POELog: a Prolog-based engine for Problem Oriented Engineering

Jon G. Hall
Lucia Rapanotti

4th April, 2008

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk
Abstract

Problem Oriented Engineering (POE) is a formal system for engineering design. In previous work, we have successfully applied POE to a variety of problems in the context of software engineering. However, even for design problems of modest complexity, the need for an automated tool to keep track of all design artefacts has become apparent. For scalability to real-world problems a tool is imperative. This paper presents a first generation tool for POE based on Prolog. It argues how the Gentzen-style basis of POE allows for a compact and elegant Prolog encoding, which we call POELog. A LaTeX based front-end provides a convenient user interface to POElog for development and testing.

1 Introduction

Problem Oriented Engineering (POE) is a formal system for engineering design. Engineering design (shortly, design) is the creative, iterative and often open-ended endeavour of conceiving and developing products, systems and processes (adapted from [ECS03]). By necessity, it includes the identification and clarification of requirements, the understanding and structuring of the context into which the designed artefact will be deployed, the specification of a design for a solution that can ensure satisfaction of the requirements in context, and the construction of arguments, convincing for all validating stake-holders, that the designed artefact will provide the functionality and qualities that are needed. The involvement of stake-holders motivates collecting and recording evidence of the designed artefact’s fitness for purpose.

The framework is still developing, although its instantiation to software engineering (POSE, for Problem Oriented Software Engineering) [HRJ08, HRJ07] is now quite mature and has been applied to a variety of safety critical developments [HMR07, MHR08, MHR07c, MHR07b, MHR07a]. This work has demonstrated the benefits of POE when applied to real-world problems. It has also highlighted the need for tool support, to help the designer focus on the creation of design artefacts and the application of sound processes, rather than wasting effort keeping track of them.

Tool support for POE is under development and takes advantage of the formal encoding of POE as a Gentzen-style system [Kle64], which maps quite naturally into Prolog [SS87]. The tree structure of POE developments, a result of its Gentzen-style basis, can be easily computed and traversed by Prolog, with unification as the means to share design artefacts in separate POE development subproblems, hence taking care of consistency. Moreover, Prolog backtracking can be exploited to rewind POE development trees in case of unsuccessful design steps.

The paper is structured as followed. Section 2 provides a brief introduction to POE. Section 3 introduces our current Prolog encoding of POE, called POElog. Section 4 gives an examples of the POE development of a safety critical system, indicating the corresponding POElog encoding at significant steps. Section 5 reflects on what has been achieved. Finally, Section 6 concludes the paper and discusses current and planned development.

2 Problem Oriented Engineering

Problem Oriented Engineering (POE) (see [HRJ08, HRJ07] for its application to software) is a Gentzen-style natural framework for engineering design (see, for instance, [Kle64, Pel99]). As such, POE supports rather than guides its user as to the particular sequence of design steps that will be used; the user choosing the sequence of steps that they deem most appropriate to the context of application. The basis of POE is the problem sequent for representing design problems requiring designed solutions. The transformations defined in POE transform problems as sequents into others in ways that preserve solutions (in a sense that will become clear). When we have managed to transform a problem to
axioms\textsuperscript{1} we have solved the problem, and we will have a designed solution for our efforts.

POE is a formal system for working with non-formal and formal descriptions, as POE is designed to work with problems not propositions as in the original natural deduction: the characteristic that distinguishes it most from natural deduction is the guarding of transformations by justification obligations, the discharge of which establishes the ‘soundness’ of the application with respect to stake-holders. Natural deduction is based on a single absolute notion of correctness provided by proof whereas, through justifications, POE caters for the engineering notion of fitness-for-purpose, something that is often very far from correctness.

A POE problem has three elements: a real-world context, $W$, a requirement, $R$, and a solution, $S$. The problem context is a collection of domains ($W = D_1, ..., D_n$) described in terms of their known, or indicative, properties, which interact through their sharing of phenomena (i.e., events, commands, states, etc.) \cite{Jac01}). The problem requirement states how a proposed solution description will be assessed as the solution to that problem. Like a domain, a requirement is a named description with phenomena; a requirement description should always be interpreted in the optative mood, i.e., as expressing a wish. A solution is a domain, intended to solve a problem, i.e., when introduced into the problem context will satisfy the problem requirement. The possible descriptions of a solution range over many forms, from high-level specification through to detailed designs.

A problem’s elements come together in POE in a problem sequent:

$$D_1, ..., D_n, S \vdash R$$

Here $\vdash$ is the problem builder and reminds us that it is the relation of the solution to its context and to the requirements that we seek to explore. By convention, the problem’s solution domain, $S$, is always positioned immediately to the left of the $\vdash$.

One way to visualise a POE problem is illustrated in Figure 1\textsuperscript{2}, where the problem is to design a solution for the safe operation of a two-button press, a medium-size press commonly used in the machine tool industry, usually working in close proximity to a human operator. The operator puts some material to be pressed on the press table, and pushes two buttons that order the press to activate. The two buttons are used for safety: the operator needs to use both hands to press the buttons at the same time (one for each button) and, as a consequence, the risk of hand injury is reduced greatly.

\textsuperscript{1}An axiomatic problem is a problem whose known fit-for-purpose solution is known.

\textsuperscript{2}The notation is reminiscent of that of Problem Frames \cite{Jac01}, which shares a similar notion of problem as POE, albeit aimed at software specification.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{2-button_press_problem.png}
\caption{The 2-button Press Problem}
\end{figure}

In the diagram, the undecorated rectangles, Operator, Button Panel and Press are context domains; the decorated rectangle, Solution, is the solution to be found; the oval, Safe Operation is the requirement; and the arc annotations are shared phenomena. We will adopt this notation throughout the paper for illustration of the various formal POE problem definitions we will encounter.

The descriptions of a problem’s elements may be in any language, different elements being described in different languages, should that be appropriate. So that descriptions in many languages may be used together in the same problem, POE provides a semantic meta-level for the combination of descriptions; notationally, this is a role of the ‘\textcircled{1}’ that collects into a problem sequent the domains that appear around the turnstile, formally making each visible to the others through their shared phenomena\textsuperscript{3}. Throughout the paper we will use natural language descriptions of problem elements.

POE problem transformations capture discrete steps in the problem solving process, by relating a problem and a justification to (a set of) problems. Problem transformations conform to the following general pattern. Suppose we have problems $W, S \vdash R$, $W, S_i \vdash R_i$, $i = 1, ..., n$, $(n \geq 0)$ and justification $J$, then we will write:

$$W, S \vdash R$$

$$W_1, S_1 \vdash R_1 \quad ... \quad W_n, S_n \vdash R_n \quad \text{[NAME]} \quad \text{[J]}$$

\textcircled{1} to mean that, derived from an application of the NAME problem transformation schema (discussed below):

$S$ is a solution of $W, S \vdash R$ with adequacy argument $\langle CA_1, ..., CA_n \rangle \land J$ whenever $S_1, ..., S_n$ are solutions of $W_1, S_1 \vdash R_1, ..., W_n, S_n \vdash R_n$, with adequacy arguments $CA_1, ..., CA_n$, respectively.

By specialising this pattern, many classes of transformations are recognised in POE (see \cite{HRJ08}), reflecting a variety of engineering practices reported in the literature or observed elsewhere. Each of them prescribes the way in which the conclusion problem (that below the line) is related to the premise problem(s) (those above the line), and

\textsuperscript{3}A situation similar to that found in the propositional calculus in which conjunction and disjunction, etc., serve to combine the truth values of the atomic propositions.
which form the justification must take, called justification obligation. When applying a pattern, the justification must be discharged by providing evidence which contributes towards the overall adequacy argument.

### 2.1 Justification general form

POE explicit requirement for discharging justification obligations at each step is a distinguishing feature of the framework. From our experience of repeated application of POE to engineering problems, the following general form of the justification has emerged:

<table>
<thead>
<tr>
<th><strong>STEP ID:</strong></th>
<th>Application of <strong>NAME</strong> to problem <strong>P</strong> for <strong>STEP RATIONALE</strong></th>
</tr>
</thead>
</table>

**JUSTIFICATION:** A justification can be named for ease of reference.

**DESCRIPTIONS & PHENOMENA:** The collection of descriptions and phenomena of the domains and requirements introduced into the problem by the step or the manipulations defined thereon by the step. For an application of the Context Interpretation step, for instance, a detailed description of the elements of \( W \) and \( W' \) would be given, alongside any relationship that holds between them, such as shared descriptions, etc.

**CONCERN:** Name

**STATUS:** Status

A concern (c.f., [Jac01]) is something that is important to the development, presumably because it relates to some stakeholder in the process. In high integrity development, for instance, the reliability concern is likely to arise; a design that does not address such a concern in such a context is likely to be unvalidatable. The status of a concern is one of pending, discharged, undischARGEABLE. The work appertaining to the discharge of a concern is structured: each concern has associated with it the following:

**CLAIM:** The statement of the claim(s) that will discharge the concern;

**ARGUMENT & EVIDENCE:** The reason to believe each claim (or the reason it does not hold);

**RISKS:** A description of the risks involved in continuing the development should the concern fail to be discharged, and/or the secondary risk introduced by the discharge of the concern. A description of the treatment of risks residual to the step.

