

# *Functional decomposition for interpretation of model based simulation*

Bell, Jonathan; Price, Chris; Snooke, Neal

*Publication date:*
2005

*Citation for published version (APA):*
Bell, J., Price, C., & Snooke, N. (2005). *Functional decomposition for interpretation of model based simulation*. 192-198. Paper presented at 19th International Workshop on Qualitative Reasoning, Graz, Austria.

# Functional decomposition for interpretation of model based simulation

**Jonathan Bell, Neal Snooke and Chris Price**
Department of Computer Science, University of Wales Aberystwyth
Penglais, Aberystwyth, Ceredigion, SY23 3DB, U. K.
jpb, nns, cjp@aber.ac.uk

## Abstract

Description of system function is already in use as the basis of an approach to interpretation of the results of simulation in design analysis, allowing an automated design analysis tool to generate a textual report detailing the results of the simulation in terms of its purpose. This paper presents a novel functional description language that allows cases where the individual elements of a required system function themselves constitute (subsidiary) functions from cases where this is not so. This increases the expressive power of the functional description, allowing the automatic generation of design analysis reports of greater precision than has previously been possible and increases the range of systems and design analysis tasks for which the approach can be used.

## 1 Introduction

Design analyses of engineered systems typically entail the production of a textual report that describes the result of the analysis. This report will be cast in terms of whether the system fulfils its intended purpose and any consequences of it failing to do so, or of any other unexpected behavior. The production of such a report therefore entails the interpretation of the results of some simulation (which might be qualitative or numerical) of a system in terms of the system's purpose. While the use of computerised simulation saves a good deal of effort on the part of the engineer, for design analysis to be fully automated the interpretation and production of the report (or at least a draft thereof) should also be undertaken by the design analysis system. As functional knowledge is generally taken to be concerned with the relationship between the behavior of a device and its purpose, this interpretation of simulation is a useful rôle for such knowledge.

All but the simplest systems have functions that depend on several inputs and outputs, so functional descriptions have to allow these to be related to one another. This paper presents a language for functional representation that allows this decomposition of function to distinguish between subsidiary functions (each with its own purpose) and collections of required effects of a function. This allows an automated design analysis tool to generate reports that are more detailed and precise than was previously possible. The greater expressiveness of the language presented also increases its usefulness in support of other design activities.

The increasing sophistication of modern systems places increasing reliance on design analysis to establish the correctness and safety of their design. It also complicates the functionality of devices, so if a functional model of a device is to be used as part of the design process, the language used must be capable of representing this greater complexity. Typical examples of such system functions are warning functions that inform the user of the state of the device.

The automation of design analysis, resulting in the automatic production of a suitable report is particularly useful with safety analyses such as Failure Mode Effects Analysis (FMEA) and Sneak Circuit Analysis (SCA). These tasks are repetitive and require an engineer who is familiar with the system to undertake them. Their automation means the analyses can be carried out early, when any changes indicated are easily made, and often, in response to such changes. FMEA is a example of failure analysis, the basis of which is comparison between the simulation of the correctly working system with simulation of the system as affected by component failures. This allows the triggers of system functions to be derived from the correct simulation (in many cases) unlike in the case of design verification analyses (such as SCA).

## 2 Background - the uses of function

Representations of function have been used for three main tasks. The functional reasoning community have used knowledge of a system's structure and of the functions of its components to derive the system's behavior. This has been done to support diagnosis [Sticklen *et al.*, 1989] and FMEA [Hawkins and Woollons, 1998]. Here system function is expressed in terms of the relations between component functions. This contrasts with a "top down" view of system function that has been used in support of the design process by [Iwasaki *et al.*, 1993], following the model of the design process in [Gero, 1990] where design is viewed as the functional refinement of a system, breaking down the system functions until they can be related to component behaviors.

A similar system level view of function is used for interpretation of results of model based simulation for automatic design analysis in [Price, 1998]. In this "functional labeling"

approach, system functions are attached to significant behaviors, showing which system outputs are required to achieve a specific purpose. These behaviors are typically represented in terms of outputs or goal states, so the function "room lit" is associated with the system state in which the light is on. The expressiveness of this approach was increased by allowing hierarchies of function, described in [Snooke and Price, 1998], so that the resulting design analysis report, instead of simply reporting that, say, a car's headlamp function had failed, might now present a more detailed report, such as "headlamps failed because left headlamp failed". The present work is primarily concerned with increasing this expressiveness further by distinguishing between cases where the presence of some of the required output mitigates the failure of the main function and cases where it does not and also cases where outputs provide alternative ways of achieving a system function.

