Proof in the Analysis of a Model of a Tracking System

J. S. Fitzgerald
Centre for Software Reliability, University of Newcastle upon Tyne, UK

C. B. Jones
Dept. of Computer Science, University of Manchester, UK
now with Applications Division, Harlequin Ltd.

Abstract

Fully formal proof is not always possible within the financial and labour constraints of a commercial project. This report shows how knowledge of the structure of a proof can guide inspections and reviews, even when the proof itself is not to be derived. The study illustrates, on a reduced example, the main issues which arose as part of the proof-based analysis of a specification of a tracking mechanism for a nuclear plant.

1 Introduction

Many of the benefits of formal techniques in software development result from the ability to model a system at a level of abstraction which may not have been possible hitherto. To some extent, this is independent of the formality of the modelling language. The particular contribution of formality lies in the high degree of rigour which is available to the developer in analysing the model. Apart from syntax- and type-checking, and testing executable parts of models, the opportunity exists to use mathematical proof to increase confidence in a model.

Because of the effort (in training and application) involved, proof is often seen as a technique to be applied only when mandated. However, proofs can be conducted at various levels of detail, each with different levels of cost and benefit. This paper discusses the application of proof at various levels of rigour in the analysis of a substantial formal model developed as part of an industrial project. Experience on the project emphasised proof, not as a machine-based activity which produces an inscrutable script, but as a technique for structuring the arguments which should take place in reviewing a formal model.

The particular application on which this paper is based is a demonstrator system for tracking the movement of nuclear material through the phases of re-processing in an industrial plant. Section 2 gives the background to the project in more detail, while Section 3 introduces a concise formal model derived from the model developed in the project but reduced in scope for the purposes of this report. The reduced model nevertheless shares many of the characteristics of its larger-scale sibling. Section 4 contains two illustrations of the rôle of proof in validating and revising the formal model of the demonstrator. A number of issues were raised by the proof activity concerning such matters as the careful delineation of system boundaries, the use of proof in the review cycle, the degree of genericity in the specification and alternative models which might have been more appropriate for the validation task. These are discussed in Section 5.

2 Context of the study

This report concerns work carried out with British Nuclear Fuels (Engineering) on the tracking of nuclear material as it passes through the various stages of industrial re-processing. For reasons of safety and security, as well as efficiency, it is necessary to track the movement of material through the reprocessing plant to ensure, for example, that there is not a build-up of fissile material in one stage of processing, and that all material is accounted for. Typically, the information about the movement of materials is distributed among the computers associated with each processing stage. A new approach to tracking was proposed, based on an architecture of *tracking managers*. Each tracking manager is responsible for monitoring, recording and permitting the movement of material through part of the plant.

The study reported here investigated the use of formal modelling in clarifying the requirements for a tracking manager architecture and validating its safety properties. Informal requirements for a *demonstrator plant* illustrating the use of the tracking manager architecture were determined in collaboration with domain experts at British Nuclear Fuels (Engineering) Ltd (BNFL). Certain properties of the system were felt to be related to the safe operation of the plant. These properties, which were expected to hold in the demonstrator plant, were stated informally. A formal model of the demonstrator plant in VDM-SL [LHP+96] was developed using SpecBox [BFM89] and the IFAD VDM-SL Toolbox [ELL94]. The model was syntax- and type-checked using the tools, but was not exercised (the Toolbox has extensive animation facilities). As a validation exercise, two formal reviews were conducted and an investigation of the use of proof in discharging proof obligations and validation conjectures (including safety properties) took place. The rest of this report concentrates on the proof activity.

3 A formal model of a tracking system

Consider a simplified waste processing plant. Material arrives in a number of *packages* stored in a *crate*. The crate and packages are opened and the contents distributed among a number of *liners*, each of which contains material of just one type (examples of material types are *glass*, *metal*, *plastic* and *liquor*). Liners are assayed to determine their fissile material content and sent to the next phase, product treatment. When liners arrive at the next phase, they may be sent to a compaction device to be crushed. A crushed liner is called a *puck*. Pucks and non-compacted liners are stored in *drums* before being passed to a storage phase. The storage phase contains sub-phases to deal with the allocation of drums to locations in a store.

In the tracking manager project, the plant described was modelled in some detail, with additional models of the tracking managers themselves and the history of movements of containers around the plant. Here consideration is confined to modelling the containers in the plant and their movement. The plant modelled here follows much the same process as that described in the example above, with five principal phases: unpacking crates, sorting contents, assaying materials, compacting materials and exporting from the plant.

The two main components of the system state are shown below:

$$
\begin{array}{l}
\mathsf{state}\ \mathit{System}\ \mathsf{of} \\
\quad \mathit{phases} : \mathit{PhaseId} \xrightarrow{m} \mathit{Phase} \\
\quad \mathit{containers} : \mathit{ContainerId} \xrightarrow{m} \mathit{Container} \\
\quad \mathsf{inv}\ \mathsf{mk}\text{-}\mathit{System}\,(\mathit{phases}, \mathit{containers}) \triangleq \\
\qquad \ldots
\end{array}
$$

$\quad \mathsf{init}\ sys \triangleq \ \ldots$
end

The *phases* component models the current status of the plant, indicating which containers are in processing in each phase. The *containers* component records all the current information about each container. In the remainder of this section, the details of the model are supplied. In many cases, these details have been added to illustrate characteristics of the larger formal model derived in the tracking manager project, and the model derived here should be viewed in that light.

Data model

First, consider the models of containers and the materials they contain. Materials and types of container are modelled as enumerated types:

$$\mathit{Material} = \mathrm{GLASS} \mid \mathrm{PLASTIC} \mid \mathrm{METAL} \mid \mathrm{LIQUOR};$$

$$\mathit{ContainerType} = \mathrm{CRATE} \mid \mathrm{PACKAGE} \mid \mathrm{LINER} \mid \mathrm{PUCK} \mid \mathrm{DRUM};$$

A container has a certain type. From the system description, it is apparent that containers may contain either "raw material" or other containers. This is modelled using optional types. In addition, each container has some associated assay data relating to the fissile material in it:

$$\begin{aligned}\mathit{Container} :: {} & \mathit{type} : \mathit{ContainerType} \\ & \mathit{material} : [\mathit{Material}] \\ & \mathit{contents} : [\mathit{ContainerId}\text{-}\mathsf{set}] \\ & \mathit{data} : [\mathit{AssayData}]\end{aligned}$$

A container may contain either raw material or other containers, but not both, so an invariant is added to record the restriction that exactly one of the *material* and *contents* fields must be nil :

$$\begin{aligned}&\mathsf{inv}\ c \triangleq \\ &\quad c.\mathit{material} = \mathsf{nil} \quad \Leftrightarrow \quad c.\mathit{contents} \neq \mathsf{nil}\end{aligned}$$

The *data* field is permitted to be nil when no assay data has yet been assigned to the container. The representation of assay data is immaterial to the study, so the type *AssayData* can be represented as token:

$$\mathit{AssayData} = \mathsf{token}$$

Now it is possible to consider the phases of the plant. Each phase has an associated input buffer and output buffer[1], each with a maximum capacity. Each phase expects a certain kind of container at its input and produces a certain kind of container at its output. No ordering among the elements of each buffer is used in the study and so each buffer is modelled as a set of container identifiers:

[1] In practice, these are often rail sidings in which materials, having been moved about the plant by rail, await treatment or further movement.

$$
\begin{array}{ll}
\mathit{Phase} :: & \mathit{input\text{-}capacity} : \mathbb{N} \\
 & \mathit{input\text{-}type} : \mathit{ContainerType} \\
 & \mathit{current\text{-}input} : \mathit{ContainerId}\text{-}\mathsf{set} \\
 & \mathit{output\text{-}capacity} : \mathbb{N} \\
 & \mathit{output\text{-}type} : \mathit{ContainerType} \\
 & \mathit{current\text{-}output} : \mathit{ContainerId}\text{-}\mathsf{set}
\end{array}
$$

