Tel Aviv University
School of Education

Research proposal towards a degree of PhD

Subject:

REQUIREMENTS ATOMIZATION IN SOFTWARE ENGINEERING EDUCATION

Submitted by: Hanania Salzer

I hereby declare my approval to supervise Hanania Salzer in his research towards the degree "Doctor of Philosophy", and I approve his research proposal.

Supervisor: **Dr. Ilya Levin**

Signature: ____________________________

Subject in Hebrew:

אטומיזציה של דרישות בהוראה של הנדסת תוכנה

Date: 18 May 2003

Number of words: **9,255**

Table of Contents

1 Abstract

The proposed research belongs to the domain of technology education. It will combine two fields of education – Software Engineering and computerized logic control, both are within the broader scope of studying programming, electronics and control.

It is based on the notion of Atomic Requirement (ATR), which is defined as a requirement or design specification that is (a) associated with a system functionality or component, (b) is well-formed and (c) would not be useful to subdivide into more elementary requirements at the abstraction level where it is being considered.

Field experience in the hi-tech industry suggests that ATRs, as opposed to non-atomic specifications facilitates identifying specification bugs, reducing implementation bugs, and identifying software bugs by tests. It is assumed that ATRs' effects result from two of their properties; they make elements of the design to be explicit, and they handle concerns one at a time. Cognitive processes might explain why these properties of ATRs would have the above effects.

This research will try to scrutinize the above claims, empirically and quantitatively and as they might become expressed in the work of students developing computerized logic control for software or software-hardware systems.

First, it will be checked whether replacing non-atomic specification with ATRs indeed results in students finding more bugs in the specifications, in making less implementation bugs, and in identifying more software bugs by tests.

Next, it will check whether students engaged in designing a controlled system and using ATRs, as opposed to non-atomic specifications, to document its design, are more successful in segregating control logic from operational functionality, thus designing the system with higher cohesion of the control unit. Also it will check whether students using ATRs are more successful in identifying the control

signals, thus designing a system with lower coupling between its control unit and its operational unit.

2 Introduction

The proposed research draws together the fields of Software Engineering (SE) and technology education. In particular, it deals with the atomization of requirements specification and design specification, and with their utilization in the teaching of computerized logic control.

The research proposal stems from the observations of Salzer (1999). His plain intuition led to develop a technique of atomizing specifications (both requirement and design), expecting for improvements in various parameters that were important in the software development industry. Indeed, he reports of success, such as reduced bug rate during software development and improved bug finding during testing, and he offers explanations for the results. However, scientific examination with quantitative data is needed in order to replace the "field experience" with useful results. SE seems to be a natural domain for such research. With the objective to examine the theory's effectiveness in practice, the research attempts to draw together SE and one of the education fields where basic SE principles are first taught to students. The chapter "Literature Review", below, presents logic control as an education field that is likely to be suitable for this purpose. The chapter "Rationale of the Research" offers reasoning why logic control is a particularly suitable subject matter upon which atomic specifications could be effectively compared to non-atomic specifications.

"*Why is it so difficult to introduce RE [Requirements Engineering] research results into mainstream RE practice?*" This question, raised by Kaindl *et al* (2002), is worth noting, even though RE (a sub-discipline of SE) deals only with requirements specifications and not with design specifications. Among the reasons for the difficulty to introduce RE research results into mainstream practice Kaindl *et al* point towards the scope of disciplines looked at by researchers: "*RE is by its very nature interdisciplinary and needs to adapt and integrate results from other disciplines, such as … cognitive science.*" Indeed, the proposed research will attempt to harness cognitive science to examine the effects of specification atomization, and with the hope that its results will be introduced into the technology education practice.

3 Literature Review

The literature review is divided into two. The first part is in this chapter. It begins by outlining the notion of Atomic Requirements (ATRs). Then, it looks into Atomic Requirements' use in the literature. Finally, it summarizes some of the Software Engineering (SE) principles that must be followed in the design of a logic control component for a controlled system. It shows, where ATRs are expected to support the adherence to these SE principles.

The literature review's second part is in the next chapter, "Rationale of the Research". It lays down foundations for a mechanism by which ATRs might supports SE.

3.1 Atomic Requirement (ATR) Specifications

The term (software) requirement has a variety of definitions, such as: "*a condition or capability that must be met by software needed by a user to solve a problem or achieve an objective*" (IEEE Std 610, 1991). Because requirements are usually natural language statements, their quality varies. "Well-formed requirements" are abstract, unambiguous, traceable and validatable (testable) (IEEE Std 1233, 1998). Atomic Requirements (ATRs) are well-formed requirements that, in addition, are also the result of splitting complex requirements into elementary, or indivisible, requirements. Usually, an ATR takes the form of a single sentence using non-formal language, nevertheless precisely expressing a specification.

3.1.1 Requirements Specifications, Design Specifications

The specification of a system, or any of its components, is a description of its interface (Britton and Parnas, 1981). The interface specifications comprise the system's (or system component's) requirements specifications. This implies that specifying the internal design of a system involves listing its components, and specifying each one. This insight establishes a recursive relationship between components along the hierarchy of a system's structure: the specifications of a component residing at a certain level of the hierarchy is, at the very same time, part of the higher level component's internal design. In other words, every specification statement is a requirement specification relative to a component at some level, and in the same time it is also an internal design specification of a higher abstraction level (Harwell *et al*, 1993, Kilov and Ross, 1994, Ghezzi *et al*, 2003, pp.161-162). Because of this duality, all specifications are called here – *requirements* specifications.

3.1.2 Requirements Atomization

The motivation for atomizing requirements is derived from the intrinsic dangers posed by the use of non-atomic requirements. For instance, when developers and testers look at a non-atomic requirement, they may recognize only some of the functionality it implies, overlooking the rest. The result could be a bug, as well as a test not looking for the bug. ATRs reduce this and other risks by making all functionalities explicit, and by listing each, elementary functionality, separately from the others. Therefore, chances are considerably better to achieve visibility of intentions, hence unambiguity, with a set of ATRs than with an equivalent non-atomic specification (Salzer, 1999).

