible to formalize the "relevant" aspects of the real-world system the user has in mind. Second, certain problems, such as redundancy, cannot be fully automated because they can be reduced to the decision problem of the first-order logic, which is undecidable in the most general setting [Chu36].

Therefore, we propose the following partial solution to the validation problem. We formulate a set of *tractable* conditions for some of the five problems listed above. These conditions serve as "alarms" for the user and are divided into two types. If the condition of the first type holds, then the software specification definitely has a problem, and the user is notified about it. If the condition of the second type holds, then the software specification *may* have a problem. In this case, the system only warns the user about the potential problem and shows the source of the problem. Then the user has to decide if he or she wants to change the specification or leave it unchanged.

We will examine each criterion in turn now.

5.1 Consistency

A Templar specification is *consistent* if it has a model, i.e. a sequence of predicates satisfying that specification[4]. To determine if a Templar specification is consistent, we propose to convert it into an equivalent TL program, as described in Section 4, and verify that the corresponding program is consistent by showing that it has a model.

Existence of a model of a set of temporal logic formulas is a hard problem. It is shown in [Har85] that for the general case of arbitrary fist-order temporal logic formulas it is a highly undecidable problem (is Π_1^1-complete). For certain restricted cases, the problem becomes decidable but still intractable. For example, assume that we impose such strong restrictions as considering only the static (snapshot) case when TL programs have no temporal operators at all and disallowing function symbols and the equality operator in the formulas. Then, as it follows from the Expansion Theorem [DG79], checking if a (temporal) logic program has a model is a decidable problem and is equivalent to the satisfiability problem, i.e. is NP-complete.

Since the verification of consistency of TL programs is an intractable problem, we propose the following partial solution. First, we want to identify the parts of a TL program that *can* produce inconsistencies. Then we try to see if these sources of inconsistencies are "false alarms," i.e. if they can never produce inconsistencies. If they can produce inconsistencies, then we alarm the user about these sources and identify them in the original Templar specification. Furthermore, we will

[4]See the definition of a model of a Templar specification in Section 2).

give the user some "reasonable" suggestions on how to avoid these inconsistencies in certain cases. Then it is the user responsibility to eliminate these potential inconsistencies.

We identify the following two sources of inconsistency in TL programs

- Predicates in the heads of two rules can conflict, i.e. we can have a situation when one rule has the form $body_1 \rightarrow q$ and another rule $body_2 \rightarrow \neg q$. These two rules can, potentially, have a conflict and, therefore, a model may not exist.
- A rule can be a constraint, i.e. have a form $body \rightarrow A < relop > B$, where $< relop >$ is a relational operator $=$, $<$, $\leq$, etc. A program may have no model because the constraint $A < relop > B$ may be violated.

We will look at these conditions in turn now.

5.1.1 Conflicting Predicates

As was stated before, it is a computationally hard problem to determine if two rules $body_1 \rightarrow q$ and $body_2 \rightarrow \neg q$ will always conflict at some time and, therefore, a TL program has no model. Therefore, we propose the following methods that check for potentially conflicting conditions and warn the user when these conditions occur.

In order to state these conditions, we first introduce some preliminary concepts. The *dependency graph* of a TL program is a graph with program predicates as its nodes (predicates p and $\neg p$ form two different nodes, but p and $\bullet p$ correspond to one node). There is an arc between nodes p and q in the dependency graph if there is a rule in the program containing predicates p in its body and q in its head, and these predicates do not have temporal operators. Dependency graphs, as defined in this paper, are very similar to dependency graphs defined in [Ull88].

Let $body_1 \rightarrow q$ and $body_2 \rightarrow \neg q$ be two conflicting rules in a TL program, where $body_i$, for $i = 1$ and 2, has subformula ϕ_i consisting of all the temporal predicates in $body_i$ (predicates preceded by the temporal operator $\bullet$) and all the relational operators. Then warn the user about a potential conflict between these two rules if one of the following two conditions holds:

1. the dependency graph of the program has a cycle going through nodes q and $\neg q$;
2. the formula $\phi_1 \wedge \phi_2$ is satisfiable.