Container identifiers can be modelled as token, as their representation is immaterial:

$$\mathit{ContainerId} = \mathsf{token}$$

It is expected that the capacities of the input and output buffers of a phase will be respected. In fact, preventing the build-up of hazardous materials in one area may be a safety issue for a plant such as this. An invariant records the restriction that the cardinalities of the buffers do not exceed the limits. An additional restriction is that no container can appear in both buffers simultaneously:

$$
\begin{array}{l}
\mathsf{inv}\ p \triangleq \\
\quad \mathsf{card}\ p.\mathit{current\text{-}input} \leq p.\mathit{input\text{-}capacity} \wedge \\
\quad \mathsf{card}\ p.\mathit{current\text{-}output} \leq p.\mathit{output\text{-}capacity} \wedge \\
\quad p.\mathit{current\text{-}input} \cap p.\mathit{current\text{-}output} = \{\}
\end{array}
$$

State invariant

Now it is possible to complete the overall data model of the plant. The state consists of mappings relating the identifiers to descriptions of phases and containers:

$$
\begin{array}{l}
\mathsf{state}\ \mathit{System}\ \mathsf{of} \\
\quad \mathit{phases} : \mathit{PhaseId} \xrightarrow{m} \mathit{Phase} \\
\quad \mathit{containers} : \mathit{ContainerId} \xrightarrow{m} \mathit{Container} \\
\mathsf{end}
\end{array}
$$

In previous projects, the BNFL participants had remarked on the value of invariants as a means of recording restrictions on systems which would otherwise be tacitly assumed. In this project, the state invariant was used to record three main kinds of restriction:

- Consistency between state components. For example, all the containers in phases should be known in the container mapping.
- Additional constraints required for system safety. For example, that the total fissile mass of containers in a phase should not exceed a certain value.
- Consistency properties on the containers in the plant. These mainly took the form of *containment laws*: regulations regarding the materials and containers which each kind of container may hold.

Each kind of restriction can be illustrated in the small model presented so far. The property that all containers in *phases* must be known in the *containers* mapping is recorded as:

$$
\begin{array}{l}
\mathsf{inv}\ \mathsf{mk}\text{-}\mathit{System}\,(\mathit{phases}, \mathit{containers}) \triangleq \\
\quad \bigcup \{p.\mathit{current\text{-}input} \cup p.\mathit{current\text{-}output} \mid p \in \mathsf{rng}\ \mathit{phases}\} \subseteq \\
\quad \mathsf{dom}\ \mathit{containers}
\end{array}
$$

An additional requirement is that a container should not appear in more than one phase:

$$
\begin{array}{l}
\mathsf{inv}\ \mathsf{mk}\text{-}System\ (phases, containers) \triangleq \\
\quad \ldots \wedge \\
\quad (\forall\, p1, p2 \in \mathsf{dom}\ phases \cdot p1 \neq p2 \;\Rightarrow \\
\qquad\qquad\qquad (phases\ (p1).current\text{-}input \cup \\
\qquad\qquad\qquad\ phases\ (p1).current\text{-}output) \\
\qquad\qquad\qquad \cap \\
\qquad\qquad\qquad (phases\ (p2).current\text{-}input \cup \\
\qquad\qquad\qquad\ phases\ (p2).current\text{-}output) \\
\qquad\qquad\qquad = \{\})
\end{array}
$$

It is a safety requirement that the containers in each phase should be of the type expected for the phase:

$$
\begin{array}{l}
\mathsf{inv}\ \mathsf{mk}\text{-}System\ (phases, containers) \triangleq \\
\quad \ldots \wedge \\
\quad (\forall\, p \in \mathsf{rng}\ phases \cdot \\
\qquad \forall\, c \in p.current\text{-}input \cdot containers\ (c).type = p.input\text{-}type \wedge \\
\qquad \forall\, c \in p.current\text{-}output \cdot containers\ (c).type = p.output\text{-}type)
\end{array}
$$

The containment laws are summarised as follows:

1. The contents of any container must be known.
2. Crates contain only packages.
3. Packages contain only raw material.
4. Liners contain only packages.
5. Drums contain only pucks and liners.
6. Pucks contain only one liner of non-liquor material.

These are formalised as conjuncts of the invariant, giving the complete invariant shown in Figure 1.

Initialisation clause

A characteristic of the formal model developed on project was that it described a particular "demonstrator" plant with a certain structure and series of processing phases. Despite this, the basic data type definitions in the model were not specific to a given phase structure. The particular structure was fixed in the state initialisation clause. In the smaller example developed in this report, the plant is initialised to five phases with different kinds of expected container and different buffer capacities:

```
state $System$ of
  $phases : PhaseId \xrightarrow{m} Phase$
  $containers : ContainerId \xrightarrow{m} Container$

  inv mk-$System\ (phases, containers) \triangleq$
     $\bigcup \{p.current\text{-}input \cup p.current\text{-}output \mid p \in \mathsf{rng}\ phases\} \subseteq$
     dom $containers \wedge$
     $(\forall\, p1, p2 \in \mathsf{dom}\ phases \cdot$
          $p1 \neq p2 \ \Rightarrow$
          $(phases\,(p1).current\text{-}input \cup phases\,(p1).current\text{-}output)$
          $\cap$
          $(phases\,(p2).current\text{-}input \cup phases\,(p2).current\text{-}output)$
           $= \{\}) \wedge$
     $(\forall\, p \in \mathsf{rng}\ phases \cdot$
          $(\forall\, c \in p.current\text{-}input \cdot$
               $containers\,(c).type = p.input\text{-}type) \wedge$
          $(\forall\, c \in p.current\text{-}output \cdot$
               $containers\,(c).type = p.output\text{-}type)) \wedge$
     $\forall\, c \in \mathsf{dom}\ containers \cdot$
         let mk-$Container\,(type, material, contents, \text{-}) =$
               $containers\,(c)$ in
         $(contents \neq \mathsf{nil} \ \Rightarrow\ contents \subseteq \mathsf{dom}\ containers) \wedge$
         $(type = \mathrm{CRATE} \ \Rightarrow$
          $contents \neq \mathsf{nil} \ \wedge$
          $\forall\, p \in contents \cdot containers\,(p).type = \mathrm{PACKAGE}) \wedge$
         $(type = \mathrm{PACKAGE} \ \Rightarrow\ material \neq \mathsf{nil}\ ) \wedge$
         $(type = \mathrm{LINER} \ \Rightarrow$
          $contents \neq \mathsf{nil} \ \wedge$
          $\forall\, c1 \in contents \cdot containers\,(c1).type = \mathrm{PACKAGE}) \wedge$
         $(type = \mathrm{DRUM} \ \Rightarrow$
          $contents \neq \mathsf{nil} \ \wedge$
          $(\forall\, c1 \in contents \cdot containers\,(c1).type = \mathrm{PUCK} \vee$
                              $containers\,(c1).type = \mathrm{LINER})) \wedge$
         $(type = \mathrm{PUCK} \ \Rightarrow$
          $contents \neq \mathsf{nil} \ \wedge \mathsf{card}\ contents = 1 \wedge$
          let $\{l\} = contents$ in
          $containers\,(l).type = \mathrm{LINER} \wedge$
          $\forall\, p \in containers\,(l).contents \cdot$
               $containers\,(p).material \neq \mathrm{LIQUOR})$
end
```

Figure 1: State Invariant for the Plant

$$
\begin{array}{l}
\mathsf{init}\ sys \triangleq sys = \mathsf{mk}\text{-}System \\
\qquad (\\
\qquad\ \{\mathrm{UNPACK} \mapsto \\
\qquad\qquad \mathsf{mk}\text{-}Phase\,(10, \mathrm{CRATE}, \{\}, 1000, \mathrm{PACKAGE}, \{\}), \\
\qquad\quad \mathrm{SORT} \mapsto \\
\qquad\qquad \mathsf{mk}\text{-}Phase\,(500, \mathrm{PACKAGE}, \{\}, 100, \mathrm{LINER}, \{\}), \\
\qquad\quad \mathrm{ASSAY} \mapsto \\
\qquad\qquad \mathsf{mk}\text{-}Phase\,(1000, \mathrm{LINER}, \{\}, 1000, \mathrm{LINER}, \{\}), \\
\qquad\quad \mathrm{COMPACTION} \mapsto \\
\qquad\qquad \mathsf{mk}\text{-}Phase\,(1000, \mathrm{LINER}, \{\}, 1000, \mathrm{PUCK}, \{\}), \\
\qquad\quad \mathrm{EXPORT} \mapsto \\
\qquad\qquad \mathsf{mk}\text{-}Phase\,(100, \mathrm{PUCK}, \{\}, 250, \mathrm{DRUM}, \{\})\}, \\
\qquad\ \{\mapsto\})
\end{array}
$$

The *containers* state component is initially empty.

