SAND 96-1192C

CONF-9610198--1

RELIABLE SOFTWARE SYSTEMS VIA CHAINS OF OBJECT MODELS WITH PROVABLY CORRECT BEHAVIOR

May 17, 1996

To be submitted to The Seventh International Symposium on Software Reliability Engineering, supported by IEEE, to be held at White Plains, NY, October 30 - November 2, 1996

Alexander Yakhnis*
High Integrity Software Research
Sandia National Laboratories, MS-0535
1515 Eubank SE
Albuquerque, NM 87123
Phone: (505) 844-0277 Fax: (505) 844-9478
E-mail: aryakhn@sandia.gov

Vladimir Yakhnis*
High Integrity Software Research
Sandia National Laboratories, MS-0535
1515 Eubank SE
Albuquerque, NM 87123
Phone: (505) 844-8672 Fax: (505) 844-9478
E-mail: vryakhn@sandia.gov

ABSTRACT Our work addresses specification and design of reliable safety-critical systems. Reliability concerns are addressed in complimentary fashion by different fields. Reliability engineers build software reliability models, etc. Safety engineers focus on prevention of potential harmful effects of systems on environment. Software/hardware correctness engineers focus on production of reliable systems on the basis of mathematical proofs. We think that correctness may be a crucial guiding issue in the development of reliable safety-critical systems. However, purely formal approaches are not adequate for the task, because they neglect the connection with the informal customer requirements. We alleviate that as follows. First, on the basis of the requirements, we build a model of the system interactions with the environment, where the system is viewed as a black box. We will provide foundations for automated tools which will a) demonstrate to the customer that all of the scenarios of system behavior are presented in the model, b) uncover scenarios not present in the requirements, and c) uncover inconsistent scenarios. The developers will work with the customer until the black box model will not posses scenarios b) and c) above. Second, we will build a chain of several increasingly detailed models, where the first model is the black box model and the last model serves to automatically generate proved executable code. The behavior of each model will be proved to conform to the behavior of the previous one. We build each model as a cluster of interactive concurrent objects, thus we allow both top-down and bottom-up development.

---

* This work was supported by the United States Department of Energy under contract DE-AC04-94AL84000.

KEYWORDS: customer, requirements, specifications, design, correctness proofs, classes, objects.

1. INTRODUCTION

Safety-critical systems, such as nuclear plants, medical lasers, transportation systems, etc. pose serious problems as to nature of criteria under which such systems can be permitted to operate. This is the problem of certification of safety-critical systems. Because of the potential harmful impact of their malfunction on the population, the certification process must be convincing and manageable. This should be achieved on the basis of automated tools and methods used to handle, view, verify, and experiment with the system requirements. Ideally, the certification process should be as follows.

Firstly, the requirements of the system submitted for certification should be complete, consistent, and transparent. Secondly, the certification process should demonstrate that the system satisfies the requirements either absolutely or with a high degree of likelihood. We understand absolute satisfaction of requirements as our ability to generate a mathematical proof that under certain conditions the system behavior satisfies the requirements. High degree of likelihood usually means the probability that the system behavior will satisfy the system requirements (for a specified period of time) is very close to 1. In other words, it means that the system has a high reliability [Lyu 1996]. We think that both concepts, i.e., provability and high reliability, are essential for system certification. Specifically, certification of the whole system is usually based on the results of certification of its components. It is highly desirable that all the software components of safety-critical systems would be mathematically proved. It also should be proved (when possible) that the hardware components function correctly in the absence of mechanical, electrical, and other physical failures. For the rest of the components a high reliability should be established.

Consider the following example of a safety-critical system: a nuclear reactor control system. One of its safety subsystems must issue a shutdown command once the sensors detect that the neutron density is above certain critical value *crit*. For simplicity, suppose there are 2 sensors whose measurements are m and n. We would like two establish that if the neutron density exceeds *crit*, then the reactor will be shut down. Now, it is possible to prove (see the section on software verification) that if at least one of the sensors does not fail and indicates the neutron density is greater than *crit*, then the safety subsystem will issue the shut down command. However, if both sensors fail then nothing can be proven. Thus a certification of the safety subsystem can be based on (a) the proof mentioned above and (b) on the high reliability of the sensor subsystem.

Here we focus on the following four aspects of the certification process for safety-critical systems:

- certification of completeness, consistency, and transparency of the informal requirements;
- certification of correct representation of the informal requirements by the formal requirements;
- mathematical proofs of system components with respect to the requirements;
- mathematical basis of automation of the two above aspects in the presence of ubiquitous partially defined operations. Note that in the above example since the sensors may fail, they should be modeled as partially defined functions.

In a future work we will investigate the mutual dependencies between correctness proofs and computations of reliability for safety-critical systems.

2. REQUIREMENTS CAPTURE: COMPLETENESS AND CONSISTENCY

In order to insure that the set of requirements captures the original idea about the system and facilitates the evolving customer intent, we suggest the following process of recursive discovery of the requirements in cooperation with the customers. At all times the customers has control over the capture/discovery process since the feedback from the developers is provided in several transparent forms, including visualization.

2.1. Requirements as formulas of classical logic

We will initially view requirements as formulas of logic. Although the requirements can be often stated as Hoare triples (e.g. [Apt 1981]), the latter are usually handled by writing down equivalent formulas of logic.

- We will collect the requirements in small increments while building a chain of executable object system models.
- For each models we will simulate and/or visualize various aspects of its behavior specified by a number of system parameters.
- We collect input from the customer with respect to the exhibited system behavior.

For each level of the chain there are several possibilities. The customer may wish to exclude some of the system behaviors allowed heretofore. This may lead us either to add constraints excluding such behavior or to change/remove some of the requirements permitting such behavior. In the latter case we should have an automated tool rebuilding the previous models in

the chain by changing/removing the aforementioned requirements. The customer may also see that some of desirable system behaviors are not observable. This may also lead us to change the system requirements.

