Component-Based Synthesis of Dependable Embedded Software*

Arshad Jhumka, Martin Hiller, Neeraj Suri
{arshad,hiller,suri}@ce.chalmers.se

Dept of Computer Engineering, Chalmers Univ., Sweden

Abstract. Standardized and reusable software (SW) objects (or SW components – in-house or pre-fabricated) are increasingly being used to reduce the cost of software (SW) development. Given that the basic components may not have been developed with dependability as primary driver, these components need to be adapted to deal with errors from their environment. To achieve this, error containment wrappers are added to increase the reliability of the components. In this paper, we first present a modular specification approach using fault intolerant components, based on the concepts of category theory. We further introduce the concept of wrapper consistency, based upon which, we present an algorithm that systematically generates *globally consistent* fault containment wrappers for each component, to make them fault tolerant. Subsequently, we enhance the initial modular specification to deal with the wrapped components, and show that safety properties of the system are preserved under composition only if the wrappers are globally consistent.

1 Introduction and Problem Perspectives

The functionality and dependability of computer systems is increasingly being defined by software (SW). However, to reduce the high cost associated with the development of SW, *components*, for example pre-fabricated or from component repositories, are being used while having to satisfy overall system/SW dependability requirements. Given that these components may not have been developed with dependability as a primary driver, i.e., the components may be fault-intolerant, i.e., they do not tolerate faults that will violate the safety specification, the components may need to be adapted or transformed to be able to contain errors coming from their environment, i.e., made fault tolerant. This can be achieved by wrapping the different components with error containment wrappers [16, 8]. Error containment wrappers can be classified into two broad categories, namely detectors and correctors [2]. Intuitively, detectors are used to detect errors, whereas correctors are detectors that correct errors whenever the system state is corrupted.

However, as pointed out in [14], the design of detectors, such as assertion checks[1], for error detection in software is a heuristic process, often with low levels

* Supported in part by Saab endowment, TFR Grants

[1] We will use the term detector, assertion or wrapper interchangeably

of efficiency to error coverage. Some conclusions mentioned were: (i) placement of detectors is crucial for them to be effective, (ii) given the randomness of the design process, some detectors detected non-existent errors in the system (so called "false positives"), leading to decreased efficiency of the system, i.e., the state of the system is such that it does not violate the safety specification [1] defined, but is however flagged as erroneous, (iii) specification-based detectors tend to have limited effectiveness, and (iv) specification-based detectors, together with code-based detectors, have a higher effectiveness in detecting errors. In other words, the state of the system is such that it does not violate the safety specification [1] defined, but is flagged as erroneous by the detector wrappers.

For problem (i) cited above, we presented an approach in [9] for locating the relevant detectors (and correctors). For problem (ii) above, in [12], we introduced the concept of *global assertion consistency* to verify global conformance across assertions. We defined global consistency as follows: Given a program P comprising a set of detectors $D = \{d_i\}$ and a safety specification S that P should satisfy, the set D is said to be *globally consistent* if $(P(D) \Longrightarrow S)$. We also showed that $(P(D) \Longrightarrow S)$ can be verified using the concept of *abstract interpretation* [3]. In this paper, we will show that the detection of global consistency of assertions is NP complete. Thus, for problems (ii)–(iv) above, in this paper, we develop an algorithm, based on global assertion consistency, that automatically generates consistent assertions, which can then be used to wrap the different components. Overall, the global consistency property ensures that detectors detect only those errors that will lead to safety violation of the program.

Paper Objectives Further, once components are wrapped with detectors (and correctors), one needs to ascertain that the wrapped components, i.e., fault tolerant components, can still be composed together into a system. We show that fault tolerant wrappers used to wrap one component do not interfere with the functionality and dependability of other components. In this paper, we show that composability of fault tolerant components with globally consistent detectors is indeed preserved, showing the viability of our algorithm. Overall, our contributions in this paper are: (i) We first present a modular specification approach of a system, using concepts from category theory, (ii) We present the concept of global assertion consistency, and show that its detection is NP complete, (iii) Thus, we provide an algorithm that generates globally consistent detectors, and (iv) fault intolerant components are transformed into fault tolerant components by wrapping them with the globally consistent fault containment wrappers, and (v) We enhance our initial specification with wrapper information, and show that the fault tolerant components are composable if the wrappers are globally consistent. We place our contribution more in context towards the end of the paper when we discuss related work in Section 6. Here, we do not address temporal aspects, which is part of our ongoing work.

The paper is structured as follows: Section 2 present the system and fault models adopted in this paper. Section 3 presents an approach to specify and verify a fault intolerant specification, based on the concepts of category theory. In Section 4, we first show that detection of global consistency of detector is

intractable, and then present a heuristic to tackle the problem of generating globally consistent detector wrappers. Section 5 extends our initial intolerant specification with detector wrapper constraints, and we identify properties that preserve composability of components. In order to develop a proper context for comparison, we present an overview of related work in Section 6, and we present a discussion of the approach and its applicability in Section 7. We conclude with a summary of the paper in Section 8.

2 System & Fault Models

We make the following system and fault model assumptions:

System model: We assume that software (SW) is made up of different components, communicating with each other through some form of communication, such as message-passing, shared memory etc. We will use the abstract term signal to denote these communication paradigms. We also assume gray-box knowledge of SW, i.e., the internals of the software are known, but are, non-modifiable. Thus, any transformation performed is achieved through the use of wrappers. The choice for reasoning at the gray-box level is three-fold: (i) Maintainability of the original SW is made easier since it is not modified, and "modification" is only through error containment wrappers, (ii) Reuse of fault intolerant SW is easier, and the relevant error containment wrappers can be generated, and (iii) reasoning at a gray-box level allows for modular SW construction, with usage flexibility for single processors or distributed systems.

