A language for functional interpretation of model based simulation

Jonathan Bell [a] Neal Snooke [a,*] Chris Price [a]

[a] *Department of Computer Science, University of Wales Aberystwyth, Penglais, Aberystwyth, Ceredigion, SY23 3DB, U. K.*

Abstract

Functional modelling has been in use for a number of years for the interpretation of the results of model based simulation of engineered systems. Its use enables the automatic generation of a textual design analysis report that interprets the results of qualitative (or numerical) simulation in terms of the system's purpose. We present a novel functional description language that increases the expressiveness of this approach, increasing the range both of systems and design analysis tasks for which the approach can be used. The language also allows closer integration of functional modelling into the design process. The language allows a device function to be decomposed either in terms of subsidiary functions or required effects. We discuss the use of such alternative decompositions and propose a logic of functional description that is used to underpin the proposed language. The language has been used in the interpretation of electro-mechanical, hydraulic and fluid transfer systems in the automotive and aerospace industries to support tasks Failure Modes and Effects Analysis, Sneak Circuit Analysis, and Diagnosis. The language is not inherently restricted to these applications and the paper makes use of indicative examples from other domains.

Key words: Functional reasoning, Automated design analysis, FMEA, Model based simulation.

1. Introduction

The automation of the design analysis of engineered systems requires both the ability to simulate the behaviour of the system, and also the ability to interpret the results of the simulation. The automation of design analysis tasks at the focus of this work employs a functional model to provide interpretation. Examples of typical tasks include Failure Modes and Effects Analysis (FMEA), Sneak Circuit Analysis (SCA), simulation explanation, Fault Tree Analysis(FTA), and diagnosis.

While simulation of an engineered system reduces the work of building actual prototypes of the system, the interpretation of the results of the simulation remains a labour intensive process. Where the interpretation can also be automated, the resulting design analysis report can be generated automatically for several design analysis techniques. Design analysis tasks can also be undertaken early in the design lifecycle and it is feasible to repeat analysis as necessary in response to changes in the system's design. The consistent interpretation of results make it easy to automatically compare these results allowing engineers to focus on the significant impacts of design changes. A design analysis report will often be couched in terms of the achievement or otherwise of a system's intended purpose(s) together with the consequences of it failing.

This paper proposes a functional description of a system, where a system's function is viewed as a relationship between its behaviour and its purpose. A Functional Interpretation Language (FIL), is presented to support the description of device function. The FIL increases the expressive power of the functional labeling approach [1] by allowing partial

* Corresponding author.

Email addresses: jpb@aber.ac.uk (Jonathan Bell), nns@aber.ac.uk (Neal Snooke), cjp@aber.ac.uk (Chris Price).

achievement of a system's purpose to be described.

Functional modelling [2,3] has been in use for a number of years, both for deriving the behaviour of a system from knowledge of its structure and component function, and also for interpreting system behaviour. System behaviour is typically derived by simulation from the system's structure, component behaviour and domain rules. For example, an electrical circuit component list and connection diagram, may be combined with component behaviour models and the laws of circuit analysis to predict overall circuit behaviour.

The automation of design analysis is particularly valuable in the case of FMEA [4,5], in which the effects of component failures on the behaviour of the whole system are traced. As a wide variety of failures are included in the FMEA, this is an extremely laborious process. This, combined with the fact that it requires an engineer's knowledge, means that FMEA is an excellent candidate for automation. Automation of FMEA through model based analysis has been demonstrated in a commercial design tool [6] and Automated Sneak Circuit Analysis (SCA) is described in [7]. These analyses differ in that FMEA is a failure analysis, and can be carried out by comparing the behaviour of the system when it is working correctly with that when there is a component failure, whereas SCA is a form of design verification in which the basis of the comparison is some representation of the relations between the intended inputs and outputs of the system with which the results of the simulation can be compared.

A pragmatic approach to the interpretation of behaviour for the above tasks is undertaken by attaching functional labels to significant system behaviours [1]. These labels identify which outputs (or goal states) of a system are required for it to fulfil some intended purpose. The system behaviour is assumed to be derived from a connected-component simulation environment where the component behaviour is considered to be reusable within the assumptions of the simulation system and is independent of its function within the system. No behavioural causality is specified or required by the functional model, allowing it to concentrate on interpreting the achievement of purpose without regard to the identification of causal propagation that can be problematic, for example in electrical analysis. This allows a wide range of simulators to be used to generate system input-output behaviour and these may or may not be based on causal propagation [8].

The approach was successfully used for several commercially deployed tools exploiting the advantages that the modelling is easy for users (engineers) to understand, and the models allow simulation results to be automatically processed at a high level of abstraction. The widespread use of software in the latest generation of many systems has has allowed a rapid increase in functional complexity and the Functional Interpretation Language presented in this work supports these developments in several ways:

- to allow the functional descriptions to capture more complex functional relationships than the functional decompositions in [9] and [1]
- to enable the functional modelling of systems with complex effects
- to extend functional labeling to assist in the automation of additional design tasks, such as design verification
- to provide a formal description of the functional labeling approach which will make the approach useful for other functional reasoning tasks

The following section discusses the background to the work and introduces approaches to and uses of functional modelling. Section 3 presents a definition of function intended to increase the expressiveness of functional labeling and support the hierarchical descriptions presented in section 4. Section 5 presents some examples of the FIL in use.

2. Earlier uses and definitions of function for modelling

There is a good deal of agreement that there are four classes of knowledge that have a rôle in model based reasoning, as suggested in [10]. These four classes of knowledge are:

Structural Concerned with what components make up the system and the connections between them.

Behavioural Concerned with how the components and system behave.

Functional Concerned with why a component is in the system and explains what the components and system do.

Teleological Describes the purpose assigned to a system by its designer or users, and provides explanation of why a component is in the system

In the context of these classes of knowledge, function is a relational concept, relating the behaviour of a device to its purpose. According to [10] there are two approaches to describing it, leading to two def-

initions. The so-called purposive definition relates behaviour to purpose while the operational definition relates input to output. This work is focused on a purposive approach to function, as the aim is to express a system's behaviour in terms of its purpose.