The functional labels in [Price, 1998] did not include any description of the triggers or preconditions of the function because they could be derived from the simulation of the system behaving correctly for the intended use of failure analysis. They were added to allow the use of functional labels for SCA. The present language adds a representation of the trigger of a function, which, it is suggested, increases the usefulness of the language, both for interpretation of model based simulation in design analysis and for support for functional refinement in the design process itself.

## 3 A language for functional description

The basis for the functional description language presented in this paper is a concern with how a device achieves an intended purpose, when viewed from the outside of the device. 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.

While this definition is novel it is consistent with the idea of function as a relation between purpose and behavior in [Chittaro and Kumar, 1998]. Indeed it could be seen as combining both their "purposive" and "operational" ideas of function. It is also not inconsistent with the idea of function as effect in [Chandrasekaran and Josephson, 1996] and with the idea of function viewed as a device's response to an external stimulus in [Sembugamoorthy and Chandrasekaran, 1986]. Unlike many other views of function used in the model based reasoning and functional reasoning communities, this view of function is distinct from both behavior and purpose.

This definition of function leads to the idea that a representation of function must include three elements, a representation of the purpose fulfilled when the system achieves the function, the trigger that stimulates the function and the function's effect. If the trigger and effect are treated as Boolean expressions, then they can be used to define the state of achievement of the function, as illustrated in Table 1. The function states in the third column of the table are defined in terms of the truth of the trigger and effect, so where a function is 'achieved' for example, its trigger and effect expressions both resolve to true. The trigger acts as the precondition for the function, so when it resolves to false then the function is not called for. Likewise, the effect is the post-condition. This captures the possibility that a function's effect might be achieved unexpectedly as well as the more likely case where the trigger does not result in the expected effect, when the function is said to have failed. The trigger and effect agreeing in value are consistent with correct behavior of the system, though it should be noted that an effect might be associated with more than one trigger, and also, as the function is seen as an external "black box" view of the system, the possibility exists that an incorrect behavior might result in (apparently correct) achievement of a function. It is not appropriate to reduce the four states of a function to these consistent and inconsistent pairs, however, as the consequence of failure of the function (trigger true and effect false) typically differ from those of unexpected achievement of the function (or strictly speaking of its effects).

| Trigger | Effect | Function |
|---|---|---|
| false | false | inoperative |
| true | false | failed |
| false | true | unexpected |
| true | true | achieved |

Table 1: Achievement of function using trigger and effect.

The keyword TRIGGERS is used to separate the trigger from the effect while ACHIEVES is used to label the purpose associated with the function. The function's recognizer (used to show the state of achievement of the function) is labelled using BY. In simple cases the recognizer is the pair of trigger and effect. A simple functional description of a room light is

```
FUNCTION room_light
  ACHIEVES "light room"
  BY
    switch_on
    TRIGGERS
    lamp_on
```

where the labels "switch_on" (the trigger) and "lamp_on" (the effect) are used to attach properties of the simulated behavior.

In practice a description of purpose will be more complex than the simple textual statement shown above. It will typically also include a description of the consequences of failure to fulfil the purpose (that is of failure of the function) and possibly also numerical values for the severity and detectability of such a failure to allow the generation of a "risk priority number" for the failure to fulfil the purpose. Therefore it is preferable to separate the description of purpose from the functional description itself. This is, of course, consistent with the idea that function is concerned with how a purpose is fulfilled as distinct from the purpose itself. It also has the practical advantage of encouraging model reuse, as different systems will fulfil a given purpose in different ways. For example, the external lighting systems of a car and motorbike fulfil the same purposes, so these models could be reused, while both the triggers and effects are different. In the functional description, then, the description of purpose is replaced by a reference to a separate model.

