Notice of Violation of IEEE Publication Principles

"Development of Embedded Software with Component Integration Based on ABCD Architecture"
by Haeng-Kon Kim, Roger Y. Lee, and Hae-Sool Yang
in the Proceedings of the 4th Annual ACIS International Conference on Computing and Information Science (ICIS'05), 14-16 July 2005, pp. 54-60

After careful and considered review of the content and authorship of this paper by a duly constituted expert committee, this paper has been found to be in violation of IEEE's Publication Principles.

This paper was found to be a near verbatim copy of the paper cited below. The original text was copied without attribution (including appropriate references to the original author(s) and/or paper title) and without permission.

Due to the nature of this violation, reasonable effort should be made to remove all past references to this paper, and future references should be made to the following article:

"An Architecture for Embedded Software Integration Using Reusable Components"
by Shige Wang and Kang G. Shin,
in the Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, ACM Press, 17-19 November 2000, pp. 110-118

Development of Embedded Software with Component Integration based on ABCD Architectures

Haeng-Kon Kim
Dept. of Computer Information & Communication Engineering, Catholic University of Deagu, Korea

Roger Y. Lee
Software Engineering & Information Technology Institute, Central Michigan University, USA

Hae-Sool Yang
Hoseo Graduate School of Venture, Korea

Abstract

The state-of-art approaches to embedded real-time software development are very costly. The high development cost can be reduced significantly by using model-based integration of reusable components. To the ABCD (Architecture, Basic, Common and Domain) architecture, we propose an architecture that supports integration of software components and their behaviors, and reconfiguration of component behavior at executable-code-level. In the architecture, components are designed and used as building blocks for integration, each of which is modeled with event-based external interfaces, a control logic driver, and service protocols. The behavior of each component is specified as a Finite State Machine (FSM), and the integrated behavior is modeled as a Nested Finite State Machine (NFSM). These behavior specifications can be packed into a Control Plan program, and loaded to a runtime system for execution or to a verification tool for analysis. With this architecture, embedded software can be constructed by selecting and then connecting (as needed) components in an asset library, specifying their behaviors and mapping them to an execution platform. Integration of heterogeneous implementations and vendor neutrality are also supported. Our evaluation based on machine tool control software development using this architecture has shown that it can reduce development and maintenance costs significantly, and provide high degrees of reusability..

1. Introduction

Agile and low-cost software development for real-time embedded systems has become critically important as embedded systems and devices are being used widely and customized frequently. However, the current practice in embedded software development relies heavily on ad-hoc implementation and labor-intensive tuning, verification and simulation to meet the various constraints of the underlying application, thereby incurring high development and maintenance costs. Although component-based software development and integration are known to be efficient for software development [1,2,] such an approach is neither well-defined nor well-understood in the embedded real-time systems do-main. Typically, embedded systems software consists of various device drivers and control algorithms, which usually exist as software components and are preferred to be reused for similar applications. Unfortunately, these components may contain dedicated information for some physical processes, and hence, can not be reused only based on their functions. The physical process for a target domain, on the other hand, is relatively static and can be modeled through component behaviors. Thus, components can be designed with customizable behavior mechanisms so that they can be reused for different applications. Besides supporting reuse and reconfiguration, the architecture for embedded software should also support separation of the specification and verification of non-functional constraints from those of functions. Such separation is essential for high-level implementation-independent specification and verification of non-functional constraints such as timing and resource constraints [4].

In this paper, we present an architecture that supports the above desired features for embedded software integration. Our reusable component model separates function definitions from behavior specifications, and enables behavior reconfiguration after structural composition. Components can be structurally integrated using their communication ports, through which acceptable external events can be ex-changed to invoke target operations. The integrated software can then be mapped onto various platform configurations by customizing service protocols of components.

Behaviors of integrated software in our architecture are modeled as *Nested Finite State Machines* (NFSMs). The NFSM model supports compositional behavior specifications. It further supports incremental and formal behavior analysis. The behavior correctness of such an integrated system can be verified using an approach similar to that in [2]. The behaviors specified in other models or

Proceedings of the Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05)
0-7695-2296-3/05 $20.00 © 2005 IEEE

languages can be converted to this model using translators. The integrated behaviors can then be specified in a *Control Plan* program for remote and runtime behavior reconfiguration. Our architecture also separates other non-functional constraints, especially timing and resource constraints, from functionality and behavior integration so that these constraints can be analyzed and verified incrementally and as early as at design phase.