Function is often viewed as less local to a device than its behaviour [11], defining function in terms of a device's response to an external stimulus. An alternative is the definition of function in [12,13] as the effect of a system or device on its environment distinguishing it from a device centred view of function:

> Function: Let G be a formula defined over properties of interest in an environment E. Let us consider the environment plus an object O. If O (by virtue of certain of its properties) causes G to be true in E we say that O performs, has or achieves the function (or rôle) G.

The purposive approach to function contrasts with the operational approach which relates function more closely to behaviour. In the Compositional Modeling Language [14], function is device centred and is used to describe an aspect of behavioural simulation rather than expressing purpose of the system.

While there is no definition of function in the earlier approach to functional labeling [1], the approach follows [11], being concerned with a system's response to a stimulus. In [1] a function is associated with the outputs of the system, so the function "stop lights on" is associated with the system state in which both of a car's stop lights are lit. There is no explicit representation of the stimulus of a function, because the application was FMEA and the stimulus can be derived from the simulation of the correctly working system. In the case of the car lights the simulation will show that when the brake pedal is pressed, the stop lights will light. This is acceptable where a function is triggered purely by an external stimulus (such as a switch being thrown) but less so if a function depends on the achievement or otherwise of some other function. Some representation of the triggering stimulus is also required for design verification, where the simulated behaviour of the system is compared to some representation of its intended behaviour. An alternative definition of function is proposed in [15]:

> An object O has a function F if there is an agent who can use O under some circumstances in some specific manner to achieve a goal.

This definition does not capture any idea of intention, so a user might misuse a chisel, say, by using it to lever open a paint can; a potentially damaging use and clearly not a purpose of a chisel. This model of the design process as functional refinement of a system follows [16].

Other researchers have sought to use knowledge of the function of components in a system to derive the overall behaviour of the system, as an alternative to qualitative reasoning. For example, [17] models a system in terms of its components' functional rôles, which contribute to the functional processes of the system and demonstrates the use of this approach for FMEA. These functional reasoning approaches have also been used to support diagnosis, [18], and have in common the idea that a system's function is expressed in terms of component functions that are related primarily by connections between the components, so capturing knowledge of the system structure. Similar models are also used for design synthesis tasks, such as conceptual design decomposition, embodiment design, parameter selection, and function identification from form. The use of function models for these tasks is particularly prevalent for mechanical systems where form and function are closely related. The FIL is not aimed at these tasks, however it is consistent with many of the requirements for these tasks [19]. The use of function models for synthesis also requires that a common library of function primitives is defined. The interpretative nature of function makes this problematic, as demonstrated by the fact that a number of function taxonomies have been proposed for different domains. For example several grammars (described in [19]) are available to describe mechanical systems functionality (including lathe, truss, vehicle, gear). Others have proposed generic functions based on energy flows [20,17] for example "conduit", "generator". These low level functions do not assist in explaining the purpose at the system level or the consequences of failures, which is the aim of this work. The functional model presented in this paper is hierarchical however, and therefore for a specific domain a library of such functions could form the lowest levels of a functional model, and indeed this would be necessary if the functional model is to be used for design synthesis tasks.

3. A logical approach to describing function

The definition of function that underlies the functional language presented here might be given informally as "how a device achieves its purpose". This

definition has the advantage of being distinct from both the notion of behaviour and of purpose (teleology). A more formal definition is

> Function: An object O has a function F if it achieves an intended goal by virtue of some external trigger T resulting in the achievement of an external effect E.

This leads to the idea that a representation of a device function must have three elements; a representation of the purpose the function is to fulfill, the *trigger* and the *effect* associated with the function. The automobile stop light function mentioned earlier might informally be described as "The stop light function achieves the purpose of warning following traffic that the vehicle is slowing down when pressing the brake pedal triggers the lighting of the left and right stop lights".

In a text based version of a FIL model, the keyword TRIGGERS is used to relate a trigger with the expected effect and the keyword ACHIEVES to relate function to purpose.

```
FUNCTION stop_light
 ACHIEVES warn_following_driver
 BY
  depress_brake_pedal TRIGGERS red_stop_lamps_lit
```

A simple functional model of the function above uses the labels "depress_brake_pedal" and "red_stop _lamps_lit" to act as hooks for linking to appropriate properties of the system's behaviour. In the case of an electrical simulation this would be the states of the pedal activated switch and the lamp respectively. The trigger and effect expression pair is subsequently called a *recognizer* for the function. They can be attached to the appropriate components in the system's structural model, following [1]. The labels themselves should describe the boundary of the system, for example the brake pedal above. The behaviour and simulation domain of interest may be restricted, and the behaviour recogniser can be used to ignore irrelevant aspects (such as the pedal lever mechanism during an electrical analysis) by attaching the trigger to the pedal switch position. Should a mechanical simulation be provided then the pedal position could be identified within the recognizer "depress_brake_pedal" to provide a more direct trigger that allows mechanical effects to be interpreted. The symbol "warn_following_driver" is a separate teleological model including a description of the purpose and the consequences of failure to fulfil the purpose, and may include measures of the *severity* and *detectability* of the function failure required for FMEA reporting. The effect recognizer similarly requires a boolean result derived from one or more outputs of the system.

The representation of the trigger and effect suggest that there are two ways the function can fail. Either the trigger can fail to result in the intended effect (the stop lights fail to light) or the effect is present without the trigger (the stop lights stay on when the pedal is not pressed). The trigger and effect are taken to be Boolean pre- and post-conditions for the function that are mapped to corresponding elements of the system's behavioural model, typically states of the input components and the effectors.