A concern established as part of a step may be addressed (and therefore discharged) in design steps subsequent to that in which it is established, i.e., when, as part of other design steps, evidence in support of its associated claim is discovered. The argument and evidence may, therefore, make reference to other concerns, arguments and evidence in the design tree. The validity concern for a step, that subject to external validation

---

4From which we have omitted elements of risk management, which are very important in engineering design, but we do not address in this paper.

by problem- and solution-owning stake-holders, will typically be required to ensure that relationships between concerns and their discharge are adequate.

**CONCERN:** Step Validity

**STATUS:** Status

The status of the step validity concern, possible values include pending, signed-off, undischARGEABLE

**ARGUMENT & EVIDENCE:** Explanation of the status after validation, including the relationships where evidence was gathered in the design, and the treatment chosen for the residual risk of the step.

**SIGNATORY:** To recognise the stake-holder or stake-holders that signed-off the step.

---

Each element is optional, typically depending on the developmental stage and context. Similarly, the collection of concerns at each step application depends on the particular transformation pattern applied.

A concern leads to a claim stated within a justification, the claim being that the concern is discharged by the development step. The justification will, eventually, contain arguments and evidence that the claim is valid so that the concerns is discharged. We say eventually because some concerns can only be discharged after the ramifications of a problem transformation are known which is, typically, later in the development tree.

One particularly important concern is the **step validity concern**—for which the associated claim is that a particular step is validateable—as it is the point of contact of the POE process with stake-holders external to the creative process of the problem solver, as explained below.

### 2.2 Process pattern

Typically, engineering design under POE proceeds through repeated application of the process pattern of Figure 2, captured as a UML activity diagram [OMG]. The pattern highlights three main constituents—activities, choice points and roles—which we will discuss in turn the following. Two main activities are present:

(Partial) Candidate Problem Exploration: to capture (increasing) knowledge and detail in the context and requirement of the problem;

(Partial) Candidate Solution Exploration: to structure the solution (or part thereof) according to a candidate architecture.

The partial nature of the candidates is so that early problem solving can focus on parts of the problem or solution, rather than the whole problem straight away.

The choice points are:
Problem solver
Problem owning
stake-holder
Solution owning
stake-holder

(Partial) candidate
problem exploration

(PCS) exploration

validation

Problem owning stake-holder

Problem solver

Solution owning stake-holder

Figure 2. POE process Pattern: to move towards a partial solution to a general engineering problem, we first understand the problem better (1), reflecting our understanding of the problem through validation with the problem holding stake-holder (2); use engineering judgement to determine a candidate solution architecture (3), then test the candidate for satisfaction of concerns, iterating if necessary (4).

(Partial) Candidate Problem Validation: to determine whether the current candidate characterisation of the problem context and requirement is appropriate to start investigating a solution, or we need explore the problem further;

(Partial) Candidate Solution Validation: to determine whether the current candidate solution is viable as the basis of a solution or whether, instead, we should backtrack the development to find another candidate solution or explore the problem further.

The role are those of problem owning stake-holder(s), solution owning stake-holder(s), and problem solver, their respective scopes indicated by shading in the figure (note that the roles, as such, do not overlap).

A problem owning stake-holder is someone whose role is to validate a (partial) candidate problem description that results from Partial Candidate Problem Exploration. There are many familiar examples of problem owning stake-holders. These include, but are not limited to, those of customer (those that pay for a product), clients (those that pay for a service), regulator (those requiring safety, for instance), end-user (those who will use the product or service when commissioned). It is the problem owning stakeholders’ role to answer the question “Is this (partial) problem description valid for you?” Depending on the problem-owning stake-holders’ responses, the problem solver may need to re-explore the problem (when the answer is “No!”), or move on to try to find a (partial) solution (when the answer is “Yes!”).

The role of the solution owning stake-holder(s) is to validate a candidate solution description, such as an architecture (a partial solution) or choice of component (i.e., something of complete functionality). Solution owning stake-holders include, but are not limited to, a development house’s chief software architect—who knows which architectures their organisation uses in solutions, an oracle—who determines which of a number of features should be included in the next release, or a project manager—who needs to timebox particular activities; there are many other roles that fit solution owning stake-holder. It is the solution owning stake-holders’ role to answer the question “Is this (partial) solution description valid?” Depending on their response, the problem solver may need to re-explore the solution (when the answer is “No!”), move back to exploring this or a previous problem (when the answer is “No, but it throws new light on the problem!”), or moving on to the next problem stage (when the answer is “Yes!”).

The role of problem solver is that of the person(s) that begins by trying to understand the problem and iterates towards a solution. As indicated by the upward pointing arrow that appears in the upper right of Figure 2, iteration is not always local: it is, for instance, possible that through the failed validation of a solution a previous problem description may be revealed as flawed, even if it has been validated by a problem-owning stake-holder and so invalid—problem-owning stake-holders make mistakes too!

It is worth emphasising that we do not preclude communication between those that will perform the role of problem- or solution-owning stake-holder, or problem solver during the process of problem solving. Indeed, this would be a very sensible option—even if just to manage the expectations of the various stake-holders before the formal validation is conducted.

Note that the step validity concern associated with a problem exploration step is dischargeable only with reference to the problem-owning stake-holder. Similarly, the step validity concern associated with a solution exploration step is dischargeable only with reference to the solution-owning stake-holder. It is the discharge of step validity concerns that require the problem solver to consult with stake-holders (although, of course, consultation with stake-holders may also take place in problem and/or solution exploration).

On the other hand, like other concerns, the discharge of step validity concerns may be postponed. Depending on the criticality of a development, the risk exposed by such a postponement may be unacceptable—given that a problem-
or solution-owning stake-holder has not validated a partial problem or solution candidate, the problem solver may be solving the wrong problem with incorrect solution technologies, or both. In this case, the future development is based on an assumption of validity. The commitment of developmental resources on this assumption is the source of the risk, although it may be more or less mitigated by problem solver experience. Of course, even if the risk is managed by discharging the step validity concern, there may be secondary risks, such as the a problem-owning stake-holder being incorrect in their validation. It may therefore be important, as part of the justification for the development step to record the explicit instance of step validity concern discharge so that it is traceable; the recording of concern discharges are properly a part of all POE steps.

Finally, it is worth noticing that POE problem exploration can be regarded as a process of capturing knowledge, aimed at producing a model which encapsulates knowledge of the real world context, and of stakeholders’ needs. Such a model is then used to synthesise an appropriate solution. Solution exploration can also be regarded as capturing knowledge, as solutions are not always the outcome of radical design [Vin90], but are often the result of reusing and adapting tried and tested knowledge: in such cases, uncovering what already exists is an essential part of solution exploration.

3 POElog

In this section we provide an overview of POElog, the Prolog-based engine for POE that we have developed. As POElog is based on Prolog, we start with a very brief introduction to logic programming. Prolog being the first and probably most famous logic programming language. Naturally, a comprehensive explanation of Prolog is outside the scope of this short paper—one can be found in [SS87]. Instead, here we recall some basic concepts which will be used in the definition of the POElog system.

3.1 Logic programming

At the heart of logic programming [Kow88] is the idea of using logic both to represent knowledge declaratively and to solve problems operationally through deduction of logical consequence. For instance, Prolog makes use of declarative formulae of the like

\[ if \ p \ and \ q \ and \ \ldots \ \text{and} \ t \ \text{then} \ u \]

which are interpreted operationally as procedures

\[ to \ solve \ u, \ solve \ p \ and \ q \ and \ \ldots \ \text{and} \ t \]

Such formulae are known as Horn clauses and are usually represented in Prolog as

\[ u :- p, q, t. \]

In general, like \( u \), the definition of \( p, q, \ldots \) is also through Horn clauses, hence the solution process under Prolog is highly recursive. \( p, q, \text{etc.} \) can be facts, like \( male(jon) \) (which states that jon is male), or predicates, like \( father(jon, gabri) \), which states that jon is gabri’s father.

Deduction in Prolog is based on three basic rules:

- **Identity**, which allows one to deduce, say, \( male(jon) \) from \( male(jon) \).
- **Generalisation**, which allows one to deduce, say, there exist \( X \) such that \( father(jon, X) \) from \( father(jon, gabri) \), where \( X \) is a variable.
- **Instantiation**, which allows one to deduce, say, \( likes(jon, holidays) \) from \( \text{for all} \ X, \text{likes}(X, holidays) \), where \( X \) is a variable.

Prolog computes deductions thorough substitution and unification. **Substitution** assigns values to variable, for instance, \( \{ X = \text{gabri} \} \) is a substitution which turns \( father(jon, X) \) into \( father(jon, \text{gabri}) \). **Unification** finds the most general substitution which makes two terms identical, should such a substitution exists; for instance, terms \( father(Y, X) \) and \( father(jon, gabri) \) can be unified by substitution \( \{ Y = \text{jon}, X = \text{gabri} \} \).

Here is a small illustrative example. Assume we have the following very simple Prolog program:

\[
\text{father(paul, ann)}.
\text{father(paul, tom)}.
\text{grandfather(X, lucy)} :- \text{father(X, Y)}, \text{father(Y, Z)}.
\]

where \( father \) is defined through facts (\( paul \ is ann’s father, etc.), while \( grandfather \) through the rule \( X \ is \ grandfather \ of \ Z \ when \ X \ is \ father \ of \ Y \ and \ Y \ is \ father \ of \ Z \).