Fault model: We assume transient data errors occurring at the communication signal level. To detect these errors in embedded systems, where the emphasis is on signal data values, assertions are often incorporated. These specify, for example, range checks, or bounds on rate of change of data values.

Next, we will present a modular specification approach for specifying a component based system, based on the concepts of category theory [6, 19].

3 Modular Specification of Embedded Systems

Our modular specification framework is based on the concept of category theory [6]. Category theory allows definition of a calculus of modules, and their respective composition, i.e., module composition. It also allows module specifications and constraints to be reasoned about in the same framework. Specification languages such as Z[2] [20] are not suitable since Z does not really allow hidden interfaces etc. Its object-oriented version, Object-Z [17], could have been used but it does not offer a framework to reason about constraints in natural way, i.e., whether constraints imposed are indeed correct etc. Similarly, formal frameworks such as CSP[3] [10] are restrictive for reasoning about transforming a component into a fault tolerant one.

[2] We refer to Z since they represent a class of state-based specification language.
[3] We refer to CSP since they represent a class of event-based specification language

Components are algebraically specified, and they are interconnected to encapsulate their interactions. The composition operation then defines and constructs an aggregated component describing the overall system from the individual components and their interactions. We will first present the specification of basic building blocks of a component, such as *export interface*, and show how they are composed together into a component specification. Components are then composed into a system.

3.1 Specification of Basic Building Blocks

A specification consists of two parts (a) a *signature* part, and (b) an *axiom* part. The signature introduces syntactical elements that are used in the axiom part, and consists of three parts: (a1) the *Sorts* part declares the domains, (a2) the *Constants* (respectively *Variables*) part declares the time independent (respectively time dependent) functions and/or predicates, and (a3) the *Action* part declares predicates and functions representing event instances. The axiom part defines the behavior of the building block, or specification.

In practice, a component is a SW module that imports some services from its environment, and provides some services used by its environment. Formally, a component can be composed of different specifications, such as *import interface*, *export interface* etc. Each building block is specified algebraically, i.e., each block specification consists of a signature and axiom part. To obtain the overall component, these specifications are combined via specification morphisms.

Specification Morphism: A specification morphism $m : A \rightarrow B$ from a specification A to specification B maps any element of the signature of A to an element of the signature of B that is compatible.

3.2 Component Specification from Basic Specifications

Syntax of Component Specifications An algebraic specification of a component C consists of four basic basic building blocks, namely (i) *parameter* (PAR), (ii) *export* (EXP), (iii) *import* (IMP), and (iv) *body* (BOD). Each building block (specification) is individually specified, and are interconnected through specification morphism, to obtain a component specification. Thus, a component specification is a tuple, *COMP = (PAR,EXP,IMP,BOD,e,s,i,v)*, consisting of 4 specifications and four specification morphisms, e, s, i and v, as shown in Fig. 1.

The BOD part of the component specification is constructed using the *pushout* operation. Specifically, the pushout of two specification morphisms $e : PAR \rightarrow EXP$ and $i : PAR \rightarrow IMP$ is an object BOD together with morphisms $v : EXP \rightarrow BOD$ and $s : IMP \rightarrow BOD$ satisfying $v \circ e = s \circ i$ and the following general property: for all objects BOD' and morphisms $s' : IMP \rightarrow BOD'$ and $v' : EXP \rightarrow BOD'$ with $v' \circ e = s' \circ i$, there is a unique morphism $b : BOD \rightarrow BOD'$ such that $v \circ b = v'$ and $s \circ b = s'$. Here, $\circ$ denotes morphism composition.

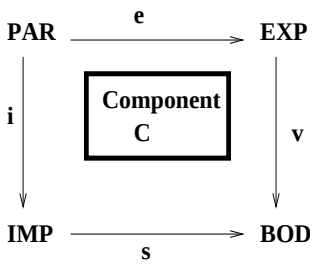

Fig. 1. Component specification with building blocks specification and specification morphisms

We briefly explain the role of each specification: (i) The *BODY*(BOD) part is the complete description of the component, i.e., it explains how resources provided by the import part are used to provide the services of the export interface, i.e., it transforms the services obtained from the environment into services provided to the environment. Given our gray-box approach, the BOD part cannot be modified to transform the component into a fault tolerant one.

The *IMPORT*(IMP), *EXPORT*(EXP), and the *PARAMETER*(PAR) parts are the interfaces of the component. The PAR part contains the parameters of the component, while the IMP (EXP) part defines the services needed (provided) by the component.

The four specification morphisms e, i, s, v of the components describe the links between the four specifications. Also, all items in PAR must have an image in IMP and EXP to ensure correct definition of the morphisms e and i. For example, IMP cannot introduce a type (sort) without the type being introduced in the PAR part of the specification. One important aspect of specification morphisms is that they preserve axioms. For example, one must prove that the axioms defined in the export interface can be deduced from those defined in the body specification. Thus, verification of a given component can be performed.