2. Component structure

Components are pre-implemented software modules and treated as building blocks in integration. The integrated embedded software can be viewed as a collection of communicating reusable components. Figure 1 shows the embedded software constructed by integrating components.

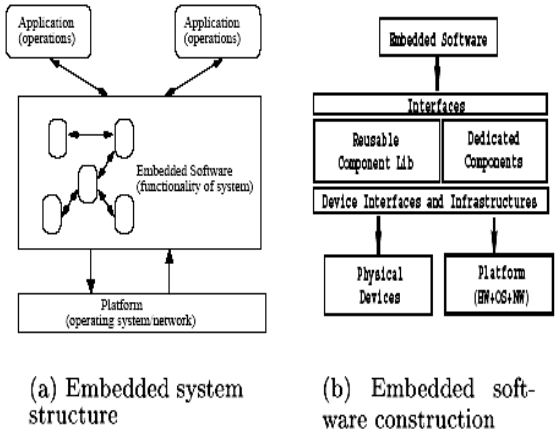


Figure 1. Integration of embedded software components

The component structure defines the required information for components to cooperate with others in a system. Our software component is modeled as a set of external interfaces with registration and mapping mechanisms, communication ports, control logic driver and service protocols, as shown in Figure 2.

External interfaces. External interfaces define the functionality of the component that can be invoked outside the component. In our model, external interfaces are represented as a set of acceptable events with designated parameters. A component with other forms of external interfaces, such as function calls, can be integrated into the system by mapping each of them to a unique event. Using events as external interfaces enables operations to be scheduled and ordered adaptively in distributed and parallel environments, and enables components from different vendors (possibly implemented

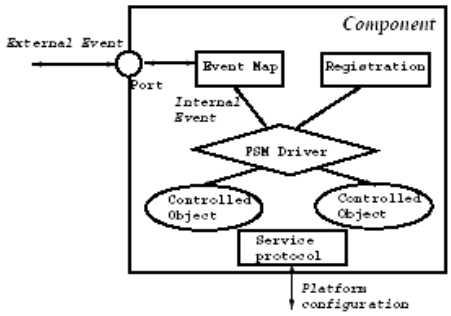


Figure 2. Reusable component structure

with different considerations) to be integrated into the system without the source code.

External interfaces are generally defined as global (external) events, which are system-wide information. A customize-able event mapping mechanism is devised in a component to achieve the translation between global events and the component's internal representations. A registration mechanism is further equipped to perform runtime check on received events. Only those operations invoked by authorized and acceptable events can be executed.

Communication ports. Communication ports are used to connect reusable components, i.e., they are physical interfaces of a component. Each reusable component can have one or more communication ports. The number of ports needed for a component can be determined and customized by the system integrator. Different types of ports with different service protocols can be determined and customized by the system integrator. Different types of ports with different service protocols can be selected to achieve different performance requirements. Multiple communication parties can share one port.

Finite State Machine Driver. The control logic driver, also called the FSM driver, is designed to separate function definitions from control logic specifications, and support control logic reconfiguration. The FSM driver can be viewed as an internal interface to access and modify the control logic, which is traditionally hard-coded in software implementation. Every component that controls behaviors should have a FSM driver inside itself. Control logic of a component can now be modeled as a FSM and fully specified in a table form [30], or a *state table*. The FSM driver will then generate commands to invoke operations of the controlled objects at runtime according to the state table and the events received.

The FSM driver and state tables also enable remote and runtime control logic reconfiguration. The state table can be treated simply as data and passed around the system. A state table can be partitioned into several small pieces with only one loaded to the FSM driver at a time. A component can also be reconfigured with a different state table when the

external environment changes or upon other components' requests. This is even more useful for devices in a system with limited resources and unreliable environments, such as an in-vehicle control system.

Service protocols. Service protocols define the execution environment or infrastructures of a component. Example service protocols include scheduling policies, inter-process communication mechanisms and network protocols. A component can be customized for use in different environments by selecting different service protocols. Such selection is based on the mechanisms available on a platform and performance constraints (such as timing and resource constraints) of the system.