Example 7 If a program has two rules $\neg p \rightarrow p$ and $p \rightarrow \neg p$ then the user is warned because the

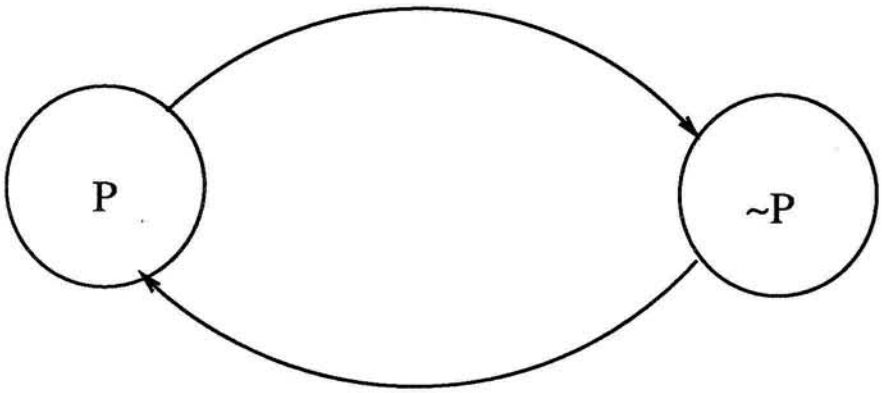

Figure 5: Dependency Graph from Example 7.

dependency graph for these two rules, as shown in Fig. 5 has a loop going through the nodes p and $\neg p$.

Also, if a program has rules $\bullet p \rightarrow q$ and $\bullet p \rightarrow \neg q$ then the warning will be issued because the formula $\bullet p \wedge \bullet p$ is satisfiable. However, if the program has rules $\bullet p \rightarrow q$ and $\bullet \neg p \rightarrow \neg q$ then the warning will not be issued because the formula $\bullet p \wedge \bullet \neg p$ is not satisfiable.

□

Both conditions warn the user that it is possible to have a situation when q and $\neg q$ are true at the same moment of time, i.e. when the program has no model. However, conflicts can occur for different reasons. Case 1 detects the situation when conflicts occur because of the "time-independent" (static) recursion, whereas Case 2 detects possible direct conflicts between the rules.

Furthermore, if a TL program has two rules of the form $body_1 \rightarrow \diamond q$ and $body_2 \rightarrow \neg q$ or of the form $body_1 \rightarrow q$ and $body_2 \rightarrow \diamond \neg q$ then always warn the user about a potential conflict. The reason for this comes from the following consideration. If the rule $body_1 \rightarrow \diamond q$ is replaced with the rule $\bullet^T body_1 \rightarrow q$ for some time T greater than any references to time in $body_2$, then $\bullet^T body_1$ and $body_2$ are satisfiable (because the predicates refer to different moments of time). Therefore, the warning to the user will always be issued for rules $\bullet^T body_1 \rightarrow q$ and $body_2 \rightarrow \neg q$. Since rule $body_1 \rightarrow \diamond q$ implies rule $\bullet^T body_1$ for some T, then to avoid potential conflicts, the user has to be issued a warning for the two original rules.

After conflicting rules in a TL program are detected and the warning to the user is issued, it is important for the user to locate the source of the conflict in Templar rules. Since each TL rule is obtained as a result of the transformation of one Templar rule, it is possible to identify conflicting Templar rules as long as the record is kept about the correspondence between TL and Templar

rules.

Furthermore, the user can also be given a suggestion on how to eliminate the source of inconsistency. If the inconsistency is of the type described in Case 1, then the cycle producing the recursion can be identified and the user should check if he or she really needs that recursive definition. If the inconsistency is of the type described in Case 2, e.g. the program has rules $body_1 \rightarrow q$ and $body_2 \rightarrow \neg q$, then the user can resolve the inconsistency by replacing these rules with $\neg q \wedge body_1 \rightarrow q$ and $q \wedge body_2 \rightarrow \neg q$, since these new rules can never conflict. This is a reasonable choice for the user because, most of the time, it is a good strategy to check if a predicate is true before making it true.