Based on such program, let us try and determine who lucy’s grandfather is by using the **query** (i.e., a possible input to the Prolog program) \( \text{grandfather(X, lucy)} \).

The Prolog system will behave as follows:

- unification will find substitution \( \{ Z = lucy \} \) which unifies \( grandfather(X, lucy) \ and \ grandfather(X, Z) \leftarrow father(X, Y), father(Y, Z) \);

- then unification will find substitution \( \{ Y = tom \} \) which unifies \( father(X, lucy) \ and \ father(tom, lucy) \), following from previous substitution \( \{ Z = lucy \} \);

- finally unification will find substitution \( \{ X = paul \} \) which unifies \( father(X, tom) \ and \ father(paul, tom) \), following from previous substitutions \( \{ Z = lucy, Y = tom \} \).

With both \( father(paul, tom) \) and \( father(tom, lucy) \) successful, \( grandfather(paul, lucy) \) will also succeed, with the outcome of the query being that paul is lucy’s grandfather.

\[
\text{u :- p, q, t}.
\]
3.2 Prolog encoding of POE

As explained in Section 2, the Gentzen-style formulation of POE is characterised by rules in which only one conclusion problem \((W, S \vdash R, \text{below the line})\) follows from its premise problems \((W_i, S_i \vdash R_i, \text{above the line})\). As a consequence, by applying POE rules, development trees are created with the \textit{null problem} as root and solved (sub-)problems as leaves.

The Prolog encoding of POE is then an engine that generates POE development trees from specific rule applications. Trees, being recursive data structures, are a very good fit for Prolog: unification works through recursion, and relatively simple yet powerful Prolog programs can be defined through recursive clauses. For POE trees, however, there is a complication in that rule application is subject to justification, hence is not always successful. Moreover, recording failure is an important practice of engineering. Therefore, the POE engine needs both to check the success of each rule application and to record any unsuccessful outcomes for future reference. This is achieved by generating not just the successful development tree, but also a forest of all backtracked (sub-)trees.

The POElog predicate \texttt{slove(Problem, DevTree)} is at the heart of such a computation: given a collection of development steps, it constructs the resulting successful development tree (if one exists), together with its collection of backtracked (sub-)trees; in doing so, it relies on predicates defining rationale, concerns, their discharge and associated risks. At the start of the computation, it is ‘invoked’ with variable \texttt{Problem} substituted with the \textit{null problem}, and \texttt{DevTree} the variable which will eventually contain the successful development tree based on successful development steps taken. It then proceeds by identifying the transformation step with the \textit{null – problem} as conclusion, checking all related concerns, and, if the step is successful, proceeds recursively taking each premise problem as conclusion for the next step. An unsatisfiable concern causes backtracking of the development tree to the conclusion problem of the step in which the concern first arises, while at the same time the backtracked sub-tree is kept as a separate record. The POElog predicate \texttt{step(StepId, RuleName, Problem, Premises)} captures development steps, with predicate \texttt{problem(Domains, Solution, Requirement)} used to capture premise and conclusion problems. Rationale, concerns and risks are unified with steps by step identifiers.

The current POElog prototype is aimed at developing and testing engine capability, rather than ease of use. Therefore rather than developing a bespoke user interface, we make use of a LaTeX system for input and output: development steps are encoded as LaTeX source, whose processing adds to the POElog engine’s knowledge base prior to the engine invocation. The engine output is also as LaTeX which is then compiled to pdf.

In the next section we will illustrate the use of POElog in the POE development of a real-world avionics system.

4 Example

The example is based on the development of systems flying in real military aircraft, specifically the \textit{Decoy Controller} component of a defensive aids system whose role is to control the release of decoy flares providing defence against incoming missile attack. Aspects of the problem were presented in [MHR07a, HMR07], where safety process and assurance issues were considered. In this paper we provide a complete POE development from system requirements to specification and early design. The development follows the POE process of Section 2.2. POElog encoding of development steps will be discussed at relevant points, where particular aspects of the encoding are deemed worth of discussion.

4.1 Initial problem exploration

The result of our initial problem exploration is the POE problem illustrated in Figure 3.

![Figure 3. The Decoy Controller Problem](image)

Formally in POE:

<table>
<thead>
<tr>
<th>STEP ID: Application of Problem Exploration as a starting point</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUSTIFICATION J1: An initial characterisation of the problem is:</td>
</tr>
<tr>
<td>\texttt{Defence System}^{init.}, Dispenser Unit^{fire, sel},</td>
</tr>
<tr>
<td>\texttt{P} \texttt{1} : Aircraft Status System^{init.}, Pilot^{ok},</td>
</tr>
<tr>
<td>\texttt{Decoy Controller^{con, out, air, ok} \vdash SDC^{con, out, air, ok}}</td>
</tr>
<tr>
<td>DESCRIPTIONS &amp; PHENOMENA: Here are the initial context and requirement descriptions:</td>
</tr>
</tbody>
</table>
**Defence System Dispenser Unit**

The computer responsible for controlling and orchestrating all defensive aids on the aircraft. Electro-mechanical device for releasing decoy flares used as defence against incoming missile attack. It has number of different flare types, and includes a safety pin that, when in place, prevents flares from being released.

The system which monitors the status of certain key aircraft parameters, including whether the aircraft is in the air.

The pilot, who can signal the controller that flare release should be allowed.

The solution to be designed

The requirement (Safe Decoy Control) is the conjunction of:

\( R_1: \) On receiving a con command from Defence System, Decoy Controller shall obtain the selected flare type information from the relevant field in con, for use in its sel message to the Dispenser Unit to control flare selection.

\( R_2: \) Decoy Controller shall issue a fire command only on receiving a con command, with fire instructions in the appropriate field, from Defence System. This shall be the only way in which a flare can be released.

\( R_3: \) Decoy Controller shall cause a flare to be released by issuing a fire command to the Dispenser Unit, which will fire the selected flare.

\( R_4: \) Decoy Controller shall only issue a fire command if its interlocks are satisfied, i.e. aircraft is in air (\( air = yes \)), safety pin has been removed (\( out = yes \)) and pilot has issued an allow a release command (\( ok = yes \)).

\( R_5: \) Decoy Controller shall mitigate \( H_1 \) and \( H_2 \), where \( H_1 \) is the inadvertent firing of decoy flare on ground. Safety Target: safety critical, \( 10^{-7} \) fph (where fph is ‘failures per flight hour’); and \( H_2 \) is the inadvertent firing of decoy flare in air. Safety Target: safety critical, \( 10^{-7} \) fph.

**Aircraft Status System Pilot Decoy Controller SDC**

and here are their phenomena:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fire</td>
<td>Command to release the selected flare type</td>
</tr>
<tr>
<td>sel</td>
<td>Command to select flare type</td>
</tr>
<tr>
<td>out</td>
<td>Pin status: ( out = yes ) when pin removed</td>
</tr>
<tr>
<td>con</td>
<td>Command to select and release a flare type</td>
</tr>
<tr>
<td>air</td>
<td>Aircraft status: ( air = yes ) when aircraft airborne</td>
</tr>
<tr>
<td>ok</td>
<td>Pilot intention: ( ok = yes ) then allow release</td>
</tr>
</tbody>
</table>

**Concern:** Validity  
**Status:** Pending  
**Claim:** This is a valid initial characterisation of the problem.

**4.1 Problem validation**

All descriptions and arguments were validated by problem-owning stake-holders, discharging all concerns, hence the step validity concern.
4.1.2 POElog encoding

The following Prolog code is the POElog encoding of this step:

```
step('id0',
  'PROB-EXP',
  null-problem,
  ['P1']).
```

together with the rationale:

```
rationale('id0', 'Establishing the initial problem description').
```

which generates problem P1 for the null problem—the start of all POE developments. Step definition requires a predicate defining P1:

```
problem('P1',
  ['$Defence\ System$', % domains
   '$Dispenser\ Unit$',
   '$Aircraft\ Status\ System$',
   '$Pilot$',
   '$Decoy\ Controller$', % solution
   '$SDC$'], % requirement
  % as well as predicates for concerns, domain, solution and requirement definitions, like the following:

  concern('id0', 'c0-01-Validity', 'Discharged',
          'The choice of domains follows ...').

domain(Problem, '$Defence\ System$',
       'The computer responsible for controlling and orchestrating all defensive aid systems on the aircraft.'),

solution(Problem, '$Decoy\ Controller$',
         'The solution to be designed.'),

requirement(Problem, '$SDC$',
            'The requirement ...').
```

Unification on problem and step identifiers allows POElog to keep track of and relate steps, problems and their parts. And backslash signs in predicates are used to take advantage of LaTeX typesetting capabilities in producing the output.

4.2 Solution Exploration for Decoy Controller

The engineers came up with an architecture for Decoy Controller consisting of three components as illustrated in Figure 4: one extant component, Interlock Input, and two to-be-designed components, Decoy μProcessor and Safety Controller. On arc annotations, the underscore _ is used to indicate that a context domain controls phenomena observed by a component, e.g. _! con observed by Decoy μProcessor.

![Decoy Controller Architecture](image)

**Figure 4. Decoy Controller Architecture**

As part of solution exploration we need to support the claim that the chosen architecture candidate should not prevent an adequately safe solution from being found. In the worst case, to continue the design without checking feasibility uncovers the risk that the final product cannot be argued safe. Traditionally, such risks are mitigated through over-engineering of the solution, but this typically adds to the development cost, hence trade-offs are normal part of development and risk management.