Semantics of Component Specification For a given component specification *COMP = (PAR,EXP,IMP,BOD,e,s,i,v)*, the specification morphisms are interpreted as *functors* (in the reverse direction), which are functions from one category to another, i.e., they map objects onto objects and morphisms onto morphisms. More specifically, they are called *forgetful* functors since they forget those resources that are not in the image of the specification morphism, i.e., for a given specification morphism $m : SPEC \rightarrow SPEC_1$, the forgetful functor corresponding to m is given by $V_m : Cat(SPEC_1) \rightarrow Cat(SPEC)$. In pratice, this means that hidden operations and hidden data domains are forgotten, i.e., not part of the export.

On the other hand, the body (BOD) specification is interpreted according to the *functorial semantics*, since it represents the construction of the export interface from the import interface. The functorial semantics is the composition of a free functor (from import algebra to body algebra) and a forgetful functor (from

body algebra to export algebra). This means that, with the functorial semantics, we allow free construction of the services to be exported with hidden data domains and hidden operations omitted from the export interface. We also have the *restriction semantics*, which is the composition of the unrestricted semantics with a restriction construction of the export with respect to the parameter part. The other specifications, i.e., PAR, IMP and EXP, have a loose interpretation, in the sense that any algebra that satisfies the specification is admissible.

Formally, from the component specification, we define a *construction semantics* of *COMP*, *FREE* as follows: $FREE : Cat(IMP) \rightarrow Cat(BOD)$ with respect to the forgetful functor $V_s : Cat(BOD) \rightarrow Cat(IMP)$. Thus, the functorial semantics of *COMP* is the functor $FUNC : Cat(IMP) \rightarrow Cat(EXP)$ which constructs for each import algebra A a corresponding export algebra B such that $B = FUNC(A)$. This construction is mainly a free construction $FREE(A)$ defined by sorts, actions and axioms in the BOD part of the component specification. The functorial semantics is a composition of the free functor and the forgetful functor with respect to the v morphism, i.e., $FUNC = V_v \circ FREE$, Fig. 2.

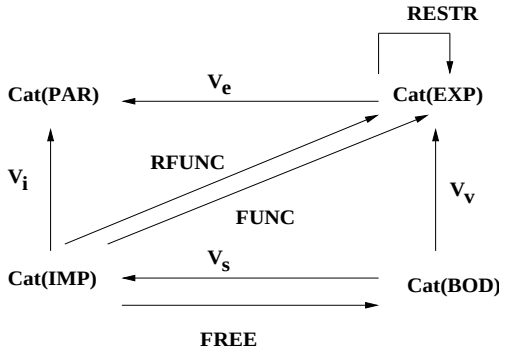

Fig. 2. Semantics of morphisms

Thus, the meaning of a component specification is thus given as a functor which maps import algebras I to export algebras *FUNC(I)*.

Definition of component correctness:
Let *COMP = (PAR,EXP,IMP,BOD,e,s,i,v)* be a component specification. *COMP* is said to be correct if the free functor *FREE* is strongly persistent, i.e., $ID = V_s \circ FREE$, where ID is the identity functor on *Cat(IMP)* and *FREE* preserves injectivity of homomorphisms. This just ensures that each import algebra I is protected by the free construction, i.e., $V_s \circ FREE(A) = A$.

3.3 System Specification from Components

Having specified and verified a given component, these components can now be composed together through component morphisms to obtain the complete system specification. Given two components C_i (a component importing services) and C_e (a component exporting the services required) such that all elements imported by C_i are exported (defined) by C_e, the composition operation builds a component C_{ie}. A module morphism is a pair (h, hp), see Fig. 3, of morphisms such that h associates services imported by C_i to corresponding services exported by C_e. Morphism hp maps parameters of C_i to those of C_e. The component C_{ie} can then be computed from components C_i and C_e, i.e., it imports what component C_e imports, exports what component C_i exports, and can be computed from the pushout operation. If components C_i and C_e and morphisms h and hp are correctly defined, then component C_{ie} is correct. Also, during composition, the following should be verified: $h \circ i_i = e_e \circ hp$, where i_i is morphism i in component C_i, and e_e is morphism e in component C_e. This ensures that the associations made by hp is compatible with those defined in h.

System verification is performed as follows: any axiom in a given component C_i is translated along a given component morphism as theorem in the C_e component, i.e., one needs to ascertain that the behavior of C_i is preserved.

Also, correctness of the resulting composition C_{ie} can be derived from the correctness of C_i and C_e (using the notion of strong persistency).

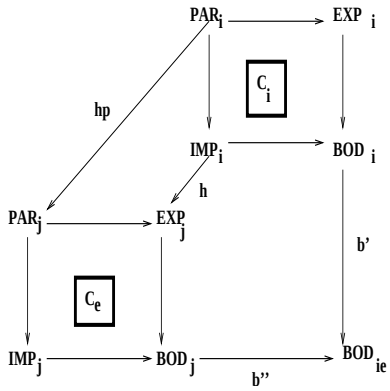

Fig. 3. Composing two components

Different components needed to construct the overall system are composed together, as has been shown. Given such a system specification *SPEC*, a corresponding implementation P satisfying the *SPEC* can be obtained through techniques such as refinement, which then obviates the need for verification. Given the free construction allowed, any body implementation that transforms the import services into the export services is allowed. Also, given the transitivity property of refinement and the fact that P refines *SPEC*, our framework is still

valid if we reason at a lower level of abstraction, i.e., at the level of P. Thus, depending on the level of implementation detail incorporated in the specification, we can reason at different levels of abstraction within the same framework.

4 Addition of Error Containment Wrappers

The system specification provided is for a fault intolerant system, i.e., though the behavior of the system has been verified, the behavior may be violated in faulty scenarios. Hence, there is a need to add fault tolerance components to the original system, i.e., adding error containment wrappers around components.

