A Goal-Oriented Software Testing Methodology
Technical Report

Cu Duy Nguyen, Anna Perini, and Paolo Tonella

SRA Division / ITC-irst
Via Sommarive, 18
38050 Trento, Italy
{cunduy, perini, tonella}@itc.it

Abstract. Goal-oriented requirements engineering methodologies have
been investigated for more than a decade, aiming at better supporting
requirements engineering. They help elicit users’ requirements, deal with
stakeholders’ goals and strategic dependencies among them. Moreover,
they allow representing alternative solutions so that stakeholders and
developers can negotiate and choose the one that meets their business
demands. Some methodologies offer specification-based formal verifica-
tion, allowing software developers to correct errors at the beginning of
the development process. However, a structured testing process for goal-
oriented methodologies that complements formal verification is still miss-
ing.
In this report, we introduce a novel methodology for goal-oriented soft-
ware testing. It specifies a testing model that complements the goal-
oriented methodology Tropos and strengthens the mutual relationship
between goal analysis and testing. Furthermore, it provides a systematic
way of deriving test cases from goal analysis. To support the proposed
methodology, a testing framework was integrated into an existing tool
(TAOM4E) that supports Tropos.

1 Introduction

Requirement definition plays an important role for the successful development of
a software system. Since goals of stakeholders provide the rationale (the “whys”)
of the intended system and lead to the system requirements that should support
them [7], many goal-oriented requirements engineering (GORE) methodologies
have been introduced, e.g., GBRAM [1], i* [23], Tropos [5], KAOS [7]. They
help elicit users’ requirements, relate requirements to organizational and busi-
ness context, clarify requirements, and deal with conflicts [24]. Moreover, they
also provide means to represent alternative solutions, allowing stakeholders and
developers to negotiate and choose the best or a reasonable one.
Requirements engineering and testing have a strong link between them [12].
First, designing test cases early and in parallel with requirements helps discover
problems early, thus avoiding implementing erroneous specifications. Secondly,
good requirements produce better tests. Moreover, early test specification pro-
duces better requirements as it helps clarify ambiguity in requirements. The link
is so relevant that considerable effort has been devoted to what is called test-driven (or test-first) development. In this approach, tests are produced from requirements before implementing the requirements themselves [3, 2].

Focusing on GORE methodologies, testing results of goal fulfillment and satisfaction can be used to evaluate system. One would accept a system if it satisfies business objectives or goals. But, she/he could also reject the system because it does not fulfill other intended goals. Additionally, if we consider different abstraction levels of goals, e.g., goals of stakeholders, goals of the system to be built, and goals of a component, we end up verifying goal fulfillment at different testing types, e.g., acceptance test, system test, and component test. This provides the basis for a goal-oriented testing methodology.

Currently, some GORE methodologies support software developers performing formal verification at the beginning of the development process. Structured testing process that complements formal verification has not been thoroughly discussed, however. Some use goals only to understand the domain in which organizational settings, human agents, and the intended system are studied; testing is often omitted. Since goal is the fundamental element that drives the development process from early requirements to implementation, the needed testing process can be naturally based on goal.

We propose in this paper a novel GO approach that follows the V-Model [18] to software testing. To make the presentation tied to a specific GORE methodology, we will describe the proposed approach with reference to Tropos [5]. The proposed methodology takes into account the strong link between requirements and test cases, as discussed above, by deriving test cases from goal analysis all along the Tropos development process. Specifically, the proposed methodology contributes to the existing Tropos methodology by providing:

- A testing process model, which complements the Tropos methodology and strengthens the mutual relationship between goals and test cases;
- A systematic way for deriving test cases from goal analysis.

From the testing perspective, goal-oriented testing is aimed at (1) verifying the capability of the system actors (agents in case a multi-agent system are chosen as the implementation platform) to fulfill their goals, and so their dependencies from/to user actors; and (2) ensuring that they do not misbehave when required conditions are not satisfied or in case unwanted events happen while reaching the goals. Different types of test case are introduced in the paper to reach these two aims.