3. Behavior model and specifications

Embedded systems normally deal with mission- or safety-critical applications. Hence, the behavior of software should be thoroughly analyzed before its implementation. In our architecture, the behavior of integrated software is modeled as a NFSM while component behaviors are modeled as traditional FSMs. Both of them can be formally specified and verified.

3.1. Control Logic Specification

Control logic specifications are used to define the static behavior of the control logic of a component. Control logic for integrated software is modeled as a NFSM. A NFSM contains a set of traditional 'flat' FSMs organized hierarchically. A NFSM at level i, M_i, can be defined as:

$$M_i =< S_i\, I_i\, O_i\, T_i\, s_{i0} > \textit{(level-I FSM)}$$

Where S_i is a set of states of the i-th level FSM, I_i and O_i are a set of inputs and outputs, respectively, T_i is a set of transitions, and s_{i0} the initial state of M_i. A non-initial state of M_i may contain a set of FSMs at the (i + 1)-th level.

The control logic of a component is modeled as a FSM (if it is a leaf node in the component hierarchy) or a NFSM (if it is a non-leaf node in the component hierarchy) and can be implemented independently. Only the top-level FSM of a component is visible during in integration.

3.2. Operation Specifications

Operation specifications define the desired runtime behavior and can be specified as a programmed operation sequence that will trigger the component actions when there are no other interferences. The operation specifications will generate predefined input events for a component FSM at run-time.

The operation specifications consist of a list of operation descriptions in the form of:

[WHEN state] [INPUT e_{input} [PARAM parameter]]
OUTPUT e_{input} [PARAM parameter]

Where *state* is the current state, e_{input} is the received event, e_{output} is the event to send out, and *parameter* is the data attached to the corresponding event. Fields in square brackets are optional. If a field is not specified, the value of the field will be ignored when executed. Such structure simplifies the specifications for pure state-triggered operations and event-triggered operations.

Events used in operation specifications should be global events for portability and reusability. However, operation specifications using internal events specific for a component can be integrated into the overall specifications as attached data of a global event. This enables runtime reconfiguration of operation sequences for a component. The process of such a global event results in execution of the new attached operation specifications in the component.

3.3. Specifications in Control Plan

A program that contains control logic and operation sequences for components is called a Control Plan. A control plan consists of two parts: *logic definitions* and *operation specifications*, corresponding to the control logic and operation specifications, respectively. The structure of a control plan is shown in Figure 3:

```
FSM{label}[location]
        StateTable-entry1
        StateTable-entry2
        ......
ENDFSM

OPERATION{label}[location]
        operation-element1
        operation-element2
        ......
ENDOPERATION
```

Figure 3. Structure of control plan program

It is possible for a control plan to have multiple FSM and OPERATION block for one component for reconfiguration. A block can also be attached to an event in either FSM or OPERATION block as data to pass around. More details of control plan specification and execution can be found in [7,8].

3.4. Behavior Specifications in Other Models

Since different subsystems of an embedded system may deal with different physical processes, it is possible that component behaviors are expressed in other models. Although the system integration of components in multiple models is an active research [9] and the results are directly applicable to our architecture, we adopt translators to solve the problem of integrating heterogeneous models. In our architecture, a translator is designed and implemented as a software component that converts a specification in a given formalism of a model to a control plan and NFSM. Translators are domain-specific and specification language-dependent, meaning that each translator can only convert programs in a designated specification language to control plan. Thus, several translators may be required in a system if there are programs written in several different specification languages.

4. System integration

Software integration includes component selection and bind-ing, and control plan construction (both control logic and operation sequence). A runtime system can be generated by mapping the integrated software onto ABCD Architectures.

4.1. Composition Model

The composition model defines how software can be integrated with given components. Since each reusable component is implemented with a set of external interfaces that uniquely define its functionality, components can be selected based on the match of their interfaces and design specifications. The integration of reusable components can be viewed as linking the components in integrated can be viewed as linking the components with their external interfaces.

Reusable components in integrated software are organized hierarchically to support integration with different granularities, as illustrated in Figure 4. The behavior of an integrated component can then be modeled as integration of its member component behaviors. The control logic and operation sequences of each component can be determined individually and specified in a *Control Plan.* The behavior specifications can further be classified as device-dependent behaviors and device-independent behaviors. The device-independent behaviors depend only on the application level control logic, and can be reused for the same application with different devices. The device-dependent behaviors are dedicated to a device or a configuration, and can be reused for different applications with the same device.