This modelling of the trigger and effect leads to a function having four possible states that we term inoperative, failed, unexpected and achieved. Where f is some device function they can be written as follows; inoperative as $\mathrm{In}(f)$, failed as $\mathrm{Fa}(f)$, unexpected as $\mathrm{Un}(f)$ and achieved as $\mathrm{Ac}(f)$. These possible states of a device function f are defined in terms of the truth of function's trigger t and effect e:

$$\mathrm{In}(f) \Leftrightarrow \neg t \wedge \neg e \tag{1}$$

$$\mathrm{Fa}(f) \Leftrightarrow t \wedge \neg e \tag{2}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg t \wedge e \tag{3}$$

$$\mathrm{Ac}(f) \Leftrightarrow t \wedge e \tag{4}$$

In addition to these states we have two additional states, termed triggered $\mathrm{Tr}(f)$ and effective $\mathrm{Ef}(f)$. These states depend on the truth of the trigger or effect respectively.

$$\mathrm{Tr}(f) \Leftrightarrow t \Leftrightarrow t \wedge (e \vee \neg e) \tag{5}$$

$$\mathrm{Ef}(f) \Leftrightarrow e \Leftrightarrow e \wedge (t \vee \neg t) \tag{6}$$

This allows the state of a function to be defined in terms of whether or not it is triggered and effective.

$$\mathrm{In}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge \neg\mathrm{Ef}(f) \tag{7}$$

$$\mathrm{Fa}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \neg\mathrm{Ef}(f) \tag{8}$$

$$\mathrm{Un}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge \mathrm{Ef}(f) \tag{9}$$

$$\mathrm{Ac}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \mathrm{Ef}(f) \tag{10}$$

This formulation of the definition of the states of a device function is useful in considering functional decomposition in Section 4, and is equivalent to rules 1 to 4 above.

For the two cases where the truth values for trigger and effect differ, the system behaviour is inconsistent with the intended functionality. For the generation of a design analysis report, it is necessary to

differentiate between these two cases because of the different consequences of these two classes of failure. For example if a "torch_lit" function fails (the lamp does not light) then the user is not helped to see, while if the effect is achieved unexpectedly (the lamp stays on) then the battery is drained. If a function's effect is true while the trigger is false, then the function itself is not achieved, strictly speaking, as it depends on both the trigger and effect. This is consistent with the consequences of unexpected achievement of a function's effects not relating to the function's purpose, as in the torch example above. The FIL model for the torch is therefore:

```
FUNCTION torch_lit
 ACHIEVES cast_light
 BY
  switch_on TRIGGERS lamp_on

PURPOSE cast_light
 DESCRIPTION "help user to see in the dark"
 FAILURE CONSEQUENCE "user can't find way in dark"
```

The FAILURE CONSEQUENCE provides the description of the consequences of the failure to fulfil the purpose of the function. The description of purpose is best not included in the functional description itself, but is instead a separate model consisting of a description of the purpose and the consequences of failure to fulfil the purpose when an associated function fails. Separating purpose from function is consistent with the idea that a description of function is concerned with how a purpose is fulfilled, and aids model reuse because the same description of purpose is common to several possible functional models. For example, the purpose of stop lights is common to any road vehicle, but the description of functions that achieve this purpose will differ according to the type of vehicle; a car's stop light function will be different from a motorbike's.

In addition to the consequences of failure, a description of purpose may include a similar description of the consequences of the effects being unexpected. UNEXPECTED CONSEQUENCES may therefore be included as in the following example to describe the function's effect being true despite the trigger being false.

```
PURPOSE warn_following_driver
 DESCRIPTION "warn other drivers - vehicle is slowing"
 FAILURE CONSEQUENCE "danger of rear end shunt"
 UNEXPECTED CONSEQUENCE "no slowing signal possible"
```

As an example of how this model is used, suppose a failure analysis of the torch is run with a wire broken, so the circuit is never completed. Following simulation, the resulting report will contain an entry:

> When torch was in state switch_on, the function torch_lit failed because the expected effect lamp_on was absent. Consequences are user can't find way in dark.

This text can be generated from the functional model above with the addition of simple linking phrases. In practice, design analysis reports have a tabular structure that simplifies the automatic generation of the text.

The reporting of function states requires a number of rules and these are described in the remainder of this section. Additional rules are required where functional decomposition is used, and these are described in section 4. In writing the rules for reporting of states of function, we use $\mathrm{R}(f)$ to report of the state of function f. This function is associated with consequences of failure c_f and an effect e. The rule for including the function in the report is written:

$$\mathrm{R}(f) \text{ if } \mathrm{Fa}(f) \vee \mathrm{Un}(f) \tag{11}$$

That is, the report will include a reference to the function if it has failed or is unexpected. If the function has failed, the report will also include the consequences of the failure.

$$\mathrm{R}(c_f) \text{ if } \mathrm{Fa}(f) \tag{12}$$

Where a device function f is unexpected, the rule for reporting the unexpected consequences u_f, if present, is

$$\mathrm{R}(u_f) \text{ if } \mathrm{Un}(f) \text{ and if } (f \text{ includes } u_f) \tag{13}$$

As the trigger and effect resolve to true or false they can be arbitrarily complex Boolean expressions, allowing partial triggers and effects to be combined using Boolean operators. The car's stop light, for example, requires both the left and right hand lamps to light for the effect to be true. A refined functional model for the stop lights might look like:

```
FUNCTION stop_lights
 ACHIEVES warn_following_driver
 BY
  depress_brake_pedal
  TRIGGERS
  left_stop_lamp_lit AND right_stop_lamp_lit
```

The choice between decomposition of a function into the required combinations of triggers and effects as

in this example, or subsidiary functions is discussed in the next section.

The design analysis report will also contain a reference to the effect whose absence (or unexpected presence) led to the function being failed (or unexpected). This is significant when a function's effect expression consists of several individual effects, e_n, each of which could be included in the report:

$$\mathrm{R}(e) \text{ if } (\mathrm{Tr}(f) \wedge \neg e_n) \vee (\neg \mathrm{Tr}(f) \wedge e_n) \quad (14)$$

This rule takes an effect to be some element of the function's effect expression. For the stop lights example in the situation where a failure causes no current to flow through the left lamp will clearly cause the "stop_lights" function to fail, and the above rule provides an explanation that effect "left_stop_lamp_lit" was absent.

Notice that this rule has no reference to the state of any associated function, so the report will include a reference to the inconsistent behaviour of the effect, even if this does not amount to failure of a function. For example, suppose a function f has a trigger t and an effect $a \vee b$. If some fault causes, say, a to fail, the report will say "When t, expected effect a was absent". This does not amount to failure of the function, provided effect b does not fail, as the presence of b in the disjunction is sufficient to ensure $\mathrm{Ef}(f)$. For the stop lamp example, a short circuit causing one brake lamp to light, may make the function inactive (trigger and effect false) but the report mentions that the effect left_stop_lamp_lit was present (Rule 14), i.e.

> Effect left_stop_lamp_lit present although associated function warn_following_driver is inactive.