The above would steps lead us to the "complete" set of system requirements. The consistency of the requirements means that they do not contradict each other. The set of requirements is consistent if a model that satisfies these requirements can be built. Sometimes a consistency check can be done by a theorem prover applied to a formula obtained by taking a conjunction of all the requirements. An obstacle to this procedure is the presence of partial functions (operations) in the requirements. Standard theorem provers presuppose total, rather than partial, functions. A remedy to this problem is suggested in a later section.

2.2. Requirements as Scenarios

Often requirements can be viewed in a different way than formulas in classical logic. An alternative view on requirements is a collection of scenarios. For example, for a security check subsystem the following sequence is a scenario

Insert Id_Card; **Read Id_Card**; **Request Acess_Code**; Type Acess_Code; **Verify Acess_Code**; etc. ...

For the set of scenarios which constitute the system requirement, we often can write a regular expression describing exactly this set or an accepting automaton for this set. Since from the expression we may automatically generate an accepting automaton for this set, we may regard that the set is represented by an accepting automaton.

We can automatically analyze the automaton for completeness of requirements (using 2-player game solving algorithms). This often can be expressed as a problem of presence of inaccessible automaton states. If there are such states, the set of requirements would often be considered incomplete.

The check of consistency of requirements presented as a collection of scenarios (as is in the case of security check subsystem) includes analysis of nondeterministic system actions (in cooperation with customers). It can automated by means of a theorem prover with the same comments as we have made earlier in this section.

Finally, a collection of interacting objects which generate the scenarios corresponding to the requirements constitute a system model satisfying the requirements. We summarize our approach to capture of the requirements by means of the following diagram.

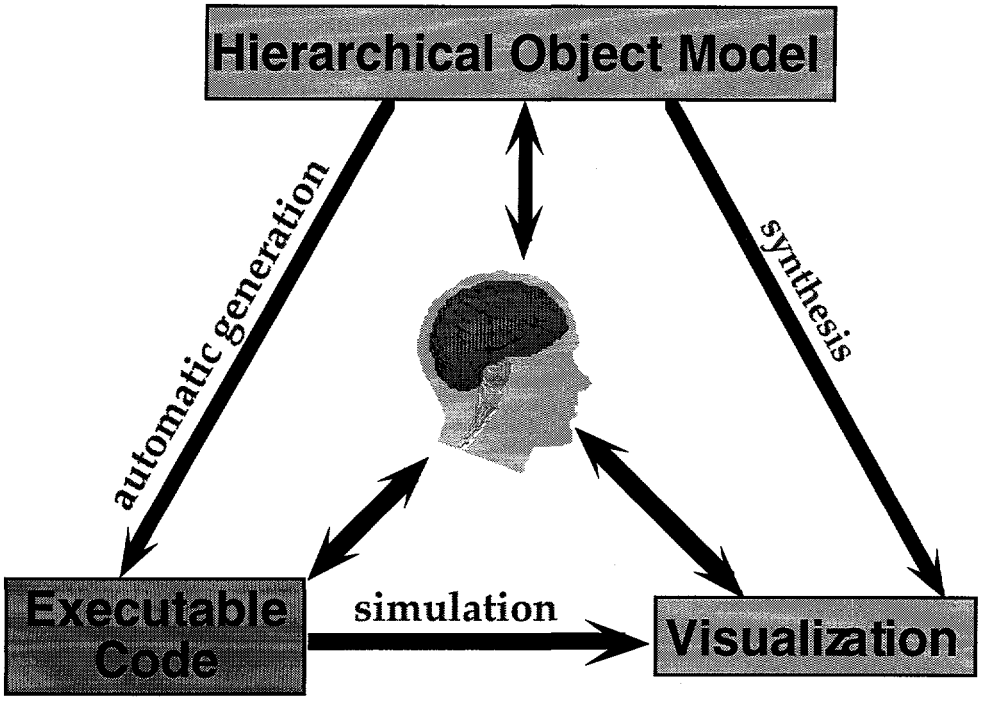

Figure 1. Recursive Capture of Requirements and System Development

The chain of system models produced by iterating the actions indicated on the diagram helps to insure 3 integrity principles:

(1) Build the right thing

(2) Build the thing right

(3) Make the customers understand what is built.

3. CHAINS OF SYSTEM MODELS

In order to simplify system certification, we look for system representation which would identify the proofs needed to be included in the certification process. A system representation which we call below a chain of models possesses this property, since the definition of such a representation below is a list of the representation properties to be proved.

Definition. A chain of models corresponding to a given collection of informal system requirements $\overline{R}$ is a sequence of quadruples satisfying the following properties. Below we view a system model as a state transition system which can generate both finite and/or infinite system runs [Yakhnis, Stilman 1995]. We think of formal requirements and of collections of informal requirements as unary relations on system runs.

- Each quadruple consists of a system model M, a "correctness preserving map" h from this model into the preceding (in the sequence) model M′, a formal requirement S, and a

collection of informal requirements R. The set of all runs of M satisfying S must not be empty. $\overline{R}$ is a union of all R in the sequence.

- The relationship between 2 adjacent quadruples $q'=(M', h', S', R')$ and $q=(M, h, S, R)$, where q is next to q' in the sequence, is as follows.
  - **Incremental Requirements Rule**: For every system run r of the model M, r satisfies S if and only if (r satisfies R and h(r) satisfies S′).
  - R and R′ are disjoint.
- The last quadruple (M, h, S, R) satisfies the following property: each system run r of the model M must satisfy S.

□

Lemma. Any run of the last model in the above chain satisfies $\overline{R}$.

□

Remark. We think of the last model either as the final software/hardware system or as such that the final system may be automatically generated from it.