Example operations

The model developed in the project described the behaviour of the demonstrator plant by means of operations. Here too, there was a mixture between the generic and the particular. Most operations were specific to each phase, for example describing the effects of assaying a container on the record maintained about that container in the system state. It was observed that some actions recurred frequently throughout the system. For example, most phases involve packing a new container at some point, filling it with a volume of material or a number of other containers. A number of generic operations were defined to describe these actions and these were then used in the phase-specific operations by quoting their post-conditions.

As an example, consider the operation describing part of the sorting process which follows the unpacking phase. Packages are placed in liners according to the kind of material they contain. The process of moving a set of packages into a liner is described by the operation given below:

$$
\begin{array}{l}
SORT\,(new : ContainerId, mat : Material, packs : ContainerId\text{-}\mathsf{set}) \\
\mathsf{ext\ wr}\ phases : PhaseId \xrightarrow{m} Phase \\
\qquad \mathsf{wr}\ containers : ContainerId \xrightarrow{m} Container \\
\mathsf{pre}\ \ \mathsf{let}\ p = phases\,(\mathrm{SORT})\ \mathsf{in} \\
\qquad packs \subseteq p.current\text{-}input \land \\
\qquad (\forall\, p \in packs \cdot containers\,(p).material = mat) \land \\
\qquad \mathsf{card}\ p.current\text{-}output < p.output\text{-}capacity \land \\
\qquad \mathsf{pre}\text{-}PACK\,(new, \mathrm{LINER}, \mathsf{nil}\ , packs, \mathsf{mk}\text{-}System\,(phases, containers)) \\
\mathsf{post}\ \mathsf{let}\ p = \overleftharpoon{phases}\,(\mathrm{SORT})\ \mathsf{in} \\
\qquad phases = \overleftharpoon{phases} \dagger \\
\qquad \{\mathrm{SORT} \mapsto \mu\ (p, current\text{-}input \mapsto p.current\text{-}input \setminus packs, \\
\qquad\qquad\quad current\text{-}output \mapsto p.current\text{-}output \cup \{new\})\} \land \\
\qquad \mathsf{post}\text{-}PACK\,(new, \mathrm{LINER}, \mathsf{nil}\ , packs, \\
\qquad\qquad\qquad \mathsf{mk}\text{-}System\,(\overleftharpoon{phases}, \overleftharpoon{containers}), \\
\qquad\qquad\qquad \mathsf{mk}\text{-}System\,(phases, containers))
\end{array}
$$

The operation takes as input the identifier of a container which will hold the sorted packages, the kind of material contained in all the packages and the set of identifiers of the packages to be sorted. The precondition ensures that the packages to be sorted are all in the input buffer of the phase, that they share a common material and that there is room for the new container in the output buffer. The postcondition requires that the state be updated with the packages removed from the phase input and packed into a container which is added to the phase output.

The pre- and postconditions of another operation, *PACK*, are quoted in the *SORT* operation. This more generic operation is used in the operations which are specific to particular phases in the plant. It specifies the process of putting arbitrary contents into a given container, updating the *containers* state component accordingly. The *PACK* operation is studied in more detail in Section 4 below. However, it can be presented here:

$$\begin{array}{l}
PACK\ (cid : ContainerId, ctype : ContainerType, \\
\qquad cmaterial : [Material], ccontents : [ContainerId\text{-}\mathsf{set}]) \\
\mathsf{ext\ wr}\ containers : ContainerId \xrightarrow{m} Container \\
\mathsf{pre}\ \ cid \notin \mathsf{dom}\ containers \land \\
\quad (ccontents \neq \mathsf{nil}\ \Rightarrow\ ccontents \subseteq \mathsf{dom}\ containers) \\
\mathsf{post}\ containers = \overleftarrow{containers} \dagger \\
\quad \{cid \mapsto \mathsf{mk}\text{-}Container\ (ctype, cmaterial, ccontents, \mathsf{nil}\ )\}
\end{array}$$

The container identified by *cid* is to be created containing the given *ctype*, *cmaterial* and *ccontents*. The precondition states that, for container packing to be applied correctly, the new container should not already exist (i.e. it should not be in the domain of the *containers* state component) and if it is to contain other containers, these other containers should be known. If this condition holds, the *containers* state component is updated with the new container.

The formal model

This concludes the introduction to the formal model. The characteristics of the model of special note are:

1. the use of the state invariant to record properties derived from the safety analysis of the plant;
2. the mixture of generic data types, which could be applied in a model of any plant (e.g. *Phase*) with types specific to the demonstrator plant (e.g. *Material*);
3. the use of the initialisation clause to fix the structure of the plant;
4. the mixture of generic operations (e.g. *PACK*) with operations specific to particular phases (e.g. *SORT*).

The full model developed in the project was much more complex in a number of respects. Additional state components were needed to record histories of container movement and the mechanism for granting permission for container movement. In addition, the plant structure itself was hierarchical, with phases divided into sub-phases.

The reader may have already identified deficiencies in the model presented so far. It should be stressed that the model presented here is being shown "warts and all". The main purpose of this paper is to show how a knowledge of proof assists in improving such a model. In fact, a fully-reviewed model of the demonstrator plant could look significantly different. In the following section, the review process applied in the project is illustrated on some examples of safety-related properties.

4 Analysing the model with proof

One aim of the tracking manager project was to see if a formal model of the plant could be useful in analysing the safety features which the proposed tacking manager architecture would have to maintain. It is in the validation of the formal model that this checking is carried out. In this section, some of the techniques employed to analyse the model in the project are illustrated on the reduced model developed above. First, two conjectures which arose in the project are introduced. The proofs of these conjectures are discussed: one involving rigorous reasoning guided by the structure of a formal proof; the second containing a fully formal element. In each case, deficiencies in the formal model are highlighted and discussed.

In VDM-SL, specifications consist of a state definition in terms of some defined data types, along with specifications of operations which can be invoked to change the system state. Typically, the state definition contains a number of state variables whose values are constrained by an invariant. Most important for this study, many of the constraints necessary to avoid hazards arising were added to the invariant: for example, the requirement that liners must contain packages of a single type. The specified operations on the system must never lead to a state which violates the invariant. It is therefore necessary to check the operation specifications to ensure that this is indeed the case

Each operation is specified in terms of a *precondition* and *postcondition*. The precondition records assumptions about the state and input parameters, while the postcondition indicates how the state may be modified and what result is returned. There is an obligation on the author of the specification to show that the precondition and postcondition are consistent in the sense that every combination of system state and inputs satisfying the precondition has a corresponding state and output combination satisfying both the invariant and the postcondition. This is termed the *satisfiability proof obligation* [Jon90]. In discharging the satisfiability obligation, one shows that the operation respects the invariant, and consequently for the tracking manager specification, respects the safety properties.

The formal model used in the BNFL project was too large (2500 lines approx.) to permit the proof that all the safety properties are respected by all the operations within the resources of the project. Instead, three typical safety-related conjectures, including one satisfiability proof obligation, were chosen. They concentrated on areas of the specification which its authors felt were the most susceptible to error and were representative of other proofs which could be undertaken.

The following section introduces three levels of rigour at which proofs may be constructed, and indicates the trade-offs between them. Section 4.2 describes the particular conjectures chosen for the tracking manager study.

4.1 Levels of rigour in proof

A proof based on a formal specification can be carried out at various levels of rigour. The three identified here are:

"Textbook" proof: This is the level of rigour found in most general mathematics texts. The argument is made in English, supported by formulae. Justifications for steps in the reasoning often appeal to human insight. This is the easiest of the three styles of proof to read, but the reliance on intuition means that such proofs can only be checked by other human beings.

Fully formal proof: A fully formal proof (of the kind discussed in [BFL^{+}94]) is a highly structured sequence of assertions in a well-defined formal language. Each step is justified by appealing to a formally-stated rule of inference. A formal proof is so detailed that it can be checked mechanically. It is possible to have a high degree of confidence in such a proof, but construction of the proof is very laborious, even with such machine assistance as is currently available. Formal proof is most frequently employed in highly critical applications.

