Reconciling behavioral mismatch through component restriction

Mark Marchukov
University of Virginia
Dept. of Computer Science
Charlottesville, VA 22903
USA
+1 804 982 2292
march@cs.virginia.edu

Kevin Sullivan
University of Virginia
Dept. of Computer Science
Charlottesville, VA 22903
USA
+1 804 982 2206
sullivan@cs.virginia.edu

ABSTRACT

In component-based software development there are often mismatches between system-level requirements and component behaviors. In general, bridging such mismatches requires mutual adaptation of system requirements and components. One kind of mismatch occurs when components permit behaviors that are not permitted by the system-level requirements. We identify restriction, the disabling of component behaviors, as an important way to bridge such mismatches. Unlike extension, which is well studied, restriction has received little attention. We present a model for reasoning about requirements for restriction, and a corresponding technique for implementing restriction, based on matching of partial models of component behaviors against state-machine-based partial system specifications. Our approach respects several difficulties in component-based development: (a) behaviorally complex components, (b) poorly documented component specifications, (c) inability to change core component implementations, and (d) a general lack of built-in restriction mechanisms in practice. To address these difficulties we use lightweight incremental specification of component operations, obtained by reverse-engineering, and external adaptors that adjust the behaviors of components by manipulating their input streams. We describe our experience using this approach to restrict shrink-wrapped package components in the Galileo fault-tree analysis tool.

Keywords
component-based development, behavioral mismatch, restriction

1. INTRODUCTION

Component-based software development (CBSD) is construction of software systems largely out of pre-fabricated executable parts called *software components*, that are usually obtained from commercial vendors [19]. The rationale behind CBSD is that massive reuse of the functionality of existing, market-proven components will lead to shorter development time and better product quality. The key assumptions of CBSD differ significantly from those of traditional coding-based approach to software development. In particular, system integrators normally cannot modify the implementations of components they use, nor is the source code of components available to them.

Like any other software project, a CBSD project is guided by a set of requirements for the run-time behavior of the system under development. We call these *system-level requirements*. Depending on context, we will use the term *behavior* to denote the set of all sequences of externally visible input and output actions that a system can exhibit at run-time (*the* behavior of a system), or one such sequence (*a* behavior). The behavior of a component-based system is synthesized out of behaviors of its components that we will collectively call its *implementation base*. Thousands of components are currently available on the market. Examples include ActiveX controls [1], Java beans [9] and Microsoft Windows applications with built-in support for OLE Automation [4], such as the members of Microsoft Office application suite. In general, any piece of executable code that can be activated from another application can serve as a component. As a rule, for most new systems there is no single component, or a collection of components whose behavior immediately satisfies all the system-level requirements. A typical situation observed in almost any CBSD project is one of mismatch between the required behavior and the behavior that can be immediately obtained from a component or a set of components. We call this situation *behavioral mismatch*. We identify two types of mismatches. A *negative* mismatch occurs when the implementation cannot or is not guaranteed to exhibit a required behavior. A *positive* mismatch takes place when some of the behaviors that the implementation may exhibit is prohibited by the system-level requirements.

Bridging mismatches between system-level requirements and the collective behavior of the system implementation base is at the heart of component-based development. It usually involves a combination of the following three approaches: (1) changing the behavior of components by adaptation and integration; (2) changing the system-level specification; (3) searching for an alternative implementation base whose behavior better matches given system-level requirements.

We concentrate on the first of these three approaches, namely, changing the behavior of implementation base in order to bring it in correspondence with system-level requirements. There has been substantial research in this area, however most of it has concentrated on reconciling negative mismatch through *extending* components or systems of components so that they can exhibit new behaviors. Inheritance [20] has been studied and applied mostly as a class extension mechanism. Frameworks exploit inheritance as a mechanism for extending collections of collaborating classes. Microsoft's Component Object Model [3], one of the most widely used standards for software components, supports *aggregation*, as an alternative to inheritance. In our previous work [17] we demonstrated that aggregation is difficult to use for purposes other than component extension. Work on extensible operating systems [2] and adaptable binary programs [7] addresses the problem of dynamic extension of executable binary programs. Research on component integration [16] and specific integration mechanisms for CBSD [15] also deals with the problem of extending a set of components with integration infrastructure, so that the resulting system satisfies certain new requirements.

In this paper, we identify the dual problem of positive behavioral mismatch as being equally important, especially for systems composed of large commercial packages with extensive functionality and rich behavior. Positive mismatch has received comparatively little attention from the research community and industry. Bridging a positive behavioral mismatch requires preventing a component-based implementation from exhibiting behaviors that violate system-level requirements. We call the corresponding operation over components *component restriction*.

In the next section we present an example that illustrates the need for restriction in a real component-based application. Section 3 discusses the rationale for restriction and the conditions under which it is necessary. Section 4 presents a formal model of component-based systems that makes our notion of behavioral mismatch precise and serves as a basis for our approach to implementing one kind of restriction. In section 5 we demonstrate how restriction can be implemented in a way that uses matching partial models of component behaviors against the relevant system-level requirements expressed in a state-machine-based form. In section 6 we describe an application of our method to restricting the implementation of a real component-based system. Section 7 describes related work. Section 8 discusses future work and concludes.