□

Essentially, a certification process must establish that the system satisfies the collection of informal requirements $\overline{R}$. In view of the last Lemma and Remark, a certifying agency may require the above chain of models for the system and verify that the chain satisfies the above definition. Thus automation of system certification requires an automated verification of the 3 properties of the chains above. This reduces frequently to automated checking of Hoare triples for (sequential) pieces of software. A complication for this problem is the presence of partial functions in the software. We will outline in the later section an approach of how to deal with the problems of this sort in a systematic way.

4. AN EXAMPLE: THE PRODUCTION CELL

As an example of the requirements capture and of system construction via a chain of models we'll consider a portion of the controller for a real-world industrial installation in a metal processing plant in Karlsruhe, Germany. The installation (called "the production cell") and several attempts by various research teams to generate a mathematically proved controller were described in [Lewerenz, Lindner 1995]. The following is a part of a prototype solution within Sandia High Integrity Software (HIS) Initiative undertaken by the authors.

The production cell consists of six sequential devices processing one or more metal plates. For the purpose of the industrial testing of the controller they are arranged in a circle, i.e., a plate passes through all six devices ad infinum. We'll consider here only two of the devices, namely

the "feed conveyor" and the "turn table", and only one plate. Figures 2 and 3 represent the two devices. The rest of the devices are represented by the dashed arrow. The plate(s) move in the direction of the arrow.

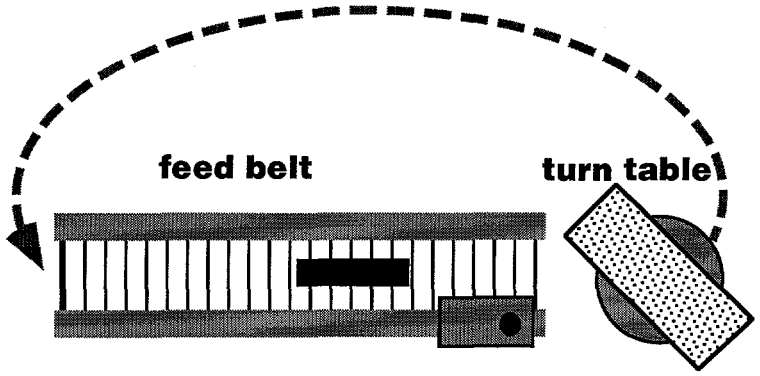

Figure 2. Production Cell: No Plate Transmission

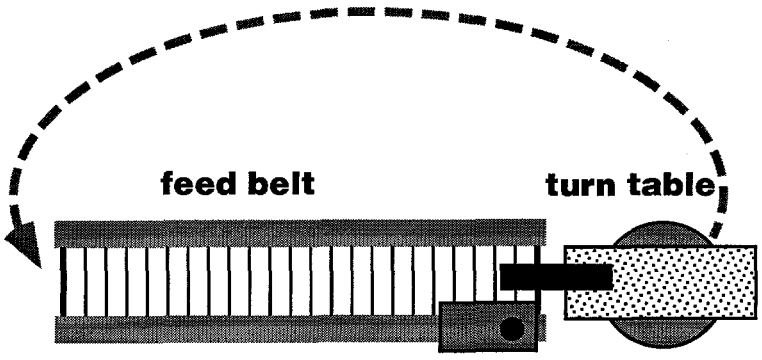

Figure 3. Production Cell: Plate Transmission

The following is the first installment of the customers' requirements:

- Plate transmissions or other activities may occur.
- There may be infinitely many plate transmissions or other activities.
- A deadlock (cessation of useful progress) occurs when:
  - for infinitely long no plate transmissions between devices are occurring,
  - while some devices perform other activities.
- An error may occur during:
  - plate transmissions;
  - other activities.
- When an error occurs, everything stops.

The top model is represented on Figure 4. System runs with either a deadlock or an error are possible.

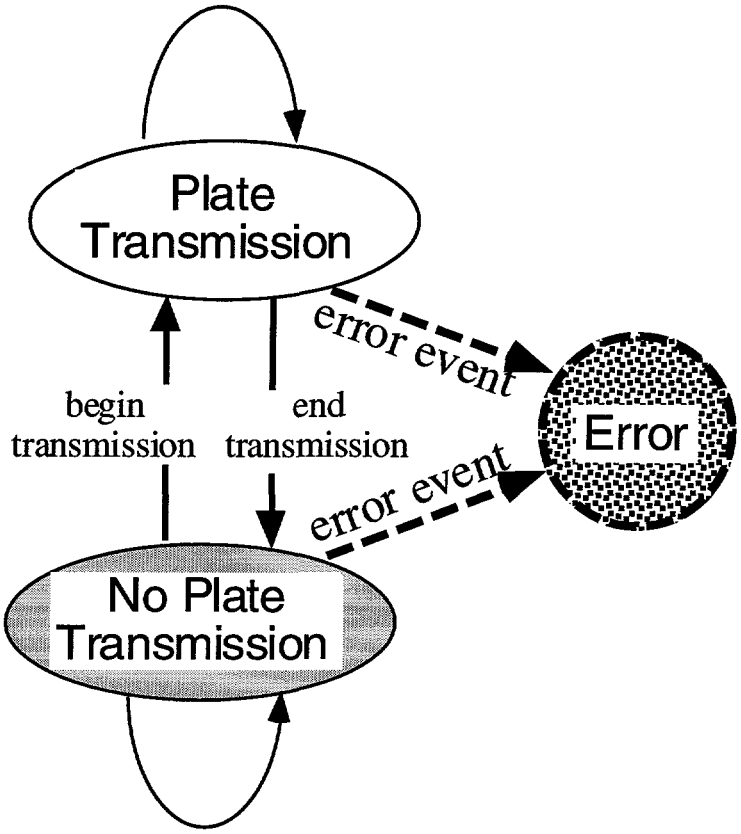

Figure 4. Top Model

The second installment of the requirements:

- The feed belt toggles between transmitting plate and not transmitting plate.
- The turn table toggles between been level with the feed belt and being turned.
- An error occurs when the feed belt transmits the plate while the turn table is turned.