3.1.3 ATRs' Nominal Definition

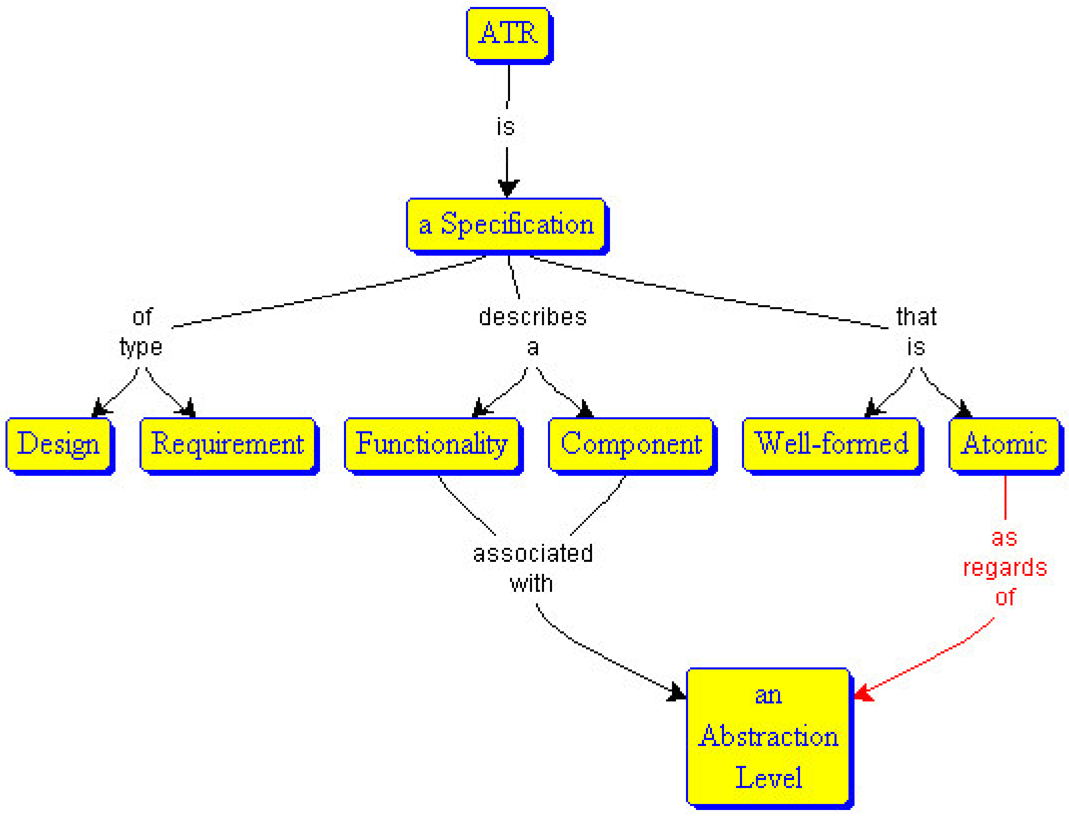

Figure 1: Concept map of ATRs' properties

Figure 1 summarizes the properties of ATRs described so far. The nominal definition for an ATR unifies design and requirement specifications and determines its atomicity:

> *An ATR is a requirement or design specification that is (a) associated with a system functionality or component, (b) is well-formed and (c) would not be useful to subdivide into more elementary requirements at the abstraction level where it is being considered.*

The ATR's atomicity is embodied by part (c) of the definition. Following are explanations for key phrases included in the definition:

- *Functionality or component.* Before designers define software and hardware components for a system, they describe the system as a list, or hierarchy, of functionalities. For example, logical processes in Data Flow

Diagrams and Use Cases are widely used for describing and organizing system functionalities before any components are identified. Specifications are associated first with functionalities, and later – with components.

When a specification is not associated with any specific component or functionality then, by default, it is associated with the whole system, which is located at the top of both the component hierarchy and the functional hierarchy.

- *Well-formed.* The term "well-formed" is defined in the IEEE standard number 1233, 1998.

- *Abstraction level.* Abstraction level refers to the degree of implementation freedom. With the progress of a system's definition process, each level's specifications are further elaborated in the next, lower level, which consequently is less abstract. Practically, an abstraction level can be identified with a set of functionalities or components that contain new specifications derived from the specifications of the higher abstraction level.

3.2 Current Uses for the Term "Atomic Requirement"

The terms "atomic requirement" and "atomic specification" have showed up incidentally in journals and on the Internet. However, Atomic Requirements (ATRs) have not been defined in the literature, and the potential advantages of their use have not been discussed, except by Salzer (1999).

Different authors make different use the terms Atomic Requirements and Atomic Specifications. Bolton *et al* (1992) and Sistla (1997) use the term atomic requirement for simple requirements in contrast with more complex requirements. Maiden *et al* (1997) too use the term atomic requirement for individual requirements, and include atomicity among the eight dimensions involved in understanding the relationship between a scenario and a requirement, but do not define this dimension.

Harn *et al* (1999) give a few examples of "atomic issues" that seem to be really atomic. On the other hand, they list many examples of what they call atomic requirements and atomic specifications, but none of them is indeed atomic in the sense of the definition given in the previous section, as demonstrated by the following example: "*R1.1-3.3: The control system must provide the function of receiving data of base coordinates, target coordinates, coordinates of safety point, target speed, and delay time of decision making; and computing data of base position, target position, missile direction, missile speed, target and missile intersection point, and missile reach time."* Lohr (1992) uses the term atomic specification to denote sequential implementation in contrast to concurrent implementation.

None of the above references includes *indivisibility* among the properties of what they call atomic requirements, or provide an account of the benefits observed or expected as a result of their atomicity.

3.3 Logic Control and ATRs

Logic control is the algorithm that controls the operation of an object by determining timely variations in its state. The algorithm models the controlled object's states. *"Each state in the algorithm maintains the object in the respective state, and a transition to a new state in the algorithm corresponds to a transition of the object to the respective state, thereby implementing the logic control"* (Shalyto, 2001).

The approach proposed in this section is based on the modular partition of a controlled system into an operational unit (OU) and a control unit (CU), where the latter implements logic control.

First, this section describes the relationship between the CU and the OU in terms of well-designed modularity. Then it elaborates on the contribution of ATRs to define the CU-OU interface in terms of well-designed modularity.

3.3.1 System Partitioning into Control Unit and Operational Unit

A controlled system can be viewed as composed of two high-level components, the control unit (CU) and the operational unit (OU). The CU is the part of the system responsible for taking the timely decisions that control the system's behavior. The OU is defined as all system components, except the CU. The communication between the system and its environment is only across the OU-environment interface. The CU does not interact directly with the system's environment; it communicates only with the OU. The OU can be viewed as an interface between the CU and the system's environment. Figure 2 presents two pairs of input and output. The left-side pair is between the environment and the OU, and the right-side pair is between the OU and the CU. From the CU design's point of view, this representation fully complies with the Four-Variable Model (Parnas, 1995, Heitmeyer *et al*, 1996).

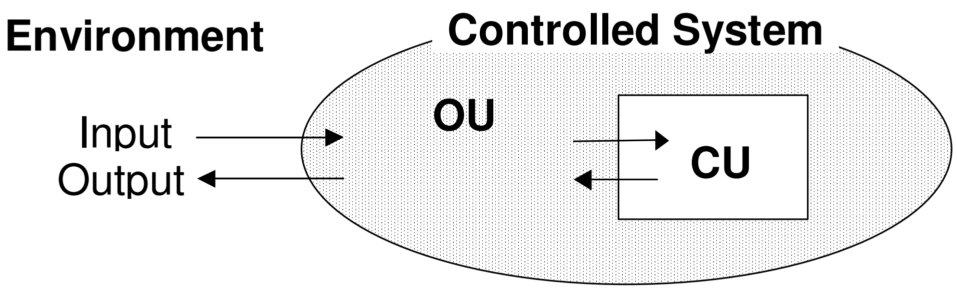

Figure 2: The system partitioning into an operational unit (OU) and a control unit (CU)

The system partition into a CU and an OU is a special case of system modularity. The concept of designing modular systems is decades old. Modular design constructs a system from a number of modules with well-defined interfaces; each one is small enough and simple enough to be thoroughly understood and well programmed (Parnas, 1972).

Parnas (1971) coined the term "information hiding". It guides the designer to decompose a system into modules that no longer correspond to steps in the processing. Instead, every module is characterized by its knowledge of a design decision, which it hides from all others. Its interface or definition is chosen to reveal as little as possible about the module's inner workings, called its "secret".

Modular design brings with it great productivity improvements through better comprehensibility and flexibility, faster and easier development, improved testing, and re-usability of work-products (Parnas, 1972, Hughes, 1989). These design properties are of significance in each one of the following cases: more than one person is involved in the project; the project takes more than a couple of weeks; the system will need to be maintained in the future or to be further developed (Ghezzi *et al*, 2003, p. 1). At least two of these conditions are true for the several months long projects of high school students building mobile robots.

Parnas (1972) suggested designing system modularity such that the design of each module will be independent of other modules' design. Two of the factors contributing to module independence are *coupling* and *cohesion*[1]. These factors are regularly discussed in software engineering textbooks, such as Ghezzi, *et al* (2003, pp. 47-49).

Module coupling is the degree of connections between modules; hence it is a measure of module interdependence. Level of coupling among modules must be kept to the minimum in order to minimize the "ripple effect" where changes in one module cause errors in other modules. The lowest level of coupling, hence the best, is *data coupling* (Myers, 1975), where two modules communicate by passing parameters of only primitive data elements. Two modules are *content coupled* if one module references data contained inside another module.