To support the methodology, we also introduce a tool and propose a structure to specify test suites and test cases.

The remainder of the paper is organized as follows. Section 2 mentions briefly background on Tropos methodology. Section 3 discusses the proposed methodology, a testing model, goal types vs. test types, test derivation, and structure of test artifacts (i.e. test suites, test cases, test scenarios). An example that illustrates how to derive test suites is presented in Section 4. Section 5 describes a tool that supports the proposed methodology. Finally, Section 6 summarizes the paper and introduces briefly literature survey and future work.
2 Background on Tropos

*Tropos* is an agent-oriented software engineering methodology [5] that adopts a requirement-driven approach, that is domain and system requirement analysis plays a pivotal role in the development process. *Tropos* is based on two key ideas. First, the notion of agent and all related mentalistic notions (for instance goals and plans) are used in all phases of software development, from early analysis down to the actual implementation. Second, *Tropos* covers also the very early phases of requirement analysis, thus allowing for a deeper understanding of the environment where the software must operate, and of the kind of interactions that should occur between software and human agents.

The *Tropos* development process consists of five stages [11]:

- **Early requirements.** The organizational settings where the intended system will operate and the relevant stakeholders are identified during this stage. Stakeholders are represented as actors while their objectives are represented as goals.
- **Late requirements.** The intended system is introduced as a new actor. It appears with new dependencies with existing actors that indicate the obligations of the system towards its environment as well as what the system can expect from existing actors in its environment.
- **Architectural design.** More system actors are introduced. They are assigned to subgoals or goals and tasks (those assigned to the system as a whole). The implementation platform is chosen, allowing designers to reuse existing design patterns. In large systems, subsystems are also specified and system actors are allocated to these subsystems.
- **Detailed design.** System actors are defined in further detail, including specification of communication and coordination protocols. Plans are designed in detail using existing modeling languages like UML or AUML [13].
- **Implementation.** During this phase, the *Tropos* specification, produced during detailed design, is transformed into a skeleton for the implementation. This is done through a mapping from the *Tropos* constructs to those of a target agent programming platform, such as JADE [19]. Recent work on mapping *Tropos* goal model to JADEX programming platform is described in [14].

The methodology provides a conceptual modeling language based on the *i* framework [22], a goal analysis technique and a diagrammatic notation to build views on a model. Basic constructs of the language are those of actor, goal, plan, softgoal, resource, and capabilities. Dependency links between pairs of actors allow to model the fact that one actor depends on another in order to achieve a goal, execute a plan, or acquire a resource and can be depicted in *actor diagrams*. As an example, Figure 1 shows how those constructs are used to model the requirements of a multi-agent system (MAS) that supports users (such as researchers) during bibliographic research. Both the user and the system are represented as actors (circles), user needs are represented in terms of goal dependencies from the actor *Researcher* to the system actor *BibFinder*, e.g. by
the hard goal *Find Bib* and the softgoal *Fast and efficient*. Further details of this example are discussed in Section 4.

Goals are analyzed from the owner actor perspective through *AND, OR* decomposition; *means-end* analysis of *plans* and *resources* that provide means for achieving the goal (the end); contribution analysis that points out goals and softgoals that contribute positively or negatively to reaching the goal being analyzed.

3 Goal-oriented testing

3.1 V-model of goal-oriented testing

The V-Model [18] is a representation of the system development process, which extends the traditional water-fall model. The left branch of the model specifies the specification stream, and the right branch of the V represents the testing stream where the systems are being tested (against the specifications defined on the left-branch). One of the advantages of the V-model is that it describes not only construction stream but also testing stream, i.e., unit test, integration test, acceptance test, and the mutual relationships between them.
Tropos guides the software engineers in building a conceptual model, which is incrementally refined and extended, from an early requirements model to system design artifacts and then to code, according to the upper branch of the V-model depicted in Figure 2.