Formally in POE:

```
STEP 1D: Application of SOLUTION EXPLORATION to P1 to identify a candidate solution architecture

JUSTIFICATION J2: We choose the following architecture for Decoy Controller:

DecoyContArch : [Interlock Input\_! ok, air, out]
                  (Decoy μProcessor\_! con, fire, _? sel, _re?, fire?, _air, _out)

in which Interlock Input is a known component, while Decoy μProcessor and Safety Controller will be subject of further design. This leads to the following sub-problems (each addressing one of the to-be-designed components):

```
Defence System\_m, Dispenser Unit\_m, SDC
P_a : Interlock Input\_m, air, out, Safety Controller_{\_? sel, fire?}
     Decoy μProcessor_{\_! con, fire?} ⊢ SDC_{\_? sel, fire?, air, out}
```

```
Defence System\_m, Dispenser Unit\_m, SDC
P_a : Interlock Input\_m, air, out, Safety Controller_{\_? sel, fire?}
     Decoy μProcessor_{\_! con, fire?} ⊢ SDC_{\_? sel, fire?, air, out}
```

DESCRIPTIONS & PHENOMENA: Here are the new descriptions:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoy Controller</td>
<td>DecoyContArch : [Interlock Input:&lt;int,ok&gt;,air,out] (DecoyµProcessor:&lt;sel,fire?&gt;, Safety Controller:&lt;con&gt;)</td>
</tr>
<tr>
<td>Decoy µProcessor</td>
<td>Component to be designed.</td>
</tr>
<tr>
<td>Interlock Input</td>
<td>Off-the-shelf component which collects together the interlock inputs and passes their status to Safety Controller (int)</td>
</tr>
<tr>
<td>Safety Controller</td>
<td>Component to be designed.</td>
</tr>
</tbody>
</table>

and related new phenomena:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fire</td>
<td>Command to release the selected flare type</td>
</tr>
<tr>
<td>int</td>
<td>Status of combined interlocks</td>
</tr>
</tbody>
</table>

**CONCERN:** Feasibility  
**STATUS:** Pending  
**CLAIM:** This is a feasible solution architecture for the Decoy Controller.  
**ARGUMENT & EVIDENCE:** The architecture is chosen to minimise the number and extent of the safety related functions, localising them to simple, distinct blocks in accordance with best practice. Specifically, the Safety Controller is intended as a simple component which handles all safety-critical elements of the interlocking.

**CONCERN:** Step Validity  
**STATUS:** Pending  
Solution-owning stake-holders’ validation of architecture and argument is required.

For completeness, we should have also included a re-expression of the requirement in terms of the chosen architectural components of Decoy Controller. For brevity, we omit what is just a simple lexical replacement within the requirement statement.

### 4.2.1 Solution validation

The chosen architecture was validated by solution-owning stake-holders, discharging the feasibility (hence, step) concern up to this point.

The step has left us with a co-design problem in that the design of each component will influence that of the other. The POE development so far is summarised in Figure 5, where: labelled circles represent problems; arrows, their transformation; their labels, step justifications; and ticked labels, validated justifications.

We consider the design of Decoy µProcessor first, assuming Safety Controller as given in this sub-problem.

![Figure 5. POE development after first 2 steps of development](image)

#### 4.2.2 POElog encoding

The introduction of a candidate architecture and the generation of related sub-problems is achieved in POElog as the combination of two steps: solution interpretation, which introduces the architecture, and solution expansion, which generates a sub-problem for each to-be-design component. This is because, POElog follows the POE Gentzen-style formal system in which corresponding transformation schema are defined. In other words, solution exploration is a composite problem transformations, defined in terms of more basic POE transformations. Here are the two steps for the above solution exploration, in which an intermediate problem P1A is generated:

```plaintext
step('id1A', 'SOL-EXP', ['P1'], ['P1A']).
problem('P1A', ['$Defence\ System$', % domains  
'\$Dispenser\ Unit$',  
'\$Aircraft\ Status\ System$',  
'\$Pilot\$'],  
'\$Decoy\ Controller\$’, % solution  
'\$SDC\$’). % requirement
solution(Problem, '$\Decoy\ Controller\$',  
'\$DecoyContArch:\[\$Interlock\ Input\$\]  
(\$Decoy\ \mu Processor, \$\Safety\ Controller\$))  
:- member(Problem, ['P1A']).
rationale('id1A', 'Choosing an architecture for $\Decoy\ Controller\$').
```

which chooses a candidate architecture for Decoy Controller, and:

---

We follow the convention of adding an A as a suffix to the name of a problem which has been transformed via solution interpretation, e.g. P1 to P1A.
10

step('id1',
'SUB-GEN',
'P1A',
['P2','P3']).

problem('P2',
['$Defence\ System$', % domains
'$Dispenser\ Unit$',
'$Aircraft\ Status\ System$',
'$Pilot$',
'$Interlock\ Input$',
'$Safety\ Controller$'],
'$Decoy\ \mu Processor$', % solution
'$SDC$'). % requirement

problem('P3',
['$Defence\ System$', % domains
'$Dispenser\ Unit$',
'$Aircraft\ Status\ System$',
'$Pilot$',
'$Interlock\ Input$',
'$Decoy\ \mu Processor$'],
'$Safety\ Controller$', % solution
'$SDC$'). % requirement

rationale('id1', 'Automatically generating
sub-problems from the chosen architecture').

which automatically generates the corresponding sub-problems.

4.3 Problem exploration to simplify the Decoy \mu Processor problem

By identify architectural components, we achieve a separation of the overall required functionality. In this development step we look at problem $P_2$ with a view to simplify it to the essential elements of the design problem for Decoy \mu Processor, one of the architecture’s components. The end point of this step is the simplified problem illustrated in Figure 6: only relevant parts of the context remain and the requirement is re-expressed accordingly.

Figure 6. Simplified Decoy \mu Processor sub-problem

STEP ID: Application of PROBLEM EXPLORATION to $P_2$ for simplification

JUSTIFICATION $J_3$: We simplify $P_2$ to the following problem:

\[
\text{Defence System}^{\text{\textsuperscript{tm}}} \text{ Dispenser Unit}_{\text{set}}, \quad P_4: \text{ Safety Controller}_{\text{set}} \text{ fire? } \rightarrow SDP_{\text{set}}^{\text{fire?}}
\]

**DESCRIPTIONS & PHENOMENA:** Here is the new requirement:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety Controller</td>
<td>To be designed to issue, safely and reliably, fire on receiving fire? with all interlocks satisfied.</td>
</tr>
<tr>
<td>SDP</td>
<td>The requirement (Safe Decoy Processing) is the conjunction of:</td>
</tr>
<tr>
<td></td>
<td>$R_a$: On receiving a con command from Defence System, Decoy \mu Processor shall obtain the selected flare type information from the relevant field in con, for use in its sel message to the Dispenser Unit to control flare selection.</td>
</tr>
<tr>
<td></td>
<td>$R_b$: Decoy \mu Processor shall issue a fire? command only on receiving a con command from Defence System. Assuming Safety Controller issues fire on receiving fire? with all interlocks satisfied, this shall be the only way in which a flare can be released.</td>
</tr>
</tbody>
</table>

**CONCERN:** Sound progression

**STATUS:** Pending

**CLAIM:** This is a sound simplification of the problem.

**ARGUMENT & EVIDENCE:** The problem is simplified based on the architectural separations afforded by the chosen architecture, particularly, the assumption that Safety Controller, yet to be designed, will safely and reliably issue fire on receiving fire? with all interlocks satisfied. This is a constraint on the design of Safety Controller.

**CONCERN:** Step Validity

**STATUS:** Pending

Stake-holders’ validation of argument is required.

4.3.1 Problem validation

Problem simplification was based on POE rules for progression [HRJ08, RHL06], which are systematic and always applicable. The validation of this step has mainly to do with validating the assumption, added to the requirement statement, that constrains the design of Safety Controller. Such an assumption was accepted as reasonable, hence the simplified problem can be used as the basis for further development.
4.4 Solution exploration for Decoy µProcessor

The chosen architecture for Decoy µProcessor consists of two components as illustrated in Figure 7: Message Buffer, which holds the received control message con and makes it available to μP (as con1); micro-controller μP, which decodes it to extract: a) a fire command request (leading to fire?), and b) the selected flare type (leading to sel).

Figure 7. Decoy µProcessor Architecture

In order to claim feasibility of this architecture and mitigate related development risks, we applied a Preliminary Safety Analysis (PSA), whose goals were to: (a) confirm the relevance of hazards allocated by the system level hazard analysis; (b) identify any further hazards to be added to the list; and (c) validate the architecture against the safety targets associated with the identified relevant hazards. Many techniques can be applied to perform a PSA. In [MHR07a] we used a combination of mathematical proof, Functional Failure Analysis (FFA) [SAE96] and functional Fault Tree Analysis (FTA) [VGRH81]. Note that PSA is not a POE per se (no POE schema defines a PSA). Instead it is a technique which we use to discharge one of the concerns in the justification obligation for SOLUTION EXPLORATION. As we shall see, PSA in this case shows that the chosen architecture is unfeasible.

Formally in POE:

\[
\text{DecoyµPArch : } [\text{Message Buffer}]_{\text{con1}} \vdash \text{SDP}_{\text{con1}}
\]

in which Message Buffer is a known component and μP is to be designed. This leads to the following problem:

\[
P_5 : \text{Safety Controller}_{\text{con1}} \rightarrow \text{Message Buffer}_{\text{con1}} \mid \muP_{\text{con1}} \dashv \text{SDP}_{\text{con1}}
\]

DESCRIPTIONS & PHENOMENA: Here are the new descriptions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoy µPArch</td>
<td>[Message Buffer]_{\text{con1}}</td>
</tr>
<tr>
<td>μP</td>
<td>To be designed component which decodes con1 to extract: a) a fire command request (leading to fire?), and b) the selected flare type (leading to sel)</td>
</tr>
<tr>
<td>Message Buffer</td>
<td>Off-the-shelf component which holds the received control message con and makes it available to μP as con1</td>
</tr>
</tbody>
</table>

Table 1. FFA Summary for Safety Controller

<table>
<thead>
<tr>
<th>Id</th>
<th>Failure Md</th>
<th>Effect</th>
<th>Haz</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>No fire?</td>
<td>Release inhibited</td>
<td>No</td>
</tr>
<tr>
<td>F2</td>
<td>fire? at wrong time</td>
<td>Inadvertent release</td>
<td>Yes</td>
</tr>
<tr>
<td>F3</td>
<td>fire? when not required</td>
<td>Inadvertent release</td>
<td>Yes</td>
</tr>
<tr>
<td>F4</td>
<td>Intermittent fire?</td>
<td>Could inhibit release</td>
<td>No</td>
</tr>
<tr>
<td>F5</td>
<td>Continuous fire?</td>
<td>Inadvertent release</td>
<td>Yes</td>
</tr>
</tbody>
</table>

A functional FTA applied to Decoy µProcessor and using the three FFA problem cases F2, F3 and F5, indicates that a failure in μP (systematic or probabilistic) could result in the fire! failing on. The Pilot’s allow input provides some mitigation, but as soon as this is set (ok = yes) a flare will be released, which is undesirable behaviour. In other words, with

\footnote{There is insufficient space to present the full PSA, and so we summarise only its main elements to demonstrate the process followed.}
this architecture, $H_2$ is only protected by the Pilot’s allow input. If fire? failed on, then as soon as the Pilot indicated an intention to allow flare release, by selecting the switch, then the flare would be released, which is not the design intention. Therefore the safety analysis indicates that fire? needs to have safety integrity, although it only needs to be safety involved and not safety critical. This can only be achieved with the existing design by making all components safety involved. That is, by assigning fire? to the $\mu P$, we require that all $\mu P$ functionality must be of fire?’s required safety integrity, including much of the $\mu P$’s functionality (timing, BIT, etc.) that is not safety-related. Further, any updates to the $\mu P$ software will have to satisfy the safety involved integrity.

**Concern:** Step Validity  
**Status:** Pending  
Solution-owning stake-holders’ validation of architecture and argument is required.

### 4.4.1 Solution validation

The involvement of solution-owning stake-holders led to a decision that making $\mu P$ safety-involved would be too expensive, hence the architecture was rejected as a suitable basis for further design, which caused the development to backtrack.

### 4.4.2 POElog encoding

Backtracking in POE occurs as a result of a concern becoming un-dischargeable. The following POElog predicate is used in this case:

\[
\text{undischargeable(‘c3A-01-Feasibility’), } \quad \text{‘The involvement of solution-owning stake-holders led to a decision ...’).}
\]

Such a predicate causes POElog to backtrack the development to the problem just before the transformation was applied in which the concern was first introduced, in this case $P_4$.

### 4.5 New solution exploration for Decoy $\mu$Processor

A revised architecture was chosen for Decoy $\mu$Processor consisting of three components as illustrated in Figure 8: buffer Message Buffer2 holds the received control message $con$ and makes it available both to $\mu P2$ and FPGA, as $con_1$ and $con_2$, respectively; micro-processor $\mu P2$ decodes $con_1$ to extract the selected flare type (leading to $sel$); field-programmable gate array FPGA decodes $con_2$ to extract a fire command request (leading to fire?).

The formal step is similar to the previous one:

**Figure 8. New Decoy $\mu$Processor Architecture**

**Step ID:** Application of Solution Exploration to $P_4$ to identify a new candidate solution architecture

**Justification J5:** We choose the following architecture for Decoy $\mu$Processor:

\[
\text{DecoysPArch2 : } |\text{Message Buffer}^\text{con}_1,\text{con}_2| ((\mu P^\text{con}_2, \text{FPGA}^\text{fire?})
\]

in which Message Buffer2 is a known component and $\mu P$ and FPGA are to be designed. This leads to the following sub-problems:

- **Defence System**\text{\textsuperscript{con}}, Dispenser Units\text{\textsuperscript{con}}, Safety Controller\text{\textsuperscript{fire?}},  
  \[  
P_6 : \quad \text{Message Buffer}^\text{con}_1,\text{con}_2, (\mu P^\text{con}_2, \text{FPGA}^\text{fire?}) \vdash \text{SDP}^\text{con}_1 \]

- **Defence System**\text{\textsuperscript{con}}, Dispenser Units\text{\textsuperscript{con}}, Safety Controller\text{\textsuperscript{fire?}},  
  \[  
P_7 : \quad \text{Message Buffer}^\text{con}_1,\text{con}_2, (\mu P^\text{con}_2, \text{FPGA}^\text{fire?}) \vdash \text{SDP}^\text{con}_2 \]

**Descriptions & Phenomena:** Here are the new descriptions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecoysPArch2</td>
<td>$</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>con1</td>
<td>con in format suitable for µP2</td>
</tr>
<tr>
<td>con2</td>
<td>con in format suitable for FPGA</td>
</tr>
</tbody>
</table>

CONCERN: Feasibility
STATUS: Pending

CLAIM: The chosen architecture for the Decoy µProcessor should not prevent an adequately safe solution.

ARGUMENT & EVIDENCE: The chosen architecture is similar to the previously chosen one (see J3), except that we take the safety involved functions out of µP2 and route them through a separate high integrity path. More precisely, the simple safety functions (those associated with the fire? request) are routed through Message Buffer2 and FPGA, while the more complex functions (that associated with the sel request) are routed through Message Buffer2 and µP2. This means that only Message Buffer2 and FPGA, which have simple functionality, have to be designed to a safety related standard, which is more economical than making µP safety related, as required by the previously chosen architecture. By applying PSA, as explained in J3, we could demonstrate the required systematic and failure rate integrity, leading to the result that the revised architecture is still safe yet more economical.

CONCERN: Step Validity
STATUS: Pending

Solution-owning stake-holders’ validation of architecture and argument is required.

4.5.1 Solution validation

The involvement of solution-owning stake-holders led to accepting this architecture as the basis for further design.

4.6 Solution exploration for FPGA

We have reached a point in which enough is known of FPGA for a precise specification to be given.

Formally, in POE:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HEADER</td>
</tr>
<tr>
<td>2</td>
<td>FIRE</td>
</tr>
<tr>
<td>2</td>
<td>SEL</td>
</tr>
<tr>
<td>3,4,5</td>
<td>OTHER</td>
</tr>
<tr>
<td>6</td>
<td>ALT. FIRE</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>P</td>
</tr>
</tbody>
</table>

The header, fire, alt. fire and P are transferred to FPGA as con2. FPGA uses HEADER to extract the FIRE, ALT. FIRE commands (each 4 bits) and the parity bit P. These are decoded and checked for validity. If an anomaly is detected then FPGA outputs the ¬fire? message. Otherwise it outputs the decoded FIRE/ALT. FIRE command, as either fire? or ¬fire?.

CONCERN: Feasibility
STATUS: Pending

CLAIM: The chosen specification for FPGA should not prevent an adequately safe solution.

ARGUMENT & EVIDENCE: An issue which will influence how the FPGA specification is implemented is the buffer’s behaviour. At this point the timing or how the serial con message is read into the buffer is not specified. The buffer may take in one byte at a time, or may be able to store a complete 6 byte message. The former will mean more decoding and storing work for FPGA.

CONCERN: Step Validity
STATUS: Pending

Solution-owning stake-holders’ validation of specification and argument is required.

4.6.1 Solution validation

The involvement of solution-owning stake-holders led to accepting this specification as the basis for detailed design.

4.6.2 POElog encoding

In POElog, solving a problem is accomplished through a combination of solution interpretation followed by expansion (see also Section 4.2.2. More precisely, it is equivalent to introducing a candidate architecture in which all components are known and then generating an empty list of sub-problems (as no component to be designed has remained). This is a further example of how elementary POE transformations combine into larger development steps. The two steps are:

```
step('id5A',
   'SOL-EXP',
   'P6',
   ['P6A']).
```

problem('P6A',
   ['$Defence\ System$', % domains
    '$Dispenser\ Unit$',
    '$Safety\ Controller$',
    '$Message\ Buffer2$',
    '$\mu P2$'],
   13)
to introduce the specification, followed by:

step('id5',
  'SOL-GEN',
  'P6A',
  []).

which automatically generates an empty list of sub-problems.

4.7 Solution exploration for $\mu P2$

Similarly to FPGA, we can provide specification for $\mu P2$. Formally, in POE:

<table>
<thead>
<tr>
<th>STEP ID</th>
<th>Application of Solution Exploration to $P_7$ to provide a candidate solution specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUSTIFICATION</td>
<td>We choose the following specification for $\mu P2$:</td>
</tr>
<tr>
<td></td>
<td>The incoming $con$ message consists of 6 bytes as explained in $J_6$. $HEADER$ and $SEL$ are transferred to $\mu P2$ as $con1$. $\mu P2$ uses $HEADER$ to extract the $SEL$ command, which is then decoded and sent to the $Dispenser Unit$ as $sel$.</td>
</tr>
<tr>
<td>CONCERN:</td>
<td>Feasibility</td>
</tr>
<tr>
<td>STATUS:</td>
<td>Pending</td>
</tr>
<tr>
<td>CLAIM:</td>
<td>The chosen specification for $\mu P2$ should not prevent an adequately safe solution.</td>
</tr>
<tr>
<td>ARGUMENT &amp; EVIDENCE:</td>
<td>As for FPGA, the buffer's behaviour will influence the implementation of this specification, resulting in more or less decoding and storing work for $\mu P2$.</td>
</tr>
<tr>
<td>CONCERN:</td>
<td>Step Validity</td>
</tr>
<tr>
<td>STATUS:</td>
<td>Pending</td>
</tr>
<tr>
<td>Solution-owning stake-holders' validation of specification and argument is required.</td>
<td></td>
</tr>
</tbody>
</table>

4.7.1 Solution validation

The involvement of solution-owning stake-holders led to accepting this specification as the basis for detailed design. We have reached a point in which enough is known of $\mu P2$ and FPGA for detail design to start, as each component implements a simple conversion function which is sufficiently specified. Therefore, we consider these two problems as solved, as least as far as the example is concerned.

It is worth summarising the POE development so far, as shown in Figure 9. Note how the backtracked sub-tree is recorded by the side.

4.8 Problem exploration to simplify the Safety Controller problem

All architectural components other than the Safety Controller are now known as their interfaces and specifications have been worked out. In this step, we aim at simplifying the Safety Controller problem $P_3$ in order to facilitate its specification definition. The end point of this step is the simplified problem illustrated in Figure 6: only relevant parts of the context remain and the requirement is re-expressed accordingly.

<table>
<thead>
<tr>
<th>STEP ID</th>
<th>Application of Problem Exploration to $P_3$ for simplification</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUSTIFICATION</td>
<td>We simplify $P_3$ to the following problem:</td>
</tr>
<tr>
<td>$P_8 :$</td>
<td>Dispenser Unit$_{fire}$</td>
</tr>
<tr>
<td>$Safety Controller_{int, fire} \vdash SC_{fire}$</td>
<td></td>
</tr>
</tbody>
</table>
The validation of this step has to do with validating the assumptions, added to the requirement statement, that constrains the design of FPGA, Message Buffer and Interlock Input. Such assumptions were accepted as compatible with the specification and/or domain properties of those domains, hence the simplified problem can be used as the basis for further development.

### 4.9 Solution exploration for Safety Controller

We can provide a Parnas-like table [PM95] specifying the Safety Controller. Formally, in POE:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| SC   | The requirement (Safe Control) is the conjunction of:  
  \( R_b: \) Safety Controller shall issue a fire command only on receiving a fire? command from Decoy \( \mu \)Processor. Assuming Decoy \( \mu \)Processor reliably decodes fire? from con, this shall be the only way in which a flare can be released.  
  \( R_c: \) Safety Controller shall cause a flare to be released by issuing a fire command to the Dispenser Unit, which will fire the selected flare.  
  \( R_d: \) Safety Controller shall only issue fire if its interlocks in \( \text{int} \) are satisfied (i.e. aircraft is in \( \text{air} \)— \( \text{air} = \text{yes} \), safety pin has been removed—\( \text{out} = \text{yes} \), and pilot has issued an allow a release command—\( \text{ok} = \text{yes} \)), assuming Interlock Input accurately calculates \( \text{int} \) from \( \text{air} \), \( \text{ok} \) and \( \text{out} \). |

### 4.9.1 Solution validation

The involvement of solution-owning stake-holders led to accepting this specification as the basis for detailed design. Therefore, we can consider this problem too as solved, which completes the development of the example. The whole POE development is shown in Figure 11.
all the intermediate problems generated by the finer granularity of some of the steps. The output includes both development and backtracked trees, together with all related descriptions and justifications.

5 Discussion

The example we have presented is part of a growing body of work on applying POE to real-world engineering design problems [HMR07, MHR08, MHR07c, MHR07b, MHR07a]. The work already carried out has provided an initial positive validation of the merits of POE in its application to complex safety engineering problems. It has helped us shape and fine-tune the framework, as well as provide some foci for further development.

One such focus was tool support, which has led to the work reported in this paper. Through the example, we have illustrated the main features of POElog, our current engine for POE development. POElog is meant as the core of a toolset to be developed, which will support engineering designers wishing to apply POE in their work.

With POElog we wanted to ensure a sound basis for such a toolset, embodying POE’s theoretical basis and philosophy. This explains our attention to the computational engine, rather than the finesse of its user interface, considered rather premature at this point.

Prolog was chosen among the numerous programming paradigms and languages available today as the most sympathetic to our purpose. There are many similarities between POE and Prolog. They both have a logical basis, Prolog as Horn clauses and POE as Gentzen sequents. Prolog computations can be represented as trees, and so can POE developments. Falsification of clauses triggers backtracking in Prolog computations, and so does the inability to discharge justification obligations in POE. Prolog variables are not references to store locations in memory, rather they are bound to single entities; hence, whenever the same variable appears in different clauses in a Prolog tree, it always refers to the same entity. Similarly a POE domain name stands for the same part of the context or solution no matter in which development sub-problem it appears, and changes in one sub-problem are reflected across the tree.

Therefore, the operational semantics of Prolog, embodied in the Prolog interpreter and based on substitution and unification, provides much of the machinery we needed to generate, backtrack and keep consistent POE development trees. The only departure from standard Prolog semantics is that fact that backtracked trees are not discarded, but put aside as a record of unsuccessful design steps.

It should be noted that POElog is mainly concerned with steps, tree structures and entities which are shared across problems in a development tree. It does not deal with the fine detail of domain descriptions, but only records that they exist and what they are at any particular point in the development. This is because POElog, like POE, sits at a meta-level above that of the particular description languages used. This is where the toolset to be built has a role to play. For instance, for behavioural descriptions of causal context domains, description languages like Statecharts [HN96] or Petri Nets [Rei85] might be appropriate. Similarly, solution specifications could be expressed, say in Z [Spi92] or VDM [Jon86]. Moreover, different types of description might, and generally will, exist within the same development tree. It would be a fruitless exercise to try and reproduce in Prolog the modelling capability of the many existing tools tailored to specific description languages. Instead, interfacing our engine with such tools is a more promising and scalable approach, and the one we have chosen. To this end, we have already initiated the integration of POE with CPN Tools, a Colour Petri Nets toolset [HJR07], as proof-of-concept.

6 Conclusion

This paper has discussed POElog, a Prolog engine for POE. Originally conceived for software engineering, POE is developing into a more general framework for engineering design. The POE notion of problem requires a separation of context, requirement and solution, with explicit descriptions of what is given, what is required and what is
object of design. This improves the traceability of artefacts and their relation, as well as exposing all assumptions to scrutiny and validation. That all descriptions are generated through problem transformation forces the inclusion of an explicit justification that such assumptions are realistic and reasonable.

POElog is a first step towards tool support for POE. It takes advantage of the formal encoding of POE as a Gentzen-style system, which maps quite naturally into Prolog. The main feature of POElog were illustrated on a POE development of a real-world safety engineering problem.

Work on POE is continuing on various fronts. Theoretically, we are trying to gain a better understanding of engineering design processes. Specifically, we have some early evidence that the simple process pattern of Section 2.2 combines in various ways to generate more complex problems. Current combinations that we are exploring are sequence, parallel and fractal compositions. We are also studying the differences between engineering design and other forms of more creative design with a view to understand whether and how creativity may fit within POE. Early observations indicate the need for an explicit reference to the target design community, within which measures of creativity are established.

