Requirements Eng (2000) 5:???–???

Requirements
Engineering

A Method for Extracting and Stating Software Requirements that a User Interface Prototype Contains

A. Ravid[a] and D. M. Berry[a,b]
[a]Computer Science, Technion, Haifa, Israel; [b]Computer Science, University of Waterloo, Waterloo, Ontario, Canada

User interface and requirements prototyping is a requirements elicitation technique. A user interface and requirements prototype is built during the requirements engineering phase of a software system development. Along with the user interface prototype are produced various documents such as the system requirement specification. When a prototype and other documents exist, they may not describe the same functionality, particularly because there may be behavior of the prototype, artifacts of prototyping, that may not be intended. The problem is that in later development stages, when there is a prototype and other documents, it is often difficult to reconcile the difference between the prototype and the other documents. This paper presents an approach for avoiding this difficulty. It demonstrates the approach by showing its application to parts of a real software development.

Keywords: Case study; Requirements elicitation; Requirements extraction; Requirements prototyping; Requirements tracing; User interface prototyping;

1. Introduction

User interface (UI) and requirements prototyping (UIRPing) is a popular requirements elicitation and validation technique, used to explore the functionality, UI, and other characteristics of a software-intensive, computer-based system that is to be built [1]. Partial implementations are developed quickly to allow the implemented features to be explored by the clients and users. A UI and requirements prototype (UIRP) permit users to relate to something tangible and to see as concretely as possible the effects of different choices for dealing with problematic requirements. Thus, the chosen requirements end up being validated by the clients and users on the basis of realistic, concrete examples.

The products of UIRPing are several artifacts, including the UIRP itself and various requirements documents (RDs). The RDs typically include a software requirements specification (SRS), a user's manual (UM), a requirements rationale (RR), etc. After the requirements are considered complete enough to begin implementation, the UIRP and RDs are given to the implementors.

Typically, the requirements engineers (REs) that developed the requirements move on to other projects. Even if some of the REs stay on the project through implementation, new members may join the project, diluting the expertise of the group. As a consequence, at any given time, knowledgeable REs may be unavailable to implementors for questioning, and the implementors may have to rely entirely on the artifacts in order to proceed with implementation.

2. The Problem

In the implementation phase, implementors work from the UIRP, SRS, UM, RR, etc. to implement the application. Sometimes, the implementors have a question. Assuming that the REs are not available for questioning, the implementors must search the UIRP, SRS, UM, RR, etc. for answers to their questions. What happens if the UIRP and the various RDs disagree? Which, if any, do the implementors believe?

Typically, the SRS, UM, and RR agree with each other as a group more often than with the UIRP. The normal

Correspondence and offprint requests to: Daniel M. Berry, Computer Science Department, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada. Email: dberry@csg.uwaterloo.ca

requirements engineering (REing) process encourages consistency checking between the written documents, the SRS, UM, and RR. In any case, lack of consistency between the SRS, UM, and RR is an old problem, studied elsewhere [2], with its own methods for overcoming it. Therefore, we consider the SRS to be the representative of the written RDs from now on and consider the problem to be that of divergence between the UIRP and the SRS. Thus, the problem considered in this paper reduces to: "When the UIRP and the SRS disagree, which do the implementors believe?"

Arguing for believing the UIRP are that

- the UIRP is more concrete than the SRS and is thus easier to validate by the client and users (C&Us); and
- therefore, C&Us' acceptance of the UIRP is more credible than C&Us acceptance of the SRS.

Arguing against believing the UIRP are that

- not all of the behavior of the UIRP is intended;
- some behavior of the UIRP is an artifact of rapid prototyping;
- some behavior of the UIRP is debris left over from functionality discarded during previous examinations of the UIRP; and
- behavior that is missing cannot be assumed as not intended, because a UIRP is intended to cover *not* all the behavior, but only the less well-understood behavior.

Arguing for believing the SRS are that

- the SRS is intended to be complete and consistent; and
- the SRS is intended to cover all and only the behavior of the application.

Arguing against believing the SRS is that

- it is harder for the C&Us to validate words describing behavior than a UIRP showing the same behavior.

It is always a possibility that neither the UIRP nor the SRS should be believed. It has happened often enough that both are wrong, even if they are consistent with each other. Thus, this possibility should always be considered.

Therefore, it is definitely *not* clear what to believe when the UIRP and the SRS differ. It is ultimately an issue of "What of the UIRP's behavior is *intended*?"

Assume in the following discussion that the UIRP and the SRS differ on a particular behavior. If the behavior *is* intended to be in the UIRP, then we should believe the UIRP more than the SRS description of the behavior, simply because the UIRP is more believable than the SRS. If the behavior is *not* intended to be in the UIRP, then we should believe the SRS more than the UIRP description of the behavior, *unless* we have seen reasons for the SRS to be considered flawed or not validatable. In any case, it is necessary to know the intents and the history of the development of the UIRP and SRS, that is, of the REing process that led to the UIRP and the SRS.

It is clear that which to believe, the UIRP or the SRS, cannot be answered *after* a disagreement has been found. If there is no documentation of intent, no access to the REs, when the UIRP and SRS disagree, there is no way to decide for sure which, if any, to believe. Therefore, it must be decided *before* beginning UIRPing, at the beginning of REing, what behavior is intended to be in the UIRP, to document that, and to set up tracing between the UIRP parts, the sentences in the SRS, and the sentences in a new document called the *intent specification* (IS). Thus, we have come back to the need for documentation. There is no escaping it.

3. Past Work

A thorough search of the literature has shown very little work addressing the problem of the meaning of a UIRP. This lack was quite surprising due to the fact that UIRPing is a well-known, widely adopted approach, which is discussed in numerous publications, e.g., the works of Lantz [3], Bischofberger and Pomberger [4], Pomberger and Blaschek [5], Connell and Shafer [6, 7], and Gould and Lewis [8]. A good source of early ideas on UIRPing is *Software Engineering News (SIGSOFT)*, 1982;7:2. Bowers and Pycock and Keil and Carmel discuss the pros and cons of UIRPing, particularly in fostering customer–developer communication and understanding. Please see [9, 10].

The problem of capturing the information a prototype contains is almost completely disregarded in these works. Two different partial solutions to the problem at hand were found. In the first, Hill [11] offers a method by which requirement statements can be attributed within models that allow for post-compilation extraction and analysis. It focuses mainly on the functional aspects of the prototype, thereby providing only a partial answer to the problem addressed by this paper. As will be explained later, a UIRP contains kinds of requirements information other than functionality that has to be extracted and stated. In the second, Kösters et al. [12] propose a requirement analysis method called FLUID, which explicitly captures the requirements of direct-manipulation UIs. The UIRP is regarded as only an executable model which helps visualize the current stage of analysis. They disregard other kinds of requirements information that a UIRP contains and other kinds of knowledge that the UIRP represents.

It is assumed that the reader is familiar with at least one approach to UIRPing from practice or the literature.

4. Solution Goals

In arriving at our solution to the problem of which of the UIRP or SRS to believe when they disagree, we do not want to invent YAREM (Yet Another REing Method). There are more than enough of them out there. Moreover, we want to be able to use whatever REing method, including its UIRPing method, we are already using. Most importantly, the problem addressed by this paper does not result from the lack of modeling methods, but rather from the fact that existing methods fail to identify the kinds of information that a UIRP contains and thus do not provide a way to capture and present that information.

Instead, the solution presented here consists of steps to be taken in addition to existing UIRPing and requirements modeling methods, to be applied *before* and *during* UIRPing and requirements modeling, to ensure that later, developers are able to answer the questions of what the UIRP specifies and what it does not. Given that the REing and UIRPing method *M* is being used, the general solution approach is to prepend an intent decision and documentation phase to *M* and add to all steps of *M* a tracing step. The intent decision and documentation phase yields the IS as its artifact. The tracing steps are to establish a trace of individual intents in the IS throughout the UIRP and the SRS.

5. The Proposed Solution

We present the steps of the solution method independent of any particular UIRPing and requirements modeling methods. Following this general description, some details of the case study UIRPing of the TSG system are presented, both to lend some concreteness to the solution approach and to help convince the reader of the effectiveness of the approach. Space limitations preclude giving all the details of the case study. The reader is referred to the first author's thesis for the missing details [13].