Fig. 2. Tropos V-model

The modeling artifacts produced along the development process are: a domain model (i.e. the organizational setting, as is), the Early Requirements model; a model of the system-to-be where system requirements are modeled in terms of system goal graph, Late Requirements model; a system architecture model, specified in terms of a set of interacting software agents, Architectural Design model; a specification of software agent roles, capabilities, and interactions, Detailed Design model; agent code Implementation artifact.

We integrate testing in Tropos by proposing the lower branch of the V-model and a systematic way to derive test cases from Tropos modeling results, i.e. the upper branch of the V-model in Figure 2.

Two levels of testing are distinguished in the model. At the first level of the model (external test – test executed after release), stakeholders (in collaboration with the analysts), during requirement acquisition time produce the specification of acceptance test suites. These test suites are one of the premises to judge whether the system fulfills stakeholders’ goals.

At the second level (internal test – test executed before release), developers refer to: goals that are assigned to the system-to-be, high-level architecture, detailed design of interactions and capabilities of single agents, and implement these agents. Based on the output of the Late Requirements and Architectural Design phases, developers derive system test suites to test the system under development at system level. Similarly, based on the output of the Architectural Design and Detailed Design phases, developers derive agent test suites to test agents individually as well as to test interactions among them. The derivation of test suites is conducted when developers specify or design the system, thus helping them refine their design and uncover early defects.


3.2 Goal types versus test types

Focusing on the internal testing level of our V-model, we consider three types of testing: Agent testing, Integration testing, and System testing. The objectives and scope of each type is described as follows:

- **Agent testing.** The smallest unit of testing in agent-oriented programming is an agent. Testing a single agent consists of testing its inner functionality and the functionality it exposes to other agents with respect to the assigned goals.

- **Integration testing.** An agent has been unit-tested, we have to test its integration with existing agents. In some circumstances, we have to test also the integration of that agent with the agents that will be developed and integrated subsequently. In other words, integration testing involves making sure an agent works properly with the agents that have been integrated before it and with the “future” agents that are in the course of Agent testing or not ready to integrate. This often leads to developing mock agents or stubs that simulate the behaviors of the “future” agents.

- **System testing.** Agents may operate correctly when they run alone but incorrectly when they are put together. System testing involves making sure the agents in the system work together as intended. Specifically, one must test the interactions among agents (protocol, incompatible content or convention, etc.) and other concerns like security, deadlock.

Goals can be classified according to different perspectives or criteria. For instance, goals can be classified into perform goals, achieve goals, and maintain goals according to the agent’s attitude toward goals [8]. Other goal types are also discussed elsewhere e.g. KAOS. In this paper, since we are interested in separating individual agent’s behavior from social behavior induced by goal delegation in Tropos, we consider two types of goal: delegated goal and own goal. The former goal type is delegated by one agent (dependee) to another agent (dependee). More often than not, this goal type leads to interactions between the two agents: the dependee is responsible to fulfill the goal whenever the depender invokes it. The later goal type requires responsibility of its owner; however, the owner agent does not necessarily run within its boundary, i.e. it can involve interactions with other agents.

One can reasons about assigned goals of the own type of a single agent to come out with agent testing level. That is, based on these goals, developers figure out which plans or behaviours of the agent, i.e. functionality, to test. Since system testing involves the operation of all agents together in the system, the goals of type delegated and those goals of type own that involve agents interactions are good starting points for system testing.

3.3 Goal-oriented testing objectives

Users’ needs are represented in Tropos in terms of their own goals and of a set of social dependencies for goal achievement, during early requirements. Dependencies between users and the system-to-be actors are explicitly specified in the
late-requirements model. Hence, goal-oriented testing has to (1) verify the capability of the system actors (agents in case a MAS are chosen as implementation platform) to fulfill their goals, and so their dependencies from/to user actors; and to (2) ensure that they do not misbehave when required conditions are not satisfied or in case unwanted events happen. For example, one of the goals of a book search agent may be to find books by publication year, testing has to verify whether the agent is capable to find out available books or not. Moreover, it also has to make sure that the agent works properly when books are not available or when the search query is not well-formed.