Rigorous proof: This refers to a proof which borrows the ideas of careful structuring and line-by-line justification from the formal proof, but relaxes some of the strictures which make the production of a formal proof so costly. Obvious typing hypotheses may be omitted, abbreviations may be used, justifications may appeal to general theories rather than to specific rules of inference.

When executed carefully, "textbook", and even rigorous, proofs should provide a structure on which a fully formal proof could subsequently be based. All three levels of rigour were viable options for proofs of safety properties in the demonstrator plant model.

4.2 Validation conjectures

This section introduces two conjectures considered in the analysis of the model of the demonstrator architecture. The conjectures are based on the following questions and observations:

- The operation for packing a container is generic. Could it violate the containment clauses, i.e. is it possible to pack something into a container which is not allowed to contain it?
- It should not be possible to compact liquor.

Although one might state conjectures in this informal way in a normal development process, the aim of verifying them against a formal specification requires greater precision in their formulation. For example, the second conjecture fails to draw a clear distinction between the specification of the system and the physical system of which the specification is merely a model. The formal model can not itself prevent compaction of liquor, but the model can be analysed to see if the specified compaction operation can be applied to liners containing any containers which contain liquor.

4.3 Container packing

The first conjecture is that the container packing operation respects the containment laws. When the model was introduced above, the containment laws were stated as part of the invariant on the overall state. It would be expected that the proof of preservation of the state invariant would form a part of the satisfiability proof of the *PACK* operation, so this is where an examination of the model should begin. Indeed, it is noted in the "Proof in VDM" book [BFL^{+}94] that the bulk of the work associated with showing satisfiability is in showing invariant preservation.

In the BNFL project, a preliminary examination of the satisfiability obligation suggested modifications to the model. A systematic attempt to show satisfiability at the "mathematical textbook" level then pointed to a number of more subtle errors which could otherwise have escaped detection until later in the development.

Following the formalism laid down in [BFL+94] (page 177), an operation specification

$$\begin{array}{l}
Op\,(x : X)\ y : Y \\
\textsf{ext rd}\ \ r : R \\
\quad\ \ \textsf{wr}\ \ w : W \\
\textsf{pre}\ \ Pr(r, w) \\
\textsf{post}\ \ Po(r, \overleftarrow{w}, w)
\end{array}$$

operating on a state

$$\begin{array}{l}
\textsf{state}\ \Sigma\ \textsf{of} \\
\quad r : R \\
\quad w : W \\
\quad u : U \\
\textsf{end}
\end{array}$$

has satisfiability obligation

$$\boxed{\text{Op-sat}}\ \frac{x : X;\ mk\text{-}\Sigma(\overleftarrow{r}, \overleftarrow{w}, \overleftarrow{u}) : \Sigma;\ pre\text{-}Op(\overleftarrow{r}, \overleftarrow{w});}{\begin{array}{l}\exists\, y : Y,\ mk\text{-}\Sigma(r, w, u) : \Sigma \cdot \\ \quad post\text{-}Op(x, y, r, \overleftarrow{w}, w) \wedge r = \overleftarrow{r} \wedge u = \overleftarrow{u}\end{array}}$$

Note that the "framing" constraints given by the operation's externals clause must be carefully handled.

Stated formally, the satisfiability obligation for *PACK* is as follows:

$$\boxed{\text{PACK-sat}}\ \frac{\begin{array}{l} cid : ContainerId; \\ ctype : ContainerType; \\ cmat : [Material]; \\ ccnts : [ContainerId\textsf{-set}]; \\ mk\text{-}System(\overleftarrow{phases}, \overleftarrow{containers}) : System \\ pre\text{-}PACK\,(cid, ctype, cmat, ccnts, \overleftarrow{containers}) \end{array}}{\begin{array}{l} \exists\, mk\text{-}System(phases, containers) : System \cdot \\ \quad post\text{-}PACK\,(cid, ctype, cmat, ccnts, \overleftarrow{containers}, containers) \end{array}}$$

Before embarking on a proof, time spent trying to construct a counter-example to the conjecture can be rewarding. If a counter-example can be found easily, it is as well to consider modification of the specification before proceeding further. Knowing that the proof of *PACK-sat* would concentrate on showing that the containment laws are respected, the reviewers of the specification tried to construct a counter-example, and one was forthcoming. The reader might wish to treat this as an exercise before reading on.

Consider invoking *PACK* on a liner l, wanting to pack inside it a crate c. The precondition evaluates to true, so the satisfiability obligation requires that a state be found to satisfy the post-condition. The post-condition requires that

$$containers = \overleftarrow{containers} \dagger \{l \mapsto \textsf{mk-}Container\,(\text{LINER}, \textsf{nil}\,, \{d\}, \textsf{nil}\,)\}$$

but this contradicts the clause in the state invariant which states that liners in the *containers* mapping may only contain packages:

```
$(type = \mathrm{LINER} \;\Rightarrow$
 $contents \neq \mathsf{nil} \;\wedge$
 $\forall\, c1 \in contents \cdot$
       $containers\,(c1).type = \mathrm{PACKAGE} \wedge \ldots)$
```

It was not therefore possible to show that the containment laws would be respected for all valid inputs to *PACK*, and so a number of alternative strategies for overcoming this were considered. The precondition of *PACK* could be modified to reflect all the containment laws. However, this would make the precondition rather large and unwieldy, certainly in a proof. It would be more practical to record the containment laws as a separate auxiliary function. However, it was also noted that the containment laws and some other conjuncts in the invariant relate solely to the *containers* state component and do not relate it to the *phases* component, so the preferred solution was to define a new type *ContainerMap* which has an invariant recording all the restrictions relating solely to containers, including the laws. This type definition is given in Figure 2. The state definition now uses the new type and the containment laws can be omitted from the state invariant. The modified state definition is given in Figure 3. Now the *PACK* operation can be modified by quoting the *ContainerMap* invariant in the precondition to ensure that the new *containers* state component to be constructed will respect the laws:

```
$ContainerMap = ContainerId \xrightarrow{m} Container$

$\mathsf{inv}\ containers \;\triangleq$
   $\forall\, c \in \mathsf{dom}\ containers \cdot$
      $\mathsf{let}\ \mathsf{mk}\text{-}Container\,(type, material, contents, \text{-}) =$
            $containers\,(c)\ \mathsf{in}$
      $(contents \neq \mathsf{nil} \;\Rightarrow\; contents \subseteq \mathsf{dom}\ containers) \wedge$
      $(type = \mathrm{PACKAGE} \;\Rightarrow\; material \neq \mathsf{nil}\ ) \wedge$
      $(type = \mathrm{LINER} \;\Rightarrow$
       $contents \neq \mathsf{nil} \;\wedge$
       $\forall\, c1 \in contents \cdot containers\,(c1).type = \mathrm{PACKAGE}) \wedge$
      $(type = \mathrm{DRUM} \;\Rightarrow$
       $contents \neq \mathsf{nil} \;\wedge$
       $(\forall\, c1 \in contents \cdot containers\,(c1).type = \mathrm{PUCK} \vee$
                            $containers\,(c1).type = \mathrm{LINER})) \wedge$
      $(type = \mathrm{PUCK} \;\Rightarrow$
       $contents \neq \mathsf{nil} \;\wedge$
       $\mathsf{card}\ contents = 1 \wedge$
       $\mathsf{let}\ \{l\} = contents\ \mathsf{in}$
       $containers\,(l).type = \mathrm{LINER} \wedge$
       $\forall\, p \in containers\,(l).contents \cdot$
            $containers\,(p).material \neq \mathrm{LIQUOR})$
```