One *caveat* is in order. We do not advise on what to prototype and what not to prototype. What is prototyped and what is not are the specifier's decisions. This decision is made based on his or her experience in specifying other systems in the same or similar application domains; it is based on what is already well understood and what is not. We do advise on what to do once the decision is made so that the developers and maintainers know what of the prototype is intended and where to look, in the UIRP or in the RDs, for answers about their questions. The advice concerns documenting the choices and creating tracing links among the documents that help search for answers to these questions.

5.1. Steps of the Approach

The basic idea of the proposed approach is to tailor the existing UIRPing method by identifying, before UIRPing begins, the kinds of requirements information that the UIRP will contain and will not contain, and by choosing modeling techniques that properly represent this information. Tailoring a UIRP construction process is done in six recurrent steps, recurrent in that as one is following the steps in sequence, he or she can go back to any previous step to redo it based on new information.

1. Define the system's operational environment and its interfaces to other systems.
2. Identify to which application domains the system belongs.
3. Characterize the principal properties and the main features of an application belonging to these domains.
4. Identify which of these properties are applicable to the system under development.
5. Decide and document which of the identified properties requirements will be prototyped.
6. Prototype the system's interfaces and chosen properties.

5.2. Application Domains

The concept of application domains is an important one. All programs in a domain share common data, attributes, and operations. Examples of domains are simulation systems, information systems, data acquisition systems, and UI-intensive systems. An application can find itself in several domains, depending on the functions it must provide. The point about a domain is that from past experience, much is known about the data, attributes, and operations that are needed by any application in the domain. Often, there are even libraries of data definitions, procedures, functions, and even requirement specifications, that can be used by simple inclusion to simplify specifying and programming the application. For example, there are many GUI building libraries available to be used to build any UI-intensive application.

5.3. Requirements Information

The kinds of requirements information that a UIRP may contain can be classified into the following types:

1. the application's functionality and behavior, that is, its reactive nature, including constraints placed on this behavior and operational logic;
2. the application's data model, data dictionary, and data-processing capabilities;

3. the application's taxonomy along with a dictionary for it;
4. a partial specification of the application's interfaces to other systems; and
5. general knowledge about the application and its intent, including about
   - the current state, i.e., before the application existed;
   - the user's work environment;
   - the user's work procedures;
   - the application domain in general;
   - the design concepts;
   - the design goals;
   - the constraints;
   - the assumptions made;
   - limitations;
   - possible tradeoffs;
   - design options;
   - decisions regarding human engineering;
   - the operational concept;
   - why things were done one way and not the other;
   - how to use the application, i.e., a description which connects all the use-cases to the operational scheme; and
   - development artifacts.

UIRPing of the system interfaces and chosen properties is conducted in an iterative manner as recommended in many publications about prototype-oriented software development. Developers and users can go over this list of properties jointly in a systematic fashion, exploring them by means of interviews, modeling, implementation, demonstration, refinement, and validation. The requirements information can be grouped and organized according to the kinds of information that the developers want to discover, leading to the specification, in the IS, of the information the UIRP will contain. The developers and customers can address all related issues and, when they come to an agreement, state the requirements formally.

5.4. Evaluation by Case Study

Ravid took an almost completed project in which he had participated, and used it for an after-the-fact case study. Very early in this project, he had noticed the problem addressed by this paper. He formulated an approach that he thought would solve the problem. He carried out the approach on the almost completed project, redoing some problematic steps in the original development.

He got better results than the first time around, and he fixed the application. Ravid is convinced that had they followed the approach from the beginning, the problems would not have been as severe. Of course, in such an after-the-fact, introspective case study, we cannot be certain that he did not do better the second time around because of learning from the first-time mistakes. However, the results are good enough to warrant further and independent case studies from projects from the beginning. Since it is infeasible to do a large project twice, with and without the approach, not to mention that it would be impossible to factor out other influences in such a small sample size, these new independent case studies would also be introspective.

Much of what follows was discovered the first time around, but some of it, particularly the explicit decisions of what to prototype, the explicit lexicon, and the explicit statements of intents, were discovered and logged only the second time around. This newly discovered information makes the whole requirements exercise cleaner. Most importantly, this information makes it possible for developers to discern what is and what is not intended to be in the prototype.

The description of the UIRPing process follows the second time history and is written as if all that is described were discovered in this second time around. Please understand that the actual requirements were discovered the first time around and are unchanged the second time around. The sole difference is that this time, there is better documentation of the decisions leading to those requirements.

6. The Target Scene Generator Case Study

This section presents the chosen case study, a simulator for generating infrared (IR) scenes, called the Target Scene Generator (TSG) [14, 15]. The TSG, a medium-scale UI-intensive system, was delivered in June 1998 with over 200,000 lines of code. Ravid, the first author, was the software system engineer for the TSG. Ravid's work on this system led to his addressing the problem being considered in this paper. This work is also the origin of some of the concepts that are introduced in this paper.

6.1. Discovering the Problem

The development team (DT) and the customer and users (C&Us) found it difficult to arrive at a satisfactory requirements specification, because of a number of, not uncommon, factors:

1. The DT had never developed a similar system in the past.
2. The system combines problems from various disciplines, requiring multidisciplinary solutions.
3. The C&Us had difficulties defining their needs and requirements because the system was supposed to

completely change their work methods and supply them with a new set of tools and aids that they had never used before.

4. The traditional methods used by the DT for requirements elicitation were not suitable for this system and did not address all needed aspects of the system.
5. The C&Us were not familiar with any requirements modeling techniques, including the ones the DT used, i.e., data-flow diagrams, state machines, object models, ERDs, etc.
6. Since a simulation system that can generate IR scenes was to be developed, it was clear that the primary purpose of the system was to enable an operator to define, execute, and analyze IR scenarios.* The difficulties were to define a scenario and to define the language used to describe the scenarios.
7. It was clear very early to the DT that very soon after the deployment of the first version of the system, the C&Us will discover and present additional user requirements to the DT [16].
8. The sequential development approach the DT was trying to follow did not seem suitable. After a series of frustrating attempts to complete the requirements specification and to reach a steady-state requirements baseline, the DT realized that the development of an innovative, complex system, such as the TSG, requires an evolutionary approach. It was possible for neither the C&Us nor the DT to instantly grasp all the details necessary to understand and to properly specify the system to be built.

Because of the many difficulties faced during the early stages of the system specification, the team decided to rapidly develop a throwaway UIRP [17]. This UIRP was to improve the communication with the C&Us and to help complete requirements specification within a reasonable amount of time. Software engineers, human engineering people, and users group representatives were involved in the development of the prototype. This approach turned out to be successful. Some major misunderstandings with the C&Us and many contradicting requirements were discovered and fixed. As is recommended by many, including Fred Brooks [18], the prototype was thrown away, and the development of the production version was started from scratch.

Creating the TSG UIRP exposed a problem, which has shown up in other projects using UIRPing to identify requirements. Given an informal problem description, it is not obvious which of the many requirement issues should be explored in a UIRP. Thus, a haphazard set of issues are explored in the UIRP. After completing the UIRP, it is necessary to integrate the UIRP with the other models of the system that are expressed in the written RDs. Even as specifiers do attempt this integration, unfortunately, a portion of the implicit information embodied in the UIRP is left implicit. It is not until the UIRP is used to answer questions that this implicit information is discovered. By the time the discovery is made, it may be difficult to recover enough of the rationale in order to determine what the implicit information should be. Instead, the UIRP and the RDs disagree, and the developers are left trying to figure out what the requirements should be. This problem was discovered unexpectedly while the DT was preparing the project to be reviewed for ISO 9000.3 compliance by the Israeli Standards Institute. Ravid presented the project, the UIRP, and the software development documents produced up to the day the review was conducted. The UIRP provoked additional questions about the system. Obviously, the UIRP was used to answer these questions. After Ravid finished answering these questions, one of the reviewers asked "Where is all this information written?" Part of the information appeared in the SRS, part was expressed indirectly by other statements in the SRS, part was written in the other documents, and part was not written anywhere even though it was known, understood, and agreed upon by all the people involved in the project by virtue of their having worked together to produce the UIRP. This undocumented information included indispensable knowledge about the system, knowledge which seemed to be essential for new programmers joining the group and for maintenance personnel who will have to support the system in the near and far future.

6.2. TSG as a Case Study

Although systems like the TSG are not that common, the TSG possesses many properties of more common systems. Therefore, it is a good source for instances of the problem at hand. Much can be learned from the experience gained during its development. The project is too big to be covered completely by this paper. Therefore, only portions of it are described.