To address the first objective, verifying the fulfillment capability of an agent with regard to a given goal, firstly, we analyze all the relationships rooted in this goal and identify the model elements that are on a path of relationships to this goal, such as plans. For each relation, a test suite that contains a set of test cases is created. Positive test cases, related to goal fulfillment, are generated, making all required resources available to the agent in order to trigger the identified element, e.g. plan. A test is considered as passed if the proper element (e.g. plan) is successfully executed successfully; otherwise, it is considered as failed.

Similarly, for the second objective, we create negative test cases, which deny the fulfillment of the goal, and add them to the test suite. A simple way to create these test cases is to provide wrong format resources while triggering the identified element. Test is passed if the agent under test keeps working properly; unveiled exceptions or misbehaviors are considered as failures.

Detailed structures of test suites and test cases are discussed in Section 3.5.

3.4 Test suite derivation

Based on the fact that goals cannot be fulfilled themselves, i.e. one must do something in order to achieve his/her goals, a very natural way of testing the achievement of a goal is to check one’s work or behavior with respect to the goal. Similarly, in order to test a goal in Tropos, we have to check what the system does or plans to do to fulfill the goal.

When applying the Tropos methodology, and moving from goal analysis to architectural and detailed design, we can find out how goals can be fulfilled by looking at their relationships with other goals and with plans. For instance, if there is a Means-End relation between goal G1 and plan P1, we say G1 is fulfilled when P1 is executed; if goal G2 contributes positively to softgoal SG2 (Contribution+ relation) then we can say SG2 is partially satisfied when G2 is fulfilled. Based on the relationships associated with a goal, we can check the fulfillment of the goal.

Goal-goal or goal-plan relationships are classified into two categories: Elementary relationships and Intermediate relationships. Elementary relationships are depicted in Figure 3. This includes (1) Means-End between a plan and a hard goal; (2) Contribution+ between a plan and a softgoal; (3) Contribution- between a plan and a softgoal. In order to test this kind of relationships, the execution of the plan corresponding to a goal is triggered and checked through
assertions on the expected behavior. Developers derive test suites from goal diagrams by starting from the relations associated with each goal. Each relationship gives raise to a corresponding test suite, consisting of a set of test cases that are used to check goal fulfillment (called positive test cases) and unfulfillment (called negative test cases). Positive test cases are aimed at accomplishing the first objective of goal-oriented testing: verifying the fulfillment capability of an agent with regard to a given goal; negative test cases, on the other hand, are used to ensure an appropriate behavior of the system under test when it can not achieve a given goal such as error management.

![Diagram](image)

**Fig. 3.** Elementary relationships. (1): a Means-End between a plan and a hard goal; (2): a Contribution+ between a plan and a softgoal; (3): a Contribution- between a plan and a softgoal.

Let us consider multiple elementary relationships that have the same goal but different plans. For instance, goal $G$ can be fulfilled by plan $P_1$ or plan $P_2$. Correspondingly, there are two elementary relationships of type (1) between $P_1$ and $G$ and between $P_2$ and $G$.

**Intermediate relationships** are shown in Figure 4. Six intermediate relationships are considered: (4) OR decomposition of a hardgoal into $N$ sub-hardgoals; (5) AND decomposition of a hardgoal into $N$ sub-hardgoals; (6) OR decomposition of a softgoal into $N$ sub-softgoals; (7) AND decomposition of a softgoal into $N$ sub-softgoals; (8) Contribution- of a hardgoal to a softgoal; and (9) Contribution+ of a hardgoal to a softgoal. In order to test the fulfillment of the root goals that are decomposed into subgoals or of those which receive contributions from other goals, we have to test the fulfillment of their subgoals or of those that contribute to them. Specifically, to test the root goal of (5) or (7), we have to test the fulfillment of the $N$ subgoals at a same time; to test the root goal of (4) or (6), we have to test the fulfillment of each subgoal; to test the root goal of (8) or (9), we have to test the corresponding hardgoals (contributing goals). The subgoals or contributing goals become, in turn, root goals of further analysis. This process continues until all considered relationships are of elementary kind. Therefore, the testing of goals related to intermediate relationships is approached eventually by the testing of goals that are part of elementary relationships.
Fig. 4. Intermediate relationships. (4): OR Decomposition of a hardgoal to $N$ sub-hardgoals; (5): AND Decomposition of a hardgoal to $N$ sub-hardgoals; (6): OR Decomposition of a softgoal to $N$ sub-softgoals; (7): AND Decomposition of a softgoal to $N$ sub-softgoals; (8): Contribution- of a hardgoal to a softgoal; (9): Contribution+ of a hardgoal to a softgoal;
There could be additional types of relationships since *Tropos* does not strictly constrain them. For instance, a common practice is to decompose a plan into several sub-plans. This could be avoidable by replacing the origin plan by a goal that is then decomposed into subgoals; the sub-plans are then means to achieve those subgoals. An example is illustrated in Figure 5.