Figure 2: New *ContainerMap* type incorporating containment laws

```
$PACK2\,(cid : ContainerId, ctype : ContainerType, cmaterial : [Material],$
         $ccontents : [ContainerId\text{-}\mathsf{set}])$
$\mathsf{ext\ wr}\ containers : ContainerMap$
```

$$
\begin{array}{l}
\mathsf{state}\ \mathit{System}\ \mathsf{of} \\
\quad \mathit{phases} : \mathit{PhaseId} \xrightarrow{m} \mathit{Phase} \\
\quad \mathit{containers} : \mathit{ContainerMap} \\
\quad \mathsf{inv}\ \mathsf{mk}\text{-}\mathit{System}\,(\mathit{phases}, \mathit{containers}) \triangleq \\
\qquad \bigcup \{p.\mathit{current\text{-}input} \cup p.\mathit{current\text{-}output} \mid p \in \mathsf{rng}\ \mathit{phases}\} \subseteq \\
\qquad \mathsf{dom}\ \mathit{containers} \wedge \\
\qquad (\forall\, p1, p2 \in \mathsf{dom}\ \mathit{phases} \cdot \\
\qquad\quad p1 \neq p2 \ \Rightarrow \\
\qquad\quad (\mathit{phases}\,(p1).\mathit{current\text{-}input} \cup \mathit{phases}\,(p1).\mathit{current\text{-}output}) \cap \\
\qquad\quad (\mathit{phases}\,(p2).\mathit{current\text{-}input} \cup \mathit{phases}\,(p2).\mathit{current\text{-}output}) \\
\qquad\quad\ = \{\}) \wedge \\
\qquad (\forall\, p \in \mathsf{rng}\ \mathit{phases} \cdot \\
\qquad\quad \forall\, c \in p.\mathit{current\text{-}input} \cdot \\
\qquad\qquad \mathit{containers}\,(c).\mathit{type} = p.\mathit{input\text{-}type} \wedge \\
\qquad\qquad \forall\, c \in p.\mathit{current\text{-}output} \cdot \\
\qquad\qquad\quad \mathit{containers}\,(c).\mathit{type} = p.\mathit{output\text{-}type}) \\
\quad \mathsf{init}\ \mathit{sys} \triangleq \mathit{sys} = \ldots \\
\mathsf{end}
\end{array}
$$

Figure 3: State after introducing *ContainerMap*

$$
\begin{array}{l}
\mathsf{pre}\ \ \mathsf{inv}\text{-}\mathit{ContainerMap}\,(\mathit{containers} \dagger \\
\qquad\qquad\qquad\qquad \{\mathit{cid} \mapsto \mathsf{mk}\text{-}\mathit{Container}\,(\mathit{ctype}, \mathit{cmaterial}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{ccontents}, \mathsf{nil}\,)\}) \wedge \\
\quad \mathsf{inv}\text{-}\mathit{Container}\,(\mathsf{mk}\text{-}\mathit{Container}\,(\mathit{ctype}, \mathit{cmaterial}, \mathit{ccontents}, \mathsf{nil}\,)) \wedge \\
\quad \mathit{cid} \notin \mathsf{dom}\ \mathit{containers} \wedge \\
\quad (\mathit{ccontents} \neq \mathsf{nil} \ \Rightarrow\ \mathit{ccontents} \subseteq \mathsf{dom}\ \mathit{containers}) \\
\mathsf{post}\ \mathit{containers} = \overleftarrow{\mathit{containers}} \dagger \\
\quad \{\mathit{cid} \mapsto \mathsf{mk}\text{-}\mathit{Container}\,(\mathit{ctype}, \mathit{cmaterial}, \mathit{ccontents}, \mathsf{nil}\,)\}
\end{array}
$$

The satisfiability obligation can now be revisited with a view to constructing a "textbook" proof. The conclusion of the conjecture is an existential expression. A common strategy for demonstrating the existence of a value satisfying some condition is actually to construct such a value which stands as a witness to the truth of the existential expression (Section 3.3.1 of [BFL$^+$94]). In this case, the proof must conclude that there exists a system satisfying the post-condition. The post-condition suggests a suitable witness value: it states that the *containers* component of the system is updated to include the new *cid* and the *Container* it points to, and the other state component remains unchanged. The witness value is therefore

$$\mathsf{mk}\text{-}\mathit{System}\,(\overleftarrow{\mathit{phases}}, \mathit{containers})$$

where

$$
\begin{array}{l}
\mathit{containers} = \overleftarrow{\mathit{containers}} \dagger \\
\quad \{\mathit{cid} \mapsto \mathsf{mk}\text{-}\mathit{Container}\,(\mathit{ctype}, \mathit{cmaterial}, \mathit{ccontents}, \mathsf{nil}\,)\}
\end{array}
$$

Call this witness value σ. To discharge the proof obligation, it is necessary to show three sub-obligations:

- that σ has the correct basic type (*System*);
- that σ satisfies *post-PACK*2;
- that σ satisfies inv-*System*.

The first sub-obligation is straightforward. For σ to be a system, its components must all be of the correct type. The unchanged components were drawn from a *System* ($\overleftarrow{\sigma}$) and so are still of the correct type. For the new *containers* component to be a *ContainerMap* it must

- be a mapping from *ContainerId* to *Container*; and
- satisfy inv-*ContainerMap*.

The constructed system σ is certainly a mapping between the correct types. It is also known to satisfy inv-*ContainerMap* because this is now guaranteed by the hypothesis *pre-PACK*2 in the conjecture.

The second sub-obligation is also straightforward: σ satisfies *post-PACK*2 by construction.

It remains to show that σ is indeed a well-formed *System*, satisfying the invariant. The approach employed when reviewing the demonstrator plant model was to consider each conjunct of the invariant in turn to see if it could fail. Although one problem was spotted and remedied before a detailed proof was considered by the definition and use of inv-*ContainerMap* in the precondition, the clause-by-clause examination of inv-*System* revealed several cases in which satisfiability was still not guaranteed. Three examples are considered below:

1.
$$\begin{aligned}&\bigcup\{p.\mathit{current\text{-}input} \cup p.\mathit{current\text{-}output} \mid p \in \mathsf{rng}\ \mathit{phases}\}\\ &\quad\subseteq \mathsf{dom}\ \mathit{containers}\end{aligned}$$

   This clause asserts that all the containers in the buffers of all phases are known in the *containers* mapping. The new state σ adds a new identifier *cid* to the domain of *containers* (the fact that *cid* is new is guaranteed by the hypothesis *pre-PACK*) and does not change any other part of the *containers* mapping so all the containers known before the *PACK* operation are still known afterwards. This conjunct is therefore preserved.

2.
$$\begin{aligned}&(\forall\, p1, p2 \in \mathsf{dom}\ \mathit{phases} \cdot\\ &\quad p1 \neq p2 \;\Rightarrow\\ &\quad (\mathit{phases}\ (p1).\mathit{current\text{-}input} \cup \mathit{phases}\ (p1).\mathit{current\text{-}output}) \cap\\ &\quad (\mathit{phases}\ (p2).\mathit{current\text{-}input} \cup \mathit{phases}\ (p2).\mathit{current\text{-}output}) = \{\})\end{aligned}$$

   This conjunct describes the requirement that no two phases should have any containers in common. This is unaffected by any change in the *containers* mapping, and so still holds after *PACK* has been applied.

3.
$$\begin{aligned}&\forall\, p \in \mathsf{rng}\ \mathit{phases} \cdot\\ &\quad (\forall\, c \in p.\mathit{current\text{-}input} \cdot\\ &\qquad \mathit{containers}\ (c).\mathit{type} = p.\mathit{input\text{-}type}) \wedge\\ &\quad (\forall\, c \in p.\mathit{current\text{-}output} \cdot\\ &\qquad \mathit{containers}\ (c).\mathit{type} = p.\mathit{output\text{-}type})\end{aligned}$$

This conjunct asserts that containers respect the container types expected for each buffer. The *phases* state component is not affected by *PACK* and, as already argued, the *containers* mapping is added to and not otherwise changed, so this conjunct is again regarded as preserved.

Depending on the level of level of confidence one has in an argument of this form, it would be possible to stop here or to go further and formalise each of the three sub-obligations as lemmas which contribute to a formal overall proof of satisfiability.

In the demonstrator plant specification, some 21 conjuncts of the invariant were affected by *PACK*. Examination in this structured but informal way revealed a number of errors which might otherwise have gone undetected.