2. A MOTIVATING EXAMPLE

In this section we give an example of positive behavioral mismatch in an existing component-based application: a fault-tree analysis tool called Galileo [18]. Galileo supports

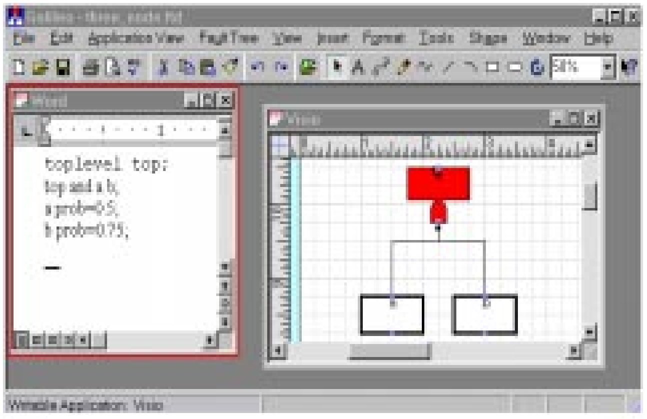
```
toplevel top;
top and a b;
a prob=0.5;
b prob=0.75;
```

Figure 1. The Galileo fault tree analysis tool.

a reliability engineering computation, the details of which are not important here. Galileo provides an interface that permits the user to edit tree-based system reliability models, called fault-trees, in either textual or graphical form. In order to build at low cost a richly functional tool that also leverages existing knowledge of PC-based application interfaces, Galileo uses Microsoft Word [8] as a component to implement its text editing and display functions, and Visio Technical [23] as a component to implement graphical editing and display (Figure 1). For performance reasons, Galileo does not require the graphical and textual views to be always consistent with each other. Instead, Galileo's specification contains a weaker consistency requirement: Once the user makes a change to one of the views, she must not be able to edit the other view without first issuing a special command that synchronizes the contents of the views. However, the user is free to browse through either view at any time.

This *semi-consistency* requirement ensures that the current state of the model is always visible to the user through one of the views. Semi-consistency requires that Galileo may not exhibit behaviors containing a pair of actions that change different views and are not separated by a synchronization request.

Obviously, a "raw" implementation base consisting of unconstrained Word and Visio running side-by side may easily exhibit such behaviors. Nothing prevents the user from making inconsistent changes into a document containing a textual definition of a fault tree, and the Visio diagram containing the graphical representation of the same tree. Enforcing semi-consistency at the implementation level requires that the editing capabilities of Word and Visio must be periodically disabled. When a Word document containing the textual representation of a fault-tree is modified, the Visio diagram with the graphical representation of that tree must become view-only until the user issues a synchronization request, and vice versa. This will effectively restrict the Word+Visio implementation base to only those behaviors that satisfy the semi-consistency requirement.

Another system-level requirement that Galileo imposes on its implementation says that the user must not be able to terminate Word or Visio separately. This requirement was not a part of the original Galileo specification, but became critical once the designers of Galileo made a decision to use a standard application integration framework based on Microsoft's Active Document Architecture, which is a part of Microsoft's OLE [4]. Reliable functioning of the current implementation of Galileo is not guaranteed if Word or Visio is terminated in the middle of a session, while the other component is still running. Like semi-consistency, this requirement rules out certain possible behaviors of Galileo's implementation, namely those where termination of Word or Galileo precedes a request to end the current session of the tool as a whole.

3. POSITIVE BEHAVIORAL MISMATCH

The Galileo example demonstrates that positive behavioral mismatch is not merely a theoretic possibility. In this section we identify two sources of mismatch and discuss its possible bridging techniques.

3.1 Enforcing system-wide abstractions

Recent case-studies of industrial CBSD projects [5,21,22] make clear that only a small part of system-level requirements can be reasonably defined before the initial search for components begins. The details of system behavior and feature set come from individual components in the implementation. For example, Galileo uses Word's search-and-replace mechanism to allow the user to rapidly change repetitive fragments of a fault tree.

Some component behaviors, however, cannot be inherited by the system in this manner, because they allow the user to break the abstraction that the system designer intends to maintain. System-wide abstractions that the components of the system are not designed to support are the first source of positive behavioral mismatch. For example, an abstraction that Galileo attempts to present to its user is a single fault-tree being edited through two separate views. The goal of the semi-consistency requirement is to enforce this abstraction. But because Word and Visio have not been specifically designed to work together as coherent views of a single abstract model, they allow the user to arbitrarily modify their contents at any time. Thus, the user can inadvertently or intentionally break the abstraction that the designers of Galileo intend. Although this will not render the tool useless, it decreases its usability.

3.2 Maintaining system integrity

Another source of positive behavioral mismatch comes not from the original intentions of the system designer, but from the requirements of a particular implementation framework that he uses. For example, taking advantage of a standard component integration infrastructure may significantly reduce the amount of integration effort needed in a component-based project [18]. However, such an infrastructure may fail to function properly, leading to system crashes, unless certain state invariants always hold for the implementation base. For example, components may have to always be running or ready to receive event notifications from the integration infrastructure. A requirement to maintain the system-wide invariants disallows certain possible behaviors of the implementation, creating a condition of positive behavioral mismatch.