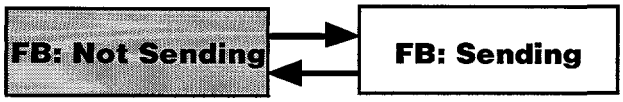

Figure 5. Feed Belt Top Model

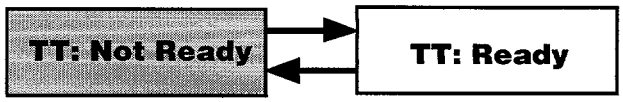

Figure 6. Turn Table Top Model

The third requirement above imposes a constraint on the Cartesian product of the models for the feed belt and the turn table. The second level model is represented on Figure 7. System runs with either a deadlock or an error are possible.

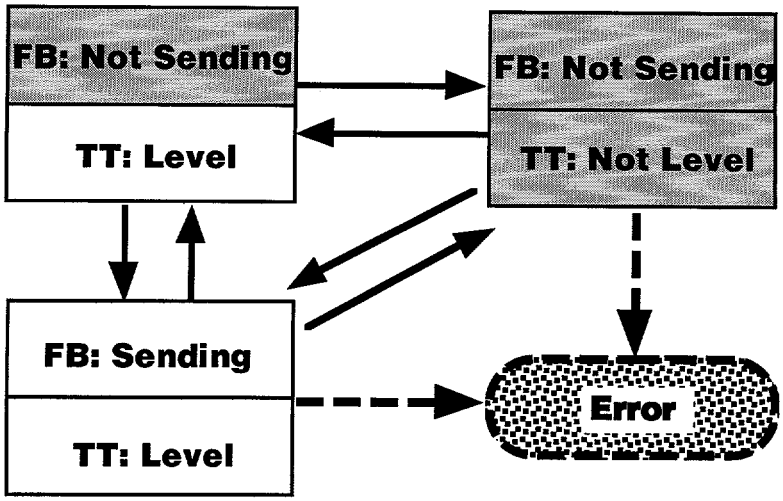

Figure 7. Second Level Model

The correctness preserving map is defined as follows: (FB: Not Sending/TT: Level)↦(No Plate Transmission), (FB: Not Sending/TT: Not Level)↦(No Plate Transmission), (FB: Sending/TT: Level)↦(Plate Transmission), (Error)↦(Error).

The third installment of the requirements:

- An error is generated if:
  - The feed belt stops before a plate transmission is completed;
  - The turn table turns before a plate transmission is completed.
- The turn table is in the state "plate loaded" only after the sensor on the feed belt detects that the transmission is completed.

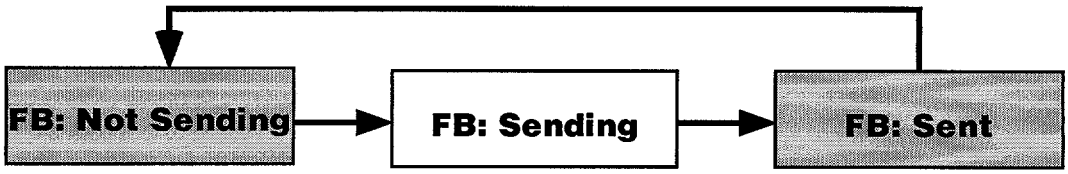

Figure 8. Feed Belt 2nd Level Model

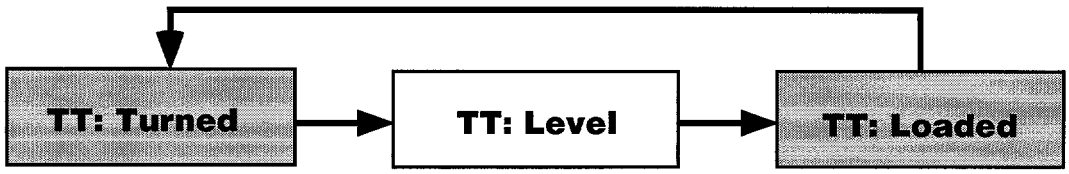

Figure 9. Turn Table 2nd Level Model

The first two requirements caused us two revise models for the feed belt and turn table, since either needs a state designating the end of the plate transmission. The third requirement above imposes a constraint on the Cartesian product of the new models for the feed belt and the turn

table. The third level model is represented on Figure 10. System runs with a deadlock are no longer possible. System runs with an error are possible.