In the following subsections, a short description of the TSG system is given, and the usage of the approach described in Section 5 is illustrated by applying it to portions of the project.

6.3. The Target Scene Generator System

The TSG system is a physical effect simulator that generates high-fidelity, multi-object IR images. An IR image generated by the system is composed of several objects that represent targets, decoys, and a background. The

* The plain word "scenario" here and in the rest of the article means IR scenario. When a scenario, as an instance of a use-case is meant, "UC scenario" is used.

system allows independent control of the appearance of and the location of these objects at any position within a large field of view. IR images are generated by continuously changing the radiometric and dynamic characteristics of the simulated objects in a controlled fashion. The primary purpose of the system is to serve as a research aid for testing and evaluating the performance of electro-optical missile seekers.

The main type of components of the TSG system are:

- optical elements, both dynamic and static;
- various types of IR radiation sources;
- several types of electro-mechanical, servo-controlled units;
- servo-control systems;
- a computer system consisting of several computers, including servo controllers, radiation source controllers; special-purpose embedded systems, and standard PC-based workstations; and
- software on the various computers.

A system-level analysis and design process preceded the software development process. Some of the products of this process are the conceptual computer system structures depicted in Fig. 1; a set of system-level requirements, some of which deal with software; and some system-level design principles and fundamental decisions that significantly influenced the software requirements and formed the basis for the software requirements analysis and specification. The most important among these principles are:

- The entire TSG system operation is controlled by computers.
- The system supports manual as well as automated operation of all the controllable elements.
- UI software controls replace hardware controls such as knobs and meters and emulate their operation.
- All the controllable elements support local control as well as remote control.
- Local control is used for start-up and maintenance purposes only; everyday operation is done using remote control.
- The system is operated from a single computer, called the host computer, with the aid of a keyboard, a mouse, and a modern graphics-based human-computer interface (HCI) system, such as Windows™ or Motif™.
- Standard off-the-shelf technologies, hardware and software, are used as much as possible.

In order to explain some of the key terms mentioned above without getting into technical details, the analogy of a video cassette recorder (VCR) can be used. Local control corresponds to the VCR's front panel. The front panel of a VCR enables the user to perform basic VCR operations. Remote control corresponds to the VCR's remote control. Remote control enables the user to perform most of the operations supported by local control and much more. In manual operation, each operation is done by the user's activating the remote control keys. In automated operation, the user can define a sequence of operations that the VCR will perform on its own, for instance, a time-triggered sequence of recordings. The users program a sequence of dates, times, channels, and durations. The VCR is turned on automatically at the specified times, performs the recording from the selected source, and then turns itself off when done. The TSG system can be compared to a recording studio, with several VCRs and lots of other related equipment, all of which are operated by a technician from a single computer-based post, as opposed to having a pile of remote controls as do most of us.

From this description, it should be clear that most of the TSG system's HCI is concentrated in the host computer. The host computer is supposed to serve the system operator and to support the operation of the entire system from a single post by means of a keyboard and a mouse. Consequently, the host computer is a UI-intensive system and is, therefore, a potential subject for UIRPing, as indeed it was. There was nothing unique in the process of UIRPing itself. It was conducted in the manner recommended in many publications about prototype-oriented software development, and eventually the prototype was thrown away.

6.4. Applying the Solution Approach to the TSG

This section describes the use of the solution approach outlined in Section 5.1 to carry out the redesign of the requirements of the TSG. Recall that the approach consists of six steps. The details of these steps are described in the following subsections.

Please note throughout these subsections:

- how the specification of the scenario-definition portions of the prototype can be made systematic;
- how requirements are presented by the prototype;
- how requirements are extracted from the prototype; and
- how the prototype relates to other requirements models.

Finally, note how the new information in these subsections allows developers to determine what is intended to be in the prototype and what is not.

6.4.1. Step 1

The first step of the approach is to define the TSG host computer's operational environment, which is illustrated in Fig. 1. The context diagram of Fig. 2 shows the scenario-related TSG host interfaces. The interfaces from the users to the control system and to other software tools are the primary interfaces that make it possible for the user to

define, execute, and analyze scenarios. UIRPing is intended to help define these interfaces. In fact, the interface to the software tools for generating scenarios and the interface to special-purpose data-analysis tools did not exist in the original customer documentation. They were discovered, specified, and agreed upon with the aid of UIRPing. The data elements that are at either end of each arrow representing a data-flow in Fig. 2 were discovered solely with the aid of the prototype.

From the scenario-related requirements, we identified one kind of user, the researcher who uses the TSG system to test and evaluate the performance of the unit under test (UUT). From what we learned from developing the UIRP, we decided that the TSG should enable this kind of user to perform the following tasks:

- control the system hardware setup;
- define scenarios;
- import scenario definitions prepared with the aid of other simulation software;
- maintain a library of reusable scenario definitions and scenario component definitions;
- check scenario definition validity;
- execute scenarios;
- acquire UUT signals and control system feedback;
- analyze scenario definition and acquired data;
- manually control all the controllable elements;
- export data acquired by the system to other data analysis tools; and
- perform basic trouble-shooting operations in case of a hardware malfunction.

Each of these tasks is represented by one or more UC scenarios [19, 20]. Their combination constitutes a preliminary list of the capabilities available to the TSG system's users, and is thus, the abstract, top-most view of the TSG's requirements.

6.4.2. Step 2

The second step is to identify the application domains with which the TSG host shares attributes. The following domains were identified:

- simulation systems;
- information systems;
- data acquisition systems;
- motion and instrument control systems;
- real-time simulation system; and
- UI-intensive systems.

6.4.3. Steps 3 and 4

Step 3 is to characterize the properties of the application domains, and Step 4 is to identify which of these properties are applicable to the system under design. After characterizing the properties of the domains, the attributes shown in the large boxes down the middle of Fig. 4 were found to be applicable to the TSG system.

6.4.4. Step 5

Step 5 is to choose from among the identified properties those that will be prototyped. The list of attributes developed in Step 4 leads to the TSG host UIRP development process model depicted in the whole of Fig. 4. The symbol "√" marks the attributes that were chosen to be prototyped. Note that the REing does not end with these prototyped attributes. It is necessary to devise requirements for *all* attributes. However, it is best to elicit the requirements for the attributes not prototyped *after* the UIRP has been written, when UIRPing has been played out, and when the UIRP can be used to help decide what the other requirements should be. The process model illustrated in Fig. 4 does not add to the information already revealed by the list of properties attributed to the TSG system. Rather, it simply details the REing process for the whole TSG system. This detailing is what is meant by the notion, introduced in Section 5.1, of tailoring the UIRPing process to the application.

Each of the listed properties is related to a complete set of questions that has to be asked. The questions represent general issues that should be discussed and that will eventually lead to the specific system requirements, for instance:

- "What is a scenario?"
- "What are the primary scenario components?"
- "How should the behavior of a scenario component be defined?"
- "Which data should be monitored and recorded in order to monitor scenario execution?"

and many others. The answers to these questions constitute a list of specific system requirements to be specified.

The point is that these questions can be grouped and organized according to the kinds of information that the developers want to discover. This information, which is also the kind of information the UIRP and its documentation will contain, includes:

- *Intent*: Why are these requirements needed? What are the main concepts on which the requirements will be based?
- *Functionality*: What functionality does the user expect

the system to provide? How does the user expect the system to react? What does the system have to do in order to make the user requests happen?

- *Application data*: What data have to be provided by the user in order to fully define a scenario? How does the system validate the data and what does the system do with these definitions?
- *Taxonomy*: Which terms does the user employ in order to interact with the system? Which language is used to describe user–system interaction? Which terms are used to describe scenarios?
- *System interfaces*: How does the user expect to interact with the system? What is required from the system interface in order to make happen what the user requested? What kinds of displays does the user expect the system to provide?

Developers and users can systematically and exhaustively go over this list of properties. They feed their joint discussions with information obtained by modeling, implementation, demonstration, refinement, and validation. They can address all related issues. When they come to an agreement, then they can state the requirements formally.