5.1.2 Constraints

As was stated before, the problem whether or not a constraint is satisfied is an intractable problem and we do not have any general "warning" techniques for that. This means that constraint specification in Templar can lead to inconsistent specifications, and the user should be aware of this situation.

5.2 Ambiguity

A specification can have many different models in general. For example, in the case of an IFIP conference, it does not matter how much time it takes a reviewer to review a paper or how much time it takes to send a paper from a reviewer back to the chairperson. Depending on the timing, we can get different sequences of events and different sequences of predicates over time, i.e. different models. For example, the program committee can meet either before or after a referee returns his or her evaluation report of a paper to the chairperson, depending on how much time it takes him/her to review the paper.

However, existence of multiple interpretations of a specification does not assume that it is necessarily ambiguous because the user might have specified the problem in this way *on purpose*. In fact, any further attempt to reduce the number of different interpretations may result in an *over-specification* of the problem, which is one of the "sins" of a specifier [Mey85].

Nevertheless, there can be specifications that are ambiguous. For instance, consider the following part of the IFIP conference specification (stated informally):

`if the chairperson receives an evaluation report from a reviewer he/she records the results of the review`

`if all the evaluation reports of a paper are available at the program committee meeting then the committee makes an acceptance or a rejection decision.`

This part of the specification is ambiguous in the following sense. If the chairperson received an evaluation report before the program committee meeting then nothing in the specification says that this report is *available* at the time of the meeting. This means that *both* interpretations of the specification are valid: the one that assumes that the report is available and the one that assumes it is not available.

To resolve this type of ambiguity the user can provide a *metarule*:

> *if condition P holds at time t and at the next moment of time $t+1$ the condition $\neg P$ does not hold then condition P holds at time $t+1$*

This type of rule is known as *inflationary condition* in logic programming [AV88, KP88]. If we assume inflationary conditions for Templar specifications and for TL programs then the ambiguity of the type described above is resolved: we conclude that the report will be available at the time of the program committee meeting. This observation is true in general: once we select the inflationary conditions, Templar specifications become unambiguous; the meaning of a Templar specification is associated with *all* the models satisfying the inflationary conditions.

However, the user may still not be satisfied with the set of models for his/her specification, i.e. he or she may still feel that some of the models are wrong. This means that the user has to provide more precise specifications to be able to reduce the set of models. Therefore, the original specification is *not* ambiguous but incomplete. We will address the problem of incompleteness of Templar specifications in the next section.

5.3 Incompleteness

According to [Mey85, DH87], 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 specification captures all the relevant facts the user has in mind. Therefore, Templar specifications cannot be formally proven to be complete in general.

However, in certain cases, it may be possible to determine if a specification is certainly incomplete even without consulting with the user. In this paper, we consider only one such case when a Templar specification has an infinite model, i.e. when there is a predicate and an instance of

time such that the predicate has an infinite instance at that time[5]. If a specification has an infinite model then it is incorrect and incomplete because some additional facts that make the model finite are missing in it. Therefore, we assume that

complete specifications have only finite models

Example 8

Consider the following rule:

When a chairperson receives a paper submission, he/she sends it to a reviewer:

```
when      receives(chairperson,paper,author)
then-do   send(paper,chairperson,reviewer)
```

This specification rule is incorrect because it says that the chairperson sends the paper to *all* the reviewers (that exist in the universe of discourse), i.e. to potentially infinitely many reviewers. Clearly, this specification is incomplete because it does not specify to which reviewers the paper should be sent.

One way to correct this specification is to assign some reviewers to review the paper, thus restricting the number of reviewers to a finite number. For example a new rule can have the form:

```
when      receives(chairperson,paper,author)
if        assigned_review(reviewer,paper)
then-do   send(paper,chairperson,reviewer)
```

□

We will state the conditions that guarantee finite models for TL programs, and therefore for Templar specifications. However, we introduce some preliminary concepts first.