3.3 Putting the burden on the user.

A positive behavioral mismatch manifests itself only when a behavior that a component-based implementation exhibits during an execution actually violates a system-level requirement. The behavior of a deterministic component is controlled by the input that the component receives from the user. Given a sufficiently "careful" user who never exercises her option to force a system into an illegal behavior, the implementation may never actually violate any requirements. For example, if the user always presses the synchronization button when switching from changing the graphical view of Galileo to changing its textual view, and back, the semi-consistency requirement of Galileo will always be satisfied. Likewise, if the user never attempts to terminate one of the components of Galileo while the other one is running, a prohibited behavior leading to an inconsistent state of the system never actualizes. This observation may suggest that given a sufficiently careful and informed user, positive behavioral mismatches do not create a problem. We believe that in most cases this "lazy solution" is unacceptable because it may significantly decrease the perceived value of the system to the user. To reconcile behavioral mismatches is a responsibility of the system designer.

3.4 Bridging behavioral mismatches

One way to reconcile a positive mismatch between the behavior of an implementation and a system-level requirement is to amend or remove the requirement. Although acceptable in some cases, this approach has a number of obvious disadvantages: (1) it may prevent system designers from presenting desirable system-wide abstractions to the user; (2) it may complicate component integration by precluding the use of a standard integration framework. In addition, as industrial case studies [21,22] indicate, changing requirements in a CBSD project often involves lengthy negotiations with the customer, may require changes in the implementation base, and negatively affects the overall development schedule.

The opposite approach to bridging positive mismatches is component restriction. It does not involve altering the requirements. Instead, restriction relies on changing the system implementation so that the implementation no longer can exhibit a behavior prohibited by the requirements. Because software components are usually distributed in an unmodifiable binary form, restriction rarely can be performed by changing component source code. In some cases, components will be designed to be restricted in the required ways; however, in practice, we have not encountered such components. (Anecdotally, Microsoft Word does have a *read-only* mode that can be set programmatically, but this mode marks the underlying *file* as being read-only as far as the system is concerned, and does not preclude editing of the displayed text.) A more practical approach to restriction relies on external component adaptors developed by system designers. Such adaptors play the role of a "careful user" that intercepts the input streams from the real users and filters them by discarding all requests that lead to an illegal system behavior. In the rest of the paper we describe this approach and its formal foundations in detail.

4. COMPONENTS AND REQUIREMENTS

In this section we present a simple formal model of component-based systems and system-level requirements that makes our notion of positive behavioral mismatch precise. In the next section we use this model as a basis for one general method of implementing restriction.

4.1 A model of software components

A software component is an executable program that can interact with its environment. For simplicity, in this paper we consider only sequential systems, i.e., those where only one thread of control can be active at a time. The environment of a typical component comprises the network and file input-output subsystems of the operating system (OS), the graphical user interface (GUI) system, and possibly other applications and components.

We call individual kinds of interactions between a component and its environment *actions*. We distinguish two types of component actions. *Input actions* are triggered by *messages* from the execution environment, i.e., transfers of control with, possibly, some data to the component. Examples of input actions are:

- getting a message from the GUI system with a notification that a key has been pressed or mouse moved;
- a return from an "open file" system call with a "file not found" error status code;
- receiving an event notification from another component;

Output actions are triggered by the component transferring control along with, possibly, some data to the environment. The following are examples of output actions:

- making an *OpenFile("readme.txt")* system call;
- returning control to the GUI system with a "message processed successfully" status code;
- notifying a registered event handler about an internal event, e.g., "new document created".

The set of actions of a component is finite.

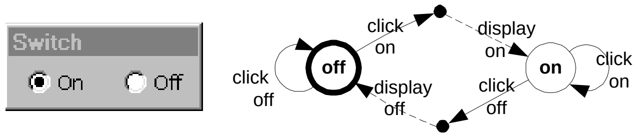

Figure 2 A model of a simple switch component

We model a software component as a state machine that changes its state as it performs input and output actions. Our model is similar to the I/O automata of Lynch and Tuttle [13] and collaboration specification of Yellin and Strom [24]. It is, however simpler and more constrained than those models. Unlike I/O automata, our model is strictly deterministic. Unlike collaboration specifications, it does not allow states in which both input and output actions may occur. Any state of a component is either an input state or an output state. In an input state a component is assumed to wait indefinitely for one of several possible input actions to occur. In an output state a component immediately executes an output action. No internal choice is allowed, namely, while several transitions may end in an output state, exactly one transition leaves it.

Figure 2 gives an example of a model of a hypothetical switch component. Its set of actions contains two input actions: *click-on* and *click-off* that the execution environment (in this case a GUI system) triggers by delivering messages to the component when the user clicks on the "On" and "Off" input fields respectively. The component also has two output actions: *display-on* and *display-off* that the switch performs in order to change its appearance on the screen. Our model of this component has two input states: **on** and **off**, and two nameless output states depicted as small black circles. **off** is the start state. Solid lines are transitions corresponding to input actions (input transitions). Dashed lines are output transitions.