3.3.2 Logo Example of Logic Control

Lego robots controlled by Logo programs are used in classrooms for exposing students to hands-on experience with controlled systems (Resnick, Stephen and Papert, 1988). Logo procedures, similar to the example in Figure 3, are frequent in programs controlling Lego robots. This procedure implements the non-atomic requirement specification:

1. *When the light at sensor number 5 drops below 40% - turn on motor B.*

[1] See definition of *coupling* and *cohesion* in the "Dependent Variables" section of the "Methodology" chapter and in the appendix "Coupling Levels and Cohesion Levels".

The procedure in Figure 3 directly references data at input port number 5 in another module (the Lego Interface, which is an electronic device communicating between the Lego robot and a PC).

```
TO TURNLEFT
   WAITUNTIL [LIGHT5 < 40]
   TTO [MOTORA]
   OFF
   TTO [MOTORB]
   ON
END
```

Figure 3: An example Logo procedure

A well-designed CU is not aware of interface ports or the threshold value determining a certain state, because these are “secrets” of OU components. Instead, the CU communicates with the rest of the system (the OU) only via binary input and output signals (Baranov, 1994, Levin and Mioduser, 1996) thus providing pure data coupling.

The set of ATRs below is the result of splitting the non-atomic requirement number 1 into several atomic requirements. The CU implements ATR number 2, and the OU implements ATRs number 3 and 4:

2. *Turn left when the robot is over a dark surface.*
3. *The robot is over a dark surface when the light at sensor number 5 drops below 40%.*
4. *To turn left, turn off motor A, and turn on motor B.*

According to these ATRs the OU sends to the CU binary signals indicating whether the robot is over a dark surface or not. The CU sends to the OU a binary signal whether to turn left or not. Obviously, the ATRs have facilitated the design of data coupling between the CU and OU.

Module cohesion is the degree of inner self-determination of the module; hence it measures the strength of the module’s independence. A module should be highly cohesive. The best is a *functionally cohesive* module, which is one in which all of the elements contribute to a single, well-defined task. The second best is the *sequentially cohesive* module, which is one whose functions are related such that output data from one function serves as input data to the next function.

The example Logo procedure in Figure 3 carries out three functions: it taps a certain hardware interface port (light sensor number 5), then it determines whether the received input is below a certain threshold (40% of maximum brightness), and finally, it controls the program flow according to the predicate's outcome. This is an example of sequential cohesion.

Replacing the non-atomic requirement specification 1 with the three ATRs 2, 3 and 4, results with the different functions implemented by separate modules. Some OU modules handle inputs from certain sensors, and make them available for the CU in the form of binary signals. Other OU modules respond to CU signals by operating their respective actuators (such as certain motors). Finally, the CU has only one function: to decide which OU functions to activate at any point of time.

Clearly, the ATRs facilitated the design of a functionally cohesive CU module, and functionally cohesive modules in the OU.

A possible outcome of the above approach is a learning process that would cut back on mistaken allocation of control functionality to sensors and to actuators (Mioduser *et al* 1996). The students would describe the functionality of the system in question, and then atomize the resulting functional specifications. At this point it would be straightforward to categorize the resulting ATRs as either control or operation. The last step, in this classroom process, would be identification of the system's CU and allocation of the control functionality to this CU. The remaining ATRs could be allocated to OU components.

Students tend to overlook the existence of control signals, which traverse between a controller and the controlled components (Mioduser *et al*, 1996, Ma, 1999). Segregation between the CU and the OU should lead students to discover the need for a communication between the two components, and hence, to the need for some kind of signals. Not only that, but also the abstract nature of the ATRs' text explicitly suggests what messages the binary signals carry.

When students build robots as a learning activity with the intention to study the basic logic control principles, they may waste most of their focus and time on non-control related issues (Martin, 1996, Hancock, 2001). As a remedy, students should be guided to plan their system, and analyze the ATRs as described above. The set of ATRs identified as the control functionality would reveal whether the control constituent is too simple, whether the operational components are beyond the students' or their tools' capabilities, or whether the predicted efforts, required for developing each of the two parts, are unbalanced. Thus, before any time has been wasted on building the actual system, teachers could assist their students adjusting their plans to the academic needs and constraints.

3.3.3 ATRs and Logic Control

This research's scope is limited to ATRs in the logic control context. In the logic control context, An ATR relates binary input signals entering the CU with its binary output signals. Following is a description of a special language of *transition formulae* useful as a formal model for logic control (Levin and Levit, 1998). Later on, transition formulae will be shown to facilitate a formal definition of ATRs in the context of logic control.

Transition formulae map binary input signals to binary output signals as follows. The set $X = x_1, x_2, \ldots, x_L$ of binary input signals is transferred from the OU to the CU. The set of binary signals $Y = y_1, y_2, \ldots, y_N$ is the set of control microoperations, transferred from the CU of the system to the OU. The CU generates control microinstructions that are subsets of the microoperations set Y, which are executed concurrently. The OU performs microoperations in one-to-one correspondence with the set Y.

A CU is associated with a set of transition formulae. A transition formula is constructed as follows. The Boolean function α_i consists of one Boolean product (product term). Each product term α_i, depending on a set of variables $X = x_1, x_2, \ldots, x_L$, is put into correspondence with a control microinstruction Y_i, which is a subset of the microoperations set Y. Product term α_i is assumed to be equal to 1 if and only if control microinstruction Y_i should be performed. The resulting transition formula F_i associated with ATR_i is defined as:

$$F_i = \alpha_i Y_i + \overline{\alpha}_i Y_0$$

where

$$\alpha_i Y_i = \begin{cases} Y_i & if\,\alpha_i = 1 \\ 0 & if\,\alpha_i = 0 \end{cases}$$

The expression $\alpha_i Y_0$ in this formula tells explicitly that the ATR specifies only the actions that should be taken when the condition α_i materializes, but refrains from explicating what should happen otherwise. When the condition in one ATR does not realize, then the action specified in that ATR does not take place. The transition formula conveys this information by stating that the microinstruction Y_0 (the empty microinstruction) is executed.

The correspondence between an ATR and its transition formula is demonstrated below. Consider the following example ATR for a mobile robot that should avoid touching obstacles:

5. *Keep turning left as long as facing an obstacle that is too close.*

ATR number 5 is one of the many specifications that define the robot's logic control. The threshold distance that is considered to be "too close" is defined in another ATR. The robot's CU receives from the OU two binary input signals: x_1=TRUE means that the robot faces an obstacle. x_1=FALSE means that the robot does not face an obstacle. x_2=TRUE means that the robot is in safe distance from any obstacle. x_2= FALSE means that the robot is within dangerous proximity to an obstacle. The CU transmits to the OU a binary signal, indicating a microoperation: y_1=TRUE signals the OU to make a turn to the right. y_1=FALSE signals the OU not to make a turn to the right.

Transition formula F_1 corresponds to ATR number 5:

$$F_1 = x_1\bar{x}_2 y_1 + (\bar{x}_1 + x_2)Y_0$$

Every ATR in the context of logic control implementation is in a one-to-one correspondence with a specific transition formula. The non-formal text in the ATR and the formal transition formula carry the same information. The product term α_i represents the condition that the ATR describes. The control microinstruction Y_i represents the operation that the ATR describes.

At the foundation of the proposed work's approach is the reality that an ATR carries the specification of the smallest meaningful quantum of functionality. The one-to-one association between an ATR and the formal representation of a corresponding transition formula makes evident the ATRs' indivisibility. A direct consequence of an ATR's oneness is that it cannot carry a functionality that is both control and operation. Therefore, after atomizing a sufficiently detailed set of system specifications, the resulting ATRs can be segregated unambiguously into two groups, control and operation.

3.3.4 ATRs' Procedural Definition

Consequently, it is possible now to formulate a procedural definition for an ATR in the context of logic control:

A control related atomic requirement specification (control related ATR) is a requirement or design specification that is (a) associated with the system's control functionality, (b) is well-formed, (c) consists of a condition and of a corresponding operation, and (d) the condition and the operation are indivisible at the abstraction level where the specification is being considered.