\[ \text{Search with preferencies} \]
\[ \text{Search by address} \]
\[ \text{Search by number} \]
\[ \text{Search by number of position} \]

\[ \text{Search with preferencies} \]
\[ \text{Search by address} \]
\[ \text{Search by number} \]
\[ \text{Search by number of position} \]

\[ \text{AND / OR} \]
\[ \text{AND / OR} \]

Fig. 5. An example of changing plan decomposition to goal decomposition. (a) shows a plan that is decomposed into two sub-plans. (b) depicts an alternative way of representing that decomposition: the plan decomposition is changed to a goal decomposition and two means-end relationships are added.

### 3.5 Test suite structure

Figure 6 presents the test suite structure in XML Schema format [20] obtained using the tool XMLSpy\(^1\). The structure consists of three elements: General, Functional, and TestCase. The first element is depicted in Figure 7 providing descriptive information concerning the test suite, such as id, name, created by whom, version of the test suite, etc. The second element, Functional as described in Figure 8, contains functional information about the agent under test, the goal-plan couple being tested, pre-/post-conditions, test suite setup and teardown activities.

The third element of a test suite, TestCase, in turn, apart from general information and test case setup/teardown activities, contains the core element of a test suite: Scenario, as shown in Figure 9. A test scenario, depicted in Figure 10, consists of a linked set of sequences, which, in turn, can be a communication act (*Initiator, Responder, Message*), a branching or sequencing condition (*Next-Sequence, NextIfTrue, NextIfFalse*), or a check point (test oracle) to validate output data (*Condition*). Scenario is also used to specify the interaction protocol, in which check points can be added for testing purposes. The comparison operators *CompareOpt* between message *Message* and *ConditionData* include

\(^1\) [http://www.altova.com](http://www.altova.com)
Fig. 6. Suite structure. A test suite contains three parts: General information, Functional information, and the third part that contains a list of test cases.

Fig. 7. Suite structure - general part. General part contains information that describes the test suite itself.
Fig. 8. Suite structure - functional part. This part contains functional information, i.e., statements, conditions specification, etc. that are used in performing tests.
eq: equal, ne: not equal, ge: greater or equal, etc., contains: message content contains condition data, not − null: message content is not null.

Fig. 9. Test case structure. Test case structure specifies a concrete test scenario and related conditions as well as initial and final operations.

The proposed test suite, test case, and test scenario are designed such that they can be used at different formality levels and with different programming languages. Informally, developers can specify their test cases using descriptive text. This format can be used by human testers to manually provide input data to the system and evaluate the output results. When used formally, the specified test cases can be read by a testing tool to test the system automatically. To obtain these purposes, the elements Precondition, Postcondition, Setup, Teardown, Message, and ConditionData are designed to contain any user-defined data type; developers can associate their data with their own parser and grammar. An example in the next section illustrates how Message and ConditionData are used to specified ACL (Agent Communication Language) messages [9].