Tool development remains a priority in order to build on current POElog limited capabilities. Beside the integration with other modeling tools to generate and analyse descriptions, which we discussed earlier on, we are developing a tactic language [OCW03] in order to capture and reply sequences of design steps in different developments—the equivalent of design patterns for development processes.

Finally, application and fine-tuning of POE continues through its application to a variety of problems, from software to other disciplines of design. More information about POE and our ongoing research can be found at www.solvemehappy.com.

Acknowledgments

We acknowledge the financial support of IBM, under the Eclipse Innovation Grants, and of SE Validation Limited, in particular Colin Brain for his many comments and insights. A particular thank you goes to Derek Mannering, at General Dynamics UK, whose work has helped us develop and validate POE for safety engineering. Thanks also go to Jens Jørgensen and Simon Tjell of Aarhus University for much fruitful discussion on how to integrate POElog and CPN Tools, and our colleagues in the Computing Department at The Open University, particularly Michael Jackson.

References


18
Preface

This document was automatically generated from POElog.

1 Preface

This document was automatically generated from POElog.
2 Backtracked developments

The backtracked development is:

\[ \begin{align*}
\text{P5} & \quad [id3] \\
\text{P4A1} & \quad [id3A] \\
\text{P4} & \quad [id3A] 
\end{align*} \]

Step details

<table>
<thead>
<tr>
<th>Step Id</th>
<th>Rule Name</th>
<th>Step Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>id3A</td>
<td>SOL-EXP</td>
<td>Choosing an architecture for Decoy ( \mu \text{Processor} )</td>
</tr>
<tr>
<td>id3</td>
<td>SUB-GEN</td>
<td>Automatically generating sub-problems from the chosen architecture</td>
</tr>
</tbody>
</table>

Development concerns for backtracked development

<table>
<thead>
<tr>
<th>Step Id</th>
<th>Concern Name</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id3A</td>
<td>c3A-01-Feasibility</td>
<td>Pending</td>
<td>The PSA analysis of this architecture indicates that ( \text{fire} )? needs to have a safety involved (not critical) integrity. This can only be achieved with the existing design by making all components safety involved. That is, by assigning ( \text{fire} )? to the ( \mu P ), we require that all ( \mu P ) functionality must be of ( \text{fire} )'s required safety integrity, including much of the ( \mu P )'s functionality (timing, BIT, etc.) that is not safety-related. Further, any updates to the ( \mu P ) software will have to satisfy the safety involved integrity.</td>
</tr>
</tbody>
</table>

Undischargeable development concerns for backtracked development

<table>
<thead>
<tr>
<th>Step Id</th>
<th>Rule Name</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>id3A</td>
<td>c3A-01-Feasibility</td>
<td>The involvement of solution-owning stake-holders led to a decision that making ( \mu P ) safety-involved would be too expensive, hence the architecture was rejected as a suitable basis for further design.</td>
</tr>
</tbody>
</table>

Problem Descriptions

<table>
<thead>
<tr>
<th>Problem</th>
<th>Context ID</th>
<th>Solution</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td>Defence System, Dispenser Unit, Safety Controller</td>
<td>Decoy ( \mu \text{Processor} )</td>
<td>SDP</td>
</tr>
<tr>
<td>P4A1</td>
<td>Defence System, Dispenser Unit, Safety Controller</td>
<td>Decoy ( \mu \text{Processor} )</td>
<td>SDP</td>
</tr>
<tr>
<td>P5</td>
<td>Defence System, Dispenser Unit, Safety Controller, Message Buffer</td>
<td>( \mu P )</td>
<td>SDP</td>
</tr>
</tbody>
</table>

Domain Descriptions

<table>
<thead>
<tr>
<th>Problem</th>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4, P4A1, P5</td>
<td>Defence System</td>
<td>The computer responsible for controlling and orchestrating all defensive aid systems on the aircraft.</td>
</tr>
<tr>
<td>P4, P4A1, P5</td>
<td>Dispenser Unit</td>
<td>Mechanical device for releasing decoy flares used as defence against incoming missile attack. It has number of different flare types, and includes a safety pin that, when in place, prevents flares from being released.</td>
</tr>
</tbody>
</table>

cont’d
Yet to be designed; it will safely and reliably issue fire on receiving fire? with all interlocks satisfied.

Off-the-shelf component which holds the received control message con and makes it available to μP as con1.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
</table>
| P4, P4A1, P5 | SDP | The requirement (Safe Decoy Processing) is the conjunction of:  
  • R_a: On receiving a con command from Defence System, Decoy μProcessor shall obtain the selected flare type information from the relevant field in con, for use in its sel message to the Dispenser Unit to control flare selection.  
  • R_b: Decoy μProcessor shall issue a fire? command only on receiving a con command from Defence System. Assuming Safety Controller issues fire on receiving fire? with all interlocks satisfied, this shall be the only way in which a flare can be released. |

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td>Decoy μProcessor</td>
<td>Component to be designed.</td>
</tr>
<tr>
<td>P4A1</td>
<td>Decoy μProcessor</td>
<td>DecoyμPArch : <a href="%CE%BCP">Message Buffer</a></td>
</tr>
<tr>
<td>P5</td>
<td>μP</td>
<td>To be designed component which decodes con1 to extract: a) a fire command request (leading to fire?), and b) the selected flare type (leading to sel).</td>
</tr>
</tbody>
</table>
3 Development Checkpoint

Step details

<table>
<thead>
<tr>
<th>Step Id</th>
<th>Rule Name</th>
<th>Step Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>id0</td>
<td>PROB-EXP</td>
<td>Establishing the initial problem description</td>
</tr>
<tr>
<td>id1</td>
<td>SUB-GEN</td>
<td>Automatically generating sub-problems from the chosen architecture</td>
</tr>
<tr>
<td>id1A</td>
<td>SOL-EXP</td>
<td>Choosing an architecture for Decoy Controller</td>
</tr>
<tr>
<td>id2</td>
<td>PROB-EXP</td>
<td>Simplifying the Decoy (\mu)Processor problem</td>
</tr>
<tr>
<td>id4</td>
<td>SUB-GEN</td>
<td>Automatically generating sub-problems from the chosen architecture</td>
</tr>
<tr>
<td>id4A</td>
<td>SOL-EXP</td>
<td>Choosing another architecture for Decoy (\mu)Processor</td>
</tr>
<tr>
<td>id5</td>
<td>SOL-GEN</td>
<td>Automatically recording a solved problem</td>
</tr>
<tr>
<td>id5A</td>
<td>SOL-EXP</td>
<td>Choosing a specification for FPGA</td>
</tr>
<tr>
<td>id6</td>
<td>SOL-GEN</td>
<td>Automatically recording a solved problem</td>
</tr>
<tr>
<td>id6A</td>
<td>SOL-EXP</td>
<td>Choosing a specification for (\mu)P</td>
</tr>
<tr>
<td>id7</td>
<td>PROB-EXP</td>
<td>Simplifying the Safety Controller problem</td>
</tr>
<tr>
<td>id8</td>
<td>SOL-GEN</td>
<td>Automatically recording a solved problem</td>
</tr>
<tr>
<td>id8A</td>
<td>SOL-EXP</td>
<td>Choosing a specification for Safety Controller</td>
</tr>
</tbody>
</table>
The choice of domains follows from the aircraft level safety analysis and the required choice of interlocks. The Defence System, Dispenser Unit, Aircraft Status System are existing components of the avionics system, with well-known properties (that could be validated through direct inspection). The Pilot is trained to follow protocol rigorously.

The customer requirement was provided as an input to the developer team. Hazard $H_1$ and $H_2$ came from an aircraft level safety analysis which allocated safety requirements to the main aircraft systems, including the Decoy Controller. Hazards $H_1$ and $H_2$ have both systematic (safety related) and probabilistic components. To counter these hazards, the following safety interlocks were required as input to the Decoy Controller to provide safety protection: an input from the pilot indicating whether the release should be allowed; an input indicating whether the aircraft is in the air; and an input indicating whether the safety pin, present when the aircraft is on the ground, is in place. The expected behaviour is that flare release should be inhibited if any of the following conditions hold: a) the pilot disallows flares; b) the aircraft is not in the air; or c) the safety pin has not been removed. These interlocks provide extra assurance for hazard $H_1$, but not for $H_2$. Therefore, the safety task is to demonstrate that $H_2$ can be satisfied, with the knowledge that if $H_2$ can be satisfied, then so can $H_1$.

The in-air indicator is obtained from the weight on wheels and landing gear up indications: if the landing gear is up and there is no weight on the wheels then the aircraft is assumed to be in the air. The landing gear is detected as being up by a number of sensor switches. The switches use a multi-pole arrangement of appropriately selected “Normally open/Normally closed’ contacts. This imbues an error detection capability that is used to achieve very good failure rates, well within the required margins.

All descriptions and arguments were validated by problem-owning stake-holders, discharging all concerns.

The architecture is chosen to minimise the number and extent of the safety related functions, localising them to simple, distinct blocks in accordance with best practice. Specifically, the Safety Controller is intended as a simple component which handles all safety-critical elements of the interlocking.

The chosen architecture was validated by solution-owning stake-holders, discharging the feasibility (hence, step) concern up to this point.

The problem is simplified based on the architectural separations afforded by the chosen architecture, particularly, the assumption that Safety Controller, yet to be designed, will safely and reliably issue fire on receiving fire? with all interlocks satisfied. This is a constrains on the design of Safety Controller.
<table>
<thead>
<tr>
<th>ID</th>
<th>Step/Feasibility</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id2</td>
<td>c2-02-Step-validity</td>
<td>Discharged</td>
<td>Problem simplification was based on POE rules for progression, which are systematic and always applicable. The validation of this step has to do with validating the assumption, added to the requirement statement, that constrains the design of Safety Controller. Such an assumption was accepted as reasonable, hence the simplified problem can be used as the basis for further development.</td>