4 Rationale of the Research

The rationale of this research has two aspects, which are reflected also in this chapter's structure:

- The potential role of ATRs in cognitive processes during system design, and
- The potential role of ATRs in learning the relationship between a controlled system's control unit (CU) and its operational unit (OU).

The first part creates the foundations for the second one.

4.1 ATRs in Cognitive Processes During Design

This section refers to cognitive processes to explain ATRs' role in preventing specification bugs from happen during system design and programming due to multiple handoffs.

4.1.1 Handoff

The purpose of a requirement is to reproduce in the mind of the reader the intellectual content, which was in the mind of the writer (Harwell, *et al*, 1993). Harwell *et al* define a requirement specification's quality as the extent to which this reproduction takes place. In this work the definition's scope is extended to cover design specifications too.

Any non-trivial software development process goes through several steps from its conception to its development and delivery. The process may continue with further cycles of maintenance and additional development. Regardless of the development methodology – whether it is "Waterfall" or "Extreme Programming" (XP) – each step in a top-down process generates a new abstraction level. Each step makes the system definition more explicit; hence it generates a lower abstraction level than the one from which it originates (Kilov and Ross, 1994).

Along the course of the system development process, specifications are handed off from step to step in order to further elaborate and detail the design. The definition of Harwell, *et al* (1993) suggests that during a hand off, a less than perfect specification is likely to lose some of the information that it intended to carry. Apparently, handoff is a weak point of the process. Mental and conceptual processes might help explain this weakness.

4.1.2 Mental and Conceptual Models

Norman (1983) described the difference between the explicit, conceptual description of a target system and its mental model. Using t to denote a particular target system, $C(t)$ to denote its conceptual model, $M(t)$ to denote its mental model, and $\rightarrow$ to denote a handoff, it is possible to describe the design process in a student assignment as a series of handoffs:

1. A teacher conceives a system t to be built by students. The teacher has a mental model of the target system: $M(t)$.

2. The teacher composes an assignment statement that describes the to be built system's functionality. The teacher creates a conceptual model of her own mental model: $M_1(t_1) \rightarrow C_2(M_1(t_1))$. The teacher believes that everything in her mental model is also in the written conceptual model, but this may be false. Hence, the conceptual model communicates a system that might be a slightly different from what the teacher had in her mental model: $C_2(M_1(t_1)) = C_2(t_2)$, where possibly $t_1 \neq t_2$.

3. Each student reads the assignment, and understands it, more or less, by creating his or her own mental model: $C_2(M_1(t_1)) \rightarrow M_3(C_2(M_1(t_1)))$, or $C_2(t_2) \rightarrow M_3(C_2(t_2))$. This handoff is just another chance for further information loss. Therefore, the student's mental model may be different from the one present in that assignment statement: $M_3(C_2(t_2)) = M_3(t_3)$, where possibly $t_2 \neq t_3$. One may also assume that $t_1 \neq t_3$.

Each subsequent handoff takes one of the two forms listed below:

- Understanding something communicated in a symbolic form, such as a written or a spoken form: $C_n(t_n) \rightarrow M_{n+1}(C_n(t_n))$.

- Communicating one's thoughts in a symbolic form: $M_n(t_n) \rightarrow C_{n+1}(M_n(t_n))$

Furthermore, the two forms alternate during a process of subsequent handoffs:

$$M_1(t_1) \rightarrow C_2(t_2) \rightarrow M_3(t_3) \rightarrow C_4(t_4) \rightarrow \ldots \rightarrow M_{n-1}(t_{n-1}) \rightarrow C_n(t_n)$$

Why would handoffs be vulnerable to information loss? Norman (1983) describes mental models as incomplete, not accurate, and containing errors and contradictions. Still, people will keep using a mental model even when they know that it is deficient. Therefore, it is reasonable to assume that every time a model passes through a mental model, it emerges in a conceptual model after, potentially, attracting bugs.

4.1.3 Bits and Chunks of Information

Miller (1956) introduced the terms *bit* of information and *chunk* of information. A *bit* is an elementary amount of information, such that it is just enough to make a decision between two equally likely alternatives. Control related ATRs have been described earlier in the context of logic control. The ATR's text, consisting of an indivisible condition and an action, is sufficient to make a decision between two alternatives, whether to carry out the action or not. Hence, the control related ATRs are the bits of a control unit (CU).

Miller (1956) proposes that people organize or group bits of information into familiar units, which he calls *chunks*. For example, a person may think of a CU's functionality in two alternative ways. Sometimes it is useful to consider it as a single chunk. At another time it is more suitable to consider a few of the individual ATRs, the bits. From Miller's findings it is possible to conclude that a person's mind can process seven, plus or minus two, ATRs simultaneously, but the same person would have no problem dealing with a CU as a whole, regardless of the number of ATRs it implements.

Transition formulae have been described earlier in the context of logic control. One may predict that in order to successfully handle seven, plus or minus two, transition formulae, each transition formula must be a chunk that can be counted as "one". The formal notation of transition formulae requires the reader to process a number of distinct input variables and output microoperations. Therefore, transition formulae are not likely to be valid chunks, since the number of variable instances in, say, seven transition formulae would pile up to a number that is beyond most people's capabilities to handle at once. ATRs, on the other hand, should be able to substitute the, sometimes complex, subsets of inputs and microoperations into a verbally expressed concept that a person can think of as a single item. Therefore, an ATR should be a useful chunk.

4.1.4 Designing and Chunking

The essence of design is invention. "Soar" is a candidate, unified theory of cognition (Lehman *et al*, 1996). According to the Soar model, each time the cognitive processes in the working memory (WM) cannot locate in the long term memory (LTM) a rule ("association") that would allow achieving a goal, it stumbles into an *impasse*. The impasse triggers a new goal to generate a new rule that should resolve the stalemate. The newly generated, and useful rule is called a chunk. Inventing a new rule is called *chunking*.

There is no contradiction between Miller's use of the chunk concept and that of Lehman *et al*, in the Soar model. In both models, older chunks are used to create newer ones, thus an old chunk is a bit for a new one. Chunking can describe the creative activity of design as follows.

For the sake of simplicity, let's assume that a designer considers a single specification. In term of the Soar model, the problem space's goal is to create a more detailed, less abstract set of specifications, which will detail how the higher-level specification will be realized. This goal implies that once the new, lower-level specifications have been created, they fully cover the higher-level specification.

In most cases, the designer does not find the more detailed specifications in his or her LTM. The result is an impasse. The impasse triggers in the WM a new problem space with a new goal: to invent a new specification that will suit the new abstraction level, and that will satisfy the higher-level specification. Once the new specification, a new chunk, has been generated, the designer adds it to the original goal's problem space. If the new specification does not fully cover the higher-level specification, then a new impasse arises. The process is repeated until the goal is achieved. At the end of these cycles the designer has a set of new chunks, which is the required set of the target abstraction level's new specifications. Note that unless the designer takes the extra effort to document each and every chunk, and unless this documentation is absolutely correct, the new abstraction level's conceptual model is deficient.

4.1.5 ATRs and Cognition

To motivate the difference between atomic (ATR) and non-atomic specifications, a simple example presents the cognitive processes of design. Assume that a programmer is assigned to write a program that retrieves from a database the necessary data to be printed on a teller machine's slip. The program collects the raw data into a data structure for further processing. The assignment includes the non-atomic specification in Figure 4.

```
Transaction-slip Data Retrieval

S-247: All transaction data are collected into a data structure
for the teller machine slip.
```

Figure 4: An example non-atomic specification

In terms of Soar, the programmer's goal is to think out its realization in program code. If the programmer likes to immediately put her thoughts in program code, then the goal for her primary problem space is to generate program code that populates the data structure. Basically, she has a single problem space for most of the task at hand: to write the requested program.

The programmer has in her long-term memory (LTM) two relevant associations. One is the list of transaction data available in the system's database. The other is that clients should be given only data that is useful for them.