Adding error containment wrappers around components is inherently difficult, as indicated in [14] and Section 1. We previously [12] showed that the consistency property of assertions help in designing effective detectors. For completeness, we will present a brief overview of the consistency property, as well as the proof of intractability of its detection.

4.1 Consistency Property of Assertions

An assertion EA_i in a program P defines a set of values, $S(EA_i)$, that a given variable V_i can take. Two EAs placed in a given SW system are linked to each other via code implementing a given function F_m (in subsequent discussion, we will use P to denote the relevant function F_m).

Definition 1. EA_1 and EA_2 are said to be consistent with each other *iff* $F_m(S(EA_1)) = S(EA_2)$. They are inconsistent *iff* $F_m(S(EA_1)) \cap S(EA_2) = \{\}$. They are partially consistent *iff* they are neither consistent nor inconsistent.

Thus, a predicate *cons*(A_i, A_j, P) is true iff A_i and A_j are consistent through P(through the relevant F_m). A set of assertions $S_A = \{A_i\}$ defined in a program P is said to be globally consistent iff, $\forall i, j : cons(A_i, A_j, P)$.

Intuitively, the global consistency property states that any valid data value should not be flagged as erroneous by any of the assertions in the system. The converse is also true. Verification of consistency property can be achieved using techniques such as abstract interpretation. Given that safety specification is defined, we want to detect those states that can potentially lead to violation of the safety specification, i.e., the set of detectors, including the safety specification, should be globally consistent. The safety specification is considered as a global property defined on the system to be satisfied. However, detection of the consistency among a set of detectors is intractable. Thus, we present a heuristic, analogous to predicate transformers [4], that generates consistent detectors.

4.2 NP Completeness of Global Consistency Detection

Before presenting the heuristic, we will briefly present the proof of NP completeness of global consistency detection.

A program P consists of a set of components, $P = \{C_1 \ldots C_N\}$. Each component C_i has a set V_i of variables defined in it. A global safety property of a

system is a boolean valued function B defined over the variables in $V = \bigcup_i V_i$ of the system. We define a set $L = \{L_1 \dots L_N\}$ of detectors placed in individual components, defined over the set V of variables. We use the notation $B(L)$ to indicate the value of predicate B in a system with $L = \{L_1 \dots L_N\}$.

Global consistency detection of wrappers (GLOB) is a decision problem. It takes the form of:
Given: Program P, global safety property B, a set of system variables V.
Determine: if there exists a set L of wrappers L_i's defined over V such that $L \cup B$ is globally consistent through P.
Claim: *GLOB is NP-complete*
Proof: First note that the problem is in NP. A verifier for the problem takes as inputs the program P, the set L of wrappers and the global property B and then verifies if the set $L \cup B$ is globally consistent. This can be done in polynomial time. Then detecting the global consistency of wrappers belongs to the set NP.

We show NP-completeness of a simplified consistency detection where all variables in V can take value "true" or "false". Assume there is only one wrapper L_i incorporated in C_i. We reduce the satisfiability problem of a boolean expression to GLOB by constructing an appropriate set of wrappers.

The set is constructed as follows: Choose an $L_i \in L$ such that $L_i =$ "false" and include a new variable $v_j \in V$ such that L_i is now "true". It is easily verified that the predicate B is true for some set L *if and only if* the global safety property is satisfiable.

4.3 Heuristic to Generate Globally Consistent Error Containment Wrappers

Given the intractability of global consistency detection, we present a heuristic that generates globally consistent wrappers. It works in a way analogous to predicate transformers. Given our access to the code (which cannot be modified), the heuristic perform a backtracking process.

We first present the main steps of the algorithm in more details, and then the implementation details:

S 1,2 The algorithm starts at the end of the sink module and backtracks to its start. Then, the process is iterated for every component.

S 3 All conditions for computing the different values of variables are calculated. Program slicing [18] techniques can be utilized for this.

S 4 Once all conditions for computing the different variables are obtained, they are substituted in the condition for computing the output signal. For example, suppose we have an output signal X and the condition for computing its value is $X := Y + 1$. Suppose that the condition for computing Y is $Y := Z + 2$, where Z is the input. Then, the new condition for X is $X := Z + 2 + 1 = X := Z + 3$.

S 5 This condition for computing the output signal is substituted in the postcondition defined on the output signal (say X), and by simplifying the constraint obtained, we can obtain a constraint defined on the input signal (say Z). This constraint will be the input detector wrapper in the given component (C_i).

S 6 This precondition is translated into the postcondition of the preceding component (C_e). Formally, we translate this precondition along morphism h to obtain the postcondition on the preceding component.

S 7+ Steps 1–6 are iterated until the source modules are reached, and constraints on system inputs obtained.

```
Derive_EA(<global_property_EA>, <module_name>,
          <module_interconnections>)
%global_property_EA is the safety specification defined
%on the system. In any round of execution, the safety spec
%is satisfied, i.e., is an invariant. Thus, the global_property_EA
%is considered as a postcondition on the output signal of the
%system, i.e., at the end of a round of execution.
1  while (NOT beginning of <module_name>)
   {
2    for (all variables V in <module_name>)
3      conditions(V) :=
       determine_from_module(<module_name>);
%Conditions for computing variable V
%determined from module
   }
4    conditions(<OutputSignal>):=
       get_all_conditions(<module_name>);
%All conditions for computing output signal

5    new_preconditions :=
output_EA[conditions(OutputSignal)/OutputSignal];
%new preconditions obtained by substituting
%output signal by the condition obtained. This is the input
%detector wrapper for the importing component.

6    postcond_preceding_module
                    := new_preconditions;
%preconditions translated as postconditions
%in preceding module. This is the output detector wrapper
%for the exporting component.
7    preceding_module :=
get_id_preceding_module(module_interconnections,
                        module_name);
%gets the id of the other module with which
%module_name is communicating

8    if (preceding_module == NIL) break;
%source module reached
9    Derive_EA(postcond_preceding_module,
               preceding_module,
               module_interconnections);
```