A rule of a TL program is *safe* [Ull88] if all of the variables appearing positively in the head of the rule also appear positively in its body. A Templar rule is safe if all the TL rules obtained from the Templar rule with the conversion algorithm described in Section 4 are safe. For instance, the first (incorrect) rule from Example 8 is not safe because the variable `reviewer` does not appear in the **when** clause of that rule. However, the corrected rule in that example is safe.

Next, we consider static and dynamic recursion in TL programs. Intuitively, we say that a TL program is *statically recursive* if one of its predicates is defined recursively and the "recursive loop"

[5]Finite and infinite models were defined in Section 2.

does not contain temporal operators ($\bullet$ and $\diamond$ in our case). To define static recursion formally, we use the dependency graph of a program [Ull88], as defined in Section 5.1. We say that a TL program is statically recursive if the dependency graph of that program has a cycle. Furthermore, we say that a TL program is *statically functionally recursive* if it is statically recursive, and there is a cycle in the dependency graph and one of the predicates in the cycle has a function symbol as an argument in one of the program rules. For example, consider the following program consisting of two rules

$$\bullet q(x) \rightarrow p(x)$$

$$p(x) \rightarrow p(x+1)$$

and a fact $q(0)$. Its dependency graph has two nodes p and q and only one arc going from p to p. Therefore, it has a cycle of length one, and no predicate in the cycle has a temporal operator. Furthermore, the predicate p has a function symbol $+$ in the rule. Therefore, the program is statically functionally recursive.

A TL program is *dynamically recursive* if it has recursive definitions of predicates involving temporal operators. For example, the program consisting of a single rule $\bullet p(x) \rightarrow p(x)$ is dynamically (but not statically) recursive.

Proposition. *A safe statically functionally non-recursive TL program can have only finite models.*

Sketch of Proof: If such a model exists, then take the first moment of time when one of the predicates has an infinite instance. Since rules are safe, this cannot happen as a result of transition from the previous to the present time. Therefore, the instances of predicates "passed" from the previous stage are finite. Since TL programs are statically functionally non-recursive, we cannot get infinite models out of finite instances of predicates "passed" by previous stages. This leads to contradiction. □

It follows from this proposition that recursion does not always lead to infinite models. For example, safe statically recursive programs without function symbols produce only finite models (a well-known result for Datalog programs and some of its extensions [Ull88]). Also, dynamic recursion does not produce infinite models by itself (unless it is accompanied by static functional recursion), as the example $\bullet p(x) \rightarrow p(x)$ shows.

It is easy to check if a TL program is safe and statically functionally non-recursive. The former requires a simple syntactic check. The latter is reduced to finding cycles in the dependency graph of a program and checking that predicates along these cycles do not take any function symbols.

Therefore, this proposition provides a tractable condition for checking if TL programs have finite models.

5.4 Redundancy

In order to study redundancy of Templar specifications, we propose to reduce them to TL programs using the mapping described in Section 4. A TL program is redundant if one of its rules is logically implied by the set of other rules.

However, the problem of logical implication is undecidable for a general class of well-formed formulas in first-order logic [Chu36]. Nevertheless, it easily follows from the techniques developed in [DG79] that this problem is decidable for TL programs containing no temporal operators, function symbols and the equality operator.

Nevertheless, this problem is still intractable even in this very simple case. It follows from the same arguments as presented in Section 5.1 that this problem is still NP-hard. This means that there is no tractable procedure that determines if a given Templar specification is redundant.

5.5 Over-Specification

As was already mentioned in the introduction, Templar was designed so that requirements specifications stated in the language should be easy to map into a broad range of existing software design methods. This means that requirements specifications written in Templar are independent of these design methods. Therefore, the language does not encourage the system developer to over-specify the real-world system by introducing design elements in the requirements stage. On the other hand, the language does not discourage such practices because the language supports hierarchical decomposition of activities into subactivities.