4.4 Safety of compaction

The second conjecture considered was that it should be "impossible to compact liquor". This conjecture is slightly more difficult to formulate than satisfiability of *PACK*. However, after consultation with domain experts, it became clear that the operation describing the compaction phase should be protected by its precondition from operating on containers with liquor in them. The specification of the compaction operation is given in Figure 4. The compaction operation updates the *phases* mapping by removing a container from the input buffer and generating a puck containing only the compressed container in the output buffer. The precondition is of more interest for the conjecture proposed. It states that the compacted container is known and that there is capacity for the puck in the output buffer of the compaction phase. The precondition of *PACK*2 is established. The last two conjuncts of the precondition were intended to ensure that the liner arriving for compaction does not contain any packages of liquor.

```
$COMPACTION\ (cid : ContainerId, new : ContainerId)$
ext wr $phases : PhaseId \xrightarrow{m} Phase$
    wr $containers : ContainerMap$
pre  let $p = phases\ (\mathrm{COMPACTION})$ in
    $cid \in p.current\text{-}input \wedge$
    card $p.current\text{-}output < p.output\text{-}capacity \wedge$
    pre-$PACK2\ (new, \mathrm{PUCK}, \mathsf{nil}, \{cid\},$
                mk-$System\ (phases, containers)) \wedge$
    $containers\ (cid).contents \neq \mathsf{nil} \wedge$
    $\forall c' \in containers\ (cid).contents \cdot$
        $containers\ (c').material \neq \mathrm{LIQUOR}$
post let $p = \overleftarrow{phases}\ (\mathrm{COMPACTION})$ in
     $phases = phases \dagger \{\mathrm{COMPACTION} \mapsto$
      $\mu\ (p, current\text{-}input \mapsto p.current\text{-}input \setminus \{cid\},$
            $current\text{-}output \mapsto p.current\text{-}output \cup \{new\})\} \wedge$
     post-$PACK2\ (new, \mathrm{PUCK}, \mathsf{nil}, \{cid\},$
                 mk-$System\ (\overleftarrow{phases}, \overleftarrow{containers}),$
                 mk-$System\ (phases, containers))$ ;
```

Figure 4: Compaction operation

The conjecture should, roughly, take the following form:

$$\boxed{\text{Compaction}}\ \frac{\begin{array}{l} cid : ContainerId; \\ new : ContainerId; \\ mk\text{-}System(phases, containers) : System; \\ pre\text{-}COMPACTION(cid, new, containers, phases) \end{array}}{\text{Liquor not in } cid}$$

How should the "Liquor not in *cid*" condition be expressed? The containment rules give a hierarchy of possible containments. Given a *ContainerMap* and a *ContainerId*, it should be possible to define a recursive function which gathers all the material types in a container and its sub-containers:

$$\begin{array}{l} gather : ContainerMap \times ContainerId \rightarrow Material\text{-}\mathsf{set} \\[4pt] gather\ (m, c)\ \triangleq \\ \quad \mathsf{if}\ m\ (c).material \neq \mathsf{nil} \\ \quad \mathsf{then}\ \{m\ (c).material\} \\ \quad \mathsf{else}\ \bigcup\{gather\ (m, c') \mid c' \in m\ (c).contents\} \\ \mathsf{pre}\ \ c \in \mathsf{dom}\ m \end{array}$$

Thus the formal conjecture should be:

$$\boxed{\text{Compaction}}\ \frac{\begin{array}{l} cid : ContainerId; \\ new : ContainerId; \\ mk\text{-}System(phases, containers) : System; \\ cid \in \mathsf{dom}\ containers; \\ pre\text{-}COMPACTION(cid, new, containers, phases) \end{array}}{\text{LIQUOR} \notin gather(containers, cid)}$$

The *gather* function was drafted purely to assist in the proof process: it was not part of the model of the plant. However, it was apparent that the *COMPACTION* function did not make use of the same kind of recursive accumulation function. Did another part of the precondition ensure that the compaction operation was only applied to containers nested one deep, or did this hint at a counter-example?

In fact, it was possible to construct a counter-example to the conjecture. Consider a drum d which contains one liner l which contains a package p which contains liquor. In this case, the precondition of *COMPACTION* is satisfied, because all the containers in d have the *material* component set to nil . However, the *gather* function would discover the liquor "hiding" in the package p.

It was clear that the compaction operation somehow relied on the input being a liner, so the precondition was modified to include an explicit check to this effect. A further discussion of the flaw in the operation specification and this resolution follows at the end of this section.

The modified compaction operation is:

$$\begin{array}{l} COMPACTION2\,(cid : ContainerId, new : ContainerId) \\ \mathsf{ext\ wr}\ phases : PhaseId \xrightarrow{m} Phase \\ \qquad \mathsf{wr}\ containers : ContainerMap \end{array}$$

```
pre  let $p = phases\,(\mathrm{COMPACTION})$ in
     $cid \in p.current\text{-}input \wedge$
     card $p.current\text{-}output < p.output\text{-}capacity \wedge$
     $new \notin$ dom $containers \wedge$
     pre-$PACK2\,(new, \mathrm{PUCK}, \mathsf{nil}\ , \{cid\},$
                    mk-$System\ (phases, containers)) \wedge$
     $containers\,(cid).contents \neq \mathsf{nil}\ \wedge$
     $\forall\, c' \in containers\,(cid).contents \cdot$
          $containers\,(c').material \in safe\text{-}materials$
post ...
```

Having modified the specification, the reviewers were not sufficiently confident about the correction to accept a "textbook" argument. Instead a rigorous proof of the conjecture was undertaken. The proof process begins by setting out the hypotheses and conclusion:

```
from $cid : ContainerId;$
     $new : ContainerId;$
     $mk\text{-}System(phases, containers) : System;$
     $cid \in$ dom $containers;$
     $pre\text{-}COMPACTION\,(cid, new, containers, phases)$
     $\vdots$
infer $\mathrm{LIQUOR} \notin gather(containers, cid)$
```

The proof's structure can be predicted by considering the informal argument. One can begin by working backwards from the conclusion, expanding the definition of *gather*. It will be necessary to show that none of the packages in the container identified by *cid* contain liquor. In order to do this, we can reason forwards from the hypotheses: the (modified) precondition ensures that *cid* identifies a liner, which must (by the containment laws) contain only packages. It will be necessary to show that the none of the packages contain liquor. This ought to follow from the last conjunct of *pre-COMPACTION*. If none of the packages contain liquor, it should be possible to show that *gather*(*containers*, *cid*) does not contain liquor.

The central point of the proof, therefore, is going to be an assertion of the form:

$$\forall\, c' \in container(cid).contents \cdot \mathrm{LIQUOR} \notin gather(containers, c')$$

Call this crucial line α. The proof is of the form:

```
     from $cid : ContainerId;$
          $new : ContainerId;$
          $mk\text{-}System(phases, containers) : System;$
          $cid \in \mathsf{dom}\ containers;$
          $pre\text{-}COMPACTION(cid, new, containers, phases)$
          $\vdots$
$\alpha$      $\forall\, c' \in container(cid).contents \cdot \mathrm{LIQUOR} \notin gather(containers, c')$
          $\vdots$
     infer $\mathrm{LIQUOR} \notin gather(containers, cid)$
```

To obtain α, a $\forall$-introduction rule ([BFL$^+$94], pg. 45) is appropriate. Applying this backwards opens a subproof β:

```
     from $hypotheses$
          $\vdots$
$\beta$       from $c' \in container(cid).contents$
               $\vdots$
          infer $\mathrm{LIQUOR} \notin gather(containers, c')$
$\alpha$      $\forall\, c' \in container(cid).contents \cdot \mathrm{LIQUOR} \notin gather(containers, c')$
                                                                          $\forall$-I, $\beta$
          $\vdots$
     infer $\mathrm{LIQUOR} \notin gather(containers, cid)$
```

Notice that line α is not exactly justified by the $\forall$-I rule, which requires a typing rather than set membership hypothesis on the subproof. Such compromises make the argument rigorous rather than fully formal.