The transition function of a component in our model need not be total. We assume that there is no transition that leaves an input state *s* and is labeled with an input action *a*, if it is known that while the component is in state *s* it cannot receive a message that triggers *a*. For example, given that the "Print" menu item is the only means to initiate printing in a text editor, the editor cannot start printing if the Print" menu item has been removed through a user interface customization procedure.

An *execution* of a component is a (possibly infinite) sequence of input and output actions that the component may perform at run time. It corresponds to a path in the multigraph of component state machine. For example, possible executions of the switch component defined above include *<click-on, display-on, click-off, display-off>* and *<click-off, click-on, display-on, click-on, click-on>*. A *prefix* is any initial subsequence of an execution. Note that a prefix of an execution is also an execution. The *behavior* of a component is the set of all its possible executions.

Our model can be trivially extended from individual components to entire component-based systems, i.e., integrated sets of components. A system of components $C_1,\ldots,C_n$ can be modeled as a "virtual" component C_b whose set of actions A_b is a disjoint union of the action sets of components $C_1,\ldots,C_n$. The state space of C_b is a subset of the cross-product of the state spaces of $C_1,\ldots,C_n$. Since our sequential execution assumption guarantees that at most one component can be active at a time, an execution of C_b can be modeled as a single sequence of actions from A_b. In the rest of the paper we apply the same model to both individual components and whole implementations containing several components.

4.2 A model of system-level requirements

System-level requirements can impose constraints on the collective behavior of components comprising a system implementation. Such requirements are often formulated in terms of abstract actions. As component selection and adaptation proceed these abstract actions are bound to concrete component actions. One abstract action can correspond to several concrete ones, but any given concrete action corresponds to only one abstract action. As a special case, system-level requirements for a known and fixed implementation base may be formulated directly in terms of component actions. The following example illustrates binding of abstract actions to concrete ones:

The semi-consistency requirement of Galileo can be formulated in terms of abstract output actions "textual view changed", which we denote *t.c*, "graphical view changed" (*g.c*) and "view contents have been successfully synchronized" (*s*). Given an implementation consisting of Word, Visio and a synchronization button augmented with view synchronization code, these abstract actions map to sets of concrete actions as shown in Table 1.

Abstract action	Concrete actions
change textual view (*t.c*)	any change in Word document, e.g., inserting a character
change graphical view (*g.c*)	any change in Visio diagram, e.g. deleting a line
successfully synchronize view contents (*s*)	successful return from the view synchronization routine that handles the "synchronization button pressed" events

Table 1. Mapping of abstract actions to concrete actions in the semi-consistency requirement of Galileo.

Collectively, the actions used in the definition of a system-level requirement form the *alphabet* of that requirement. The alphabet of the semi-consistency requirement is {*t.c, g.c, s*}. We model a system-level requirement as a language in the alphabet of that requirement. For example, the semi-consistency requirement corresponds to the regular language

$$(s+(t.c^* + g.c^*)\, s)^* \, (t.c^* + g.c^* + \varepsilon).$$

We model the process of binding the abstract actions of a system-level requirement to concrete actions of a particular component-based implementation as an application of *abstraction function.* An abstraction function is a partial mapping from the action set of a component or system of components onto the alphabet of a system-level requirement. Table 1 read from right to left defines an abstraction function for the semi-consistency requirement and the current implementation of Galileo. An abstraction function is generally partial and many-to-one. If a system-level requirement is defined in terms of component-level actions, then the mapping is one-to-one and is a partial identity function.

We say that an execution of a component-based implementation *satisfies* a system-level requirement, if replacing concrete actions in the execution with abstract actions of the requirement according to the abstraction function produces a string in the language of that requirement. Only component actions in the domain of the abstraction function need to be replaced. All other component actions are ignored and not included in the resulting string of abstract actions. Their presence and order in the execution are not constrained by the requirement. For example, the semi-consistency requirement imposes no constraints on the browsing actions, such as "move cursor up" or "scroll down". In fact, most operations that Word and Visio can perform are not affected by the semi-consistency requirement. If an execution satisfies a requirement *R*, we call it *legal* with respect to *R*. Otherwise an execution is *illegal*.

Finally, we can precisely define the notion of positive behavioral mismatch. We say that there is a positive behavioral mismatch between a component-based implementation *C* and a system-level requirement *R* if the behavior (the set of all possible executions) of *C* contains an execution that is illegal with respect to *R*.

4.3 An algorithm for detecting illegal executions.

Whether a particular execution *e* of component *C*, and all of its prefixes, satisfy a system-level requirement *R* can be detected by the following simple algorithm.