However, the boundary between the stages of requirements specifications and design is not clearly and formally defined. Therefore, unless such boundary is well-defined, it is impossible to prove formally that a Templar specification is not over-specified.

6 Summary

In this paper, we studied the problem of analysis and validation of requirements specifications written in language Templar. We were interested in analyzing software specifications in terms of completeness, unambiguity, non-contradiction, non-redundancy, and minimality [DH87].

Since Templar has a rich set of modeling primitives, some of which are procedural in nature,

it is difficult to reason about Templar specifications. To solve this problem, we mapped Templar specifications into equivalent temporal logic programs and then analyzed these programs in terms of the five properties listed above.

We encountered two problems with formal validations of Templar specifications in terms of the five criteria listed above. First, some of these properties, such as completeness and minimality cannot be formally proven because they are not formally defined. Second, other properties, such as contradictions and redundancy, can be undecidable in general. Even for special cases, when these properties become decidable, the solutions are still impractical because they are still intractable (NP-hard).

To solve these two problems, we proposed the following solution. We stated *tractable* conditions for some of the properties that served as "alarms:" if these conditions do not hold then the property either does not or may not hold. For example, if a Templar specification admits an infinite model then the specification is incomplete. Then Templar specifications can be partially validated by checking these conditions.

We believe that some of these conditions are good working partial solutions for the original undecidable or intractable problems. For example, detecting if the dependency graph of a TL program has a cycle going through nodes of the type q and $\neg q$ and detecting satisfiability of certain simple formulas described in Section 5.1 is a good practical approximation to the intractable problem of detecting conflicts among predicates in Templar specifications.

On the other hand, some of the other partial solutions constitute only "mild" checks if a certain property holds. For example, checking if a specification accepts only finite models is a "mild" check for completeness. This observation raises an issue if there are "stronger" checks for some of these conditions that are still tractable and issues of theoretical trade-offs between the strengths of these checks and tractability. These issues constitute a topic of future research.

References

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

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proceedings of PODS Symposium*, pages 240–250, 1988.

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

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

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

[Chu36] A. Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 1936. correction: JSL, 1, 101–102.

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

[DG79] B. Dreben and W.D. Goldfarb. *The Decision Problem: Solvable Classes of Quantificational Formulas.* Addison-Wesley, 1979.

[DH87] E. Dubois and J. Hagelstein. Reasoning on formal requirements: A lift control problem. In *Proceedings of the 4th International Workshop on Software Specification and Design*, pages 161–167, Monterey, CA, 1987.

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

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

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

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

[Har85] D. Harel. Recurring dominoes: Making the highly undecidable highly understandable. *Annals of Discrete Mathematics*, 24:51–71, 1985.

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

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

[KKN+90] D. Kato, T. Kikuchi, R. Nakajima, J. Sawada, and H. Tsuiki. Modal logic programming. In *VDM and Z - Formal Methods in Software Development.* Springer-Verlag, 1990. LNCS 428.

[KP88] P. G. Kolaitis and C. H. Papadimitriou. Why not negation by fixpoint? In *Proceedings of PODS Symposium*, pages 231–239, 1988.

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

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

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

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

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

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

[Mey85] B. Meyer. On formalism in specification. *IEEE Software*, pages 6–26, January 1985.

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

[Mos86] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, England, 1986.

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

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

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

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

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

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

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

[Tuz91a] A. Tuzhilin. Templar: A knowledge representation language for requirements specifications. Working Paper IS-91-27, Stern School of Business, NYU, 1991.

[Tuz91b] A. Tuzhilin. Temporal logic as a simulation language. In *Proceedings of the International Conference on Artificial Intelligence and Simulation*, New Orleans, Louisiana, April 1991.

[TWL90] C. Theodoulidis, B. Wangler, and P. Loucopoulos. Requirements specification in TEMPORA. In *Proceedings of the Second Conference on Advanced Information Systems Engineering*, Lecture Notes in Computer Science 436, pages 264 – 282, 1990.

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

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

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