The programmer's first impasse could lead to generating the new chunk:

```
If using the "Collect all Transaction Data for a Teller Machine
Slip" problem space,
and the data is for a client of the bank,
then include only data useful for clients.
```

A new problem space is generated in the programmer's WM, with the goal to include only data that is useful for the clients. The programmer may have in her LTM the information that through a few more steps will let her generate a set of new chunks, such as:

```
If using the "Collect all Transaction Data for a Teller Machine
Slip" problem space,
and the data is for a client of the bank,
then include "transaction date" in the data structure.
```

This chunk is not yet useful to create the program code. The programmer, keeping the above chunk in her working memory (WM), has a new impasse that induces an additional problem space with the goal to write a piece of program code that will retrieve the data from the data base. The resulting chunk is a mental model of the program code, which the programmer immediately hands off, in the form of a conceptual model, by writing down the actual program code.

At this point, the programmer's WM has created a considerable hierarchy of problem spaces. But that is not all, because the goal for the main problem space, "*Collect all Transaction Data for a Teller Machine Slip*", is far from being achieved. The whole process repeats until the goals of all subordinate problem spaces have been achieved, such as: "*Include all Data Useful for the Client*", "*Retrieve XYZ from the Database*", "*Add XYZ to the Data Structure*", etc.

The seven, plus minus two, limit on the number of concurrent chunks in WM, is extended through shuffling information between WM and LTM, and between WM and external memory, such as written text. Every shuffle, or handoff, that involves the mental model is vulnerable to damage. This damage is cumulative, and seems to build up fast.

An alternative design process may eliminate many of the chances for information degradation. In the above example, the programmer performed complex design activities in her mind. In the following example, on the other hand, the designer performs a number of design steps and documentation before handing off the design to the programmer.

The designer breaks down the specification in Figure 4 into ATRs suitable for the next, second abstraction level (Figure 5). Then he handles each ATR of the second abstraction level – separately. In the next step, again he breaks down the second abstraction level ATRs, creating the third abstraction level ATRs (Figure 6). At this time, or even earlier, the designer completes the design with standard ATRs, as described by Salzer (1999). The standard ATRs cover issues such as data retrieval from the database, and error handling. (The example does not show them.)

```
Transaction-slip Data Retrieval

S-601: Slip data is collected into a data structure for the
teller machine slip.

S-602: Only information useful for the Client is included in
slip data structure.
```

Figure 5: Second abstraction level

```
Transaction-slip Data Useful for the Client

S-603: Transaction's short description

S-604: Transaction date

S-605: Transaction amount
```

Figure 6: Third abstraction level

In this, alternative process, the designer works on small hierarchies of problem spaces. External memory is used to store chunks generated during the process, thus trading-off extra physical action for reduced mental complexity (Norman, 1983).

It is expected that because of the difference in the cognitive processes, a step-by-step process, such as described for the designer (second example), will accumulate considerably less bugs then a single-shot process, such as described for the programmer (first example).

In summary, this work hypothesizes that the vulnerability to information loss during handoff is affected by the individual specification statements' complexity. One may expect that an ATR, which carries the smallest possible amount of meaningful information, hence presents the simplest goal for a problem space, would be the optimal item for a loss-less handoff of information. Therefore, this research will examine whether the loss of information when handing off **atomic** specifications (ATRs) is less than the loss of information when handing off **non-atomic** specifications.

The practical effect of improved information retention along the development process should be a better identification of bugs by specification validation (see hypothesis number 1.a) and by software testing (see hypothesis number 1.d) as well as less bugs making their way into programs (see hypothesis number 1.c). For the same reason, views of teachers and students regarding to the expectations from an assignment will be less dissimilar (see hypothesis number 2).

4.2 ATRs' Role in Learning the CU-OU Interaction

This section refers to ATRs' potential ability to enlighten the segregation between a controlled system's OU and CU, and the communication between them in a way that facilitates students' understanding the basics of logic control.

4.2.1 Modularity and Logic Control

Most ATRs, at the lowest abstraction level, can be assigned to exactly one software unit, while this is not true for non-atomic specifications (Salzer, 1999). It is speculated that teaching students to allocate each ATR to either a control component or to an operational component, should lead them to recognize the functional difference between a system's control and operational components. Thus students are expected to apply functional (i.e., the best) module cohesion when they design a system's control component (See hypothesis number 1.b).

Furthermore, ATRs facilitate data (i.e., the best) coupling between the CU and the OU, as described in the next section.

4.2.2 Control vs. Operation

Building physical artifacts is an important educational tool in learning the basics of logic control (Lewis, 1994). Some researchers observed that students, when were given the freedom of creativity, engaged into tasks that they did not have the skills to make into reality. Many of Martin's undergraduate students (Martin, 1996) started off their mobile robot projects with high level plans for control only to find out rather late in their project's lifecycle that it was too complex to implement. Programming a robot to behave in interesting or intelligent ways – in ways that have some apparent autonomy – can be surprisingly hard. Students often find that goals must be repeatedly scaled back as the complexity of seemingly simple behaviors is revealed (Hancock, 2001).

These observations could be explained by students' inability to distinguish between control and operational functionalities.

4.2.3 Control Signals

The observations of Mioduser *et al* (1996) lead to the conclusion that even in the case of correct identification of components and functionality (device knowledge) in a feedback system, children may misallocate control functionalities relative to the components. Ma (1999) reports a case with a high-school student who failed to allocate more than one function to the same physical component. Both studies observe three constituents of the mental model for a feedback system: components, functions and signals. They find that students are not always aware of the signals moving between a system's control and operational components. Control signals may be viewed as taking the role of "stuff" in and among "autonomous objects" in the conception of mental model proposed by Williams *et al* (1983). de Kleer and Brown (1983) define the "stuff" as the (sometimes abstract) means to transmit information among a machine's comstituents as represented by a mental model.

These findings raise a problem; students tend to ignore the signals that operational and control components send to each other. The section “Logic Control and ATRs”, in the “Literature Review” chapter, elaborated on the one-to-one correspondence between control-related ATRs and transition formulae. It is speculated that students will identify the textual counterparts of the transition formulae variables (the phrases that stand for the “x”-es and for the “y”-s) as the control signals traversing between the CU and the OU (See hypothesis number 1.f).

It is expected that crisp identification of control signals will help students to comprehend the notion of data coupling between the CU and the OU (See hypothesis number 1.b).

It is expected that understanding the coupling between the CU and OU, and the cohesion within each of these two parts of a controlled system, will have the overall effect of understanding the CU-OU functional segregation (see hypothesis number 1.e) and comprehending the CU’s role as the control module in a controlled system.

5 Research Hypotheses

The hypotheses in this research revolve around the proposition that ATRs, through their atomicity lead to a number of measurable benefits, in the area of software programming education and in learning basic logic control concepts.

5.1 Scope

The notion of Atomic Requirement (ATR) provokes questions concerning their utilization and questions regarding to their composition. The research focuses on questions regarding to ATRs' utilization and effectiveness only. Questions that look into the process of ATRs' composition are left for a possible future work.

This research will evaluate the hypotheses listed below.

5.2 The Hypotheses

All hypotheses listed below relate to students learning programming, software development or controlled system development. In addition, these hypotheses are presented in the context where students receive assignments in the form of written requirement or design specifications, and develop software programs that should comply with those specifications. In this context we propose the following hypotheses:

1. **Students** receiving requirement or design specifications in the form of **ATRs** as compared to students receiving requirement or design specifications in the form of **non-atomic specifications**, perform better in the following areas:
    a. Through review, they identify more of the **bugs** existing in the requirement or design specifications.
    b. They design software applications with better modularity in terms of lower **coupling** between modules and higher **cohesion** within modules.
    c. While **programming**, they make fewer **bugs** in their programs.

d. Through **testing**, they identify more of the bugs that they make in the software programs written by them or by others.

e. They learn faster and better the notion of **CU** and the segregation between the CU module(s) and the OU modules.

f. They learn faster and better the notion of **control signals**, which traverse between the CU and the OU.