Here, although not necessary, the inactive function associated with the effect is also reported:

$$R(f) \text{ if } (\neg \mathrm{Tr}(f) \wedge e_n) \wedge In(f) \quad (15)$$

Of course, if both lamps are lit due to a fault, $\mathrm{Ef}(f)$ is true so the function state is $\mathrm{Un}(f)$. The report includes "no slowing signal possible" (Rule 13), and a description of the consequences of unexpected achievement of the function:

> When depress_brake_pedal trigger not present, function warn_following_driver achieved unexpectedly. Consequences are that no slowing signal is possible.

Consequences do not always relate to the purpose of the system. For example, the significant consequence of the torch lamp staying lit is the draining of the battery, it has little to do with lighting the dark and so it would be inappropriate to add "UNEXPECTED CONSEQUENCE battery drained" to the "cast_light" purpose of the torch. Therefore we attach a description of the consequences of each effect being achieved unexpectedly to that effect. The following describes the effect for the torch example and links effects to the behaviour of specific structural elements such as the current flow in the "lamp" component identified in the structure (schematic). In this case a qualitative electrical simulation behaviour is used resulting in "ACTIVE" as a valid value of the current flow in the lamp. The behaviour is qualified with the function name, allowing several effect descriptions in a single file to link all effects (and triggers) to the physical system components.

```
lamp.filament.i = ACTIVE
 IMPLEMENTS torch_lit.lamp_on
 UNEXPECTED CONSEQUENCE "battery drained"
```

This allows the design analysis report to include the consequences of an effect, even where that effect's presence does not amount to unexpectedly achieving any function.

If $\mathrm{Un}(f)$ does occur and the function has no unexpected consequences included, we can still include consequences of the unexpected effect u_e. The rule for reporting unexpected consequences of effects $\mathrm{R}(u_e)$ is:

$$R(u\_e) \text{ if } \neg \mathrm{Un}(f) \ \vee \ \neg(f \text{ includes } u\_f) \quad (16)$$

A trigger for a device function is represented using the IMPLEMENTS keyword in the same way as an effect. Triggers will typically be a representation of the user's intentions for the system, expressed in terms of the state of those components that allow the user, whether a human operator or a larger, surrounding system, to interact with the system. Switch positions are a typical example. In most cases, the input of energy to the system is not part of the trigger, as its source might either be treated as part of the system – such as a battery – or beyond the scope of the system's design analysis, as would be the case for a mains electrical appliance. In this case, the public electricity supply can be regarded as a separate entity and taken for granted unless there are specific safety issues arising from failure of the supply, in which case it is best modelled as part of the system, rather than a triggering interface to the system.

4. Decomposition of function

In all but the simplest of cases, a device function will depend for its achievement on more than one trigger and effect. There are also cases where a function is better considered in terms of subsidiary functions, each with their own purpose, trigger and effect, beside being a necessary part of the top level function. One simple example of this is a cooking hob with several rings, where each ring has its own control knob, output of heat for cooking, and has the purpose to allow cooking. In addition, the (identical) functions of the rings also contribute to a top level "cook on hob" function – using logical OR – so that the correct working of any one ring allows some use of the hob. For a two ring cooker, the subsidiary functions "cook_on_left" and "cook_on_right" take the place of the recogniser in the top level function.

```
FUNCTION cook_on_hob
  ACHIEVES cook_food
  BY  cook_on_left OR cook on_right

FUNCTION cook_on_left
  ACHIEVES cook_on_ring
  BY switch_on_left TRIGGERS heat_left_ring

PURPOSE cook_food
  DESCRIPTION allow preparation of cooked food
  FAILURE CONSEQUENCE hob cannot be used for cooking

PURPOSE cook_on_ring
  DESCRIPTION use burner for cooking
  FAILURE_CONSEQUENCE burner cannot be used for
                cooking, limits cooking
```

This captures the idea that a failure of some component affecting all the rings, such as a wire connecting all the knobs to the power supply going open circuit, is more serious than one that only affects one ring, such as failure of a wire connecting a ring with its knob. It is worth noting that subsidiary functions must have a distinct purpose. Decomposing the stop lamps into "FUNCTION left_stop_light" would be inappropriate, since this function has no distinct purpose in the system, and is merely an effect that forms part of the real function described earlier. In general, subsidiary functions may be combined using the Boolean operators AND, OR and XOR.

One consequence of decomposing functions is that it becomes necessary to express the four states of achievement of the top level function in terms of the four possible states of the children. Where some function f is composed of two subsidiary functions a and b related using an arbitrary Boolean operator $\otimes$ the following rules are used to determine whether f is triggered and effective.

$$\mathrm{Tr}(f) \text{ if } \mathrm{Tr}(a) \otimes \mathrm{Tr}(b) \tag{17}$$

$$\mathrm{Ef}(f) \text{ if } \mathrm{Ef}(a) \otimes \mathrm{Ef}(b) \tag{18}$$

The state of the top level function f can then be determined using rules 7 to 10. Therefore, subsidiary functions can only be used where the same logical relation is appropriate for both the trigger and effect. This is consistent with the idea that the subsidiary function has its own trigger and effect. Where the expressions for the triggers and effects differ, either the top level function has to be composed using triggers and effects instead of subsidiary functions or, if lower level descriptions of purpose are to be included, incomplete subsidiary functions described later in this section can be used.

The resulting states of the top level function, expressed in terms of the subsidiary functions combined using AND, OR and XOR are shown in Table 1. In the table, for the cases where a fault does not affect the top level function – typically because it remains inoperative – the state of the top level function has been indicated in parentheses. In these cases, the design analysis report should simply ignore the top level function and report the problem with the child function. This follows the rule for reporting on the state of a function, rule 11. For example if a top level function depends on **Child 1 AND Child 2** then when Child 2 fails (its trigger is true but effect false) and Child 1 is inoperative (trigger and effect both false) then the top level function is also regarded as inoperative (both triggers are not true) so all that is needed in the report in this case is a note of the failure of Child 2.

It might appear anomalous that a top level function can be achieved despite this depending upon incorrect states of subsidiary functions but in fact it is consistent with fulfilling the purpose of the top level function. The clearest example of this is the penultimate row of Table 1 for the decomposition **Child 1 OR Child 2**. This apparently unlikely scenario, can easily arise in complex systems and failure modes and can be demonstrated in the hob example if connections between switches and effectors were inadvertently swapped. For example the knob intended for the front left ring actually controlled the left back ring and vice versa. In this case the top level function is achieved, as a necessary precondition is associated with a necessary postcondition.