Overall, pre- and postconditions (input and output wrappers) are generated for each component upon backtracking. When a component C_i imports services

exported by another component C_e, the precondition defined in C_i is transformed into a postcondition in C_e. Formally, the input wrapper in C_i is translated along morphism h (see before) into an output wrapper in C_e.

One potential limitation of the above algorithm is the handling of loop constructs among components, or within a component. There are techniques that can be used, such as *instrumented semantics* [7]. Techniques used for determining weakest preconditions [4] can also be used here. In the worst case, EA's which are not fully consistent may be obtained. Determining the applicability of partially consistent EA's is an avenue for future work.

4.4 Proof of Correctness of the Algorithm

Having presented the algorithm, we now present an informal proof of correctness, i.e., that the algorithm will return EAs that are consistent with each other. We make use of three lemmas for the proof.

As inputs to the algorithm, we have the global EA monitoring the output signals. We also know the input signals to the modules. States of programs can only be changed through assignment statements, however, the changes depend on the state of the program at that point, i.e., there can be multiple data paths linking the input signals and output signals, and the data path taken is determined by the state at that time. In the proof, we denote the output signal as $OutputSignal$ and F_M denotes the function implemented by a given module M.

Lemma 1 *Along any data path taken, there exists at least one assignment of the form* $OutputSignal := F(...)$ *for the output signal to have updated data.*

Proof of Lemma 1: If there is no assignment where $OutputSignal$ is the target destination, then it will only contain its initialization data value or a constant value. Hence, for it to have updated data, the assignment $F(...)$ should be a function of some other variables, i.e., $F()$ should hold the conditions (variables) used to compute the $OutputSignal$.

Lemma 2 *The values held by the variables determining the value of* OutputSignal *should either be input signal data values or result from applying a function on the input signals.*

Proof of Lemma 2 If the variables which determine the data value of $OutputSignal$ does not hold input signal data values (or values that depend on input signal data values), it implies that $OutputSignal$ does not depend on the input signals. This is inconsistent with having input signals going into that module.

Using the two lemmas, we can deduce that $OutputSignal$ can be expressed as a function (F_M) of the input signals. Hence, in the EA monitoring $OutputSignal$, we can substitute $OutputSignal$ by the function executing on the input signals. Thus, for a global EA of the form $(a < OutputSignal < b)$ we replace $OutputSignal$ by $F_M(inputsignals)$, resulting in $(a < F_M(inputsignals) < b)$. This expression can be simplified as appropriate. Thus, preconditions monitoring the input signals are obtained. From Lemma 1 and 2, we show that the algorithm does generate preconditions from output signals specification.

Lemma 3 *The preconditions are consistent with the global EA.*

Proof of Lemma 3 From the above, the precondition is as follows: $(a < F_M(inputsignals) < b)$. Executing F_M on the input signals will result in $OutputSignal$, hence $(a < OutputSignal < b)$, which is the global EA. Thus, the precondition and postcondition will be consistent.

Lemmas 1, 2 and 3 constitute the overall proof of correctness of the algorithm.

5 Specification of Fault Tolerant Embedded Systems

Initially, we have provided a modular specification of a fault intolerant embedded system, and explained how verification can be performed. To add fault tolerant components to the system, error containment wrappers are added to the components, given our focus on gray-box components. Since detection of globally consistent wrappers is intractable, we have proposed a heuristic that can generate globally consistent wrappers, starting from a given safety specification, i.e., the heuristic generate wrappers that are consistent with the safety specification. In subsequent sections, we show (i) how to enhance our specification with the wrapper information, i.e., constraints provided by the wrappers, and (ii) that the consistency condition on wrappers allows the fault tolerant components to be composed together. Wrappers can be added both at the input and output of a component for fault tolerance.

5.1 Addition of Detector Wrappers to Basic Specifications

Once input and output wrappers are obtained, it implies that there are constraints imposed on the components. Input wrappers constrain the values allowed by the imported functions, while output wrappers constrain the data values outputted. These constraints formulate requirements for interface data types, which may not be expressed in equational logic (as in the axiom part of the specification). Given our focus on gray-box components, we allow these constraints to be defined only at the interface level, not in the body specification. This allows for the reuse of the free construction of the equational case for corresponding specifications with constraints, except that we restrict the free constructions to those interface algebras that satisfy the given constraints.

Given (input and output) wrappers W of constraints over a component C, which is a free construction, we define *SPECW=(SPEC,W)*, as a specification with constraints. A $SPECW$-algebra is a usual $SPEC$-algebra that satisfies all constraints defined in W. Given the constraints, we include one more part to a specification, namely a *constraints* part, in addition to the *variables, axioms* and *sorts* parts already defined in the specification. We enhance each basic specifications with the constraints part that define the imposed data requirements, i.e., we wrap both import and export interfaces with relevant wrappers.