2. **Teachers** who work with students receiving requirement or design specifications in the form of **ATRs** as compared to **Teachers** who work with students receiving requirement or design specifications in the form of **non-atomic specifications**, make a less biased evaluation of the extent to which students' software programs meet the assignment goals.

6 Significance of the Research

If indeed ATRs prove to reduce the frequency of specification bugs, then they could be a significant supplement to requirements and specifications' validation. While validation help identify and remove bugs after they have occurred, the use of ATRs would prevent bugs form happening on the first place.

This research intends to contribute to the following fields of education:

- Learning sound Software Engineering (SE) notions and practices, in particular:
  - Software modularity in terms of cohesion and coupling
  - Requirement based software unit development and testing (verification).
  - Requirement coverage in software unit development and testing.
- Underlying theory of computerized logic control.

Research results may justify the inclusion of ATRs among the techniques that students use with the aim to facilitate the following insights during the students' studies:

- The role of the Control Unit (CU) as a special module within a controlled (or automated) system will be comprehended.
- Control signals and other "stuff" that traverse within and through systems (Mioduser *et al.* 1966) would naturally emerge.

Research results may have the following implications:

- Improvements in curricula for teaching programming and teaching computerized logic control
- Improvements in training students in the discipline of programming and the discipline of logic control.

7 Methodology

7.1 Research Questions

For an explanation of the dependent and independent variables mentioned in this chapter, please refer to the correspondingly named sections in this chapter.

1. Can students identify more of the **bugs** existing in the requirement or design specifications when the specifications are in the form of **ATRs** than when the specifications are in the form **non-atomic specifications**?

2. Does the students' design demonstrate better modularity in terms of (a) lower **coupling** between modules and (b) higher **cohesion** within modules when they receive requirement or design specifications in the form of **ATRs** than when they receive requirement or design specifications in the form of **non-atomic specifications**?

3. Do students make fewer **bugs** when programming from requirement or design specifications in the form of **ATRs** than when programming from requirement or design specifications in the form of **non-atomic specifications**?

4. Do students identify more of the **bugs** existing in programs when they test against requirement or design specifications in the form of **ATRs** then when they test against requirement or design specifications in the form of **non-atomic specifications**?

5. Do students learn (a) faster and (b) better the **notion of CU** and the segregation between the CU module(s) and the OU modules when the system is described with requirement or design specifications in the form of **ATRs** then when the system is described with requirement or design specifications in the form of **non-atomic specifications**?

6. Do students learn (a) faster and (b) better the notion of **control signals**, which traverse between the CU and the OU, when the system is described with requirement or design specifications in the form of **ATRs** then when

the system is described with requirement or design specifications in the form of **non-atomic specifications**?

7. Do **teachers** make a less biased evaluation of the extent to which students' software programs meet the assignment goals when they work with students receiving requirement or design specifications in the form of **ATRs** as compared to **teachers** who work with students receiving requirement or design specifications in the form of **non-atomic specifications**?

7.2 Research Population

The research population (subjects) will consist of high school and undergraduate students learning one of the following topics:

- Programming with a Third Generation (3G) procedural language, such as C or Pascal
- Principles of computerized logic control
- Mobile robot design and construction, including its programming.

7.3 Independent Variable - Specification Style

Specification Style is an independent variable describing an object that communicates functionality, such as an assignment, a requirement statement or an oral description of another object's functionality. Specification Style has two values:

- **Atomic.** All specifications in the statement are ATRs, that is, all of them are well formed (IEEE Std 1233, 1998) as well as atomic (Salzer, 1999).
- **Non-atomic.** All specifications in the statement are well formed (IEEE Std 1233, 1998) but many are not atomic.

The explicit information in atomic and non-atomic specifications can be compared, as described in "Appendix: Deriving Variables from ATRs".

Specification Style is useful as both independent and dependent variable, as described in the ”Dependent Variables” section below.

7.4 Dependent Variables

Some of the dependent variables require the identification of the equivalents of ATRs present within objects such as statements including non-atomic specifications, and software components. The “Appendix: Deriving Variables from ATRs” explains how this is done.

7.4.1 Number of Specification Bugs

A specification bug in a requirement specification or in a design specification is an explicit or implied ATR that is incorrect. Some of the discrepancies, defined below as cases of bugs, are subjective. A missing ATR, as well as an incorrect one, are both specification bugs. An ATR is deemed to be incorrect in the following cases:

- An ATR in a design or requirement specification contradicts another ATR in the same specifications.
- An ATR in a requirement or design specification contradicts another ATR in a specification at a higher abstraction level.
- A reviewer determined that an ATR is incorrect.
- One or more ATRs are missing when a reviewer has determined that the ATRs in a requirement or design specification do not fully cover an ATR that is present in a specification at a higher abstraction level.
- A reviewer has determined that an ATR is missing, although that ATR has no explicit roots in a specification at a higher abstraction level.

This variable represents the number of specification bugs that students discover.

7.4.2 Number of Software Bugs

Comparing the list of ATRs actually implemented by a software component with the list of ATRs in the design specifications of that software component is the underlying approach to reveal software bugs. The two main techniques for making such comparison are *test* and *code review*.

One should assume that there are no bugs in the respective design specifications. A software bug is defined as one of the following:

- **A missing ATR.** An ATR that is present in the design specification but is missing from the software component.

- **An unwanted ATR.** An ATR in the software component that does not exist in the design specifications, and that hampers the software component's functionality. Note that this definition includes a subjective, hence not absolute opinion.

7.4.3 Coupling Level

Module coupling is the degree of connections between modules; hence it is a measure of module interdependence. Level of coupling among modules must be kept to the minimum in order to minimize the "ripple effect" where changes in one module cause errors in other modules. The lowest level of coupling, hence the best, is data coupling (Myers, 1975), where two modules communicate by passing parameters. Two modules are content coupled if one module references data contained inside another module.

See "Appendix: Coupling Levels and Cohesion Levels" for an ordered list of coupling level names.

Coupling level between the CU and OU in a particular design is a measure of the segregation between the CU and OU in that design. Therefore, a CU-OU interface designed with data coupling is considered to be an indication of understanding the notion of control signals.

7.4.4 Cohesion Level

Module cohesion is the degree of inner self-determination of the module; hence it measures the strength of the module's independence. A module should be highly cohesive. The best is a *functionally* cohesive module, which is one in which all of the elements contribute to a single, well-defined task. The second best is the *sequentially* cohesive module, which is one whose functions are related such that output data from one function serves as input data to the next function.

See "Appendix: Coupling Levels and Cohesion Levels" for an ordered list of cohesion level names.

A CU designed with high level of cohesion indicates comprehension of logic control implementation in a CU.

7.4.5 Specification Style

Specification Style is useful as both independent and dependent variable. As an independent variable, students are exposed to an object that communicates functionality in one or the other specification style. As a dependent variable it describes an object that is the work product of students, such as design specifications and oral descriptions. The values that Specification Style takes are listed and defined in the "Independent Variable - Specification Style" section above.

7.5 Variable Dependency

Table 1: Dependency of variables related to students

		Dependent Variables	
		Non-atomic Specification Style n=40	Atomic Specification Style n=40
Independent Variables	Number of specifications bugs identified		
	Number of software bugs		
	Number of software bugs identified		
	Coupling Level		
	Cohesion Level		
	Specification Style		

Table 2: Dependency of variables related to teachers

		Dependent Variables	
		Non-atomic Specification Style n=4	Atomic Specification Style n=4
Independent Variable	Number of specifications non-bugs identified as		
	Number of software non-bugs		
	Number of specifications bugs missed		
	Number of software bugs		

7.6 Research Tools

This section lists, and describes briefly the various tools that are required in this research. Only a small number of the subjects will be interviewed, time permitting.