</tr>
<tr>
<td>id4A</td>
<td>c4A-01-Feasibility</td>
<td>Discharged</td>
<td>The chosen architecture is similar to the previously chosen one, except that we take the safety involved functions out of $\mu P2$ and route them through a separate high integrity path. More precisely, the simple safety functions (those associated with the <code>re?' request) are routed through Message Buffer2 and FPGA, while the more complex functions (that associated with the </code>sel' request) are routed through Message Buffer2 and $\mu P2$. This means that only Message Buffer2 and FPGA, which have simple functionality, have to be designed to a safety related standard, which is more economical than making $\mu P$ safety related, as required by the previously chosen architecture. By applying PSA, we could demonstrate the required systematic and failure rate integrity, leading to the result that the revised architecture is still safe yet more economical.</td>
</tr>
<tr>
<td>id4A</td>
<td>c4A-02-Step-validity</td>
<td>Discharged</td>
<td>The involvement of solution-owning stake-holders led to accepting this architecture as the basis for further design.</td>
</tr>
<tr>
<td>id5A</td>
<td>c5A-01-Feasibility</td>
<td>Discharged</td>
<td>An issue which will influence how the FPGA specification is implemented is the behaviour of the buffer. At this point the timing or how the serial `con' message is read into the buffer is not specified. The buffer may take in one byte at a time, or may be able to store a complete 6 byte message. The former will mean more decoding and storing work for FPGA.</td>
</tr>
<tr>
<td>id5A</td>
<td>c5A-02-Step-validity</td>
<td>Discharged</td>
<td>The involvement of solution-owning stake-holders led to accepting this specification as the basis for detailed design.</td>
</tr>
<tr>
<td>id6A</td>
<td>c6A-01-Feasibility</td>
<td>Discharged</td>
<td>As for FPGA, buffer behaviour will influence the implementation of this specification, resulting in more or less decoding and storing work for $\mu P2$.</td>
</tr>
<tr>
<td>id6A</td>
<td>c6A-02-Step-validity</td>
<td>Discharged</td>
<td>The involvement of solution-owning stake-holders led to accepting this specification as the basis for detailed design.</td>
</tr>
<tr>
<td>id7</td>
<td>c7-01-Sound-progression</td>
<td>Discharged</td>
<td>The problem is simplified based on the architectural separations afforded by the chosen architecture, particularly, the assumptions that: FPGA reliably decodes `fire?' from con1; Message Buffer reliably decodes con1 from con; and Interlock Input accurately calculates int from air, ok and out. These are constrains on the design of FPGA, Message Buffer and Interlock Input.</td>
</tr>
<tr>
<td>id7</td>
<td>c7-02-Step-validity</td>
<td>Discharged</td>
<td>The validation of this step has to do with validating the assumptions, added to the requirement statement, that constrains the design of FPGA, Message Buffer and Interlock Input. Such assumptions were accepted as compatible with the specification and/or domain properties of those domains, hence the simplified problem can be used as the basis for further development.</td>
</tr>
</tbody>
</table>
Software implementation of this logic specification is straightforward.

The involvement of solution-owning stake-holders led to accepting this specification as the basis for detailed design.
P2  Safety Controller Component to be designed.
P3  Decoy µProcessor Component to be designed.
P4, P4A2, P6, P6A, P7, P7A  Yet to be designed; it will safely and reliably issue fire on receiving fire? with all interlocks satisfied.
P6, P6A, P7, P7A  Off-the-shelf component which holds the received control message and makes it available to µP as con1 and to FPGA as con2.
P6, P6A  To be designed component which decodes con1 to extract the selected flare type (leading to sel).
P7  To be designed component which decodes con2 to extract a fire command request (leading to fire?).
P7A  We choose the following specification for FPGA. The incoming con message consists of 6 bytes as shown in the table below. The first byte is a fixed header to indicate start of message, the remaining bytes contain the FIRE, SEL, OTHER commands, alternate form of fire ALT. FIRE and the parity bit P. X indicates 3 Don't Care bits.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3, 4, 5</th>
<th>6</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADER</td>
<td>FIRE</td>
<td>SEL</td>
<td>OTHER</td>
<td>ALT. FIRE</td>
<td>P</td>
</tr>
</tbody>
</table>

The header, FIRE, ALT. FIRE and P are transferred to FPGA as con2. FPGA uses HEADER to extract the FIRE, ALT. FIRE commands (each 4 bits) and the parity bit P. These are decoded and checked for validity. If an anomaly is detected then FPGA outputs the ¬fire? message. Otherwise it outputs the decoded FIRE/ALT. FIRE command, as either fire? or ¬fire?.

### Requirement Descriptions

<table>
<thead>
<tr>
<th>Problem</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
</table>
| P1, P1A, P2, P3 | SDC | The requirement (Safe Decoy Control) is the conjunction of:
- \( R_a \): On receiving a con command from Defence System, Decoy Controller shall obtain the selected flare type information from the relevant field in con, for use in its sel message to the Dispenser Unit to control flare selection.
- \( R_b \): Decoy Controller shall issue a fire command only on receiving a con command from Defence System. This shall be the only way in which a flare can be released.
- \( R_c \): Decoy Controller shall cause a flare to be released by issuing a fire command to the Dispenser Unit, which will fire the selected flare.
- \( R_d \): Decoy Controller shall only issue a fire command if its interlocks are satisfied, i.e. aircraft is in air (air = yes), safety pin has been removed (out = yes) and pilot has issued an allow a release command (ok = yes).
- \( R_e \): Decoy Controller shall mitigate \( H_1 \) and \( H_2 \), where \( H_1 \) is the inadvertent firing of decoy flare on ground. Safety Target: safety critical, \( 10^{-7} \) fpfh (where fpfh is failures per flight hour); and \( H_2 \) is the inadvertent firing of decoy flare in air. Safety Target: safety critical, \( 10^{-7} \) fpfh. |

cont’d
<table>
<thead>
<tr>
<th>Page Numbers</th>
<th>Requirement (Label)</th>
<th>Description</th>
</tr>
</thead>
</table>
| P4, P4A2, P6, P6A, P7, P7A | SDP | The requirement (*Safe Decoy Processing*) is the conjunction of:
- \( R_a \): On receiving a `con` command from *Defence System*, *Decoy \( \mu \)Processor* shall obtain the selected flare type information from the relevant field in `con`, for use in its `sel` message to the *Dispenser Unit* to control flare selection.
- \( R_b \): *Decoy \( \mu \)Processor* shall issue a `fire?` command only on receiving a `con` command from *Defence System*. *Assuming Safety Controller issues fire on receiving fire? with all interlocks satisfied*, this shall be the only way in which a flare can be released. |
| P8, P8A | SC | The requirement (*Safe Control*) is the conjunction of:
- \( R_b \): *Safety Controller* shall issue a `fire` command only on receiving a `fire?` command from *FPGA*. *Assuming FPGA reliably decodes fire? from con1, and Message Buffer2 reliably decodes con1 from con*, this shall be the only way in which a flare can be released.
- \( R_c \): *Safety Controller* shall cause a flare to be released by issuing a `fire` command to the *Dispenser Unit*, which will fire the selected flare.
- \( R_d \): *Safety Controller* shall only issue `fire` if its interlocks in `int` are satisfied (i.e. aircraft is in air—`air = yes`, safety pin has been removed—`out = yes` and pilot has issued an allow a release command—`ok = yes`), *assuming Interlock Input accurately calculates int from air, ok and out*. |
## Solution Descriptions

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Decoy Controller</td>
<td>The solution to be designed.</td>
</tr>
<tr>
<td>P1A</td>
<td>Decoy Controller</td>
<td>DecoyContArch : [Interlock Input](DecoyµProcessor,Safety Controller)</td>
</tr>
<tr>
<td>P2, P4</td>
<td>Decoy µProcessor</td>
<td>Component to be designed.</td>
</tr>
<tr>
<td>P3</td>
<td>Safety Controller</td>
<td>Component to be designed.</td>
</tr>
<tr>
<td>P4A2</td>
<td>Decoy µProcessor</td>
<td>DecoyµPArch2 : <a href="FPGA,%C2%B5P2">Message Buffer2</a></td>
</tr>
<tr>
<td>P6</td>
<td>FPGA</td>
<td>To be designed component which decodes con2 to extract a fire command request (leading to fire?).</td>
</tr>
<tr>
<td>P6A</td>
<td>FPGA</td>
<td>We choose the following specification for FPGA. The incoming con message consists of 6 bytes as shown in the table below. The first byte is a fixed header to indicate start of message, the remaining bytes contain the FIRE, SEL, OTHER commands, alternate form of fire ALT. FIRE and the parity bit P. X indicates 3 Dont Care bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The header, FIRE, ALT. FIRE and P are transferred to FPGA as con2. FPGA uses HEADER to extract the FIRE, ALT. FIRE commands (each 4 bits) and the parity bit P. These are decoded and checked for validity. If an anomaly is detected then FPGA outputs the ¬fire? message. Otherwise it outputs the decoded FIRE/ALT. FIRE command, as either fire? or ¬fire?.</td>
</tr>
<tr>
<td>P7</td>
<td>µP2</td>
<td>To be designed component which decodes con1 to extract the selected flare type (leading to sel).</td>
</tr>
<tr>
<td>P7A</td>
<td>µP2</td>
<td>We choose the following specification for µP2. The incoming con message consists of 6 bytes as explained in J6. HEADER and SEL are transferred to µP2 as con1. µP2 uses HEADER to extract the SEL command, which is then decoded and sent to the Dispenser Unit as sel.</td>
</tr>
<tr>
<td>P8</td>
<td>Safety Controller</td>
<td>Yet to be designed; it will safely and reliably issue fire on receiving fire? with all interlocks satisfied.</td>
</tr>
<tr>
<td>P8A</td>
<td>Safety Controller</td>
<td>The logic specification for the Safety Controller output of its fire command is as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="markdown_table" alt="Logic Table" /></td>
</tr>
</tbody>
</table>