5.2 Building Fault Tolerant Components from Fault Tolerant Building Blocks

Up to now, we have generated wrappers (constraints) on a given specification. However, given that input constraints in an importing component $C_i$are translated as output constraints for an exporting component C_e, and output constraints into input constraints in a given component, this translates into the ability to reason about translation of constraints along specification morphism.

Semantically, we make use of a *Constraints* functor, $Constraints : CatSpec \rightarrow Sets$, that associates a set of constraints with each specification, where $CatSpec$ is the category of specifications. This functor also maps every morphism $m : SPEC1 \rightarrow SPEC2$ onto a function $Constraints(m) : Constraints(SPEC1) \rightarrow Constraints(SPEC2)$ which assigns to every constraint $C \in Constraints(SPEC1)$ on *SPEC1* the translated constraint $Constraint(m)(C)$, denoted $m'(C)$, defined on $SPEC2$. Intuitively, this means that any constraint defined over the import interface is translated into a given constraint over the export interface of a given component. In other words, input constraints defined on a component are transformed into output constraints for the same component.

Given wrapped specifications $SPECW_1 = (SPEC_1, W_1)$, and $SPECW_2 = (SPEC_2, W_2)$, a specification morphism m from $SPECW_1$ to $SPECW_2$, denoted $m : SPECW_1 \rightarrow SPECW_2$, is called *consistent* if W_2 implies the translated constraints $m'(W_1)$, i.e., $W_2 \Longrightarrow m'(W_1)$. Intuitively, it means that the predicate *cons(*W_1, W_2, BOD*)* (defined in Sec. 4.1) evaluates to true under the consistency condition of morphisms, and where BOD represents the component implementation. Note that for consistency condition, we do not require $W_2 \cup A_2 \Longrightarrow m'(W_1)$, where A_2 is the set of axioms for $SPEC_2$, because since we define W_2 on the specification, rather than on signatures, any algebra satisfying W_2 is already a $SPEC_2$ algebra satisfying A_2.

At this point, we need to show that the wrappers generated by our heuristic satisfy the consistent specification morphism condition.

Lemma 4: The specification morphisms of a given component are consistent.

We want to prove that the specifications, enhanced with the constraints imposed by the wrappers, yield consistent specification morphisms. To prove the above, we need to prove that *COMPW = (PARC,EXPC,IMPC,BOD,e,s,i,v)* is correct. A given component with wrappers, *COMPW*, is *correct* if it satisfies:

1. *COMPW* is constraints preserving, i.e., for all IMP-algebras I with $I \models CI$, we have $FUNC(I) \models CE$
2. *COMPW* is strongly persistent with respect to the import constraints *CI*, i.e., $FREE_s$ is strongly persistent on all IMP-algebras I where $I \models CI$
3. $e : (PAR, CP) \rightarrow (EXP, CE)$ and $i : (PAR, CP) \rightarrow (IMP, CI)$ are consistent.

The first part amounts to proving that for every $I \models CI$, then $(I \models CI) \Longrightarrow (V_f(I) \models CE)$, where $f : EXPC \rightarrow IMPC$. Given our definition of global consistency, input constraints are translated into output constraints. Thus, the component is constraint preserving.

For the second part, it is easy to see that *COMPW* is indeed strongly persistent, i.e., the free construction, starting from an import algebra $I \models CI$, protects the algebra. Since we adopt a gray-box approach, we reuse the original free construction. So, if the original construction was strongly persistent, so is the construction with constraints.

For the third part, consistency of morphisms e and i means only that the constraints part of PAR, *CP*, are reflected in the constraints part of the export and import interface, *CE* and *CI*.

Theorem: If *COMPW* is correct, then all specification morphisms *e, s, i, v* are consistent.

Proof: Consistency of morphisms *e, i* is proven by part (3) of the conditions for component correctness.

Consistency of morphism v is proven by part (1) of the correctness condition. Consistency of morphism s is based upon how induced constraints on the body part, BOD, of the component are obtained. In fact, given that we allow free construction, we do not allow any constraints in the BOD part of the component. However, given that there are constraints imposed on the import and export interface (CI, and CE), these translate into induced body constraints. Thus, $CB = s'(CI) \cup v'(CE)$. This ensures that morphism s is consistent. Note that $s' = Constraints(s)$ and $v' = Constraints(v)$, i.e., the *Constraints* functor applied to the morphisms.

It follows that the described approach of transforming a component into a fault tolerant one satisfies the consistency condition on morphisms.

Theorem: *The category of specifications with constraints and consistent specification morphisms has pushouts.*

The proof of the above theorem is direct and can be found in [6], and can be built in a similar way to that of the specification without constraints.

Thus, a component specification with constraints,
COMPW = (PARC,EXPC,IMPC,BOD,e,s,i,v), consists of three specifications with constraints, namely PARC, EXPC, and IMPC where PARC = (PAR,CP), EXPC = (EXP,CE), and IMPC = (IMP,CI), a specification without constraints, namely BOD, and four specification morphisms. However, given that constraints are translated along morphisms, there is a set of constraints, *CB*, induced on BOD, and is given by $CB = s'(CI) \cup v'(CE)$. In [12], the induced constraints are called *annotations*. Given that we used abstract interpretation for global consistency verification, that builds an abstract context for each variable and that each abstract context is derived from constraints imposed on the import interface (by wrappers), these abstract contexts, denoted as annotations, are the induced constraints on the BOD part of the specification.

Thus, from the above theorem, the wrappers generated using the heuristic (that preserves consistency of wrappers) allow for component specification.