During its execution the algorithm incrementally builds a string α of abstract actions in the alphabet of *R*. Initially, the string is empty. Let *A* be the abstraction function from the actions of *C* to the alphabet of *R*. The algorithm consecutively reads in the actions from *e*. Let *x* be the next action read. If *x* is not in the domain of the abstraction function *A*, the algorithm goes on to reading the next action, or stops and declares success, if it reached the end of *e*. If *x* is in the domain of *A*, the algorithm appends *A*(*x*) to the end of the string of abstract actions α and checks whether the result is in the language of *R*. If it is, the algorithm continues to next action in *e*, or declares success if there are no more actions. Otherwise, it declares that an illegal prefix of the execution has been found and stops.

5. IMPLEMENTING RESTRICTION

In this section we present a general method of implementing component restriction. Our method is based on the model presented in section 4.

5.1 Internal and external restriction.

Component restriction (or simply restriction) is an operation on a component or a system of components that prevents it from exhibiting at run time a behavior that is prohibited by system-level requirements but otherwise permitted by the underlying components. There are essentially two ways to prevent a component from engaging in an undesirable behavior: internal and external, which can be used in combination. *Internal* restriction involves changing component implementation, or using facilities that component specifically provides to allow its clients to disable some of its features or behaviors statically, i.e., before runtime. For example, user-defined shapes in Visio drawings can be pinned to specified coordinates, made unselectable, and restricted in several other ways. However, the characteristics of CBSD and current state-of-the-practice in component design appear to make internal restriction infeasible in many cases.

First of all, because component source code is usually unavailable, the obvious approach to restricting component behavior — through changing component implementation — is generally infeasible. Even when source code is available, it may be difficult to understand and modify correctly, and the changes may not work for later versions of component. Second, most software components available on the market are not designed to support restriction. The problem of positive behavioral mismatch and the need for restriction do not appear to be widely recognized yet by software component vendors.

External restriction is based on the observation that the sequence of input and output actions that a component performs during execution is defined by two factors. The first factor is component implementation. The second is the sequence of messages (requests to perform an input action) that a component receives from its environment. Thus, an undesirable component behavior can be prevented by properly filtering the message stream that comes to a component from its environment.

External restriction requires no special support from the component or the availability of its source code. On the down side, it is implementable only in those execution environments that allow interception of messages directed to components. Fortunately, many popular component execution environments allow such interception. For example interception of user interface messages is possible in both

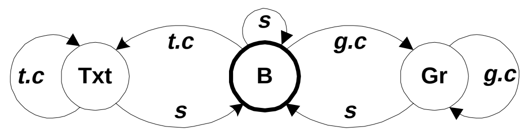

Figure 3. A requirement machine (requirement language recognizer) for the semi-consistency requirement. *t.c* and *g.c* are "text changed" and "graphics changed" actions respectively. *s* is the "views synchronized successfully" action. Only text can be changed in state *Txt*, only graphics in state *Gr*. *B* is the start state. Both graphics and text can be changed in state *B*. All states are accepting states.

the X-Window System and Microsoft's Win32, the *de facto* window system standards on the UNIX and PC platforms.

5.2 Implementing external restriction

To maximize the applicability of implementing restriction to real-world software components, our proposed method follows the external approach. The key questions that any external restriction implementation must address are:

1. When a filter intercepts a message from the execution environment to a component, how does it determine whether to drop the message or deliver it to component?
2. What messages must be intercepted?

To determine what message to drop, we use a version of the algorithm for detecting illegal executions, described in section 4.3. The use of this algorithm also determines our answer to the second question. Namely, two kinds of messages need to be intercepted by a restriction filter:

a) messages that may trigger input actions in the domain of the abstraction function of the requirement, and

b) messages that may trigger input actions that in turn immediately initiate output actions in the domain of the abstraction function.

All other messages need not be intercepted, as they do not change the outcome of test for legality of an execution. In more detail, our proposed method for implementing filters for external restriction has the following three parts:

a) A model of a system level requirement.

To bring the behavior of implementation in correspondence with a particular system-level requirement, system designers must first clearly define the set of behaviors allowed by that requirement. First, the designers must define the alphabet of actions of the requirement. Then they implement a recognizer for the language of the requirement, which we call a *requirement machine.* All system-level requirements that we have encountered — albeit simple ones — can be recognized by finite-state machines. For example, figure 3 contains the diagram of an FSM recognizing the semi-

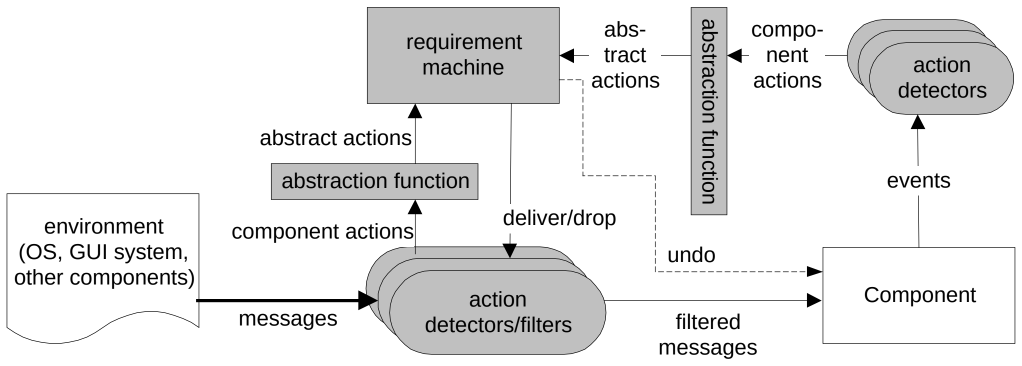

Figure 4. The structure of an external restriction filter.