The unexpected achievement of a function's effect will also have a similar set of consequences, though the content will

differ in most cases. As these consequences do not typically relate to the purpose of the system, and are more specific to the system, they are better associated with the coupling between the functional model's effect and the system property that implements that effect.

The inclusion of the trigger is an important difference from the functional labeling approach in [Price, 1998]. Not only does it allow the functional model to be used for design verification, as well failure analysis, it also allows functional models to be used in cases where a function's trigger cannot be unambiguously derived from the simulation of the correctly working system. These include cases where a given function is triggered by the state of some other system function. Examples include telltale and warning functions and fault tolerant backup functionality. This is illustrated later. As the precondition for the function, the trigger captures the external stimuli that call for the function to be achieved, so model the user's intention for the system, for example by describing switch positions.

The description of the trigger, together with the use of labels in the trigger and effect allow the functional model to be constructed independently of any system to which it is to be attached. This aids reuse of the functional model and supports its use for functional refinement of the design similarly to the approach in [Iwasaki *et al.*, 1993]. It also allows the language to be used for establishing the functional requirements of a system.

While in simple cases, the trigger and effect of a function might simply be labels, as illustrated in the example above, the fact that they are treated as Boolean expressions allows the use of logical operators to combine expected triggers and / or effects. These might be the Boolean operators while to describe more temporally complex combinations the sequential operators described in [Bell and Snooke, 2004] can also be used. This is the simplest case of functional decomposition, which area forms the subject of the following section.

## 4 Decomposition of function

The idea that a function can depend on more than one trigger and effect can also usefully be extended to cases where a given system function is best decomposed in terms of subsidiary functions. This allows cases where achievement of some but not all of a top level function's required effects have different consequences from failure of all of the effects to be described, and also cases where a function can be achieved by one of several alternative combinations of triggers and effects. These are illustrated below.

If functions are decomposed into subsidiary functions, this raises the necessity of relating the state of the top level function (as in Table 1) to the states of the child function(s). These relations are shown in Table 2. In the table, where an entry is in brackets, the automatically generated failure report text will not refer to the top level function, but rather to the failure of the child function. The rule used is that the triggering and effect of the top level function depend on the relations between the triggers and effects of the child functions. For example, in the third row of Table 2, Child 1 is inoperative (defined as its trigger and effect both false) and Child 2 is failed (trigger true but effect false). A function dependent on Child 1 AND Child 2 is therefore inoperative as the trigger of Child 1 AND the trigger of Child 2 resolves to false and the effect of Child 1 AND the effect of Child 2 also resolves to false. As this function is inoperative the report can ignore it in this case and instead draw attention to the failure of Child 2. This approach gives rise to apparently anomalous results, such as where there are two children combined using OR and one child has failed and the other effect is unexpected. It is perhaps questionable whether this amounts to achieving the top level function. Where a function is achieved by alternative effects (as here) arguably its purpose can be fulfilled if an unexpected alternative effect is present. For example, a hob might have a function that is achieved provided that at least one ring heats up in response to being turned on. Arguably this function is achieved if the wrong ring comes on (you could still cook on the hob, at least if you know which ring is on) even though it is not consistent with correct behavior of the system. In such cases, the report will include entries for the subfunction failures, which reduces the significance of such possible anomalous results.

Where a function is composed of subsidiary functions, these might be complete representations of a function, each with its own (reference to) purpose, trigger and effect or they can be incompletely represented, because they share an element with other linked subsidiary (child) functions. Given that there are three elements in a complete functional representation, there are three possible incomplete representations of function, pairing trigger and effect, pairing effect and purpose and pairing trigger and purpose. As a trigger cannot fulfil a purpose without resulting in an effect, the last of these can be ignored, so we have two classes of "incomplete functions" that can be used in a hierarchical functional decomposition. These have been named consistently with the alternative approaches to function in [Chittaro and Kumar, 1998] as "operational incomplete function" (abbreviated to OIF) for one that pairs trigger and effect (so relates input and output) and "purposive incomplete function" (PIF) for those that relate effect and purpose.