Table 1
States of functions and sub-functions.

Child 1	Child 2	AND	OR	XOR
inoperative	inoperative	inoperative	inoperative	inoperative
inoperative	achieved	inoperative	achieved	achieved
inoperative	failed	(inoperative)	failed	failed
inoperative	unexpected	(inoperative)	unexpected	unexpected
achieved	achieved	achieved	achieved	inoperative
achieved	failed	failed	(achieved)	(inoperative)
achieved	unexpected	unexpected	(achieved)	failed
failed	failed	failed	failed	(inoperative)
failed	unexpected	(inoperative)	(achieved)	(achieved)
unexpected	unexpected	unexpected	unexpected	(inoperative)

The hob could be used for cooking once you found that a wrong ring was on, so the purpose is achieved. Notice that according to rule 11, although the top level function is not included in the design analysis report, the two child functions will be.

It is possible for a function to be achieved despite the failure of a child function, as is the case where the top level function depends on **Child 1 OR Child 2** and Child 2 has failed. In this case, the report will be cast in terms of the failure of Child 2, along the lines of "function top level achieved when Child 1 and Child 2 are triggered, despite Child 2 having failed" together with the consequences of the failure of Child 2. Using the hob example, the report would produce:

> function cook on hob achieved when cook on left and cook on right triggered despite cook on right having failed. Burner cannot be used for cooking, limits cooking.

In practice a properly set up design analysis will reveal the failure of Child 2 in terms of failure of the top level function as, of course, when only Child 2 is triggered, the expected top level function will fail. For example:

> function cook on hob failed when cook on right triggered because subsidiary function cook on right failed. Hob cannot be used for cooking.

The XOR operator completes Table 1 and describes the situation when one, but not both of the subsidiary functions must be achieved. The operator leads to some results that are not necessarily intuitive. Row seven describes the situation when one child is unexpected and the other is achieved. For this case, when the effects are both true, equation 17 provides a false effect for the parent function leading to an overall failed state for the parent function. This result is consistent with the specification provided by using the XOR operator to combine functions.

As a functional description has three elements (the purpose, trigger and effect) it is useful to introduce the concept of *incomplete* functions that combine two of these elements. There are three possible combinations. The first is a "purposive incomplete function" (PIF) which associates an effect with a purpose, but does not have its own trigger. Instead it shares a trigger with other PIFs that contribute to a top level function. A possible example is a warning system that both flashes a lamp and sounds a horn.

```
FUNCTION fire_warning
 ACHIEVES fire_notification
 BY
   smoke_dection OR alarm_button_pressed
   TRIGGERS
    PIF siren
    AND
    PIF strobe_light

PIF siren
 ACHIEVES audible_fire_notification
 BY sounder_activated

PIF strobe_light ...
```

In this case, both warnings are triggered by the same event, but each has its own effect. This also illustrates one use of distinguishing between subsidiary functions and merely the required effects for the top level function. In this case, a failure in the warning system that prevents one of the warnings being

given, but not both, does at least mean there is some warning, so the achievement of one of the subsidiary functions mitigates the failure of the main function. This contrasts with the case where both effects are simply associated with the top level function, so the absence of either is tantamount to complete failure of the main function. This is one of the motivations for distinguishing between decomposing a function in terms of subsidiary functions and in terms of effects. To allow achievement of one of a function's effects to mitigate failure of the function, that effect is associated with a subsidiary function and the consequences of failure of the functions follow rules below. We define f as the top level function with consequences of failure c_f, and a and b as subsidiary functions with consequences of failure c_a and c_b. Rules 19 and 20 are used where the subsidiary functions are combined using AND or OR.

$$\mathrm{R}(c_f) \text{ if and only if } \mathrm{Fa}(a) \wedge \mathrm{Fa}(b) \tag{19}$$

$$\mathrm{R}(c_a) \text{ if } \mathrm{Fa}(a) \wedge \neg\mathrm{Fa}(b) \tag{20}$$

These rules are inappropriate where the subsidiary device functions are combined using XOR and the opposite approach has to be taken:

$$\mathrm{R}(c_f) \text{ if } (\mathrm{Fa}(a) \wedge \neg\mathrm{Fa}(b)) \vee (\neg\mathrm{Fa}(a) \wedge \mathrm{Fa}(b)) \tag{21}$$

This is because failure of one of the subsidiary functions, but not both, is tantamount to failure of the top level function. These rules can be used whether the subsidiary functions are complete functions or PIFs. Note that the failure of the top level function will still be reported following rule 11.

While it might be supposed that there is no need for an incomplete function that maps a trigger to a purpose thereby sharing the effect with other subsidiary functions, there is actually a need for such "triggered incomplete functions" (TIFs). This is because PIFs and TIFs cannot always be replaced by complete subsidiary functions that share a trigger. If the subsidiary functions are combined using XOR, then if they share a trigger in the case of PIFs, or an effect in the case of TIFs, the triggers of the subsidiary functions will always have a common value. Following rules 17 and 18, the top level function will never be triggered if composed of PIFs, or effective if composed of TIFs, as one of subsidiary function's will never be true without the other. Therefore these incomplete functions are necessary for completeness of the language, even though TIFs will seldom be used. An example might be a Yale style lock which allows allows a door to be opened from outside with a key and from inside using a knob:

```
FUNCTION unlock_door
 ACHIEVES door_open
 BY
  TIF unlock_using_key
  OR
  TIF unlock_using_knob
  TRIGGERS lock_released

TIF unlock_using_key
 ACHIEVES entry_to_building_from_outside
 BY
  key_inserted_and turned

TIF unlock_using_knob
 ACHIEVES exit_from_building, allow_visitor_in
 BY
  turn_knob
```

Here "lock_released" is an effect with two purposes dependent upon the trigger used. The lack of effect in a TIF makes unexpected consequences inappropriate for the TIF functions, although the failure consequences are associated with the purpose in the usual way. Unexpected consequences may be added for the intermediate level effect (i.e. lock_released in the example), and this provides a rôle for TIF functions combined using AND. Clearly the subsidiary functions can never fail individually, however the effect can appear unexpectedly in response to only one of the triggers occurring.