4 An example

4.1 Introduction

In scientific research and writing, bibliography search is a time-consuming activity. BibFinder is a MAS for the retrieval of bibliographic information in BibTeX format². BibFinder is capable of scanning the local drivers of the host machine,

² http://www.ecst.csuchico.edu/~jacobsd/bib/formats/bibtex.html
Fig. 10. Test scenario structure. A test scenario contains a sequence of test actions. The considered actions are sending, receiving, verifying data. After finishing an action, the three references NextSequence, NextIfTrue, and NextIfFalse point to the next action to be executed.
where it runs, to search for bibliographic data in the format of BibTeX. It consolidates databases spread over multiple devices into a unique one, where the queried item can be quickly searched. BibFinder can also exchange bibliographic information with other agents, in a peer-to-peer manner, thus augmenting its search capability with those provided by other peer agents. Moreover, BibFinder performs searches on and extracts BibTeX data from the Scientific Literature Digital Library³, exploiting the Google search Web service⁴.

Let’s go back to the Figure 1 that shows the late requirements analysis of BibFinder. Basically, the actor Researcher depends on the BibFinder system by the hard goal Find Bib and the softgoal Fast and efficient. Inside BibFinder, the former goal is decomposed into subgoals Interface to external systems, Manage local bib database, Auto-extract bibtex either From existing files or From the Internet, and Exchange bibtex with other BibFinders. The three goals Manage local bib, Auto-extract bibtex, and Exchange bibtex with other BibFinders contribute positively to the softgoal Fast and efficient.

Figure 11 shows the architecture of BibFinder. BibFinder is composed of three agents, namely: BibFinderAgent, BibExtractorAgent, and BibExchangerAgent. The goals identified in the late requirements analysis phase are assigned to these agents. For instance, Manage local bib and Interface are assigned to the BibFinderAgent agent while Exchange bibtex with other BibFinders is the BibExchangerAgent’s task. The BibExtractorAgent agent is responsible for the goal Auto-extract bibtex; and the BibFinderAgent depends on the BibExtractorAgent agent for that goal in order to serve requests from external systems.

4.2 Test suites derivation

Testing BibFinder consists of testing its three agents (BibFinderAgent, BibExtractorAgent, BibExchangerAgent) and the interactions among them (presented as goal dependencies). Inside each agent, we analyze the top root to find out all elementary relationships. A test suite will be derived for each elementary relationship.

For instance, test suite TS1 is derived from the Means-End relationship between the goal Search in local bib and the plan Search in local bib database. Since the goal is delegated by the BibFinderAgent agent to the BibExchangerAgent agent, two test cases are defined to check if the latter agent is able to search for an existing BibTeX item and if it behaves properly in case of badly formatted request; two test cases address the former agent’s ability to delegate the request.

Figure 12 depicts the test suite TS1 in XML format. Apart from general and functional elements, the test suite contains four test cases. In the first test case, a test scenario of two sequences are specified. The first sequence consists of sending a request (request content is shown in Figure 13) to the BibFinderAgent while the second one specifies a check point sequence, that is the tester waits for receiving a result and then compares it to a defined data.