Neither of these classes of incomplete function should be used other than as subsidiary functions of a top level (complete) function. Their use allows a function to be decomposed in four ways, as illustrated in Figure 1. Introducing a description of purpose lower in the hierarchy (by using subsidiary functions or purposive incomplete functions) allows a more precise identification of the failure. Purposive incomplete functions allow effects that share a trigger to be associated with their own purposes, perhaps mitigating failure of the top level function. For example, where a warning system gives both an audible and visual signal, the presence of one of these signals means some warning is given. In this case, purposive incomplete functions can be used as the two signal child functions will share a trigger (the failure that is the subject of the warning).

Operational incomplete functions allow the grouping of triggers and effects that provide alternative means of achieving a function and where the effects of the subsidiary functions fulfil the same purpose. A simple example is where a room has two lamp circuits, each with its own switch and

| 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) |

Table 2: States of functions and sub-functions.

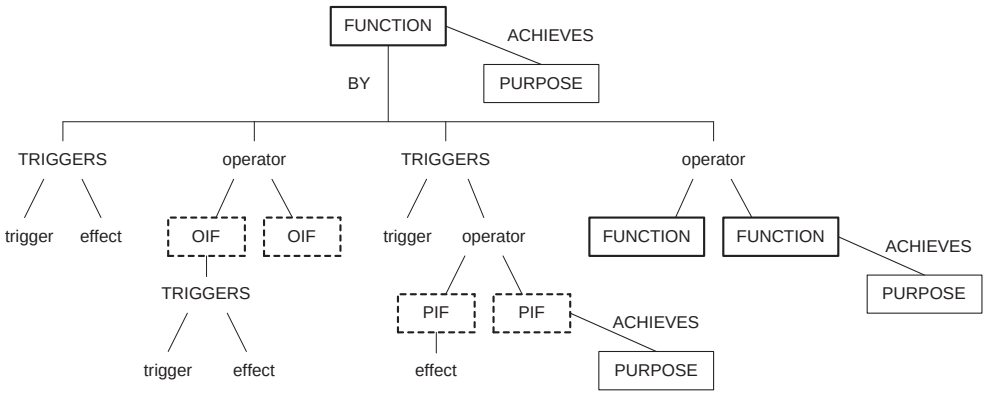


Figure 1: Four ways of decomposing a system function

lamp and either of which will serve to light an occupant's way around the room. Notice that

```
FUNCTION room_light
  ACHIEVES light_way_around_room
  BY
    OIF ceiling_light
    OR
    OIF wall_light
```

where each operational incomplete function associates a switch with its lamp, differs from

```
FUNCTION room_light
  ACHIEVES light_way_around_room
  BY
    wall_lamp_switch_on
      OR ceiling_lamp_switch_on
    TRIGGERS
    wall_lamp_on OR ceiling_lamp_on
```

as in this case, either switch could switch on either (or both!) lamps and any fault that caused the wrong switch to trigger the wrong lamp would go undetected. This could be avoided by having each switch and lamp in its own clause, but this loses the idea that a function has a trigger and an effect. Another benefit from the use of operational incomplete functions is the possibility that as the design is refined, they can readily have a purpose added, promoting them to complete functions. For example, the wall_light function might later have the purpose "light desk" added as the design of the room proceeds, so promoting it to a complete (if subsidiary) function.

As will be seen in the example above, where a function is composed of subsidiary functions (whether complete or incomplete), an expression relating these subfunctions replaces the trigger and effect as the recognizer of the main function.

Having introduced the idea that using subsidiary functions can be used to generate more precise reports, its use can now be discussed. This follows the approach in [Snooke and Price, 1998], but the area is developed more fully than was the case in that paper. The basis of the approach is that subsidiary functions that have their own reference to purpose are used to represent cases where achievement of one of the top level function's effects mitigate the failure of the function. This contrasts with cases where this does not apply. Using Boolean AND and Boolean OR with either subsidiary functions or with effects gives four possible outputs when describing failure of a function. Each will be briefly described in turn.