The final class of incomplete function is an "operational incomplete function" (OIF) which is a combination of trigger and effect which is not associated with a purpose, but merely contributes to a top level function. A case in point is a room with, say, a wall light and a ceiling light, each with its own switch. Either one of these lights will allow an occupant to move around the room without hitting the furniture, so either contributes to a "light room" function, while – at least arguably – having no distinct function of its own. The "light room" function can be expressed as "wall light" OR "ceiling light":

```
FUNCTION room_light
 ACHIEVES find_way_around_room
 BY
  OIF wall_lamp
  OR
  OIF ceiling_lamp

OIF wall_lamp_switch_on TRIGGERS light_on

OIF ceiling_lamp_switch_on TRIGGERS light_on
```

This has the advantage of associating a trigger with its effect. After all, it is not the case that either switch can switch on either light as would be the case if we used:

```
FUNCTION room_light
ACHIEVES find_way_around_room
 BY
   wall_lamp_switch_on OR ceiling_lamp_switch_on
   TRIGGERS
   wall_light_on OR ceiling_light_on
```

The use of OIFs also allows the introduction of a finer grained functional model as the design proceeds. This might be done by promoting the OIFs to complete functions, by associating them with a purpose. For example, the wall light might be associated with the purpose of lighting a desk, for which the ceiling light might be inappropriate because it casts the user's shadow on the work.

These "incomplete functions" are only to be used as child functions of some other function, and not to try and represent component or sub assembly functions for a specific implementation. Sometimes subsystems may provide both a structural and functional decomposition because the physical structure has been organised functionally. We briefly consider local functions of implementation dependent components, and their relationships, in the context of an example in the following section. This section has presented a pure function decomposition using triggers and effects leading to four classes of subfunction:

- Complete child functions each having its own triggers, effects and purpose.
- Purposive incomplete functions which share a trigger.
- Triggered incomplete functions which share an effect.
- Operational incomplete functions each with its own trigger and effect but that do not by themselves fulfil a purpose but simply contribute to the top level function's fulfilling of its purpose.

5. Practical Examples

Having described the Functional Interpretation Language, we can discuss its application to a real world automotive example system, although only one function is presented for reasons of space. The passenger safety system includes car's seat belt warning system that lights a lamp on the dashboard and sounds a chimer if the car moves off with one of the front seats occupied but without the seat belt buckled. A functional description might be:

```
FUNCTION belt_warning
 ACHIEVES unbuckled_warning
 BY
  vehicle moving
  AND (driver unbuckled OR
    (passenger present AND
     passenger unbuckled))
  TRIGGERS
   PIF warning_lamp
   AND
   PIF chimer

PURPOSE unbuckled_warning
 DESCRIPTION
 "Warn that passenger is unbuckled"
 FAILURE_CONSEQUENCE
 "no warning given of dangerous state"

PIF warning_lamp
 ACHIEVES visual_warning
 BY lamp_on

PURPOSE visual_warning
 DESCRIPTION
  "Show driver that a belt is unbuckled"
 FAILURE_CONSEQUENCE
  "no persistent visual warning given"

PIF chimer
 ACHIEVES audible warning
 BY chimer_sounding

PURPOSE audible_warning
 DESCRIPTION
  "Draw attention to unbuckled seat belt"
 FAILURE_CONSEQUENCE
  "no audible warning given"
```

This example illustrates the point that PIFs can be used to model cases where the triggers and effects are combined using different logical expressions.

This system is electrical but some of the functionality is implemented using electronic control units, so some of the components use software. We assume the use of a mixed electrical and state machine based simulator, such as described in [21]. The elements in the trigger expression of the belt_warning function and effect expressions of the two PIFs can then be mapped to appropriate states in the component behavioural models.

Having built the functional model and attached the labels to the structural model, a failure analysis is run on the system. Assuming the design is correct, simulating the system with all components behaving correctly results in all the device functions be-

ing achieved as expected. Suppose that the simulation is now run with wire connecting both the lamp and chimer to ground open circuit, so a circuit is never completed for either component, and despite the trigger condition being true (maybe the driver is unbuckled) neither effect will be achieved. The design analysis report will include an entry similar to:

> Function belt_warning failed because function warning_lamp failed because expected effect lamp_on was absent and function chimer failed because expected effect chimer_sounding was absent. Consequences are no warning given of dangerous state.

On the other hand, if a wire connecting the lamp to ground goes open circuit, at least the chimer will sound, so some warning is given. This will be reported:

> Function belt_warning failed because function warning_lamp failed because expected effect lamp_on was absent. Consequences are no persistent visual warning given.

In the second case, the failure of the top level function is still reported but its effects are mitigated by the consequences of the failed subsidiary function being used.

This allows the design analysis report to differentiate between the failure of both subsidiary functions and either one of them. If the two effects are not associated with child functions, then the failure of either or both will result in the same report. This decomposition of a function in terms of effects and triggers would be used in the case of the stop lights system example mentioned earlier, because the lighting of one but not both of the stop lamps will not be regarded as mitigating the failure of the function, if only because of the legal requirement for a car to show two stop lights.

For FMEA, the system's behaviour will be simulated with all possible component failures, and the effects of each will be found. For example, if the detector in the passenger seat sticks, this will result in the warning sounding when there is no passenger in the car. This example has considered only one function. In many cases a system will embody several functions and the FIL can be used to relate these functions to each other. This is especially true of software systems.