5.3 Fault Tolerant System Synthesis from Fault Tolerant Components

Having discussed the construction of components with wrappers, we now look at how to construct a system from components with wrappers, i.e., components with constraints. We will keep the same approach as for the initial case of no constraints, but we will identify requirements that allow for composition of such components with wrappers.

Given two components with constraints, $COMPW_i$ and $COMPW_e$, and a a component morphism $cm : COMPW_i \rightarrow COMPW_e$, i.e., a pair $cm = (h, hp)$ of specification morphisms where $hp : (PAR_i, CP_i) \rightarrow (PAR_e, CP_e)$ and $h : (IMP_i, CI_i) \rightarrow (EXP_e, CE_e)$, the composition $COMPW_3$ of $COMPW_i$ and $COMPW_e$ via cm, written $COMPW_3 = COMPW_i \circ_{cm} COMPC_e$is given as in Fig. 4.

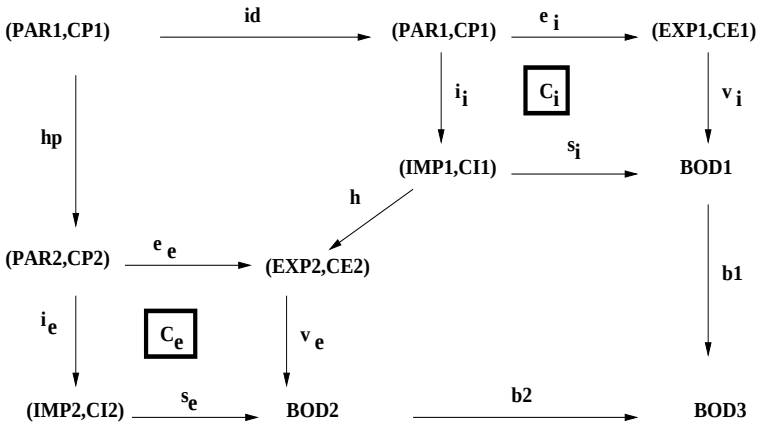

Fig. 4. Module specification from fault tolerant components

The component morphism $cm = (h, hp)$ is called *consistent* if h and hp are consistent specification morphisms. If the morphisms h and hp are not consistent, then correctness and composability is not guaranteed.

Lemma 5: Morphism h is consistent. **Proof:** Given that we translate input constraints W_i in an importing component C_i into output constraints W_e on an exporting component C_e, then $h'(W_i) = W_e$, which satisfies the condition for consistent morphism.

Lemma 6: Morphism hp is consistent

Proof: Components C_i and $C_e$to be composed have consistent specification morphisms. Given that during composition, $e_e \circ hp = h \circ i_i$, and from lemma 1, we get that morphism hp is consistent, where e_e, i_i means the e, i morphism of the exporting (C_e) and importing (C_i) components respectively.

Claim: The wrappers generated by the heuristic preserve the composation of the wrapped components.

Proof: The proof follows naturally from the previous two lemmas.

Given that input and output wrappers within a component are consistent (via consistent specification morphisms) and that the consistency of wrappers are preserved across different components (via consistent component morphisms), the set of wrappers in the system is called globally consistent.

6 Related Work

An approach to transform a fault intolerant system into a fault tolerant one has been presented in [13]. They represented a program as a state machine, and they proceeded to identify those transitions that may violate a given safety specification (invariant), according to the level of fault tolerance needed (i.e., fault masking, fail safe etc). Their work differs from ours in the sense that they assume white box software. In this paper, we assume gray-box software, i.e., knowledge of the component implementation is known, but is however non-modifiable. Such components may be obtained from repositories, which for maintenance reasons, are not modified.

[15] presents an transformational method for specifying and verifying fault tolerant, real time programs. The set of transitions possible in a given intolerant program is increased with faulty transitions. The authors used the concept of refinement for obtaining fault tolerant programs, and they provided different refinement conditions for fault tolerance verification. However, the authors reasoned at a white-box level, as opposed to our gray-box level approach here.

Work in [14] showed that design of detectors (assertion checks) is difficult. We have shown that detection of globally consistent detectors is NP complete, thus explaining the observation made. We further provided a heuristic that effectively tackled the problem of generation of globally consistent wrappers.

[5, 19] showed usage of category theory in system specification. Our initial intolerant specification is similar to their approaches. However, work in this paper differs from theirs in many respect, namely: (i) we showed how wrappers can be automatically generated, (ii)identified consistency properties of morphisms that preserve composability.

Formal methods such as Z [20] present analogous approaches of verifying consistency of assertions, by verifying whether an output assertion is consistent with an input assertion. In Z, postconditions are always verified against preconditions. Using category theory, we have looked at a calculus of components, however we believe that a calculus of schemas, as in Z, would retain most of the properties mentioned here. However, the assertions may not be globally consistent. Invariants are defined, which need to be preserved by operations. However, their suitability is rarely assessed, giving rise to defensive programming style. CSP [10] though allows hiding during composition, but is however not suitable for our purpose here. We referred to Z and CSP here as they are representative of a class of state-based and event-based formal methods.

Finally, we mention work in [16, 8] that advocates the use of wrappers for adding fault tolerance.

7 Discussion

In this section, we provide a general discussion on the heuristic and consistency of wrappers generated, and on the applicability of the approach in general.