Commonly, of course, a function might depend on two (or more) effects, the failure of either of which is regarded as tantamount to failure of the function. In this case, subsidiary functions are not used, as in Figure 2. In all these figures, a thick lined box indicates a function and a thin one a description of purpose. In this case there is no distinction between

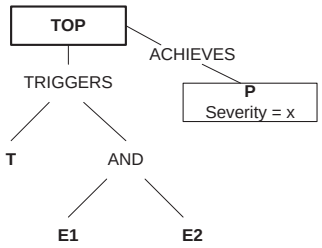


Figure 2: The result of combining effects using AND

failure of one of the effects or both, as either amounts to failure of the function itself, although the textual output will indicate which effect was missing. In this case, then, the failure of one effect has the same severity as the failure of both, and the consequences are also identical. A simple example might be a car's stop lights, if only because failure of either stop light renders the car legally unroadworthy. In this case, the failure of either or both of the stop lights to light in response to the brake pedal being pressed might be "Function brake light not achieved, because expected effect right lamp lit was absent. Consequences are no warning given to following driver. Severity 8".

This contrasts with the case where the failure of a top level function is mitigated by achievement of one of the required child functions (which might be either a complete function with its own trigger or one of a set of purposive incomplete functions that share a trigger). This is illustrated in Figure 3. This allows the effects of the failure of one of the child functions to be distinguished from the effects of the failure of both. In the warning system mentioned earlier, failure of the visual warning will result in the output in the report using the consequences and severity value associated with the visual warning child function, but it will include a reference to the failure of the top level function, "Function warning failed because of failure of function visual warning. Consequences are no lasting visual indication of system failure. Severity 7". If both effects (subfunctions) fail the report will use the consequences and (greater) severity value associated with the top level function. The report will still include a reference to missing effects. This is particularly valuable where the child functions themselves depend on more than one effect. Arguably the reporting of the consequences and severity of the subsidiary failure is not entirely consistent with AND, but it does seem useful to be able to distinguish between this case and the earlier case, and this does provide a simple approach to drawing this distinction. The report does still include the reference to the failure of the top level function. Notice that if the consequences of failure of the top level function are reported, there is no distinction between this case and the earlier one, and nothing gained by using subsidiary functions as their failures will not be reported. An alternative would be to include addition operators, but this merely adds complexity without appearing to result in any useful increase in the expressiveness of the language compared to the approach adopted.

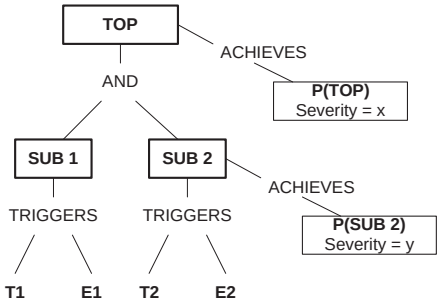


Figure 3: Using subfunctions to mitigate AND

Another possible decomposition is where a function is satisfied by any one of its subsidiary functions being achieved, as illustrated in Figure 4. In this case, the failure of one subsidiary function does not prevent achievement of the top level function, so the report need not refer to that function but only to the failed subsidiary function. A possible example is the hob functional model, where a top level "cook on hob" function is achieved by any of the four rings' identical "cook on ring" functions. Suppose the left front ring fails, then the "cook on hob" function is achieved by using another ring (subject to possible limitations on the sophistication of the cuisine!). This case might be reported as "Function cook on ring failed for left front ring because expected effect of ring heating was absent. Consequences are left front ring not available for cooking. Severity 4." This captures the differ-

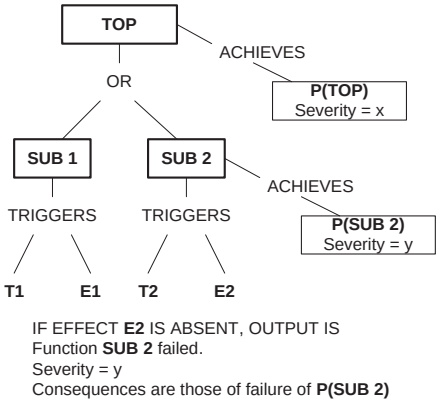


Figure 4: Combining subfunctions using OR

ence in severity between failures that cause any one ring to fail and failures that cause all the rings to fail, which will be missed if the rings' outputs are simply treated as effects of the "cook on hob" function.