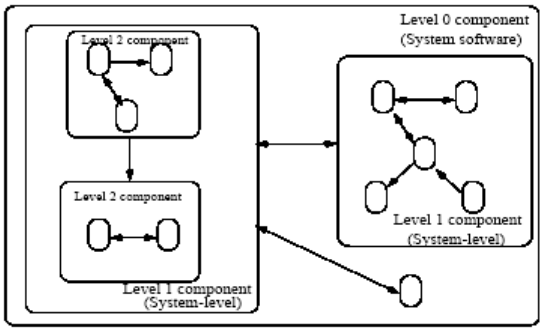


Figure 4. Hierarchical composition model

With such a composition model, both components for low-level control such as algorithms and drivers and for high-level systems can be constructed and reused.

4.2. Runtime System Construction

The integrated software obtained from the composition model cannot be executed directly on a platform since the composition model only deals with functionality. To obtain executable software, components have to be grouped into tasks, which are basic schedulable units in current operating systems.[1] Each task needs to be assigned to a processor with proper scheduling parameters (e.g., scheduling policy and priority) determined by an appropriate real-time analysis. Also, communications among components should be mapped to the services supported by the platform configuration. After these pieces of information are obtained, the components can be mapped to the platform by customizing their service protocols. Figure 5 shows the mapping form functional integrated software to a runtime system with our architecture.

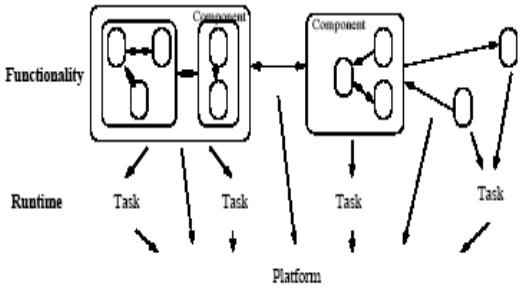


Figure 5. Runtime system generation from composition

5. Evaluation

We evaluated our architecture by constructing two software prototypes for machine tool control with the same set of reusable components. The software is running on two control boxes (with their own processors and memory) connected with a peer-to-peer Ethernet. The software components are implemented with the proposed structures and mechanisms. The evaluation is carried by examining the reusability of components and reconfigurability of integrated software.

5.1. Mobile Motion Controller

We first developed motion control software for a 3-axis milling machine, called Mobile. This controller is used to dynamically coordinate 3-axis motion with given algorithms and computed federate based on sensed forces.

The reusable components include control algorithms, phys-ical device drivers and subsystems. Some high-level compo-nents used in the Mobile motion controller are:

- **AxisGroup:** receives a process model form the user or predefined control programs, and coordinates the motion of the three axes by sending them the corresponding set points.
- **Axis:** receives set points from AxisGroup and sends out the drive signal to the physical device according to the selected control algorithm (PID or FUZZY).
- **G-code Translator:** translates a G-code program into a control plan.
- **Force Supervisory:** calculates the federate for AxisGroup at runtime.

Figure 6 shows the corresponding software structure.

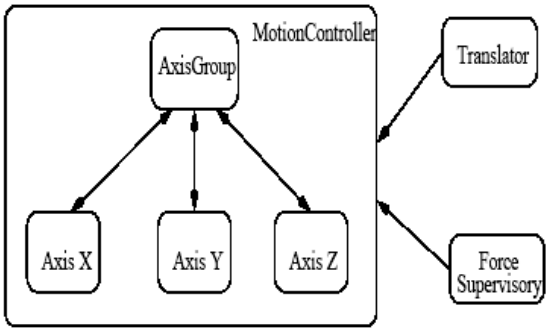
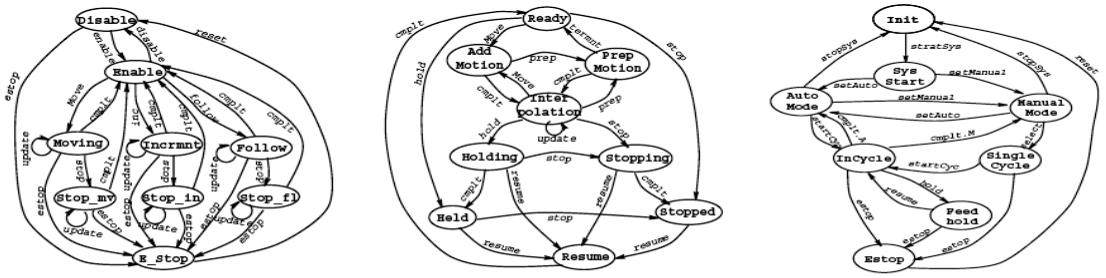