Subsidiary functions within the FIL are designed to form a purely functional hierarchy. For design synthesis tasks it is common to refine systems to include the product structure decomposition, and to link functions provided by components to support the system functions. For example in mechnical design [22] provides a model that aims to associate function into the product representation. The FIL includes a PERFORMED BY attribute to provide a link to lower level components or sub assemblies if a relationship between structure and function decomposition is required. This information implicitly encodes many implementation choices into the function hierarchy however, resulting in a model that is no longer a purely functional description. As an illustration consider the Yale lock presented earlier. The lock may be built of three modules such as the bolt, the latch and the lock barrel perhaps considered as 'subsystems' with individual functions below:

```
barrel FUNCTION turn
 ACHIEVES use_key_to_move_latch BY
   key_inserted_and_turned
    TRIGGERS
   open_barrel_lever

latch FUNCTION release
 ACHIEVES retract_bolt BY
   turn_knob OR open_barrel_lever
   TRIGGERS bolt_lever_retracted

latch FUNCTION sprung_extend
 ACHIEVES return_bolt_to_extended_position BY
   NOT turn_knob AND NOT open_barrel_lever
   AND NOT lever_output_pushed
   TRIGGERS bolt_lever_extended

bolt FUNCTION connect
 ACHIEVES secure_lock_to_slam_plate BY
 extend_bolt TRIGGERS bolt_restrained_by_slam_plate

bolt FUNCTION disconnect
 ACHIEVES release_lock_from_slam_plate BY
 retract_bolt TRIGGERS bolt_released_from_slam_plate
```

At this level, these components are often designed with a specific relationship to an adjacent component, resulting in for example a component called 'bolt_lever' as part of the latch. The name does not actually imply any specific relationship within the functional model however. In some cases an output from one element becomes the trigger for another function (e.g. 'open_barrel_lever'), however these links do not build a complete causal model, as can be seen in the lock example, because a trigger for a function only describes the behaviour that causes the function to occur rather than all the inputs that may be required for the expected behaviour to occur. The structure and other simulation is required to determine all of the behavioural interactions between functions (e.g. between latch and bolt above).

The purpose of each module is in terms of its role at the next higher level of the decomposition, in this case adjacent modules or the lock itself, consistent with the notion of purpose used for the FIL. The choice of components for a decomposition will depend upon the level of detail required for the design tasks. For example many products define the idea of a replaceable unit for the purpose of diagnosis and repair and the functions of these modules may be considered as the most detailed level of granularity required.

The implementation details required to specify the relationship between the system and module function means that it cannot form part of the system functional hierarchy and it is provided alongside details of the *recogniser* details of the FIL. This maintains the implementation details separate from the system function decomposition. For example, for the lock in section 4:

```
bolt.position=bolt.geometry.retracted
 IMPLEMENTS lock_released
..etc..

TIF unlock_using_key
 PERFORMED BY
   barrel.turn
   AND latch.release
   AND bolt.disconnect

TIF unlock_using_knob
  PERFORMED BY
    latch.release
    AND bolt.disconnect
```

Notice that triggers such as "key_inserted_and _turned", "open_barrel_lever" need to be defined in terms of force applied since the trigger is that an attempt is made to move the component, and when the component is stuck (or something linked component is stuck) the trigger is still required to be effective or else the function will be inactive and not failed. For component functions completed as follows an analysis of the latch mechanism becoming stuck will report:

> When key_inserted_and_turned and when turn_knob, unlock_door failed because unlock_using_key failed and unlock_using_knob failed. Cannot enter building from outside, cannot exit building, cannot allow visitor in. Sub assembly functions barrel.turn and latch.release failed. Consequences are no movement of latch lever and no movement of bolt mechanism.

```
barrel PURPOSE use_key_to_move_latch
 DESCRIPTION "key engages with tumblers and allow
          lock lever to move"
 FAILURE CONSEQUENCE "no movement of latch lever"

latch PURPOSE retract_bolt
 DESCRIPTION "the knob or barrel lever force
          is used to retract the bolt"
 FAILURE CONSEQUENCE "no movement of bolt mechanism"
```

In this case the bolt function is simply inactive because no force was transferred to the trigger and the bolt didn't retract. If the key is faulty, when both unlock functions are attempted, the report states that unlock_using_key fails (unlock_door can be achieved due to unlock_using_knob). The worst case is:

> When key_inserted_and_turned unlock_door failed because unlock_using_key failed. Cannot enter building from outside. Sub assembly function barrel.turn failed. Consequences are no movement of latch lever.

It is feasible however that at the lower levels of product decomposition, many common components have regularly used functions and these could be reused from a library together with their recognisers which are all local to the component model. At the lowest levels functions such as "conduit" can be identified, with a purpose "transfer fluid from A to B" for a pipe connecting A and B. The details of these functions might conceivably be derived from the structure model or the global behaviour models such as electrical circuit simulation. PERFORMED BY links with the high level system function would allow more detailed explanation and fault localisation, perhaps forming part of a functional design synthesis tool. Clearly such models can become complicated and would not be recommended for many design tasks, particularly at such a low level where the additional detail is unnecessary, even for diagnosis tasks that only require localisation to a replaceable assembly. The discussion was intended to demonstrate possibilities for decomposition using the FIL that may be applied to large systems where large functional sub assemblies are used.

Two observations are worth making about the above model. Firstly, that bolt component 'bolt_released_from_slam_plate' effect would probably be implemented as:

```
bolt.position=bolt.geometry.retracted
 IMPLEMENTS bolt_released_from_slam_plate
```

This is the same recogniser as 'lock_released' demonstrating that this single effect is the output of the 'disconnect' function of the bolt and the

'unlock_door function effect of the system.

Secondly, it might be tempting to specify the trigger of performer functions in a causal sequence (such as the lock) by using the achievement of another component function to create an explicit dependency between functions. This leads to interpretation problems when functions are in the Un(f) state or when partial effects exist since the functions are not actually related directly, but only to common behaviour elements and the higher level functions. It *is* possible and useful to use dependencies between functions in cases such as warning, fault mitigation (backup, fault tolerence, limp home), interlocking, and recharging functions. In these cases one function is explicitly intended to be active (or in some other state) based on the state of another. We do not pursue these specialised *dependent functions* in this paper, suffice to say the FIL is able to support them.

6. Conclusion

The functional description language proposed herein allows a more precise specification of required system behaviour than the one used for functional labeling in [1,8]. The specific inclusion of the triggering condition for the function allows the language to be used for interpretation (and so for report generation), for design verification as well as failure analysis. The specification of the triggering condition of a function has several advantages compared to the earlier functional labeling approach, though these must be set against an increase in complexity. Where the trigger is not included, then the use of the language for interpretation is limited to failure analysis where the triggers can be derived from the simulation of the system working correctly. This is not appropriate for design verification, as some description of the triggers and effects is required for comparison with the simulation. Even for failure analysis the trigger has to be specified for device functions such as warning or fault mitigation that are triggered by the state of some other device function, since the trigger of such functions cannot be unambiguously derived from the simulation of the system working correctly. The use of the trigger allows these cases to be unambiguously specified so different kinds of warning or telltale functions can be distinguished. The trigger of a function can also be specified to an arbitrary degree of precision at the expense of reducing the reusability of the functional model. This, combined with the use of labels for attaching system properties, allows a functional description of the system to be created independently of the system, so it can be used both to assist in capturing detailed requirements for the system and also to support the design process, where this follows the model in [16]. This differs from the approach in [1] where functional labels are added to an existing structural description of the system to be analyzed.