7.6.1 Object Analysis

The purpose of object analysis is to evaluate objects and compare objects through bug counts, missing and excessive functionality and, where relevant, also Specification Style. The objects will include teacher work products, such as assignments, student work products, including documents, interview records, software source code and, in the case of mobile robots, also hardware.

To analyze an object, first it will be studied to reveal the lists of ATRs that it comprises. For a software component, the list includes any ATRs that the software can demonstrate. For this reason, demonstrating an ATR by running tests is more reliable than identifying it through review of its source code.

Second, ATRs lists from different objects will be compared and analyzed to reveal the existing functionality, missing functionality and incorrect functionality (bugs).

In addition, analysis of the objects that communicate functionality will show its Specification Style, whether atomic or non-atomic.

7.6.2 *Interviews*

In the interviews subjects will be asked to describe the specifications in various contexts. Thus, the record of an interview is just another case of an object that communicates functionality.

The researcher in the present work will interview individual students, possibly detached from the regular course of classroom lessons. Teachers will be interviewed too. The reviews will be documented by taking notes as well as by recording on a voice recorder.

The written and recorded dialogues will be analyzed to identify and list ATRs. *Such lists are assumed to reflect the ATRs present*[2] *in the interviewees' mental models.*

Researchers routinely pick their subjects' brains by interviewing them or by recording subjects' conversations. To analyze the obtained free-style texts, researchers make various assumptions that enable conversion of this raw material into abstract, comparable data, which is supposed to reflect the interviewee's mental model. The ATRs' list extracted from an interview does just that.

7.7 Research Course

During the research we shall monitor students from several learning fields[3] (programming, basics of logic control and mobile robots). The students will receive assignments in two different Specification Styles.

[2] At the time of the interview.
[3] A "learning field" is not considered as a variable in this research.

We shall split the subjects in each learning field into two groups:

- Group A students will receive assignments using the **non-atomic** Specification Style
- Group B students will receive assignments using the **atomic** Specification Style.

The subjects will do the following activities:

- **Design.** The subjects will document their detailed design specifications to meet the assignment requirements.
- **Construction.** The subjects will construct the software that they have designed.
- **Testing and bug fixing.** The subjects will test the software against the requirement and design specifications. They will document and fix any bugs that they may find.

7.7.1 Data Collection

Both groups within a learning field will be assigned with the same task and will carry it out.

After the student complete their assignment, the objects that comprise their work products will be collected. The collected objects will include program source code, program executables, working robots and documentation written by the students.

In addition, a few of the subjects will be interviewed. Subjects may be asked, for example, to describe the functionality of the objects that they have created or to explain the roots of a bug.

All objects will be analyzed, as described earlier, to create an ATRs list for each object. The work products will be further analyzed by means of the respective

ATRs list with the aim of characterizing them according to the dependent variables listed earlier.

7.7.2 Data Analysis

The dependent variables will be correlated with the independent variable with the purpose of responding to the research questions.

8 Appendix: Terms and Definitions

Atomic Requirement (ATR). The nominal definition for an ATR, in this work, is: *a well-formed requirement or design specification associated with a system component that would not be useful to subdivide into more elementary requirements at the abstraction level where it is being considered.*

ATR. See Atomic Requirement.

Control Unit (CU). A software or hardware component of a controlled system whose sole role is to decide what actions the operational unit (OU) takes at any moment. This thesis only deals with discrete-event logic control. For this effect the CU sends the OU bi-level operation initiating signals called microoperations and receives bi-level signals about the OU's state.

Control signal. The CU and OU communicate via two types of control signals:

The CU sends binary signals to the OU, called microoperation, that tell the OU whether to do something or not to do it.

The OU sends binary signals to the CU to inform the CU of the OU's and its environment's state.

Control Related ATR. The procedural definition for a control related ATR, in this work, is: *a requirement or design specification that is (a) associated with the system's control functionality, (b) is well-formed, (c) consists of a condition and of a corresponding operation, and (d) the condition and the operation are indivisible at the abstraction level where the specification is being considered.*

Controlled System. In this work, a controlled system is one that contains the functionality to control the system itself. In many controlled systems it is possible to identify the components that carry out the control functionality; these components are called Control Units. (Note: for the purpose of this work, a system controlled by an entity external to the system is not a controlled system.)

CU. See Control Unit.

Operational Unit (OU). The OU is defined as all system components, except the CU.

OU. See Operational Unit.

Software Engineering (SE). Software Engineering is the field that deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers. Usually, these systems exist in multiple versions and are in service for many years. During their lifetime, they undergo many changes: to fix defects, to enhance existing features, to add new features, to remove old features, or to be adapted to run in a new environment. (Ghezzi *et al*, 2003, p 1.)

OU. See Operational Unit.

9 Appendix: Coupling Levels and Cohesion Levels

Myers (1975) defined the notions of *module coupling* and *module cohesion* (strength) and their respective levels.

9.1 Levels of Coupling

Levels of coupling in increasing order of their relative strength:

1. **Data.** Only primitive data elements are passed as parameters between components.
2. **Structure.** Data structures are passed as parameters between components. Also called **stamp** coupling.
3. **Control.** Control flags are passed as parameters between components.
4. **External.** Individual data items are organized into a common store.
5. **Common.** Data structures are organized into a common store.
6. **Contents.** One component directly modifies data or control flow of another.

9.2 Levels of Cohesion

Levels of cohesion in decreasing order of their relative strength:

1. **Functional.** Every processing element is essential to single function, and all such essential elements are contained within one component.
2. **Sequential.** Output from one function is input to the next one.
3. **Communicational.** Functions operate on or produce the same data set.
4. **Procedural.** Tasks grouped together to ensure mandatory ordering.
5. **Temporal.** Performs several tasks in sequence, related only by timing (not ordering).

6. **Logical.** Logically related tasks or data placed in same component.

7. **Coincidental.** Component's parts are unrelated.

10 Appendix: Deriving Variables from ATRs

One may wonder whether the information given to the different groups of research subjects differ not only in the specification style, as defined in the "Methodology" chapter, but also in the amount of explicit information. The amount of information in non-atomic specifications can be compared with the amount of information in atomic specifications after atomizing the former ones. This appendix lists definitions by which one may compare the equivalents of ATRs among lists of non-atomic specifications, software components, etc., and argues that the comparison is valid. The definitions are useful for interpreting the research variables.

10.1 Definitions Useful for Comparing ATRs

ATRs' comparison is one of the underlying techniques planned for this research. The ATR's procedural definition (see earlier in this document) lead to the following basic definitions useful for comparison of ATRs and for ATR based comparison of objects:

- **Object.** An object in this context is something that explicitly and/or implicitly communicates or implements specifications.

  Example objects are programming assignments, oral descriptions of systems, and software components' source code. A programming assignment is an object that the teacher gives to students. It communicates the features (functional and non-functional) that the teacher expects from the students to implement in a program. Depending on the style of the assignment, the teacher's expectations may be fully or only partially explicit. Similarly, a system's oral description is an object that communicates some of system's perceived specifications. A software component source code is an example for an object that implements a set of specifications.

- **Matching ATRs.** Two ATRs match if they translate into identical formal expressions.

- **Contradicting ATRs.** Two ATRs contradict if they translate into two logically contradicting formal expressions.

  $F_1 = x_1Y_1 + \bar{x}_1Y_0, \quad F_2 = \bar{x}_1x_2Y_1 + (x_1 + \bar{x}_2)Y_0$ are examples for two contradicting formal expressions. F_1 says that Y_1 is executed only when x_1 is true, while F_2 says that x_1 must be false in order to execute Y_1. More formally: the product of F_1 and F_2 is zero (false).

- **ATRs' list.** By analyzing an object, it is possible to make an inventory – In the form of an ATRs' list – that includes all the functionality implemented or communicated by the object.

  ATRs' lists are important tools in this research by making possible the comparison among very different objects.

- **Missing ATR.** An ATR is not present in an ATRs' list if it does not match any of the ATRs in the list. An ATR that should be present in a list, but is not – is missing from that list.