consistency requirement. We anticipate requiring more computationally complex recognizers for more demanding restriction situations. However, we expect that the ideas developed here in a finite state machine context will extend to these more demanding situations.

b) The relevant message set and action detectors

The next step in implementing a restriction is to map the abstract actions of the requirement to the concrete actions of the component-based implementation, thus implicitly defining an abstraction function. Table 1 gives an example of such mapping. Then the designer must identify the set of *relevant messages*, i.e., those messages that initiate component actions in the domain of the abstraction function. For example, changes to the content of a document are caused by messages from the GUI system notifying the editor that the user pressed a key or chose the "cut selection" command from the menu. Only relevant messages need to be intercepted at run-time by the restriction filter. Identification of relevant messages may involve a certain degree of reverse engineering of components. In particular, the designer must discover casual relationships between the messages the component receives and the input and output actions it performs in response to these messages.

At run-time, what action a component is about to perform or has just performed is determined by the part of restriction filter called the *action detector*. One filter may contain several such detectors. There are two types of detectors. Detectors of the first type, called *message interceptors*, intercept raw system messages that the environment sends to the component, and analyze their attributes. By the attributes of a system message an action detector determines whether the message is a relevant one, and if it is, what action the component will perform if the message is delivered. Examples of message attributes that a detector may analyze include message type (keyboard, mouse, network), a key code, mouse pointer position, window or screen that the message is associated with, or a data packet returned from the network. A detector may also keep track of the history of previously observed messages and take it into account when deciding whether the current message is relevant and what component action it will trigger.

Action detectors of the second type listen to events generated by components themselves and determine from event parameters what action the component is about to perform or has just performed. These detectors are not as universal as message interceptors. They can only detect actions that components announce through events. However, they require no reverse engineering and can be the best option in situations where the same message may initiate several different component actions at different times, depending on the state of component.

Essentially, action detectors are reusable partial models of component operations built by reverse engineering. They are not specific to a particular restriction implementation and can be reused across projects.

c) An abstraction function

When an action detector recognizes a component action, it emits a code that uniquely identifies that action. In order to check whether the execution so far has satisfied the system-level requirement the filter must translate this code to the code of the corresponding abstract action that can then be submitted for checking to the requirement machine. This translation is accomplished by the third element of restriction filter: an implementation of the abstraction function. This implementation can be as simple as a look-up table, or it can use a more efficient algorithm.

When the abstraction function is a one-to-one correspondence, the translation of action codes can be skipped altogether by designing a requirement machine that accepts the outputs of action recognizers directly. This may reduce the response time overhead of the filter, but will also reduce the reusability of filter parts by creating a dependency between its requirement machine and action detectors.

5.3 Functioning of a restriction filter.

Put together, the three parts of a restriction filter provide a practical implementation of the illegal execution detection algorithm described in section 4.1. However, they do not only detect illegal executions, but also prevent them from happening. Figure 4 shows an interaction diagram that illustrates the run-time operation of such a filter (the internals of the filter are shown in gray).

When a restriction filter intercepts a message from the execution environment to a component of the implementation being restricted, it routes the message to an array of action detectors. If none of the detectors recognizes the message as relevant, the filter delivers the message to the component. If a detector recognizes the message as initiating an action, it generates a component action code. This code is translated by the abstraction function to the code of an abstract action in the alphabet of the requirement, which is then fed to the requirement machine. If the machine in its current state does not have a transition marked with this action code, then appending this abstract action to the string of abstract actions recognized so far would produce a string not in the language of the requirement. At this point the illegal execution detection algorithm described in section 4.1 would stop and announce the detection of an illegal execution. The goal of a restriction filter, however, is not to just detect such executions, but to avert them. Had the message intercepted by the filter been delivered to the component, an illegal execution would result. To prevent this, the filter drops the message. The requirement machine does not change its state. Similarly, when the transition marked with the abstract action code exists but leads to a non-accepting state, the intercepted message is dropped and the machine does not advance. Only if a transition marked with the abstract action is present in the current state of the requirement machine and leads to an accepting state does the machine perform the transition and the filter delivers the intercepted message to the component.

It appears that it is often possible to set up a restriction filter in such a way that most of the messages it intercepts are relevant messages. In other words, external restriction does not require complete wrapping of component. For example, if all relevant messages are user interface messages, the interface between the component and the network subsystem of the OS need not be affected.

Handling component events follows a similar scenario. An event is examined by one or more action detectors, and if one of them recognizes a component action, the code of that action is translated by the abstraction function to the corresponding abstract action code, which is then delivered to the requirement machine. As in the previous case, the requirement machine determines whether the component action detected leads to an illegal execution. However, an action detected by listening to component events usually cannot be averted, as events are typically notification devices only. The closest approximation to blocking an action in this case, is "undoing" the action immediately after it occurs. Undoing may not work for actions that break a system-wide invariant and may cause an immediate system to crash. However, it is often acceptable as a means of enforcing an abstraction that the system designers want to present to the user.