The list of user tasks and the corresponding UC scenarios form a unique view of the required functionality, a view of the TSG system in use. They bind the properties attributed to the system to an operational scheme that explains for what the system is good, why these capabilities are needed, and how these tasks will be performed with the aid of the system. The structured list of requirements resembles a partial table of contents of the system's SRS, and the basis and the origin to which requirements will be traced. The list organizes requirements according to topics. It offers the same view of the system requirements as the view offered by a user's manual. This view is intuitive to users who are trying to understand the system. The resulting process model enables developers to carry out prototyping explicitly as part of a fully traced requirements elicitation process in which it is decided and documented ahead of time what aspects, usually in the UI, are being modeled in the UIRP. Requirements information about issues that are not included in this list has to be sought elsewhere. The tracing links allow easy access to the documentation of any decision so that in the future, when one is following the trace links to track down an answer to a requirements issue, he or she sees the explicit decision and knows whether to consult the prototype or another document in the requirements specification suite.

6.4.5. Step 6

Step 6 is to prototype the system's interfaces and chosen properties. Doing so involves carrying out UIRPing under control of the process defined in Step 5, while tracing each requirement through the UIRP and the RDs. UIRPing has its own steps.

The first step of UIRPing was to decide where to start. The main idea was to pick a central issue and evolve from it. It was expected that the discovery of less central issues would follow. The main goal of the host computer's UI was to enable the TSG system's user to define, execute, and analyze IR scenarios. After all, the project's RFP and, therefore, each response by the DT to the RFP included the following requirement: "The system will enable an operator to define scenarios, execute them and analyze the results of the executed scenarios".* Consequently, scenario definition seemed like a good starting point for UIRPing. The purpose of UIRPing was to understand these vague requirements in sufficient detail to specify them straightforwardly and completely. The UIRPing started by defining the use cases. These use cases had to say at least;

- what is a scenario definition;
- what does the user define;
- what elements a scenario is composed of;
- whether the definition for each element is time dependent;
- what is the element data type;
- which values are legal and which are not legal;
- how to check the validity of the definition;
- what processing has to be done in order to translate this definition to commands understood by the servo-control system.

The use cases had to describe the UI by which the scenarios would be defined, saying which options will be available and which operations will be allowed in each step, and defining the dialog boxes, menus, toolbars, and feedback displays, etc.

At this point, due to space limitations, the description of the case study must focus on only part of the rather large specifications that were eventually developed. We focus on the parts dealing with IR scenario definition and validation. The next subsections give the relevant

- UC scenario descriptions;
- HCI;
- lexicon entries;
- functional models; and
- intent.

* This sentence is typical of sentences that can be found in many customers' requirements documents. It asks for a lot but says very little.

6.4.6. The IR Scenario Definition Use-Case Scenario Descriptions

Due to space limitations, this section shows the detailed definitions of only two UC scenarios, namely those concerned with IR scenario definition. It shows only the skeletons of the definitions of other sub-UC scenarios referred to by the detailed definitions. The description is given in the form presented by Leite [21]. The bold-faced words represent terms from the lexicon, and Italicized words represent UC scenario and sub-UC scenario names.

TITLE: *Define a **scenario***
OBJECTIVE: To define to the system a **scenario** to be executed
CONTEXT: The system must be running in off-line mode.
ACTORS: **User**
RESOURCES: The TSG host software, a workstation
EPISODES:

The user chooses to define a **scenario**.

The system enters the **scenario**-definition state.

The user can either

- *Load a preexisting definition from the library* and edit it, or
- *Create a new **scenario** definition* and edit it, or
- Edit the **scenario** already loaded (if one exists).

The **scenario** definition is displayed.

The user may edit the **scenario**'s general properties, i.e. name, description, and duration. He may also add or remove **simulated objects** such as a **main target** and **decoys** to or from the **scenario**, and *edit* each of the defined ***simulated object** properties*.

After completing, the user can ask the system to accept the definition and end the **scenario** definition activity.

In response, the system checks the validity of the defined **scenario**.

If the definition is valid, the system accepts it and exits the **scenario**-definition state. Otherwise, the system refuses to accept the **scenario** definition. It displays a list of all the **scenario validity checks** that failed in an error message window and returns to the **scenario**-definition state. The user can fix the definition and ask the system once more to accept the **scenario** definition.

At any time during this operation, the user can ask to *save the definition in the library* for future use.

In any time during this operation, the user can ask to cancel the operation and exit the **scenario**-definition state. In response, the system revokes all the changes made by the user, restores the definition that existed prior to the time the operation was initiated, and exits the **scenario**-definition state.

TITLE: *Load a preexisting definition from the library*
OBJECTIVE: To load a definition of a **scenario** or a **simulated object** from the library for reuse.
CONTEXT: The system must be running in the edit-**scenario** state.
ACTORS: **User**
RESOURCES: The TSG host software, a workstation
EPISODES:

The user chooses to load a definition from the library.

The system issues a warning to the user if the current definition (if one exists) was not saved, and asks the user if he wants to save it. The user can choose to ignore the warning, *save the definition in the library*, or cancel the operation.

If the user did not cancel the operation, the system enters the load-definition state.

The system displays a list of all the definitions that exist in the library and asks the user to choose one. The definition are ordered by name. The definition's identifying name is given by the user when he decides to *save the definition in the library*.

The user can browse the list and get a preview of the definition.

After the user made his choice, the system loads the chosen definition overwriting the existing definition (if one exists) and displays it.

At any time during this operation, the user can ask to cancel the operation, exit the load-definition state, and return to the edit state.

TITLE: *Create a new definition*
OBJECTIVE: To create a new empty definition of a **scenario** or a **simulated object**.
...

TITLE: *Edit **simulated object** properties*
OBJECTIVE: To define **radiometric properties** and **dynamic properties** of a **simulated object**.
...

TITLE: *Save a definition in the library*
OBJECTIVE: To save a definition of a **scenario** or a **simulated object** in the library for future use.
...

6.4.7. The Scenario Definition Human–Computer Interface

Figures 3, 5, and 6 illustrate simplified versions of some of the scenario-defining forms. The arrows that connect the forms represent the user's actions and the transitions between the forms. A picture is worth a thousand words. Thus, the presentation of the UIRP's display output simplifies and shortens the description of the UC scenarios. A portion of the functionality is implied by the UI. The use of UI conventions and standard UI designs obviate the need to textually describe some of the intended behavior.

The UIRP demonstrates the UC scenarios, explains them, complements them, and eventually becomes an indispensable part of them. Because the UIRP is so strongly related to the UC scenario-based requirement model, the lexicon, additional functional models, the data model, and all the other models can be produced based on the information extracted from the UI.

6.4.8. The Scenario Definition Lexicon

Leite [22, 21] states that the objective of populating a language-extended lexicon (LEL) is to promote understanding the problem language without requiring full understanding of the problem. The goal of UIRPing is to understand the problem. Thus, the application lexicon is discovered simultaneously with other requirements information, and at the same time it is reflected in the other requirements models and the UIRP. To demonstrate this duality, a subset of the scenario-related lexicon entries is given in the form presented by Leite [21]. Details for terms not needed for the detailed use-case definitions are omitted.

Scenario

- Notion
  - — a sequence of IR images projected on the focal plane of the UUT.
  - — represents a real-life IR scene viewed by a seeker.
  - — also called an IR scenario.
- Behavioral Response:
  - — A scenario is composed of one or more simulated objects and a background.
  - — The system maintains a library of scenario definitions.
  - — The actual scenario duration is determined by the user.
  - — The system executes only valid scenarios, namely scenarios which have passed the scenario validity checks.

Scenario duration

- Notion
  - — the scenario length measured in seconds.
  - — the time elapsed since scenario starts till scenario ends.
  - — the total time that IR images will be generated and projected.
- Behavioral Response:
  - — A scenario duration should be greater than zero and less than 120 seconds.
  - — The scenario definition must be valid throughout the scenario duration.
  - — UUT signal and control system feedback are acquired at a rate of 200Hz throughout the scenario duration.

Simulated object

- Notion
  - — simulated entities representing real-life IR images such as targets and decoys.
  - — also called scenario components.
- Behavioral Response:
  - — A scenario is composed of one or more simulated objects and a background.
  - — A simulated object is defined by stating the way its dynamic and radiometric properties vary over time.
  - — The system maintains libraries of all kinds of simulated objects definitions.

Main target ...

Decoy ...

Dynamic properties ...

Radiometric properties ...

Motion mode ...