This function decomposition in section 4 uses binary logical operators, however a function might also depend on a combination of temporally more complex behaviours resulting in intermittent or sequential effects, so the sequential operators described in [23] might also be used. These give rise to few complications as they are unary operators (as are the temporal logic operators to which they are related) so decompositions are simple. If a function depends on two child functions being achieved in sequence then naturally if one of the children fails, so will the top level function.

The FIL allows functional descriptions of a system to be made before the physical design of the system is arrived at. For example, such a functional model might be used as a way of capturing the functional requirements of a design in such a way that the behaviour of a candidate physical design can, in due course, be verified against these requirements.

The FIL is intended to be used for the description of high level system function and therefore the majority of the model will be specific to each system since it is designed to capture intention. The modularity of the description does make it possible to reuse various elements of the models on occasion, especially for new versions of a previous system.

The proposed language shares the advantages of simplicity, reusability and capability claimed for functional labeling in [1] with the additional advantage of allowing the functional model to be constructed independently of the system, so supporting the use of the language in functional refinement of a design and for constructing a complete functional description of a proposed system for comparison with the (simulated) behaviour of that system in design verification. This, combined with the possibility of using the language to specify the functional requirements of the system, allows the closer integration of functional modelling in the design process.

References

[1] C. J. Price, Function-directed electrical design analysis, Artificial Intelligence in Engineering 12 (4) (1998) 445–456.

[2] J. Sticklen, A. Goel, B. Chandrasekaran, W. E. Bond, Functional reasoning for design and diagnosis, in: Proceedings Model Based Diagnosis International Workshop (DX-89), 1989.

[3] Y. Iwasaki, R. Fikes, M. Vescovi, B. Chandrasekaran, How things are intended to work: Capturing functional knowledge in device design, in: Proceedings of 13th International Joint Conference on Artificial Intelligence, 1993, pp. 1516–1522.

[4] J. E. Hunt, C. J. Price, M. H. Lee, Automating the FMEA process, Intelligent Systems Engineering 2 (2) (1993) 119–132.

[5] J. B. Bowles, C. Wan, Software failure modes and effects analysis for a small embedded control system, in: Proceedings Annual Reliability and Maintainability Symposium, IEEE, 2001, pp. 1–6.

[6] C. J. Price, AutoSteve: automated electrical design analysis, in: Proceedings ECAI-2000, 2000, pp. 721–725.

[7] C. J. Price, N. A. Snooke, J. Landry, Automated sneak identification, Engineering Applications of Artificial Intelligence 9 (4) (1996) 423–427.

[8] C. J. Price, N. A. Snooke, S. D. Lewis, A layered approach to automated electrical safety analysis in automotive environments, Computers in Industry 57 (5) (2006) 451–461.

[9] N. A. Snooke, C. J. Price, Hierarchical functional reasoning, Knowledge-Based Systems 11 (5–6) (1998) 301–309.

[10] L. Chittaro, A. N. Kumar, Reasoning about function and its applications to engineering, Artificial Intelligence in Engineering 12 (4) (1998) 331.

[11] V. Sembugamoorthy, B. Chandrasekaran, Functional representation of devices and compilation of diagnostic problem-solving systems, in: J. L. Kolodner, C. K. Riesbeck (Eds.), Experience, Memory and Reasoning, Erlbaum, 1986, pp. 47–73.

[12] B. Chandrasekaran, J. R. Josephson, Representing function as effect: Assigning functions to objects in context and out, in: Proceedings of AAAI-96 Workshop on modelling and reasoning about function, American Association for Artificial Intelligence, 1996.

[13] B. Chandrasekaran, J. R. Josephson, Function in device representation, Engineering with Computers 16 (3–4) (2000) 162–177.

[14] D. Bobrow, B. Falkenainer, A. Farquhar, R. Fikes, K. Forbus, T. Gruber, Y. Iwasaki, B. Kuipers, A compositional modeling language, in: Proceedings 10th International Workshop on Qualitative Reasoning QR-96, AAAI Press. ISBN 978-1-57735-001-9, 1996.

[15] Y. Iwasaki, M. Vescovi, R. Fikes, B. Chandrasekaran, Causal functional representation language with behavior-based semantics, Applied Artificial Intelligence 9 (1) (1995) 5–31.

[16] J. Gero, Design prototypes: A knowledge representation schema for design, AI Magazine 11 (4) (1990) 26–36.

[17] P. G. Hawkins, D. J. Woollons, Failure modes and effects analysis of complex engineering systems using functional models, Artificial Intelligence in Engineering 12 (4) (1998) 375–397.

[18] J. Sticklen, B. Chandrasekaran, Integrating classification-based compiled level reasoning with function-based deep level reasoning, in: W. Horn (Ed.), Causal AI Models, Steps Toward Applications, Hemisphere Publishing Corporation, 1989, pp. 191–220.

[19] J. Summers, N. Vargas-Hernandez, Z. Zhao, Comparative study of representation structures for modelling function and behaviour of mechanical devices, in: Proceedings of DETC2000: Computers in Engineering, 2001, pp. 775–787.

[20] M. van Wie, C. R. Bryant, M. R. Bohm, D. A. McAdams, R. B. Stone, A model of function-based representations, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 19 (2005) 89–111.

[21] N. A. Snooke, Simulating electrical devices with complex behaviour, AI Communications 12 (1,2) (1999) 45–58.

[22] J. Baxter, N. Juster, A. D. Pennington, A functional data model for assemblies used to verify product design specifications, in: Proceedings of the Institution for Mechanical Engineers, Part B - Journal of Engineering Manufacture, Vol. 208, 1994, pp. 235–244.

[23] J. Bell, N. A. Snooke, Describing system functions that depend on intermittent and sequential behavior, in: Proceedings 18th International Workshop on Qualitative Reasoning, QR2004, 2004, pp. 51–57.