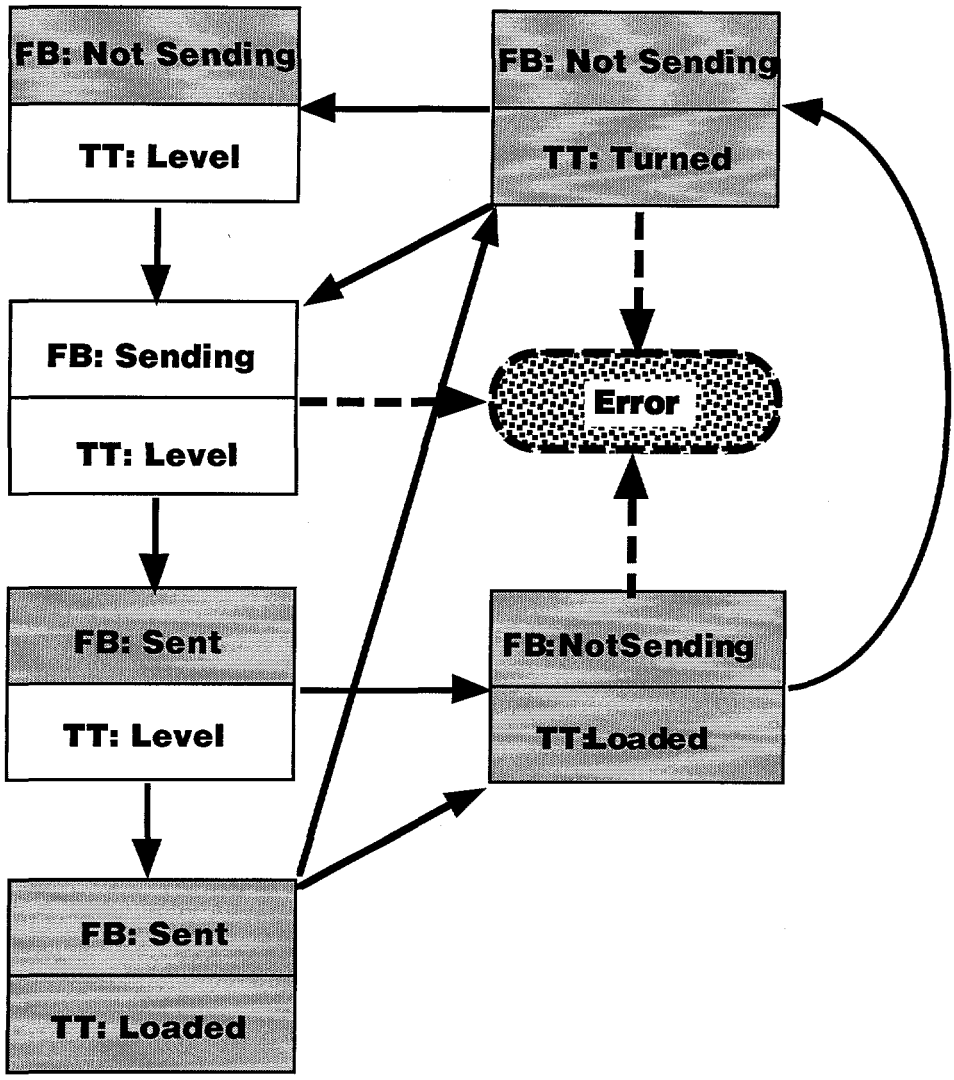

Figure 10. Third Level Model

The correctness preserving map is defined as follows: The top 4 states of the third level model are mapped into corresponding states of the second model. This is an injective map onto the second model of the 4 states. The remaining 3 states are mapped as follows. {(FB: Sent/TT: Level), ↦(FB: Not Sending/TT: Level), (FB: Not Sending/TT: Loaded)↦(FB: Not Sending/TT: Turned), (FB: Sent/TT: Loaded)↦(FB: Not Sending/TT: Level).

The fourth level model which we do not show in this paper, eliminates runs leading to the error state. This model is defined by a strategy which can be represented by 4 distributed communicating automata representing the feed belt, its controller, turn table, and its controller. The correctness preserving map projects this strategy into a subgraph of the third level model not

containing the error state. By the Lemma of section 3, it follows the 4th level model satisfies the complete set of customer requirements.

5. A BASIS FOR AUTOMATED VERIFICATION OF SOFTWARE WITH PARTIALLY DEFINED OPERATIONS

5.1. The Problems to be Solved

5.1.1. Extending Logical Connectors

Consider a typical safety-critical system: a nuclear reactor control system. One of its safety subsystems must issue a shutdown command once the sensors detect that neutron density is above certain critical value *crit*. For simplicity, suppose there are 2 sensors whose measurements are m and n. If the Boolean value of F(m,n)=(n≥*crit* OR m≥*crit*) is true, the shutdown must follow. If any of the sensors fail to deliver a value, the corresponding inequalities do not make mathematical sense, and therefore the logical value of the Boolean expression is undefined in the classical 2-valued (**true**, **false**) logic. This means that we have to extend the meaning of the logical connector OR to the case when one of its Boolean inputs is undefined. Logical connectors such as OR can be extended over 3-valued Boolean domain in a variety of ways. E.g., [Gries 1981] and [Jones 1990] introduced a 3rd Boolean value "undefined", however, while Gries provided an asymmetric extension of OR, Jones provided a symmetrical one. The above example corresponds to the "symmetric" extension of OR. Since a single safety-critical system may need several different extensions of each classical Boolean connector, we need to provide a uniform treatment of all such extensions. This is one of the goals of this section.

5.1.2. Formalizing the D. Gries Technique for Correctness Proofs

Consider the following example from [Gries 1981]. What is the logical value of (x = 0 OR y/x = 5) when x = 0?. The first disjunct holds, while the second has no standard meaning since y/0 is undefined. If, however, we choose an arbitrary value "w" for y/0 thus extending division to a total function, the logical value of (x = 0 OR y/x = 5) will be **true** independently of the choice of w. This is the essence of the following idea of Gries:

- If all partial functions in a formula are somehow extended to total functions, then we can try to prove the formula as if the functions were indeed total;
- If during the proof we would never take an advantage of the values extending the partial functions into total, the proof would be valid.

This technique is very convenient since it enables the classical 2-valued first order logic to be applied to formulas with partial functions. However, in order to make this technique both rigorous and amenable to automation, the following questions must be answered:

- which formulas may be treated in this fashion?
- since the classical Tarski's semantics of classical first-order logic formulas does not treat partial functions, in which sense can we speak about the validity of the proofs within the Gries technique?
- is there a rigorous meta-proof of the validity of the technique?

Another goal of this section is to provide positive answers to the above questions.

5.1.3. Functions whose Argument Lists May Be only Partially Available

Curiously enough, the mere usage of partial functions introduces another problem with semantics of proofs within the realm of total functions. Consider the "selection" function (b ? x : y) from C/C++. It is obviously a total function. However, what is the meaning of (**true** ? 1 : 1/0)? It is 1, even if the third argument is undefined. Thus, although, by itself, (b ? x : y) is total, its usage does not conform to the classical Tarski's semantics, since it does not allow undefined arguments. The third goal of this section is to extend the classical 2-valued 1st order logic to such usages of total functions.