Scenario definition validity check

- Notion
  - — an overall test that checks that the complete scenario definition is legal, meaning, that it can be executed by the system.
  - — checks that the combination of simulated objects is legal and that each of the defined simulated objects validity check has passed.
- Behavioral Response:
  - — A scenario definition validity check is performed automatically when a user asks to execute a scenario or upon a user's request.
  - — If a scenario definition validity check fails, it produces a list of all the checks that failed.
  - — The system executes only scenarios that pass successfully this check.

Simulated object definition validity check ...

It is recommended to maximize the use of words from the lexicon when describing scenarios. In the beginning, the combination of UC scenarios and the UIRP seems to be describing the requirements very clearly and understandably. However, only after reading the lexicon entries description does it become apparent how much information was not included and cannot be deduced from either the text of the UC scenarios or the UIRP. The lexicon enhances the clarity and readability of use-cases. It minimizes ambiguities that might be caused by the use of natural language to describe functionality, by basing the scenario descriptions on a minimal set of well-defined, unambiguous terms. It helps ensure that the same term is used each time a particular concept is mentioned.

The combination of the UC scenarios, the UIRP, and the lexicon forms the primary top-most abstract view of the TSG system's requirements, the intended user's view. The combination helps understand the purposes of the TSG system and facilitates understanding other requirements models. The next level of detail requires producing functional models and a data model. These models are the subject of the next subsection.

6.4.9. The Scenario Definition Models

The functional models are almost trivial for the scenario definition example. Most of the functional model is implied by the use of modal dialogs and standard event generators such as push buttons. Thus, there is no need to create such a model for this portion of the system. For the rest of the system, the functional models include data models and data-flow diagrams, which are not shown here due to space limitations. Their link to the UI is straightforward. In order to complete the specification of the data-related requirements, the following have to be specified:

- the processing performed by each of the bubbles that appears in the data-flow diagrams, either by means of structured English or by means of pseudocode;
- the complete definition of each of the data entities that appear in the entity-relation diagram and in the data dictionary;
- the I/O masks that filter out illegal user inputs;
- the algorithms for translating the scenario definition to commands to the control system;
- the algorithms for translating the feedback acquired from the control system to the actual scenario; and
- the algorithms needed in order to perform the extensive scenario validity checks that are required by the users.

6.4.10. Information about Other System Interfaces that is Implied by the UI

The TSG system interfaces are listed in Fig. 4 as system-specific properties. The UI partially hides these interfaces from the system users. The requirements information about them is discovered through the UI displays in the UIRP, through the flow of information from and to the system, and through the description of the system functionality. The system interfaces are part of the functional entity being specified, and they make the functional specifications happen.

The interface to software tools for generating scenarios is an example of an interface that is expressed by the UI. The interface to the control system is an example of an interface that is not expressed directly by the UI but can be specified with the aid of the UIRP. The TSG host translates an IR scenario definition to commands for a servo system. Since the servo control is done by another computer, there is a real-time interface between the two computers. The servo system is not aware of the fact that when it executes a scenario, it is executing only a sequence of coordinated motion commands. During and after scenario execution, the user can ask the TSG system to display feedback information about how well the scenario was actually executed. The scenario-related UI and operational logic helped define this interface.

6.4.11. The Intent Behind the Scenario Definition Models

A description of the intent is needed in order to complement the requirements models.

- It was decided that the TSG system would support a work flow that resembles the work flow of a compiler-based development environment. The user's work cycle will look like:

  ```
  edit <>;
  check (compile) <>;
  run and monitor execution <>;
  edit <>; ....
  ```

  The users demanded that the system would not allow executing a scenario unless it passes a complete validity check in order to avoid uncertainties that might be caused by the unexpected results of an illegal scenario execution. The only uncertainties that were allowed were the ones related to the unit being tested. This constraint required us to define an extensive set of validity checks that can be run on a scenario before executing it and a mechanism for producing meaningful error messages in case the checks fail.
- The users asked for a visual scenario definition language rather than a textual language. They did not want to be obliged to learn the syntax of a textual language and have to remember it for the 10 to 15 years that the TSG system was supposed to serve them.
- The primary role of the library of scenarios and scenario components was to enable the user to assemble scenarios from a reusable set of definitions. It was planned that the users will initially invest a lot of work in creating a set of reusable component definitions. In the future, the task of defining an IR scenario will become much simpler and quicker, because all the users will have to do is to create a new scenario or load an existing one, add or remove predefined components to the scenario, check the new combination, and execute it. Consequently, the TSG system has to support storing and retrieving of scenario components and complete

scenarios to and from the library, and to possess library management capabilities. That is, the system is required to possess some attributes of an information system.

- We made a distinction between two data sets, the *expected* scenario and the *actual* scenario. The expected scenario is the scenario the user defines. The actual scenario is that performed by the TSG system, and it is slightly different from the expected scenario because the control system is not optimal. Therefore, the TSG system must track errors. The expected data set is required in order to analyze the planned scenario, and the actual data set is required in order to know how well the system performed the scenario. The performance of the UUT is evaluated relative to the actual scenario. Users have to know for sure that when they observe a certain response from the UUT, it is due to the scenario defined and not due to an artifact caused by the control system. Therefore, there is a requirement to continuously monitor and record scenario execution in real time, a requirement to provide data analysis tools that support the comparison and analysis of acquired signals from several sources, and a requirement to provide on-line data monitoring capabilities. That is, the TSG system is required to possess some attributes of a data-acquisition system.

All of this intent information is written up and stored. Each item is linked to the relevant artifacts in the other documents.

It is clear, from these few examples, that the dimension of intent contributes to the clarity and understandability of the software requirements. It offers the "why" information that is very important for someone who is trying to understand the TSG system's requirements. Without this information, the requirements specification is incomplete. It is impossible to extract this kind of information from the UIRP or other requirements models in the absence of knowledge of how the UIRP was developed.

6.4.12. Summary

In the end, the requirements specification for the TSG had been redone. Now however, besides the SRS and the UIRP, which had been produced before, there were other documents that did not exist before. These new documents include the UC scenarios, a lexicon, functional models, and most importantly, an intent specification that states what behavior is in the UIRP and what is not. Moreover, there were links between the items in these documents to allow developers to trace any requirement throughout these documents. The origin of the links is the structured list of requirements that is described in Section 6.4.4. Now the developers are able to answer any question that might arise about intent and to deal with inconsistencies between the UIRP and SRS with a hope of deciding as intended by the specifiers.

The documents were produced using MS Word. They were stored under the source-control features offered by MS Source Safe and were backed up by the routine backup procedures of the organization. The latest version of the documents were available also in hardcopy format from the project's Software Development File. The tracing links were implemented using a traceability matrix in the various documents and in the project's context-sensitive help files.

The portion of the case study documents presented in this paper deals with only that part of the TSG concerned with IR scenario definition. The IR scenario definition example is a typical one. It represents a pattern that repeated itself for nearly all of the TSG system properties that were prototyped. It demonstrates the usability of the method proposed by this paper and the quality of the requirements specification it produces. Having such a specification satisfies the needs of all the customers of the requirements specification process, especially those of the system users and software maintenance personnel. This portion should be enough for the reader to see how the approach works, to judge whether it is worth applying in other situations, and to carry out the approach on other problems. The first author's thesis [13] gives additional portions of the case study documents for the benefit of those who need more information.

Some comments are necessary:

1. Some of the information appears more than once or is expressed in more than one way. Although this duplication appears redundant, it is necessary because it reflects multiple perspectives, and because none of the individual models represents the system requirements completely.
2. For some parts of a system being specified, one or more of the models are minor, while others are major. Which is which depends on the modeled aspect. In this example, the functional models are almost trivial. Most of the function is implied by the use of modal dialogs and standard event generators such as push buttons.

6.5. Lessons Learned from the Host Prototype Construction

Before discussing the lessons learned from building the TSG host computer prototype, it should be said that UIRPing worked for us. The primary benefits of the UIRP were the improved communication with the customers, the completion of the requirements elicitation process within a reasonable amount of time, and the validation of the basic

conceptual requirements and the concepts on which the design was based. In the course of prototyping, we discovered some major misunderstandings with the customer and many contradicting requirements. Actually, most of the first prototype was thrown away because it was totally wrong. We started the development of the production version from scratch based on a second prototype.