³ http://citeseer.ist.psu.edu/
⁴ http://code.google.com/apis/soapsearch/
Fig. 11. Architecture of BibFinder. BibFinder contains three “physical” agents: BibFinderAgent, BibExchangerAgent, and BibExtractorAgent. These agents are responsible for the goals of the system specified in the Late requirements phase. Plans are identified in order to achieve those goals.
<?xml version="1.0" encoding="UTF-8"?>
  + <General>
    - <Functional>
      <AgentInCharge>BibFinderAgent</AgentInCharge>
      <GoalPlan>
        <Goal><Plan><Search in local bib</Plan></Search in local bib Database></Plan></Goal>
      </GoalPlan>
    </Functional>
  </General>
  <tc:TestCase>
    <tc:TestCase>
      <tc:TestCase>
        <tc:ID>T51-TC1</tc:ID>
        <tc:Name>Search Bib</tc:Name>
        <tc:Type>positive</tc:Type>
        <tc:Description>Search bibtex items for a given title</tc:Description>
      </tc:TestCase>
    </tc:TestCase>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1001</tc:ID>
      <tc:Initiator>TesterAgent</tc:Initiator>
      <tc:Responder>BibFinderAgent</tc:Responder>
      <tc:SequenceType>initial</tc:SequenceType>
      <tc:NextSequence>TC1002</tc:NextSequence>
    </tc:Sequence>
    <tc:Message>
      + <acl:fipa-message act="REQUEST" conversation-id="C12">Search bib</acl:fipa-message>
      <tc:Message>
        <tc:Priority>5000</tc:Priority>
      </tc:Message>
    </tc:Message>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
  <tc:TestCase>
    <tc:Sequence>
      <tc:ID>TC1002</tc:ID>
      <tc:Initiator>BibFinderAgent</tc:Initiator>
      <tc:Responder>TesterAgent</tc:Responder>
      <tc:SequenceType>checkpoint</tc:SequenceType>
      <tc:Message>
        - <acl:fipa-message act="INFORM">Search bib</acl:fipa-message>
        + <acl:msg-param>
          <acl:content>John Mylopoulos</acl:content>
        </acl:msg-param>
      </tc:Message>
    </tc:Sequence>
  </tc:TestCase>
</TestSuite>

Fig. 12. TS1 - Search Bib test suite

Fig. 13. Content of the request
4.3 Testing BibFinder against the derived test suites

We manually derived three test suites from the three Means-End relations in the architectural design of BibFinder (Figure 11). Test suite TS1 was derived from the Means-End relation between the goal Search in local bib and the plan Search in local bib database. Since the goal is delegated by the BibFinder Coordinator agent to the BibExchanger agent, two test cases are defined to check if the latter agent is able to search for an existing BibTeX item and if it behaves properly in case of badly formatted request; two test cases address the former agent’s ability to delegate the request.

Test suite TS2 contains two test cases to check the goal-plan couple Update - Update new items. Test suite TS3 contains two test cases to check the couple From the Internet - Search URL via Google.

BibFinder has been implemented under JADE [19]. The testing execution was performed on a Windows machine, equipped with a 1.1 GHz processor and 768 MB of RAM memory. BibFinder was tested by the tool that will be introduced in the next section. Those manually-derived test suites have revealed four moderate severity errors, related to the BibExtractor agent, which, in some cases, does not reply to an admissible request, replies to the wrong address or wrong performative. The last defect is related to the BibFinder Coordinator agent, that is it raises a fatal error while updating a wrong BibTeX item.

5 eCAT: a supporting tool

5.1 Introduction

In order to support the proposed methodology, we are developing a tool, called eCAT (environment for the Continuous Agent Testing). This tool interoperates with TAOM4E, a tool that supports Tropos, taking the output of goal analysis to generate skeletons of test suites. Developers then use a GUI editor to complete those test suites. This type of test suites is “manually-derived”, and they are of the type system testing and agent testing of the internal level in the V-model.

The specific features of MAS (autonomicity, proactivity, learning ability, cooperation, etc.) demand for a framework that supports extensive and possibly automated testing. In our future work, we intend to complement manually-derived test cases with automatically generated ones, which are continuously run in order to reveal errors associated with conditions that are hard to simulate and reproduce. The MAS deployed in the testing environment is free to evolve according to the behavior of the composing agents, but a monitoring agent and a tester agent that are included in it monitors continuously the state of the other agents and automatically generate test data that are submitted to the agents in order to guide their evolution towards revealing defects of the system. Details of this technique are discussed in a separate report.

Figure 14 depicts a high-level view of the proposed testing framework. It consists of three main components: Test Suite Editor, allowing human testers...
to derive test suites from goal analysis diagrams, such as those produced by TAOM4E; *Autonomous Tester Agent*, capable to automatically take manually-derived test suites and generate new ones and to execute them on a multi-agent system; and *Monitoring Agent*, that monitors communication among agents, including the *Autonomous Tester Agent*, and all events happening in the execution environment. The *Monitoring Agent* traces and reports errors as soon as they occur.