- **Redundant ATRs.** Two ATRs are redundant if they match and they are in the same ATRs' list.

10.2 ATR's Suitability as a Variable

Tal-Levy, *et al* (2001) identified atomic condition-action statements, which they called "rules", in young children's descriptions and definitions for simple controlled systems. They counted rules and even half-rules to quantify the children's maturity levels in regard to technology comprehension. Similarly, this research uses atomic specifications to identify elementary specification items, but in a larger scale and from many different sources.

ATRs are suitable for use as variables in this research because even when two, very differently phrased ATRs are compared, it is possible to tell with high confidence whether they carry the same information or not. This is the direct result of ATRs' very nature. Each ATR encompasses only a single, atomic, functionality. Because of their simplicity, an analyst (in this case, the researcher) can concentrate on comparing two specific ATRs by simultaneously holding the two respective mental models in his or her “working memory”.

Thanks to this property of ATRs, an analyst can compare two lists of ATRs and identify the ATRs that match, the ATRs that contradict, the ATRs that are present in one list and not in the other list, and the redundant ATRs.

10.3 ATR’s Validity as a Variable

ATRs’ validity and usefulness in comparing two functionality lists is itself a direct outcome of this works’ hypotheses. Therefore, it is not possible to validate this kind of use for ATRs before the research concludes. This is like a person trying to lift himself up by his own bootstraps.

11 Bibliography

Baranov, S. (1994). *Logic Synthesis for Control Automata*. Kluwer Academic Press.

Bolton, D., Jones, S., Till, D., Furber, D. and Green, S. (1992). Knowledge-based support for requirements elicitation: A progress review. *Technical Report TCU/CS/1992/23, City University, 1992. GMARC Project Report R44*.

Britton, K. H., and Parnas, D. L. (1981). A-7E Software Module Guide. *Naval Research Laboratory (NRL) (NRL Memorandum Report 4702)*.

de Kleer, J. and Brown, J. S. (1983). Assumptions and Ambiguities in Mechanistic Mental Models. In *Mental Models*, Gentner, D. and Stevens, A. L. (Eds.), Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 155-190.

Ghezzi C., Jazayeri, M., and Mandrioli, M. (2003). *Fundamentals of Software Engineering, 2nd Ed.* Upper Saddle River, New Jersey: Prentice Hall.

Hancock, C. (2001). Children's Understanding of Process in the Construction of Robot Behaviors. *Varieties of Programming Experience, AERA 2001*, Seattle.

Harn, M., V. Berzins, and Luqi (1999). Evolution of C4I Systems. *Command and Control Research and Technology Symposium*, United States Naval War College, Newport, Rhode Island, June 29 - July 1, 1999, pp. 1361-1380. Naval Postgraduate School

Harwell, R., Aslaksen, E., Hooks, I., Mengot, R., Ptack, K. (1993). What Is A Requirement? *Proceedings of the Third International Symposium of the NCOSE*.

Heitmeyer, C. L., Jeffords, R. D. and Labaw, B. G. (1996). Automated Consistency Checking of Requirements Specifications. *ACM Trans. on Software Eng. and Methodology*, 5(3), 231-261.

Hughes, J. (1989). Why Functional Programming Matters. *Computer Journal*, 32(2), 98-107.

IEEE Std 1233, 1998 edition. Guide for Developing System Requirements Specifications. *IEEE Standards, Software Engineering, Volume One, Customer and Terminology Standards*. IEEE, Computer Society.

IEEE Std 610.12-1990, 1991 edition. IEEE Standard Glossary of Software Engineerirng Terminology. *IEEE Standards, Software Engineering*. IEEE, Computer Society.

Kaindl, H., Brinkkemper, S., Bubenko Jr, J. A., Farbey, B., Greenspan, S. J., Heitmeyer, C. L., Leite†, J. C. S., Mead, N. R., Mylopoulos, J., Siddiqi, J. (2002). Requirements Engineering and Technology Transfer: Obstacles, Incentives and Improvement Agenda. *Requirements Engineering*, 7(3), 113-123.

Kilov, H. and Ross, J. (1994). *Information Modeling: An Object-oriented Approach*, pp. 28-32. Prentice-Hall (1994)

Lehman, J.F., Laird, J.E. and Rosenbloom, P.S., (1996). A gentle introduction to SOAR, an architecture for human cognition. In S. Sternberg and D. Scarborough (Eds.) *Invitation to Cognitive Science*, Volume 4.

Levin, I. and Levit, V. E. (1998). Controlware for Learning with Mobile Robots. *Computer Science Education*, 8(3), 181-196.

Levin, I., and Mioduser, D. (1996). A Multiple-Constructs Framework for Teaching Control Concepts. *IEEE Transactions of Education*, 39(4), 488-496.

Lewis, P. H. (1994). Introducing Discrete-Event Control Concepts and State-Transition Methodology into Control Curricula. *IEEE Transactions of Education*, 37(1), 65-70.

Lohr, K. P., (1992). Concurrency Annotations for Reusable Concurrent Software. In *OOPSLA '92, Proceedings*, p. 327–340, Vancouver, Canada, October 1992.

Ma, J. (1999). A Case Study of Student Reasoning About Feedback Control In a Computer-Based Learning Environment. *29th ASEE/IEEE Frontiers in Education Conference* (12d4), 7-12. San Juan, Puerto Rico:.

Maiden, N., Minocha, S., Manning, K. and Ryan, M., (1997). A Software Tool and Method for Scenario Generation and Use. *Third International Workshop on Requirements Engineering: Foundation for Software Quality RESFQ'*, June 16-17, 1997, Barcelona, Spain

Martin, F. G. (1996). Ideal and Real Systems: A Study of Notions of Control in Undergraduates Who Design Robots. In *Constructionism in Practice: Rethinking the Roles of Technology in Learning*, Y. Kafai and M. Resnick (Eds.), pp. 297-322. Mahwah, NJ: Lawrence Erlbaum.

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Limits on Our Capacity for Processing Information. *The Psychological Review*, 63, 81-97.

Mioduser D., Venezky, R. L., and Gong, B. (1996). Students' Perceptions and Designs of Simple Control Systems. *Computers in Human Behavior*, 12(3), 363-388.

Myers, G. J. (1975). *Reliable Software Through Composite Design*, pp. 19-54. New York: Petrocelli/Charter.

Norman, D. A. (1983). Some Observations on Mental Models. In *Mental Models*, Gentner, D. and Stevens, A. L. (Eds.), Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 7-14.

Parnas, D. L. (1971). Information Distribution Aspects of Design Methodology. *Proceedings of the 1971 IFIP Congress*, 339-344.

Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), 1053-1058.

Parnas, D. L. (1995). Functional Documentation for Computer Systems. *Science of Computer Programming, 25*(1), 41-61.

Resnick, M., Stephen, O. and Papert, S. (1988). LEGO, Logo, and Design. *Children's Envs. Quart.*, 5(4), 14-18.

Salzer, H. (1999). ATRs (Atomic Requirements) Used Throughout Development Lifecycle. *12th International Software Quality Week (QW99)*, 1, (6S1), San Jose, CA.

Shalyto, A. A. (2001) Logic Control and "Reactive" Systems: Algorithmization and Programming. *Automation and Remote Control*, 62(1), 1-29.

Sistla, P., Yu, C. T. and Venkatasubrahmanian, R. (1997). Similarity Based Retrieval of Videos. In *Proceedings of IEEE ICDE*, Birmingham, UK, pp. 181-190.

Tal-Levy, S., Mioduser, D. and Talis, V. (2001). Concrete-Abstractions Stage in Kindergarten Children's Perception and Construction of Robotic Control Rules. *PATT 2001 Proceedings*.

Williams, M. D., Hollan, J. D. and Stevens, A. L. (1983). Human Reasoning About a Simple Physical System. In *Mental Models*, Gentner, D. and Stevens, A. L. (Eds.), Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 131-154.

(This page has been intentionally left blank.)