Having said this, we now consider the primary lessons learned from the UIRP-oriented development of the host computer:

- The lack of communication with the users at the beginning of requirements elicitation exposed two serious disadvantages of our manual work methods.
  1. Unless customers sufficiently understand the requirements modeling technique, and the developers can see that they do, it is almost impossible for the developers to be convinced that the customers really mean what they say when they say "This is what we want." More often than not, they really mean, "It seems that you did a professional job. However, we do not understand most of what you wrote. On the other hand, it does mention most of the things we asked for." Even when there was someone who could understand and validate the models we presented, he generally expressed only his own opinion. We were interested in hearing the opinions of other user representatives as well.
  2. It is pointless to create complete and consistent models for the wrong requirements. A model, even if complete, consistent, and very formal, can easily represent wrong requirements. During the first stages of requirements elicitation, it turned out to be more efficient to use high-level, architecture-level methods in order to understand the overall picture and to find the primary details of this picture. The use of more precise methods can be postponed until later stages, after the overall picture is clearer. These methods can be used in order to find and define the finer details. Thus, a top-down hierarchy of requirements modeling techniques is recommended.
- The classification of the application according to domains with which it shares properties proved to be useful. It helped to simplify the complexity of the overall software requirements by classifying and grouping requirements. It allowed us to focus on the TSG system's unique properties. The more standard properties were much simpler to elicit since we could literally copy requirements, and the users were already familiar with similar properties. It led to requirements reuse, which, in turn, led to software reuse, because we could purchase standard software components that address standard domain requirements. It also assisted in making prototype construction systematic. We could conduct a systematic process according to topics, i.e. scenario definition, data acquisition, scenario execution control, on-line and off-line analysis, etc.
- It was stated earlier that a prototype, as a tangible model, is known to cause inflation of functional requirements. This effect is so commonplace that it is considered one of the drawbacks of requirements prototyping [23]. It would seem that the reuse of standard development environments and requirements from application-related domains would worsen this effect. It is thought that using this approach would lead developers to voluntarily implement unnecessary function for which they will never be paid, especially if the project budget is already determined. We learned that this is partially true. Users do seem to always ask for more, no matter how the requirements are being gathered. Prototyping does tend to exacerbate this effect. However, most of these new requirements are requirements that, in a conventional development approach, would have been discovered much later, after the users start to exercise the system. The problem of which requirements to accept and which not, or which requirements are within the scope of the development effort and which requirements are not, is a managerial contractual issue that has to be resolved in any case. Moreover, late discovery of requirements can undermine the system's user friendliness and adequacy to users' needs. Creating a non-user-friendly, non-useful system can kill a project just as easily as being overdue and over budget. A requirement discovered later costs more to implement than the same requirement discovered earlier.
- The application's taxonomy is very important. It helps to reveal inconsistencies and contradicting requirements. It contributes to the application's user friendliness. Terms, whether well or poorly chosen, live far longer than expected. Terms can be found in the UI, the requirements models, the requirements statements, the help system, the user's manual, and the code, in the form of class names, variable names, function names, etc. Poor choices are very hard to change, because a change creates a huge ripple effect and requires so many other changes. What seems to be vague or nonintuitive now will always remain that way and will undermine the readability of the requirements and design and the system's ease of use. We had several examples of poor choices of similar terms that expressed totally different concepts. For example, "flare sequencer" and "sequential flare" sound alike, and "flipping mirror", "switching mirror", and "transition mirror" sound alike. Yet, each of the six terms represents a different concept. We always mixed the sound alikes, never remembered which is which, never remembered the exact meaning,

and often talked about one when we meant another. These inconsistencies and duplicates originated from the requirements statements and the prototype. They existed because we did not pay proper attention to lexicon issues. The lexicon contributed to reducing this problem. It enabled the users to use the terms, view their usage, and comment about their inadequacy when they exercised the UIRP and when they discussed the requirements. The design of the UI itself, which was based on the principle of uniformity, helped reduce duplicates, but it did not prevent us from uniformly using wrong or inadequate terms.

- The UIRP cannot serve as the only requirements model. Other models are needed, in particular, to describe intents and functions. Otherwise, extracting requirements from the prototype becomes a reverse engineering activity, even if the prototype were complete.
- We observed another possible side effect of the proposed method. Existing methods produce requirements documents that are unreadable even to some of the people involved in the requirements specification process. It appears that the documented explicit decisions of what is intended in the UIRP and the tracing links between the intents and the various documents help make all the documents more readable. Certainly the consistent use of the customer's language with the help of the lexicon contributed to increased readability. Finally, it is possible that the use of less formal methods, such as UC scenarios, helped.

7. Conclusions

There is a problem in evaluating the benefits of research work in the field of software engineering. We rarely, if ever, get the opportunity to develop a project larger than a toy in more than one way and to compare the implementations. Certainly it will not happen in a $6,000,000 project. Even if we do get such an opportunity, the various implementations are hard to compare for several reasons. The opportunity typically arises only when the first implementation fails and then, if we do better in the second trial, it might be thought that the second way is more successful. In this case, the first version of the system becomes a very expensive throw-away prototype of the deliverable system. However, conclusions cannot be drawn comparing the methods since it cannot be certain that the second-time success is not due to learning in the first-time failure. What can be done is to exercise a proposed approach over a relatively long period of time on several projects, gather information that can assist in evaluating the benefits of the approach, and compare this information to information gathered from past projects and projects that followed a different approach.

Still the question remains as to how valuable the new intent information is. Since the case study was done after the fact, we have no data on how helpful the new information was to the requirements specification process and to the subsequent system development. However, there are several concrete indications of the high value of the new information.

1. After the first development, when the project went into what is commonly called the maintenance phase, most of the original development team was replaced by new programmers. We believe that it was easier for the new programmers to get a good understanding of the system than is normally the case. They got up to speed more speedily than normal. Moreover, the new programmer's increased understanding paid off when they had to identify wrong requirements. Their understanding of the "why" made it easier to tell when the "what" was wrong or not good enough.
2. After the first delivery, Ravid was involved in preparing a course for the users of the new system. The course material consisted mostly of intent and specification information. It took Ravid only one day total time to prepare a two-day course, because all of the needed information was there. Had he not had the intent information generated during the case study, it would have taken considerably longer to prepare the course.
3. Ravid was recently approached by people from the company that built the TSG. They were trying to redesign parts of the control system and were having difficulties specifying what they needed to do. Ravid referred them to the intent documents, and they used this information to get onto the right track. They had not been able to understand the specification that existed before the case study and that had been used in the original development. They found the specification too hard and too non-standard, and their questions were all "why" questions. They had even proposed some inadequate solutions, whose inadequacy became clear once they understood the intents.

The bottom line is that the additional documents produced in the case study made the requirements specifications more readable and more understandable.

This paper presents the problem encountered in the course of development of a recently delivered project, and the research that was conducted afterwards in order to find an approach that will address some of these problems. Therefore, the required information for a full evaluation is not available. Thus, we are left with a lessons-learned, introspective evaluation of a one-time case study.

The TSG host computer UIRPing example is presented in order to illustrate the reasoning and the systematic

process, and an example is given. One example is sufficient to demonstrate the use of the approach proposed by this paper, since the pattern illustrated by the example repeated itself for the rest of the UI-related software requirements.

This paper does not intend to create the impression that once developers follow the proposed approach, all their elicitation and specification problems are solved. Like most of the work in software engineering, this work is about techniques that assist in dealing with software development problems. The real problem of finding what has to be done is still left to be solved anew by developers for each project. We hope that the lessons learned, the examples, and the demonstrated usage given in Section 6 show

- that this approach is useful;
- that it yields valuable results from which other prototyping-oriented projects can benefit;
- that it addresses the problems described in Section 2; and
- that it helps to decrease these problems.

The illustrated modeling method captures all the important properties of the system and provides sufficient information about requirements that adhere to the concept of intent specification [24]. Presenting requirements in additional models would be unnecessary, because as Leveson [24] states, attempts to include everything in a specification are not only impractical, but are wasted effort, and are unlikely to fit the budgets and schedules of industrial projects.

Acknowledgments. We thank Dr. Ephi Pinski from the Missile Division Electro Optical department at Rafael, who made it possible to publish the work done for the TSG project. We thank also the referees who have read earlier versions of this paper and provided many comments that showed us how to improve this paper.

Berry was supported in parts by a University of Waterloo Startup Grant and by NSERC grant NSERC-RGPIN227055-00.