5.1.4. Meaning and Correctness Proofs of Hoare Triples with Partial Functions

In order to allow for partial functions within Hoare triples of the form $\{P\}\mathscr{R}\{Q\}$, we extend their "total correctness" meaning as follows:

- the assertion $\{P\}\mathscr{R}\{Q\}$ is
  - **true**, if precondition P is both *defined* and **true**, then the program $\mathscr{R}$ terminates and upon its completion the postcondition Q is both *defined* and **true**.
  - **false**, otherwise.

Although the Hoare triples are the major mechanism for proving correctness of terminating programs, the current state of the correctness proofs practice does not adequately address Hoare triples with partial functions. Consider an example from [Kaldewaij 1990], an excellent book on program correctness and derivation. It is suggested there (and in many other books and papers, e.g., [Gries 1980; Cohen 1990; Yakhnis, Farrell, Shultz 1994], etc.) that in order to prove a

Hoare triple of the form $P\{x:=E\}Q$, one has to show that $P \Rightarrow Def.E \wedge Q(x/E)$, where $Q(x/E)$ is the result of substitution of E for x, holds. There are three problems with such treatment:

- the expression transformer Def is not formally defined. This makes the approach less amenable to automation;
- the meaning of connector $\wedge$ must be extended to cover undefined inputs. This is done in a limited form in several works (e.g., [Gries 1980; Yakhnis, Farrell, Shultz 1994]);
- even if $\wedge$ is extended, the formula would become undefined if P or Q contain partial functions. Moreover, the occurrence of partial functions in P or Q is quite common.

For instance, consider a program using a one dimensional array f of length 100. Then f is a partial function (over integers) whose domain is the segment [1..100]. If we would want to require some property of f upon the completion of the program then f must be included in the postcondition Q. E.g., {**true**}n := 101{f(n)>0} is **false**, since upon the execution of n := 101 the postcondition f(n)>0 is not defined. Kaldewaij's formula gives us **true** $\Rightarrow$ Def(101) $\wedge$ f(102)>0, which is equivalent to f(101)>0 which is neither **true** nor **false**. Thus an automatic checking based on this proof rule would not yield a definite answer.

Our final goal is to provide a formal definition of Def and to modify the Hoare and Dijkstra proof rules for all the program connectors, so that one would be able to find by automatic means the logical values of the Hoare triples in the presence of partial functions.

5.2. The Existing Research on Partially Defined Operations

Many researchers worked in the area of partial functions and their applications in computing [Stephen Kleene 1936-1950, David Gries 1981, 1983, Horst Reichel 1987, Cliff Jones (VDM) 1986, 1990, Martin Wirsing 1990, Ruth Breu 1991, Yuri Gurevich 1992].

In order to reason about partial functions, 3-valued logic was used by Kleene in his classical "Introduction to Mathematical Logic", 1952. Kleene described several 3-valued logics developed by him (1938) and others (e.g., the Lukacevich logic 1920). One of these 3-valued logics is identical to that of Jones 1986, 1990, however, Kleene's purpose was to elucidate partial functions in recursion theory, rather than to reason about software. Beginning from Gries all the authors used 3-valued logics described in [Kleene 1952] and introduced various versions of explicit domains for partial functions.

Our explicit domains are substantially different from the ones previously considered:

- to accommodate computations with functions whose argument lists may be only partially available, we impose an additional structure on the explicit domains;

- we represent the explicit domains for atomic functions in a form suitable for uniform automated computation of domains for compound terms built up from the atomic partial functions;
- we proved that all 3-valued logics of atomic Boolean functions [described in Kleene 1952] can be reformulated using classical 2-valued 1st order logic;
- we provide a uniform reduction of Hoare triples with partial functions to formulas of classical 2-valued 1st order logic making them more amenable to automated proofs.

5.3. Outline of Our Selected Results on Verification of Software with Partially Defined Operations

As we have demonstrated in the previous examples, while overall computation may be correct, some of the subordinate computations may not yield any definite result or even may not terminate. We would like to make definite conclusions about correctness of such software in a uniform way within 2-valued 1st order logic. This would be the basis for automated verification of correctness of software. We model computations that may not yield any definite result or may not terminate by means of partial functions. Partial functions were dealt with in mathematics rigorously for quite a long time. However, with respect to software there is more difficulty in handling them. This is because, while in mathematics overstepping the domain of a partial function is prohibited and is watched over very closely, computation of a partial function outside its domain on computers is a common occurrence. Another common occurrence is a computation of a "total function" on an invalid input which is the same as regarding the function as partial on a larger domain. This makes it a challenge to reason in an uniform and practical way about using partial functions in software engineering.

The simplified outline of our approach is as follows. For every pair consisting of a piece of software and a requirement imposed upon it (either of which may contain partial functions), we construct a classical 2-valued 1st order logic formula such that:

- all the symbols denoting partial functions are considered as if they denote total functions. Each total denotation coincides with the corresponding partial one over the domain of the partial denotation;
- if it has classical 1st order logic proof then the software is correct with respect to the requirement;
- if its negation has such proof, then the software is faulty with respect to the requirement;
- if neither of the 2 proofs above exist, then nothing can be said about the software;

- Now, in order to check automatically such software, we need to run an automatic theorem prover on the formula.