Figure 6. The structure of the Mobile motion control software

The test application on the Mobile is a sequence of milling operations. The integrated behavior of the Mobile controller software consists of FSMs of Axis and AxisGroup components, and a G-code program for operation sequences. A higher-level motion control FSM is developed to specify the overall machine-level control logic. Figure 7 shows the control logic of each component and Figure 8 shows the desired operations.

Reconfiguration to include broken tool detection. A broken tool detection algorithm is then developed and integrated into the motion controller to evaluate the reconfigurability of the software. A broken tool detection algorithm is developed separately and implemented as an individual component. The function of the broken tool detection component is to detect abnormal forces at runtime, and send a stop signal to the motion controller when such a force is observed. The software structure with the broken tool detection algorithm is shown in Figure 9.



(a) Axis FSM (b) AxisGroup FSM. (c) Machine-level FSM.

Figure 7. Behavior specifications of Tobotool motion controller.

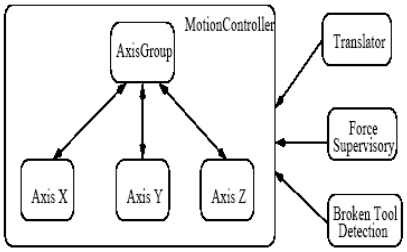


Figure 9. Software with broken tool detection.

To react to the new signal from the broken tool detection algorithm, the machine-level control logic needs to be changed, while the rest of behaviors remain the same. Figure 10 shows the new machine-level FSM.

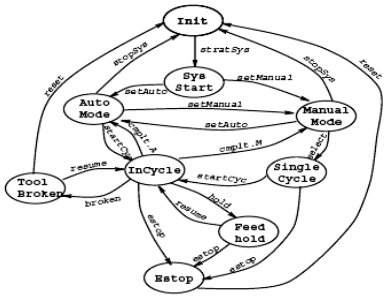


Figure 10. Machine-level FSM with broken tool detection

5.2. Reconfigurable Machine Tool Controller

We then modified the Mobile motion control software to construct another motion control software for a reconfigurable Machine Tool (RMT). RMT is a modularized and composable machine tool with 2-axis and a 2-position discrete device. Unlike the Mobile, the RMT motion controller needs neither coordinated motion nor monitoring. The same axes and translator components are used to con-struct the RMT motion controller. A new component, *Spindle*, is added into the system to control the discrete device. The software structure for RMT controller is illustrated in Figure 11.

The behavior specifications for Axis components are the same as those used for the Mobile. The behavior specifications for the new added Spindle component is simple with only 3 states representing the position of in or out and the situation of *estop,* and transitions with corresponding events, as shown in Figure 12.

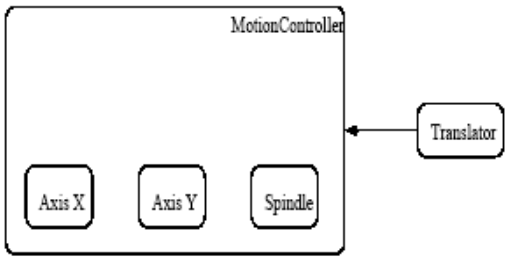


Figure 11. The structure of the RMT motion controller software

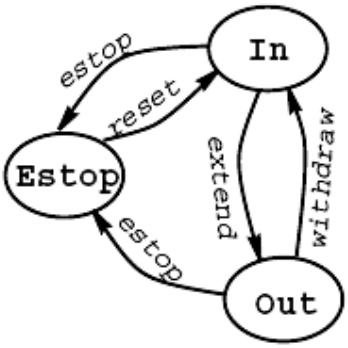


Figure 12. FSM for the spindle component.