As mentioned in the introduction, it may be the case that fault intolerant components are available, which have been formally verified. However, given the level of dependability required in embedded systems, wrappers are needed to transform these fault intolerant components into fault tolerant ones. The heuristic eases generation of wrappers that maintains composability of the components. Thus, generation of such wrappers can be automated, such as with the use of compilers. Also, given that we reuse the same free construction by not allowing constraints being imposed on the construction of the export services, verification need not be performed over again. Given the consistency condition imposed, and that the component is correct by construction, this implies that the morphisms are consistent and that the corresponding component with wrappers is correct by construction.

We also note that the global consistency property of wrappers allow for compositional verification, which is useful in software maintenance. Initially, we mentioned that to verify such a specification, axioms in one component are translated into theorems or lemmas along component morphisms in another component. However, if a component is modified for software maintenance, the whole proof procedure needs to be re run, to ascertain correctness. This means that one free construction is chosen over another free construction. However, having consistent wrappers, hence consistent morphisms, any subsequent verification remains local, i.e., we only need to ensure that the input and output wrappers of that given component are still consistent, i.e., the change in the component preserves consistency of the specification morphisms. Given that the module morphisms are still consistent, we do not have to do a whole proof procedure again for correctness verification.

Our approach can also be automated, by incorporating the heuristic in a compiler. After the first pass of performing syntax analysis, the heuristic can be run as a second pass, whereby wrappers are generated. Also, tools such as Moka [19] can be enhanced and used to automate the consistency check.

One potential limitation, in addition to having loops, may be that the wrappers generated may be cumbersome. However, we argue that with the level of dependability and performance needed in safety-critical systems, generation and use of such wrappers is justified, especially with the fact that their generation can be automated.

8 Summary and Conclusions

In this paper, we have explained the concept of global consistency of detectors. We have shown that its detection is intractable, and we have provided a heuristic, analogous to predicate transformers, that generate globally consistent wrappers (detectors). We have first specified a fault intolerant system, and explained its

subsequent verification. Then, using our heuristic, we have generated a set of fault tolerance components, i.e., detector wrappers in the form of assertions, we have shown how to systematically transform a given fault intolerant specification into a fault tolerant one. We have also shown that having globally consistent wrappers do preserve composability of components.

Earlier, we pointed out a limitation of our heuristic, which is that, due to loop structures within and among components, consistent wrappers may not be obtained. In such cases, we endeavor to take advantage of the fact that, for consistency, we require $W_2 \implies m'(W_1)$, and not the stronger $W_2 = m'(W_1)$ condition, as in this paper. Thus, as future work, we will look into generating wrappers that are consistent in the more general case, as mentioned above.

References

1. B. Alpern, F.B. Schneider, "Defining Liveness", *Information Processing Letters, 21(4):181–185, 1985*
2. A. Arora, S. Kulkarni, "Detectors and Correctors: A Theory of Fault-Tolerance Components", *Proc ICDCS*, pp 436–443, May 1998.
3. P. Cousot, R. Cousot, "Static Determination of Dynamic Properties of Programs", *Int. Symposium on Programming*, 1976
4. E.W. Dijkstra, "A Discipline of Programming", *Prentice Hall*, 1976
5. M. Doche et al, "A Modular Approach to Specify and Test an Electrical Flight Control System", *4th Intl. Workshop on Formal Methods for Industrial Critical Systems*, 1999
6. H. Ehrig, B. Mahr, "Fundamentals of Algebraic Specification 2: Modules Specifications and Constraints", *EATCS Monographs on Theoretical Computer Science*, Vol. 21, Springer Verlag, 1989
7. A. Ermedahl, J. Gustafsson, "Deriving Annotations For Tight Calculation of Execution Time", *Proc EuroPar'97, RT System Workshop*
8. T. Fraser et al, "Hardening cots software with generic software wrappers", *IEEE Symposium on Security and Privacy, pp. 2–16*, 1999
9. M. Hiller, A. Jhumka, N. Suri, "An Approach for Analysing the Propagation of Data Errors in Software", *Proc. DSN'01, pp. 161-170, 2001*
10. C. A. R. Hoare, "Communicating Sequential Processes", *Prentice Hall*, 1985
11. T. Jensen et al, "Verification of Control Flow Based Security Properties", *Proc. IEEE Symp.on Security and Privacy, pp. 89–103, 1999*
12. A. Jhumka, M. Hiller, V. Claesson, N. Suri, "On Systematic Design of Consistent Executable Assertions For Distributed Embedded Software", *to Appear ACM LCTES/SCOPES, 2002*
13. S. Kulkarni, A. Arora, "Automating the Addition of Fault Tolerance", *Proc. Formal Techniques in Real Time and Fault Tolerant Systems, pp. 82–93, 2000*
14. N.G. Leveson et al, "The use of self checks and voting in software error detection: An empirical study.", *IEEE Trans. on Soft. Eng., 16:432–443, 1990*
15. Z. Liu, M. Joseph, "Verification of Fault-Tolerance and Real-Time", *Proc. FTCS 1996, pp220-229.*
16. F. Salles et al, "Metakernels and fault containment wrappers", *Proc. FTCS*, pp. 22–29, 1998
17. G. Smith, "The Object-Z Specification Language. Advances in Formal Methods", *Kluwer Academic Publishers*, 2000

18. F. Tip, "A Survey of Program Slicing Techniques," *Journal Prog. Languages, Vol.3, No.3, pp.121–189*, Sept. 95
19. V. Wiels, "Modularite pour la conception et la validation formelles de systemes", *PhD thesis, ENSAE - ONERA/CERT/DERI*, Oct 97
20. J. Woodcock, J. Davies, "Using Z: Specification, Refinement, and Proof", *Prentice Hall*, 1996