We proceed as follows. We extend the universe over which we consider the functions occurring in a piece of software by a single value denoting undefined value $\perp$. To every partial atomic function, say, f(x, y) we attach another atomic function Edom.f(z, x, w, y) which is boolean-valued and total over the extended universe and represents a classical formula of 1st order 2-valued logic. We do not distinguish later between that formula and the function. Here z, and w are Boolean variables (i.e., they are taken from $\mathbb{B}$ = {**true**, **false**}) and "Edom" stands for "explicit domain". The meaning of Edom.f(z, x, w, y) is the following:

- the standard set-theoretical domain may be computed as Dom.f = {(x, y) | Edom.f(**true**, x, **true**, y) = **true**};
- if Edom.f(**true**, x_0, **false**, y) = **true** then (x_0, y)$\in$ Dom.f, f(x_0, y) does not depend on y and, moreover, f(x_0, y) may be computed without knowing y;
- if Edom.f(**false**, x, **true**, y_0) = **true** then (x, y_0)$\in$ Dom.f, f(x, y_0) does not depend on x and, moreover, f(x, y_0) may be computed without knowing x.

For every compound term t = f(t_1, ..., t_k) (where t_1, ..., t_k are other terms) we inductively define the expression transformer Def by Def.t = Edom.f(Def.t_1, t_1,..., Def.t_k, t_k). It follows by induction that Def.t is a total Boolean valued function over the extended universe representing a formula of classical 2-valued 1st order logic.

Theorem 1. If Def.t = **true** then the value of t may be computed without attempts to find the values of atomic partial functions outside of their domains.

□

Theorem 2. Let φ be a formula of classical 2-valued 1st order logic (φ does not have free variables. Note that formulas representing Hoare triples are of this kind.) Suppose that some of the functional symbols of φ are interpreted as partial functions over the original universe. Then the following holds:

- If a classical 2-valued 1st order logic proof of Def.φ ($\neg$Def.φ) exists, then Def.φ ($\neg$Def.φ) is **true** over the original universe.

□

Remark 1. Without Theorem 2 the existence of the classical proof mentioned above implies only that the formula Def.φ (¬Def.φ) is **true** over the extended universe.

□

We say that φ is total if Def.φ is **true**. Otherwise, we say that φ is not defined.

Theorem 3. Let φ be as in Theorem 2. If φ is total then:

if a classical 2-valued 1st order logic proof of φ (¬φ) exists, then φ (¬φ) is **true** over the original universe.

□

Theorem 4. A Hoare triple {P}$\mathscr{R}${Q} holds if there is a well-formed classical 2-valued 1st order logic proof of Def(P) ∧ P ⇒ wpp($\mathscr{R}$, Q) using our proof rules. Here, wpp($\mathscr{R}$, Q) is the "weakest precondition for $\mathscr{R}$, Q in the presence of partial functions". The table rules defining wpp is at the end of the paper.

□

Thus in order to check a piece of software $\mathscr{R}$ with respect to a precondition P and postcondition Q, it is sufficient to run a theorem prover on the formula from Theorem 4 for the corresponding Hoare triple. If there is a classical proof of the formula, the software is correct. If there is a classical proof of the negation of the formula, the software is faulty. Otherwise, it is inconclusive.

6. CONCLUSION

6.1. What We Have Achieved

- Provided a basis for automated certification of software/hardware systems.
- Enhanced capability of customers and developers to express requirements for safety-critical systems. By means of extended logical connectors we enable customers and developers to express requirements on system behavior accounting for possible failures of system components (such as potentially faulty sensors or software subsystems, etc.).
- Provided a solid foundation for automating proofs of correctness of safety-critical systems. We developed a systematic (and mathematically sound) way to convert expressions with partially defined operations into expressions acceptable by existing automated tools for mathematical correctness verification.
- Developed mathematical foundations for reasoning about such features of programming languages as operations with argument lists of variable length. Operations of this kind are

intended to be applied when some elements on the argument list are generated by potentially faulty system components. Thus programming languages having such operations (e.g., Eiffel, C/C++, etc.) would be better suited for:

- implementing of safety-critical systems;
- automated proofs of correctness of safety-critical systems.

6.2. Constructing a Prototype Tool for Automated Verification of Safety-Critical Systems via off-the-shelf Technology

We take advantage of two existing tools, TAMPR (an automated term-rewriting system) and OTTER (an automated theorem prover). This work on the prototype tool is joint with Victor Winter (Sandia National Laboratories, High Integrity Software Research). To combine the above tools and the results of the present paper, we do the following:

- Load into OTTER the proof rules and the "explicit domains" described in this paper;
- Given a system requirement with partially defined operations do the following:
  - apply TAMPR (using the definition from the present paper) to construct a formula φ of classical logic representing the domain where the requirement is meaningful;
  - apply OTTER to φ. If OTTER tells us that φ is **true** then the requirement is meaningful, otherwise the requirement is not meaningful. If the requirement is not meaningful, it is not acceptable;
  - if the requirement is meaningful, apply TAMPR to convert the requirement into a formula ψ acceptable by OTTER;
  - apply OTTER to ψ to establish either satisfaction or not satisfaction of the requirement.

REFERENCES

Apt, K. R. (1981) Ten Years of Hoare's Logic, a Survey," *ACM Trans. on Prog. Lang. and Sys.*, 3, 431-483, 1981.

Apt, K. R., Olderog, E. R. (1991) *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991.

Bohorquez, J., Cardoso, R. (1993) Problem Solving Strategies for the Derivation of Programs, *Logical Methods* (J. N. Crossley et al, ed.), Birkhauser, 1993.

Breu, R., *Algebraic Specification Techniques in Object Oriented Programming Environments*, Springer-Verlag, 1991.

Burmeister, P. (1986) A Model Theoretic Oriented Approach to Partial Algebras, *Mathematical research 31*, Akademie-Verlag, Berlin , 1986

Burstall, R.M., Goguen, J.A. (1977) Putting Theories together to Make Specifications, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 1045-1058, 1977.

Cohen, E., *Programming in the 90s: An Introduction to the Calculation of Programs*, Springer-Verlag, 1990.

Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, 1976.

Dijkstra, E.W., Feijen, W.H.J., *A Method of Programming*, Addison Wesley, 1988.