The subproof β contains the bulk of the argument for this conjecture. Working backwards from its conclusion, it is possible to see from the definition of *gather* that

$$gather(containers, c') = \{containers(c').material\}$$

From the last conjunct of *pre-COMPACTION*, it should also be possible to infer that

$$containers(c').material \neq \mathrm{LIQUOR}$$

and hence the conclusion. Updating the proof with this line of reasoning:

from $hypotheses$

$\vdots$

β from $c' \in container(cid).contents$

$\vdots$

a $gather(containers, c') = \{containers(c').material\}$

b $containers(c').material \neq \mathrm{LIQUOR}$

infer $\mathrm{LIQUOR} \notin gather(containers, c')$ Lemma1, a, b

α $\forall\, c' \in container(cid).contents \cdot \mathrm{LIQUOR} \notin gather(containers, c')$

$\forall$-I, β

$\vdots$

infer $\mathrm{LIQUOR} \notin gather(containers, cid)$

Lemma 1 is used to justify the subproof's conclusion, along with rules for the substitution of equal values:

$$\boxed{\text{Lemma1}}\ \frac{m, n : A;\ m \neq n;}{n \notin \{m\}}$$

We need to show that line a holds, by appealing to the definition of *gather*. This function is based on a conditional (if ... then ... else ...), so it is necessary to show which arm of the conditional applies. In a formal proof it would also be necessary to show that the condition itself is defined. In this case, the first arm of the conditional is taken, because

$$containers(c').material \neq \mathsf{nil}$$

and this is known because c' must refer to a package. This is in turn known because c' is in the *contents* component of the container identified by *cid*, and *cid* must refer to a liner. The containment laws state that liners may only contain packages. Adding this chain of backwards reasoning to the proof, we have:

```
    from $hypotheses$
        ⋮
$\beta$    from $c' \in containers(cid).contents$
            ⋮
k          $container(cid).type = \mathrm{LINER}$                                            ???
j          $cid \in \mathsf{dom}\ containers$                                                 ???
i          $container(cid).type = \mathrm{LINER} \Rightarrow$
             $containers(cid).contents \neq \mathsf{nil} \wedge$
             $\forall c1 \in containers(cid).contents \cdot$
               $containers(c1).type = \mathrm{PACKAGE}$
                                                     $\wedge$-E, $\forall$-E inv-$ContainerMap$, j
g          $\forall c1 \in containers(cid).contents \cdot$
             $containers(c1).type = \mathrm{PACKAGE}$                          $\Rightarrow$-E-left, i, k
f          $containers(c').type = \mathrm{PACKAGE}$                                        $\forall$-E, g
e          $c' \in \mathsf{dom}\ containers$                                                  ???
d          $containers(c').type = \mathrm{PACKAGE} \Rightarrow$
             $containers(c').material \neq \mathsf{nil}$          $\wedge$-E, $\forall$-E inv-$ContainerMap$, e
c          $containers(c').material \neq \mathsf{nil}$                          $\Rightarrow$-E-left, d, f
a          $gather(containers, c') = \{containers(c').material\}$            defn of $gather$
b          $containers(c').material \neq \mathrm{LIQUOR}$                                     ???
        infer $\mathrm{LIQUOR} \notin gather(containers, c')$                      Lemma1, a, b
$\alpha$    $\forall c' \in container(cid).contents \cdot \mathrm{LIQUOR} \notin gather(containers, c')$
                                                                              $\forall$-I, $\beta$
        ⋮
    infer $\mathrm{LIQUOR} \notin gather(containers, cid)$
```

The fact that *cid* indicates a liner is guaranteed by the modified precondition which is a hypothesis of the conjecture. It remains to establish that *cid* and c' are both in the domain of the *containers* mapping. In the case of *cid*, this is guaranteed by the fourth hypothesis of conjecture. In the case of c' it is guaranteed by the the invariant on *containers*, because c' is contained in the container identified by *cid*.

Finally, the last conjunct of *pre-COMPACTION* ensures that packages do not contain liquor. This allows completion of the subproof (and numbering of the lines) as follows:

	from $hypotheses$	
	$\vdots$	
β	from $c' \in containers(cid).contents$	
$\beta.1$	$container(cid).contents \neq \mathsf{nil} \Rightarrow$ $containers(cid).contents \subseteq \mathsf{dom}\ containers$	$\wedge$-E, $\forall$-E inv-$ContainerMap$, h4
$\beta.2$	$containers(cid).contents \subseteq \mathsf{dom}\ containers$	$\Rightarrow$-E-left, $\beta.1$, pre-$COMPACTION$
$\beta.3$	$container(cid).type = \mathrm{LINER}$	$\wedge$-E, pre-$COMPACTION$
$\beta.4$	$container(cid).type = \mathrm{LINER} \Rightarrow$ $containers(cid).contents \neq \mathsf{nil} \wedge$ $\forall c1 \in containers(cid).contents \cdot$ $containers(c1).type = \mathrm{PACKAGE}$	$\wedge$-E, $\forall$-E inv-$ContainerMap$, h4
$\beta.5$	$\forall c1 \in containers(cid).contents \cdot$ $containers(c1).type = \mathrm{PACKAGE}$	$\Rightarrow$-E-left, $\beta.4$, $\beta.3$
$\beta.6$	$containers(c').type = \mathrm{PACKAGE}$	$\forall$-E, $\beta.5$
$\beta.7$	$c' \in \mathsf{dom}\ containers$	$\beta.2$, subset
$\beta.8$	$containers(c').type = \mathrm{PACKAGE} \Rightarrow$ $containers(c').material \neq \mathsf{nil}$	$\wedge$-E, $\forall$-E inv-$ContainerMap$, $\beta.7$
$\beta.9$	$containers(c').material \neq \mathsf{nil}$	$\Rightarrow$-E-left, $\beta.8$, $\beta.6$
$\beta.10$	$gather(containers, c') = \{containers(c').material\}$	defn of $gather$
$\beta.11$	$containers(c').material \neq \mathrm{LIQUOR}$	$\forall$-E, pre-$COMPACTION$, β.h1
	infer $\mathrm{LIQUOR} \notin gather(containers, c')$	Lemma1, $\beta.10$, $\beta.11$
α	$\forall c' \in container(cid).contents \cdot \mathrm{LIQUOR} \notin gather(containers, c')$	$\forall$-I, β
	$\vdots$	
	infer $\mathrm{LIQUOR} \notin gather(containers, cid)$	

The remainder of the proof is left as an exercise. The crucial point is the expansion of *gather* from the overall conclusion.

Remarks

The flaw in the compaction operation which admitted compaction of liquor was a consequence of the specification relying on the fact that the container identified by *cid* would be a liner. There was a check in the precondition that *cid* was in the compaction phase, but there is no formal link between the compaction phase and the kind of containers which appear in the input. The link is initially present (the init clause in the state definition sets the expected input type to LINER for the compaction phase), but the compaction operation cannot rely on this still holding at the time it is applied.

When this discussion arose in the inspection of the full specification in the BNFL project, it was argued that there were no operations capable of modifying the expected input types of phases. This relies on an argument that, starting from the initial state, there are no reachable states in which anything other than a liner can be accepted into compaction. Thus, the argument relied on the initial state and the operations to maintain the property, rather than having the property stated explicitly in the invariant: the property was emergent, rather than being an integral part of the model. The risk associated with using this approach is that future modifications to the model may fail to respect the emergent property because it is not documented anywhere in the model. Recording the property in the invariant ensures that future modifications respect it because they must meet their satisfiability proof obligations.

5 Issues raised by the study

This section brings together evidence from the small study just presented and the full tracking manager project on which it was based, to raise a number of issues which the authors feel are applicable anywhere formal modelling is to be used.

5.1 Review cycle

The full tracking manager project divided the phases of specification and proof completely: first deriving a specification from the informal requirements document and, having reviewed this and confirmed that it was satisfactory, proceeding to the proof stage. Furthermore, the first review of the formal specification was conducted as an *inspection* at the stage where the complete specification was available. In the event it was only found possible to review the state in the first inspection and a second inspection was scheduled which reviewed the operation descriptions.

It is clear that a number of the issues raised during the proof work could have been determined earlier had extra appropriate reviews been scheduled. In particular, it seems that it would be constructive from all points of view to have a formal inspection at the stage where the system state has been defined and to ensure that this inspection is attended by people who are expert in proof matters. (It should also be anticipated that this inspection will uncover enough alternative suggestions that at least one revision cycle with re-inspection should be allowed for in the schedule.) It is important to realise that simplifications of the state at an early stage can economise not only on the effort in specifying individual operations but can, more importantly, have a major impact on the effort required to complete satisfiability proofs etc. Although not undertaken in this project, other experiences suggest that similar observations could be made about implementation proofs.