To undo an action, a filter may send a message to the component, initiating an action that cancels the effects of the last one and completely restores the component state. Unfortunately, in some cases an appropriate message may not exist, or it may be difficult to identify. The "undo" facility, which has become a standard part of any high-quality user interface may greatly help with this task. When an "Undo" operation is unavailable, the filter may still be able to determine what action could cancel the effects of the one just observed, e.g., by analyzing the attributes of event.

6. IMPLEMENTING SEMI-CONSISTENCY

We applied our method of implementing external restriction to restrict the current implementation of Galileo with respect to a version of the semi-consistency requirement. The execution environment for Galileo is the GUI subsystem of the Microsoft Windows 95 OS. Our restriction filter intercepts the keyboard, mouse and menu item selection messages that Windows sends to Word and Visio. Each of these types of messages is handled by a separate action detector. These detectors identify those messages that may trigger a change in the Word document or Visio drawing containing the textual or graphical representations of the fault-tree respectively. Another action detector is built into the integration infrastructure of Galileo. It observes the notifications that the integration code generates when the user selects a view synchronization command from the tool menu. Once an action detector identifies a component action, it uses an abstraction function to map the action code to the corresponding system-level action code and then submits it to the requirement machine similar to the one shown in Figure 3, which is implemented as a simple C++ object. Depending on the result returned by the machine, the filter drops the message or allows it to reach its target component. View synchronization actions cannot be prevented as only notifications of them are observed after the actions themselves have completed. However, there is never a need for this, since the semi-consistency requirement allows view synchronization in any state. Our action detectors and requirement machine are completely independent and can be reused in other projects that either have the same semi-consistency requirement or require detection of editing actions in Word and Visio.

Our work on implementing an external restriction filter for Galileo required considerable knowledge of the details of interface between components and the operating system. It also involved a considerable amount of reverse-engineering work to determine what messages components receive from the operating system and what actions these messages trigger. The only means to obtain this information was by using a general purpose OS message inspector and direct observation of changes in the visible component state.

Our filter appears to be sufficient to enforce the semi-consistency of views in Galileo for most usage scenarios. Because it relies on the knowledge obtained by reverse engineering of Word and Visio, which are complex software packages, there may still be some sources of positive mismatch through which a user of Galileo can force the tool to violate the semi-consistency of views. However, this would most likely require an intentional attempt to break the abstraction presented by the tool. The goal of the work reported in this paper was not the detection of all sources of positive behavioral mismatches in a component-based system, but rather developing a general mechanism for bridging identified mismatches. Possible approaches to testing or the formal verification of a particular restriction to ensure that it prevents all behaviors that do not satisfy a particular system-level requirement is a topic for future work.

7. RELATED WORK

The idea of disabling features and restricting behavior of programs has been explored mostly as a source of program optimization. Program specialization (partial evaluation) [10] is a well-known approach to optimization by transforming a program *P* into a more efficient program *P'* that is equivalent to *P* for a restricted subset of inputs. Specialization applies only to programs in a high-level language and uses a simple and straightforward model of programs as functions that map inputs to outputs. Neither of these assumptions holds in CBSD, where source code is not available and components are stateful and often highly interactive.

Work on open implementation [11,12] is also driven mostly by the issue of optimization: a black-box component (including source code components) can be more reusable if besides its "functionality" interfaces it exposes a meta-interface through which a user can tune the component's implementation parameters to improve component's efficiency in a particular usage context. The ideas of open implementation can be used in design of components that natively support restriction. The restriction method we described does not rely on any such support.

Parnas in his "Designing Software for Ease of Extension and Contraction" [14] discusses the problem of removal of services or features of programs in the context of program families developed within the same organization with full access to the source code. The question that Parnas addresses in his paper is how to design software so that its *designers* can easily modify its source code to obtain another program that has a subset of features of the original one and is as efficient as an equivalent program written from scratch. In contrast, we concentrate on the mechanisms that the *composers* of software components with no access to source code may use to obtain a subset of behaviors (essentially by temporarily disabling features).

Gentleman [6] acknowledged the importance of masking unwanted component functionality in CBSD. However, he did not describe a general approach to such masking.

8. CONCLUSION AND FUTURE WORK

Positive behavioral mismatch is an important problem arising in component-based software projects. Its primary sources are system-level abstractions that the implementation components are not designed to support, and constraints on component behavior imposed by the integration infrastructure. Restriction, or preventing component-based implementations from exhibiting illegal behaviors, is often the best approach to bridge positive mismatches. To make the problem of positive behavioral mismatch precise, we developed a formal model of software components and system-level requirements. Our model is not the only one possible. We used it to develop one general method of implementing restriction. Our method utilizes an external filter that detects and averts or cancels component actions that lead to a behavior prohibited by a system-level requirement. The method does not rely on the availability of source code, formal specification, or native support for restriction in components. We have used it to restrict an existing component-based application — a fault-tree analysis tool Galileo — with respect to the semi-consistency requirement. We see several directions for future work on component restriction:

Automatic generation of restriction filters

Our method for implementing external restriction provides a basis for automatic generation of restriction filters from high-level reusable declarative specifications containing a requirement machine description, an abstraction function and a set of action detector specifications. Special tools may assist with the reverse engineering necessary to develop such action detector specifications. Such tools may be used, for example, to test hypotheses about the casual connections between system messages and component actions by matching component input and output against a set of predicates given by the designer.