It is possible (though perhaps unlikely) that there are several ways of achieving some function each of which does not itself really have its own distinct purpose. The room light example mentioned above is a possible case, as the alternative lights' functions might be felt to be too imprecise for useful modeling, and for safety analysis all that is felt to matter is that an occupant can find his or her way around after dark. In this case, there is no call for a description of purpose below the top level function, as in Figure 5. Here once both lamps

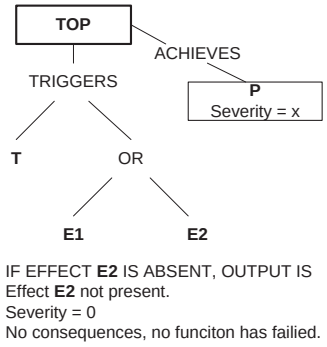


Figure 5: Combining alternative effects using OR

are switched on the report need, strictly speaking, include nothing as the function is achieved. This is clearly unhelpful, so a reference to the absence of the missing effect will be included, but will have no consequences or value for severity. If the wall lamp fails, the report might read, "Expected effect wall lamp lit absent". At some point in the testing, of course, only the failed part of the system will be active (only the wall lamp switched on) and this will, of course result in the function.

It will seldom (if ever) be the case that OR will be used to combine effects or purposive incomplete functions, as it is extremely unlikely that a trigger can result in one or other (or both) of two different effects being achieved. In cases where such a non-deterministic model applies, it will often be the case that the trigger needs to be more closely specified. For example, in a software system it might be the case than an input can result in one of two alternative paths of execution (so effects) but this will almost invariably depend on the value of the input and will not actually be non-deterministic. It will be necessary to model the trigger in such a way that both paths are tested, of course, in this case.

Exclusive OR can be used in much the same way as OR, but its use will, it is suggested, be extremely rare. There can be few top level functions that can be correctly achieved by either one, but not both, of two subsidiary functions (or effects). In general, where XOR might be used the subsidiary functions are generally better viewed as separate functions. A case in point might be a car's direction indicators, where "indicate turn" might (carelessly!) be modelled as "indicate left XOR indicate right". This is clearly incorrect, as in any given situation substituting one subfunction for the other is simply misleading. Those two functions are better modelled as separate functions.

In addition to the use of these conventional logical operators, the sequential operators described in [Bell and Snooke, 2004] can readily be used either to combine expected effects or subsidiary function so, for example, a washing machine's wash function might decomposed as a sequence of wash, rinse and spin functions, each of which can be considered to have its own associated purpose and the consequences of failure of each function are different.

## 5 Using the functional language

With the increasing sophistication of many systems, the functional models required for design analysis are increasingly complex, not only individually, but in their relationships. For example a modern domestic washing machine will have indication and warning functions to inform the user of the current state of the wash cycle and more specifically to indicate that the wash is complete and the machine can be unloaded or that a problem has arisen. A simple model of a function to indicate that the wash is complete might look like this.

```
FUNCTION wash_completed_indicator
  ACHIEVES show_wash_completed
  BY
    FUNCTION wash ACHIEVED
    TRIGGERS
      PIF telltale_lamp
      AND
      PIF chimer

PURPOSE show_wash_completed
  DESCRIPTION
  "Indicate wash cycle is complete"
  FAILURE_CONSEQUENCE
  "User not told machine can be unloaded."

PIF telltale_lamp
  ACHIEVES visual_indication
  BY lamp_on

PURPOSE visual_indication
  DESCRIPTION
    "Show user wash cycle is complete"
  FAILURE_CONSEQUENCE
    "No lasting indication that wash is complete"
```

Each section of the example is taken to be a separate model. In addition to referring to a separate description of purpose, a functional decomposition might involve subsidiary functions that might themselves be separate models, with the advantage of encouraging reuse of these subsidiary functions. As in the example above, a function might also refer to some other related system function. Here, the functional model and both the subsidiary purposive incomplete functions, as well as the three related purpose descriptions can all be separate files. An alternative approach would be to use a database for storing these models, using the database keys as the references, instead of the filenames.