Since the new Spindle component introduced new control logic in to the system, the overall machine-level FSM was implemented, as shown in Figure 13. The G-code program

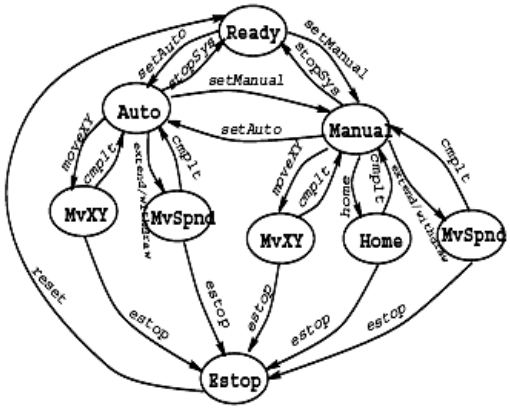


Figure 13. The machine-level FSM for the RMT

5.3. Evaluation Results

According to our experience in implementing these controllers, the proposed architecture provides excellent reusability and reconfigurability of software. The component structure separates the functionality definition from the behavior specification so that the component can be reused for different applications. Table 1 illustrates the numbers of components and behavior specifications in both controllers and the number of mobile components and behaviors reused in RMT. We were able to achieve over 80% reuse of components. On the other hand, the reuse of behavior specifications was not at the same high percentage because a lower-level behavior change may require changes at higher-levels. However, since the behavior specifications and component functional specifications have been separated, the behavior changes will not affect the integration of components, and can be done with- out much of programming skills. These numbers are also consistent with the fact that functions are stable regardless of the application but the behaviors are not.

Table 1. Number of components and behavioral specifications

	Mobile	RMT	Reused
# of components	52	37	34
# of behavioral spec	6	5	2

It also took less effort to reconfigure an existing controller or construct a new controller. As illustrated in Table 2, reconfiguring the Mobile controller software using our architecture with broken tool detection took much less effort than doing so to the software implemented using traditional approaches (0.8 human-month vs. 2 human-month). The time spent on the RMT software construction was also reduced by 50%.

Table 2. Efforts needed for two application with traditional and proposed architectures (in human month)

	Mobile reconfiguration	RMT construction
Traditional approach	2	8
Proposed approach	0.8	4

Although the proposed architecture demonstrated benefits in software construction, we experienced some performance penalties associated with it. Table 3 shows the execution times of the Motion Controller component,[3] collected by a special designed hardware, VMEStop Watch card, which has a built-in high resolution timer 25 nanoseconds

Table 3. Execution times for software constructed with different architectures (in millisecond).

	Mobile	RMT
Traditional approach	2.1	1.3
Proposed approach	3.0	1.5

6. Conclusions and future work

In this paper, we presented a component-based ABCD architecture for embedded software integration. This architecture defines components and a composition model as well as a behavior model. A reusable component in our architecture is modeled with a set of events as external interfaces, communication ports for connections, a control logic driver (FSM driver) for separate behavior specification and recon-figuration, and service protocols for executing environment adaptation. Such a structure enables multi-granularity and vendor-neutral component integration, as well as behavior reconfiguration.

The control logic of the integrated software is modeled using the formal method of NFSM, where each FSM rep-resents the control logic of each component in the integration. The control logic of each component is specified in a state table separately form the component implementation, and can be reconfigured remotely and dynamically. Verification can also be done independently of implementation, and incrementally as the integration continues.

References

[1] A. M. Zaremski and J. M. Wing, "Specification matching of software components", ACM Transactions on Software Engineering & Methodology, 6(4):333-369, October 1997.

[2] B. Henderson-Sellers, B. Unhelkar, OPEN modeling with UML, Addison-Wesley, 2000.

[3] D.F. D'Souza, A.C. Wills, Objects, Components, and Frameworks wit UML, Addison-Wesley, 1998.

[4] P. Allen, Realizing e-Business with Components, Addison-Wesley, 2001.

[5]. X.Cai, M.R. Lyu, K. Wong, Component-Based Soft-ware Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes, Pro-ceedings APSEC 2000, Seventh Asia-Pacific Software Engineering Conference, Singapore, December 2000, pp 372-379

[6] C.D. Turner and D.J. Robson „The state-based testing of CBD programs" Proc. IEEE Conf. Software Maintenance, 1993, pp. 302-310.