Restriction as a source of optimization.

Restriction can serve not only as a bridging technique for positive behavioral mismatches, but also as a source of component optimization. When a component specifically designed to supports restriction receives a request to disable a certain feature or behavior, it may specialize itself to perform the rest of its functions faster or with smaller memory requirements. How exactly a component should be designed to better support restriction for both optimization and bridging positive behavioral mismatches is an open research question.

REFERENCES

1. A. Williams, *Developing ActiveX Web Controls*, Coriolis Group, 1996
2. Bershad, B.N., Savage, S., Pardyak., P, Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., Extensibility, Safety and Performance in the SPIN Op-

erating System, in *Proc. 15th Symposium on Operating Systems Principles*, Copper Mountain, Colorado, Dec. 1995, pp. 267–284

3. Brockschmidt, K., “How OLE and COM Solve the Problems of Component Software Design”, *Microsoft Systems Journal*, v.11, no.5, pp. 63-82, May 1996
4. Brockschmidt, K., *Inside OLE*, Microsoft Press, 1995
5. Dean, J.C., Vigder, M.R., “System Implementation Using Commercial Off-The-Shelf (COTS) Software”, *NRC Report No. 40173*, 1997
6. Gentleman, W.M., Effective Use of COTS (Commercial-Off-the-Shelf) Software Components in Long Lived Systems (tutorial summary), in *Proc. ICSE’97*, Boston, MA, Springer, pp. 635 – 636.
7. Graham, S.L., Lucco, S., Wahbe, R., Adaptable Binary Programs, *in Proc. 1995 USENIX Technical Conference*, New Orleans, Louisiana, January 1995, pp. 315-325
8. Hart-Davis, G., *Word 97 Macro & VBA Handbook*, Sybex, 1997
9. JavaSoft, *The JavaBeans 1.01 API Specification*, July 24, 1997, available on the WWW at http://www.javasoft.com/beans/spec.html
10. Jones, N.D., An Introduction to Partial Evaluation, *ACM Computing Surveys*, Vol. 28, No. 3, (September 1996), pp. 480 – 504.
11. Kiczales, G., Beyond the Black Box: Open Implementations, *IEEE Software*, Vol. 13, No. 1, January 1996, pp. 8 –11
12. Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G., Open Implementation Design Guidelines, in *Proc. ICSE’97*, Boston, MA, Springer, pp. 481 – 490.
13. Lynch, N.A., Tuttle, M.R., An Introduction to Input/Output Automata, *MIT Technical Memo MIT-LCS-TM373*, 1988
14. Parnas, D.L., Designing Software for Ease of Extension and Contraction, *IEEE Transactions on Software Engineering*, Vol. 5, No. 2, (March 1979), pp. 128 – 138.
15. Rader, J.A., Mechanisms for Integration and Enhancement of Software Components, in *Proc. 5th Int. Symp. On Assessment of Software Tools and Technologies*, Pittsburgh, PA, 2-5 June 1997, pp. 24–31
16. Sullivan, K.J. and D. Notkin, “Reconciling Environment Integration and Software Evolution,” ACM *Transactions on Software Engineering and Methodology* vol. 1, no. 3, July 1992, pp. 229–269
17. Sullivan, K.J., Socha, J., Marchukov, M., "Using Formal Methods to Reason About Architectural Standards," *Proc. ICSE’97*, Boston, MA, May 1997, pp. 503-513
18. Sullivan, K.J., Knight, J.C., Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse, in *Proc. ICSE-18*, Berlin, Germany, March 1996, pp. 220 – 228
19. Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998
20. Taivalsaari, A, On the Notion of Inheritance, *ACM Computing Surveys*, Vol. 28, No. 3, (September 1996), pp. 438–479
21. Tran, V., Hummel, B., Liu, D., Le, T.A., Doan, J. Understanding and Managing the Relationship Between Requirement Changes and Product Constraints in Component-based Software Projects, in *Proc 31st Annual Hawaii International Conference on System Sciences*, Jan. 6-9, 1998, Kona, Hawaii, pp. 132–142
22. Tran, V., Liu, D., Hummel, B., Component-based Systems Development: Challenges and Lessons Learned, in *Proc. 18th International Workshop on Software Technology and Engineering Practice*, London, England, 1997, pp. 452–462
23. Visio Corporation, *Developing Visio Solutions*, 1997. Also on the WWW at http://www.visio.com
24. Yellin, D., Strom. R., Protocol Specifications and Component Adaptors, *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 2 (February 1997), pp. 292 – 333