References

1. Working papers from the ACM SIGSOFT rapid prototyping workshop, Software Eng Notes (Special issue on rapid prototyping) 1982;7(5)
2. Easterbrook, S and Nuseibeh, B. Managing inconsistencies in an evolving specification. In Proceedings of the second IEEE international symposium on requirements engineering, York, UK, March 1995, pp 48–55
3. Lantz, KE. The prototyping methodology. Prentice-Hall, Englewood Cliffs, NJ, 1986
4. Bischofberger, W and Pomberger, G. Prototyping oriented software development: concepts and tools. Springer, Berlin, 1992
5. Pomberger, G and Blaschek, G. Object orientation and prototyping in software engineering. Prentice-Hall, Englewood-Cliffs, NJ, 1996
6. Connell, JL and Shafer, LB. Structured rapid prototyping. Yourdon/Prentice-Hall, Englewood Cliffs, NJ, 1989
7. Connell, JL and Shafer, LB. Object-oriented rapid prototyping. Yourdon/Prentice-Hall, Englewood Cliffs, NJ, 1995
8. Gould, JD and Lewis, C. Designing for usability: key principles and what designers think. Commun ACM 1985;28(3):300–311
9. Bowers, J and Pycock, J. Talking through design: requirements and resistance in cooperative prototyping. In Proceedings of the 1994 computer-human interaction conference (CHI'94), ACM SIGCHI, New York, NY, 1994, pp 299–305
10. Keil, M and Carmel, E. Customer-developer links in software development. Commun ACM 1995;38(5):33–44
11. Hill, M. Parasitic languages for requirements. In Proceedings of the second international conference on requirements engineering, IEEE Computer Society, Colorado Springs, CO, 1996, pp 69–75
12. Kösters, G, Six, HW, and Voss, J. Combined analysis of user interface and domain requirements. In Proceedings of the second international conference on requirements engineering, IEEE Computer Society, Colorado Springs, CO, 1996, pp 199–207
13. Ravid, A. A method for extracting and stating software requirements that a user interface prototype contains. M.Sc. Thesis, Faculty of Computer Science, Technion, Haifa, Israel, March 1999, Available at ftp://www.cs.technion.ac.il/pub/misc/dberry/alon.ravid/Thesis.doc
14. Pinski, E and Sturlesi, D. Generation of dynamic IR scene for seekers testing. In Proceedings of SPIE (international society for optical engineering) infrared technology and applications XXIII, Orlando, FL, April 1997, pp 20–25
15. Sturlesi, D and Pinski, E. Target scene Generator (TSG) for infrared seeker evaluation. In Proceedings of SPIE (international society for optical engineering) technologies for synthetic environments: hardware-in-the-loop testing II, Orlando, FL, April 1997, pp 111–119
16. Lehman, MM. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 1980;68(9):1060–1076
17. Andriole, SJ. Fast cheap requirements: prototype or else!. IEEE Software 1994;14(2):85–87
18. Brooks, FP Jr.. The mythical man-month: essays on software engineering (2nd edn). Addison-Wesley, Reading, MA, 1995
19. Jacobson, I. Object-oriented software engineering. Addison-Wesley, Reading, MA, 1992
20. Douglass, BP. Real-time UML: developing efficient objects for embedded systems. Addison-Wesley, Reading, MA, 1998
21. Leite, JCSP, Leonardi, MC, and Rossi, G. Deriving object-oriented specifications from external scenarios. Technical Report, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brasil, 1998
22. Leite, JCSP. Application language: a meta level requirements strategy. Technical Report, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brasil, August 1990
23. Gordon, VS and Bieman, JM. Rapid prototyping: lessons learned. IEEE Software 1995;15(1):85–95
24. Leveson, NG. Intent specification: an approach to building human-centered specification. In Proceedings of the third international conference on requirements engineering, IEEE Computer Society, Colorado Springs, CO, 1998