![fig14.png](https://example.com/fig14.png)

**Fig. 14.** eCAT framework. *Test Suite Editor* connects the framework to TAOM4E and provides GUI to edit test suites; *Autonomous Tester Agent* runs and evolves test suites continuously in background; *Monitoring Agent* monitors all sort of communication and events that take place in the environment.

Currently, eCAT is implemented as an Eclipse plug-in and it supports the JADE platform [19]. The *Autonomous Tester Agent* and the *Monitoring Agent* are implemented as a JADE agent and a JADE tool-agent, respectively. The *Autonomous Tester Agent* reads testing commands from an XML file, where test case generation technique, test suites (in the format presented in Section 3.5), protocol and data definition are specified.

### 5.2 Test suite editor

The test suite editor provides graphical interfaces allowing developers to derive test suites directly from goal-plan relationships. It is implemented as an extension of TAOM4E. Developers can choose any elementary relationship to create a test suite that is then used to test the goal and plan connected by that relationship.

Information that can be extracted from TAOM4E models helps automate test suite derivation. For instance, some general and functional information such as id, name, agent-in-charge, goal, plan, relationship can be taken automatically from the architectural model of TAOM4E. However, since plans are specified at
the conceptual level in the architectural model, and detailed design of plan uses
an extension of UML for agents (AUML [13]) that is not integrated in TAOM4E,
the core content of a test suite, i.e. the test scenario, is not currently generated
automatically. Test Suite Editor provides a visual environment in which devel-
opers can easily specify test scenario and data.

6 Conclusion

This report has introduced a goal-oriented testing methodology that takes goal
artifacts from modeling diagrams as central elements to derive test cases. The
methodology provides a systematic guidance to generate test suites along the
Tropos development process. These test suites, on the one hand, are used to
refine goal analysis and detect problems early at the requirement phase. On the
other hand, they are executed afterward to test the achievement of the goals from
which they were derived. Our preliminary application of the proposed method-
ology and tool in testing BibFinder, which has been introduced in Section 4, is
very promising in revealing errors.

In addition, different goal types, testing types, and the connection between
them were discussed in the report. The test suite structure was also discussed.

The remainder of the report introduces a short literature survey and our
future work.

6.1 Literature review

Research on agent verification focused mainly on static verification [21, 10, 4].
Existing techniques, which can be applied, are mathematical-based verification,
program inspections, Cleanroom software development, etc. Despite the effective-
ness of static verification, its application is still limited as it requires substantial
formalization and analysis effort, and mathematical skills as well. In addition,
user’s requirements can be incorrectly formalized, thus subsequent steps are not
able to detect wanted errors. To fill this gap, we commonly apply both static
and dynamic verification.

Dynamic verification (testing) of MAS which are usually distributed and con-
sist of concurrent entities, is a challenging task. Current researches on testing
MAS mainly investigate monitoring MAS at run-time in order to observe abnor-
mal behaviors. Coelho et al. [6] focus on unit testing (i.e. single agent testing)
with the help of mock agents that simulate real agents that the agent under test
communicates with. A monitoring agent is involved to monitor the interactions
among agents. Poutakidis et al. [15] use knowledge from diagrams that spec-
ify communication protocols during the architectural design phase to monitor
execution and detect problems of a MAS.

Saff and Ernst [16, 17] have introduced and evaluated a testing technique
that used spare CPU resources to run continuously tests in the background,
providing rapid feedback about test failures while the source code is being edited.
This technique is called continuous testing and suitable for regression testing,
that is to make sure that new changes to a program do not effect previously-tested functionalities. They reported that this technique reduced wasted time by 92-98%.

6.2 Future work

The methodology has focused mainly on the goal concept, leaving out other dependencies among actors like resource dependency, plan dependency. The methodology will be extended to cover those types of dependency as well.

Currently, the internal structure of a plan has not been fully exploited. Detailed information concerning operations and interactions could facilitate detailed test generation. In the future work, we will investigate plan modeling as well in order to extract this information.

Furthermore, we will investigate continuous testing and test cases generation techniques in order to deal with testing MAS.

References