This is an example of subsidiary functions combined using AND. The achievement of either subfunction mitigates the failure of the wash_completed_indicator function, as some indication is given. The chimer purposive incomplete function has been omitted to save space. It will contain a series of buzzes as the required effect. This can be described using the 'sequence' operator discussed in [Bell and Snooke, 2004]

Another interesting point that has not been discussed is the relationships between unexpected achievement of the effects. In this and many other cases, this is cumulative. Here, if the lamp output is achieved unexpectedly (so the telltale lamp is on continuously) then the user will still know that the wash cycle is complete if the buzzer gives the correct indication (and the user hears it!) while if both outputs occur continuously, no indication is given. This means that while in most cases the consequences of unexpected achievement of an effect are best attached to the link between the effect label and the system property that implements the effect, it might be necessary to attach descriptions of unexpected consequences at other, higher, levels of the functional hierarchy.

The specification of the trigger as the wash function being ACHIEVED does, of course, have the specific meaning that the trigger and effect of that function are both true. This distinguishes the indication function from a warning function that is triggered by failure of the triggering function. In practice it is likely that specific triggers and effects of the triggering function will be used, rather than the state of the function itself, so as to specify the trigger of the dependent (warning) function more precisely. As the trigger can be specified to whatever degree of precision the model builder thinks fit, and the use of labels for the trigger and effect allow the functional model to be built without reference to the target system, the language can be used to support functional refinement of the system design, following the model of the design process in [Gero, 1990]. These features also allow the language to be used to clarify the requirement of the system's functionality.

The approach to functional decomposition described herein differs from the earlier approach in [Snooke and Price, 1998] by using the difference between decomposing a function into subsidiary functions and into expected effects to increase the expressive power of the language. It also differs from that paper (and the functional reasoning approach) by not relating the system's functional hierarchy and its structural hierarchy. There is no explicit modeling of component functions, though it will typically be the case that subsidiary functions will be related to at least one possible structural decomposition of the system.

## 6 Conclusion

The functional language described herein increases the expressive power of descriptions of system function over earlier approaches. This increases the range of design analyses that can be automated using the functional labeling approach, to include both design verification and failure analysis, and also increases the range of systems to which functional descriptions can be applied. The distinction between decomposition using subfunctions and using effects allows cases where partial achievement of a function is better than nothing to be differentiated. The explicit inclusion of the trigger of a function allows system functions whose trigger is the achievement or failure of some other function to be unambiguously described as well as enabling the use of the approach for design verification. The modeling of the and use of labels for triggers and effects also allows for the functional models to be built independently of the target system. This allows the use of the language for functional refinement of a design and also for the related task of specifying the behavioral requirements of the system (from an external viewpoint), so increasing the use of the language beyond its rôle in interpretation of model based simulation of engineered systems.

## References


[Bell and Snooke, 2004] Jonathan Bell and Neal A. Snooke. Describing system functions that depend on intermittent and sequential behavior. In *Proceedings 18th International Workshop on Qualitative Reasoning, QR2004*, 2004.

[Chandrasekaran and Josephson, 1996] B. Chandrasekaran and John R. Josephson. Representing function as effect: Assigning functions to objects in context and out. In *Proceedings of American Association for Artificial Intelligence*, 1996.

[Chittaro and Kumar, 1998] Luca Chittaro and Amruth N. Kumar. Reasoning about function and its applications to engineering. *Artificial Intelligence in Engineering*, 12(4):331, 1998.

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

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

[Iwasaki *et al.*, 1993] Yumi Iwasaki, R. Fikes, M. Vescovi, and B. Chandrasekaran. How things are intended to work: Capturing functional knowledge in device design. In *Proceedings of 13th International Joint Conference on Artificial Intelligence*, pages 1516–1522, 1993.

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

[Sembugamoorthy and Chandrasekaran, 1986] V Sembugamoorthy and B Chandrasekaran. Functional representation of devices and compilation of diagnostic problem-solving systems. In Janet L. Kolodner and Christopher K. Riesbeck, editors, *Experience, Memory and Reasoning*, pages 47–73. Erlbaum, 1986.

[Snooke and Price, 1998] Neal A. Snooke and Christopher J. Price. Hierarchical functional reasoning. *Knowledge-Based Systems*, 11(5–6):301–309, 1998.

[Sticklen *et al.*, 1989] Jon Sticklen, A. Goel, B. Chandrasekaran, and W. E. Bond. Functional reasoning for design and diagnosis. In *Proceedings Model Based Diagnosis International Workshop (DX-89)*, 1989.