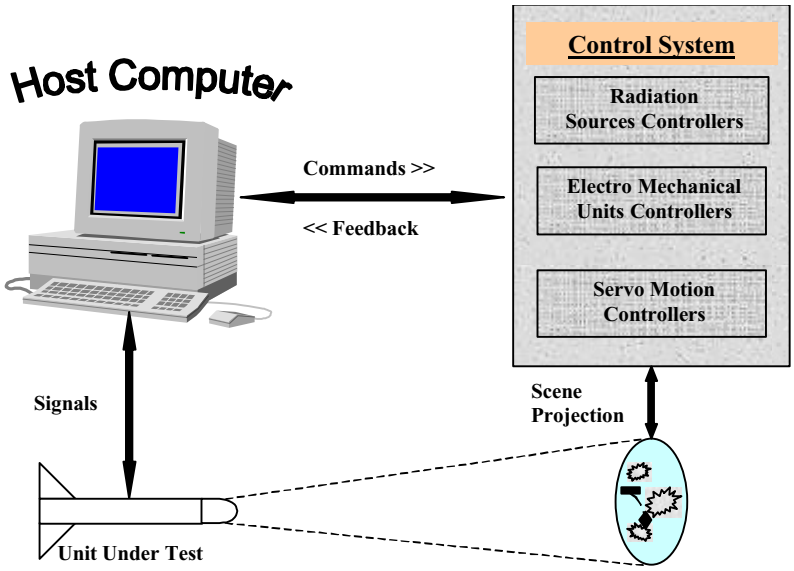

Fig. 1. The Principal Structure of the TSG Computer System

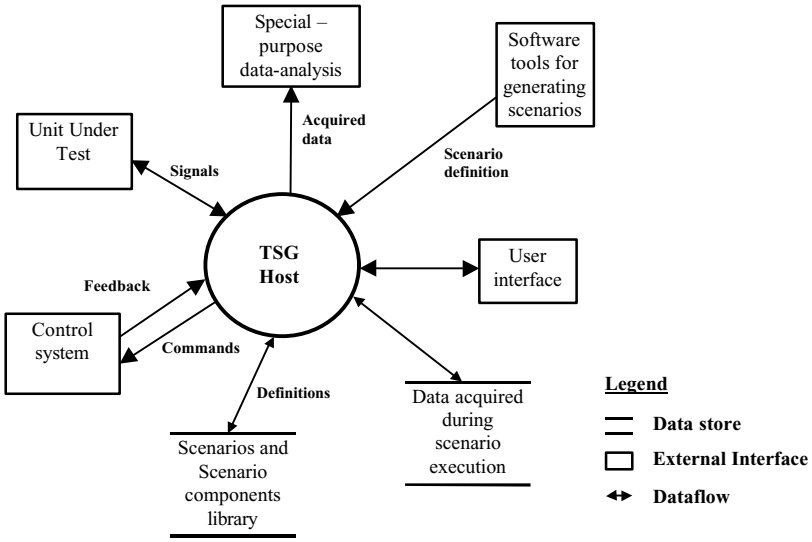

Fig. 2. The TSG Host Context-Diagram

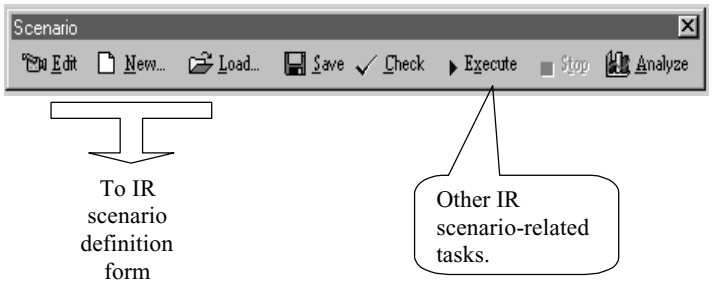

Fig. 3. The Infrared Scenario Related Tasks Toolbar

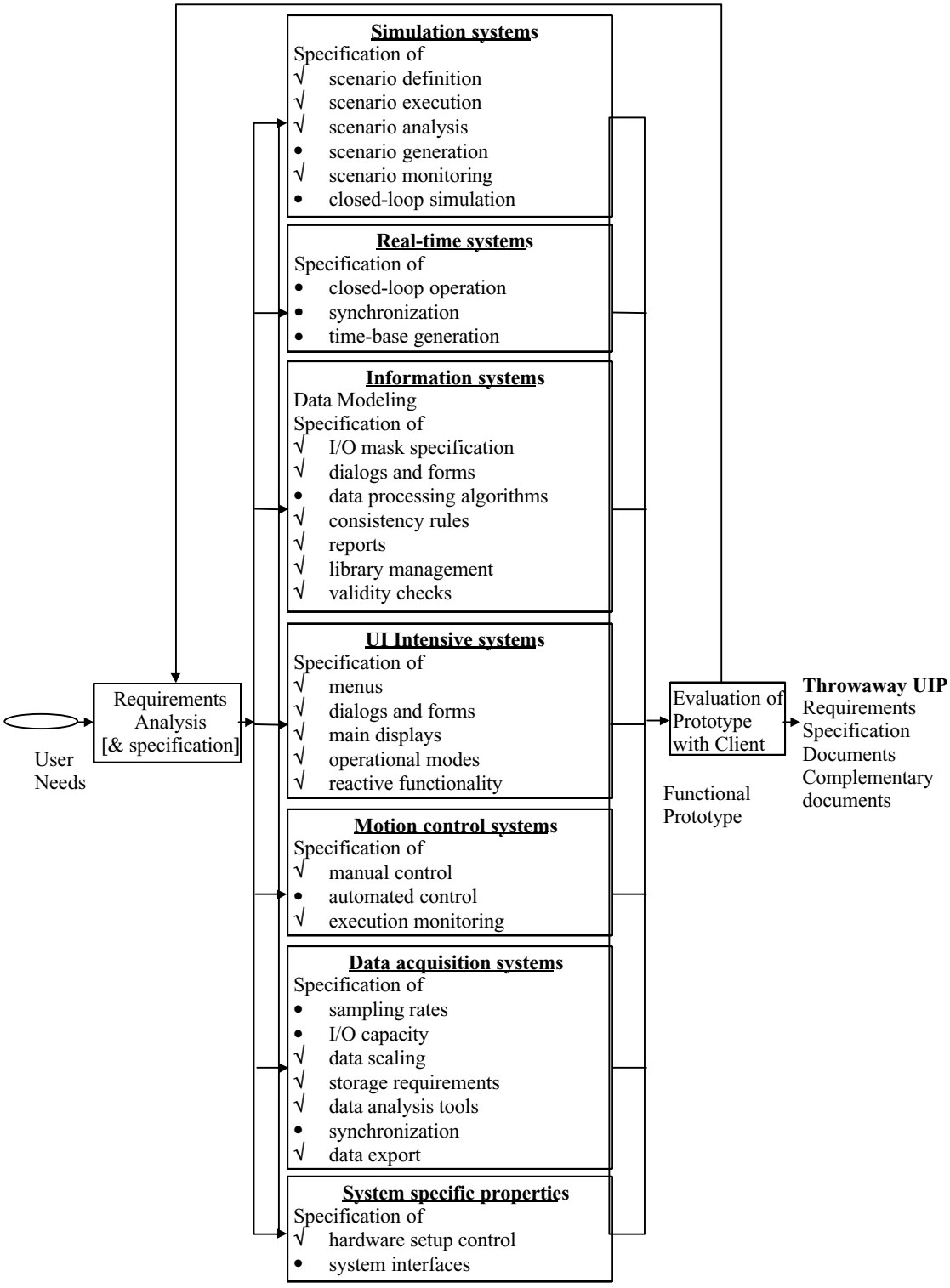

Fig. 4. The TSG Host User Interface Prototype Development Process

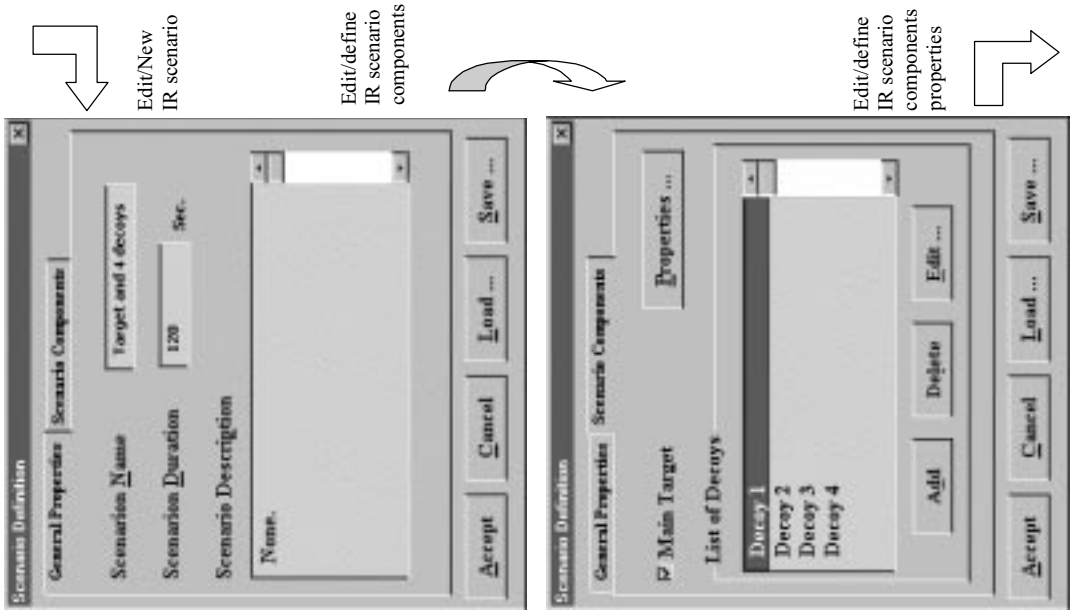

Fig. 5. Infrared Scenario Definition Main Form

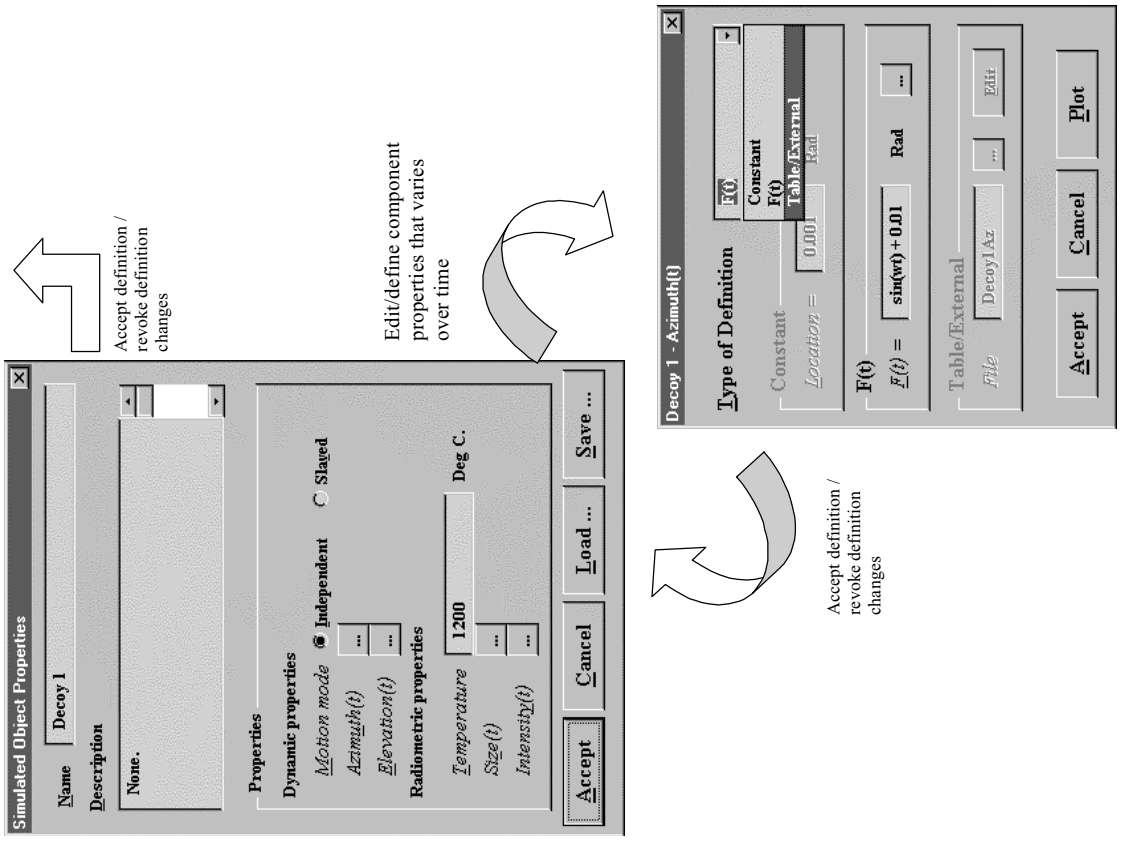

Fig. 6. Simulated object Definition