Dijkstra, E.W., Scholten, C.S., *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.

Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

Gumb, D., *Programming Logics*, John Wiley & Sons, 1989.

Gurevich, Y. (1995) Evolving Algebras 1993: Lipari Guide, *Specification and Validation Methods*, pp. 7-36, Oxford University Press, 1995.

Guttag, J.V., Horning, J.J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

Harel, D., *First-Order Dynamic Logic*, Springer-Verlag, 1979.

Hoare, C.A.R., "An Axiomatic Approach to Computer Programming," in *Essays in Computer Science*, C. A. .R. Hoare and C. B. Jones (eds), Prentice-Hall, 1989.

Hopcroft, J., Ullman, J. (1979) *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

Jones, C.B., *Systematic Software Development using VDM*, Prentice-Hall International 1990.

Kaldewaij, A., *Programming: The Derivation of Algorithms*, Prentice Hall, 1990.

Kieburtz, R.B., Shultis, J. (1981) Transformations of FP Program Schemes, *Proceedings of the Conference on Functional Programming and Architecture*, pp. 41-48, 1981.

Lyu, M, editor, *Handbook of Software Reliability Engineering*,McGraw-Hill, 1996.

Loeckx, J., Sieber, K., *The Foundations of Program Verification*, John Wiley & Sons, 1987.

Morgan, C., *Programming from specifications*, Oxford University Press, 1991.

Nerode, A., Remmel, J. B., Yakhnis, A. (1993) Hybrid System Games: Extraction of Control Automata with Small Topologies, Mathematical Sciences Institute, Cornell University, Technical Report 93-102, 61 pp., 1993.

Nerode, A., Remmel, J. B., Yakhnis, A. (1995) Controllers as Fixed Points of Set-Valued Operators, *International Conference on Intelligent Control*, Monterey, California, 1995.

Nerode, A., Remmel, J. B., Yakhnis, A. (1995) Differential Inclusions and Fixed POints for Design and Verification of Hybrid Systems, *Hybrid Systems II, Lecture Notes in Computer Science #999*, pp. 344-358, Springer, 1995.

Nerode, A., Yakhnis, A. (1992) Modeling Hybrid Systems as Games, *Proceedings of the 31st IEEE Conference on Decision and Control*, pp. 2947-2952, 1992.

Nerode, A., Yakhnis, A., Yakhnis, V. (1992) Concurrent Programs as Strategies in Games, in *Logic From Computer Science*, MSRI series (Y. Moschovakis, ed.), pp. 405-479, Springer-Verlag, 1992.

Nerode, A., Yakhnis, A., Yakhnis, V. (1993) Distributed Concurrent Programs as Strategies in Games, in *Logical Methods* (J. N. Crossley et al, ed.), pp. 624-653, Birkhauser, 1993.

Owicki, S., Gries, D., An Axiomatic Proof Technique for Parallel Programs, *Acta Informatica*, No. 6, pp. 319-340, 1976.

Reichel, H. (1987) *Initial computability, Algebraic Specifications, and Partial Algebras*, Claredon Press, Oxford, 1987

Stilman, B. (1995b) Multiagent Air Combat with Concurrent Motions, Symposium on Linguistic Geometry and Semantic Control, *Proc. of the First World Congress on Intelligent Manufacturing: Processes and Systems,* Mayaguez, Puerto Rico, Feb. 1995.

Tucker, J.V., Zucker, J.I., *Program Correctness over Abstract Data Types with Error-State Semantics*, North-Holland, 1988.

Wirsing, M., "Algebraic Specification," in *Handbook of Theoretical Computer Science*, pp. 675-788, Elsevier Science Publishers B.V., 1990.

Woodcock, J., Loomes, M., *Software Engineering Mathematics*, Pitman, 1988.

Woodcock, J.C.P., "The Rudiments of Algorithm Refinement," The Computer Journal, vol. 35, num. 5, October 1992.

Yakhnis, A. (1989) Concurrent Specifications and their Gurevich-Harrington Games and Representation of Programs as Strategies, *Transactions of the 7th (June 1989) Army Conference on Applied Mathematics and Computing*, pp. 319-332, 1990.

Yakhnis, A., Yakhnis, V. (1990) Extension of Gurevich-Harrington's Restricted Memory Determinacy Theorem: a Criterion for the Winning Player and an Explicit Class of Winning Strategies, *Annals of Pure and Applied Logic*, Vol. 48, pp. 277-297, 1990.

Yakhnis, A., Yakhnis, V. (1993) Gurevich-Harrington's Games Defined by Finite Automata, *Annals of Pure and Applied Logic*, Vol. 62, pp. 265-294, 1993.

Yakhnis, A., Yakhnis, V., First-Order Basis for Automated Checking of Software Build from Partial and Nondeterministic Operations, to be submitted to *CADE-13 Workshop on Mechanization Of Partial Functions*, July 30, 1996.

Yakhnis, A., Yakhnis, V., Semantics of Concurrent Communicating Objects, in preparation.

Yakhnis, V. (1989) Extraction of Concurrent Programs from Gurevich-Harrington Games, *Transactions of the 7th (June1989) Army Conference on Applied Mathematics and Computing*, pp.333-343, 1990.

Yakhnis, V., Farrell, J., Shultz, S. (1994) Deriving Programs Using Generic Algorithms, *IBM Systems Journal*, vol. 33, no. 1, pp. 158-181, 1994.

Yakhnis, V., Stilman, B. (1995a) Foundations of Linguistic Geometry: Complex Systems and Winning Conditions, *Proceedings of the First World Congress on Intelligent Manufacturing Processes and Systems (IMP&S)*, February 1995.

Yakhnis, V., Stilman, B. (1995b) A Multi-Agent Graph-Game Approach to Theoretical Foundations of Linguistic Geometry, *WOCFAI 95*, Paris, 3-7 July 1995.