The inclusion of safety-related properties in invariants means that their proof is part of the satisfiability obligation. A change to the specification (state space, operation definitions or invariant) would necessitate re-discharging the obligations on affected operations, thus ensuring that safety is re-assessed on each change and reissue of the specification. To take advantage of this, it is worth setting up an inspection process which concentrates on discharging satisfiability proof obligations at a suitable level of rigour. Further experience is needed to measure how cost-effective such an approach would be. The specification of this system, possibly in a revised and more general form, could form a useful basis for such an experiment.

5.2 Scope of system

There is a class of computer systems which can be regarded as "closed world" systems. Such systems compute a neat mathematical function and their specification can easily be documented in terms of pre/post conditions which say all that is required for safe execution. There is another class of systems where the overall requirements should actually be stated in terms of the connection between what goes on in the computer and what goes on in the physical world: *controlling* the movement of nuclear material would clearly fall into this category. The tracking manager systems which we were asked to specify somehow or another tried to avoid the overall linking with reality by saying that it is an advisory system which would be employed to check functions determined in other ways. In spite of this, one of the conjectures which was to be considered was informally termed "*LIQUOR* cannot be compacted". It is clear that there is here some danger of misunderstanding about what can actually be proved. The tracking manager which was specified cannot compact anything –*LIQUOR* or otherwise– nor can it prevent such compaction taking place. It is important to emphasize that the result of proof exercises conducted on a formal model of a controlling system does not by itself establish safe function of the overall factory site.

5.3 Tools

In the tracking manager project, the full formal specification was created with the aid of Adelard SpecBox VDM tools and later checked with the aid of the IFAD VDM-SL Toolbox[2]. Both of these tools offered considerable help to the specifier and in particular the latter was successful in removing a number of type errors in the specification. It is clear that it would be a waste of effort to begin undertaking proofs –at whatever level of rigour– before such type errors are eradicated by use of appropriate tools.

However, for the purposes of the proof exercise the available tools offered very little support. It was not felt that any particular proof tool was appropriate for the range of proof styles which have been employed in the study. The proofs in this paper have therefore been constructed with no other support than a text editor and the LaTeX formatting system. This clearly makes them vulnerable to sources of inconsistency. It is, for example, possible that the statement of a lemma above has been erroneously copied and, however formal the proof is, the lemma will not match its alleged applications.

It is difficult to see how a theorem proving system can offer a significant degree of extra security except for the very formal proofs but this is clearly a topic which justifies further research.

5.4 Genericity and proofs

The present specification describes a specific demonstrator architecture. This is witnessed by the use of operations specific to particular phases, and the use of the initialisation clause to set up a phase structure. Yet some parts of the specification are clearly generic: the container packing and unpacking operations for example. The complex and large-scale task of proving properties about each tracking manager application (which could well be different in each plant) would be eased if more general properties of the generic tracking architecture were proved separately. This implies a modular specification, with a parametric module giving the model of the generic architecture, and its instantiation in the demonstrator. The authors feel that the safety case

[2] The reduced specification in this paper was developed with the aid of the IFAD VDM-SL Toolbox only.

for each tracking manager application could be easier to construct if based on such a generic model and would suggest this as a next step in research.

One area in which the tracking manager system has been made generic is that the phase structure is not fixed by the state itself but is determined by initialization of *System*. One could question whether the genericity so produced is in fact the area where change is most likely: one could, for instance, envisage the sorts of containers as being more likely to change than the phases through which containers are processed. Leaving aside the specification issue of whether the application of genericity is even across the system, it is more interesting here to investigate the impact on the proof work of such genericity as has been included. As indicated in the proofs, the way in which the generic system has been instantiated to a particular phase pattern by means of initialisation made it unnecessarily difficult to prove a number of desirable results. Earlier work of the authors [FJ90, Fit91, Fit92] has, however, suggested that there is little point in generality in specifications unless the level is so chosen that proofs about the general system particularize to subsequent instantiations. In the case in hand, one would wish to be convinced that there were useful general theorems about the generic phase system which lent themselves to easy understanding in any particular instantiation in that generic system. Indeed, the authors feel that the proof work reported here would have uncovered fewer errors if the specification had been more biased to the specific demonstrator. The work on the demonstrator architecture has not sought to identify such general results, but there would appear to be scope for considerable research into the area of proofs about generic systems.

5.5 Testing as a way of detecting problems

A number of problems have been detected in the specification during the attempt to construct the proofs contained in this document. The authors suggest that many of these problems would not have been detected by animation of the specification based on testing. This in no way questions the overall value of tools which can perform simple execution style checks on a specification: such checks can frequently detect errors before one starts the laborious effort of proof. Indeed, where a property appears not to hold, it may be less costly to come up with a test case which serves as a counter-example than to initiate the process of proof, as in the phase entry case above. However, testing often exercises those parts of a specification which one expects to function rather than detecting the unexpected gaps in the specification by conducting proofs about universal properties. An obvious example of this in the work above is the proof about non-compaction of *LIQUOR*. It would have been easy for somebody familiar with the intent of the system to set up tests showing the attempt to compact a *LINER* which contained or did not contain *LIQUOR* but the observation which is detected in the proof attempt is precisely that it is the derivation of the assumption that the *Container* is or is not a *LINER* which is not clearly established by the mechanism of instantiating the generic specification. In the longer term, one can envisage automated test case generation tools which make some contribution to the identification of pathological test cases.

6 Conclusions

- This report has illustrated, though a compact version of a larger specification, the use made of proof at various levels of rigour in the analysis of the larger formal model of an industrial system.

- Fully formal proof has its place, but we have stressed the use of less detailed proofs as guides to structuring the arguments which should take place during validation and review of a system model.
- In the commercial application of formal modelling, it may well be desirable to minimise the size of the skill base required for successful application of formal techniques, but the experience gained on this study leads the authors to the view that a knowledge of the structure and process of formal proof is desirable in teams undertaking this kind of analysis in future.
- Proof should play a role in the early stages of formal modelling, as part of a process of incremental specification development. This would allow the outcome of a proof study to influence the overall design of a specification, affecting issues such as specification structure, genericity and other "tradeoffs" between alternative formal models.

Acknowledgements

The tracking manager study took place as part of the *Research Study into the use of Computer-based Tracking Systems as valuable support to Safety Cases in Nuclear Power Technology*, a collaboration of Manchester informatics Limited, BNFL (Engineering) and Adelard under the UK Health and Safety Executive's Nuclear Safety Research Programme. The authors are especially grateful to Martyn Spink for composing the original tracking manager model, Bill Neary and Paul Vlissidis, then of BNFL Engineering, for their valuable domain expertise and to Ian Cottam for work in managing the project. JSF gladly acknowledges the support of the Engineering and Physical Science Research Council. CBJ acknowledges the support to his research by grants from the EPSRC and the Royal Society. Finally, both authors thank Juan Bicarregui and Peter Gorm Larsen for their helpful comments on earlier drafts of this paper.

References

[BFL+94] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[BFM89] Robin Bloomfield, Peter Froome, and Brian Monahan. SpecBox: A toolkit for BSI-VDM. *SafetyNet*, (5):4–7, 1989.

[ELL94] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, September 1994.

[Fit91] J. S. Fitzgerald. *Modularity in Model-Oriented Formal Specifications and its Interaction with Formal Reasoning.* PhD thesis, Dept. of Computer Science, University of Manchester, UK, 1991. Available as Technical Report UMCS 91-11-2 from Dept. of Computer Science, University of Manchester, UK.

[Fit92] J. S. Fitzgerald. Reasoning about a modular model-oriented formal specification. In David J. Harper and Moira C. Norrie, editors, *Proc. Intl. Workshop on Specifications of Database Systems, University of Glasgow 1991*, Workshops in Computer Science. Springer-Verlag, 1992.

[FJ90] J.S. Fitzgerald and C.B. Jones. Modularizing the Formal Description of a Database System. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International(UK), second edition, 1990. ISBN 0-13-880733-7. Out of print. Available by `ftp` from `ftp.cs.man.ac.uk` in directory `pub/cbj` in file `ssdvdm.ps.gz`.

[LHP+96] P. G. Larsen, B. S. Hansen, H. Brunn N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.