SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-7

A Precise Examination of the Behaviour of Merlin Process Models

Kelvin Ross and Peter Lindsay

September 1994

Phone: +61 7 365 1003
Fax: +61 7 365 1533

A Precise Examination of the Behaviour of Merlin Process Models

Kelvin J. Ross & Peter A. Lindsay
kjross @ cs.uq.oz.au & pal @ cs.uq.oz.au
Software Verification Research Centre
Department of Computer Science
The University of Queensland
AUSTRALIA

September 23, 1994

Abstract

For large software developments projects, process modelling is an important technique for guiding and monitoring the use of development tools. This paper explores the addition of "behavioural properties" to process models as a mechanism for reasoning about the status of a software development as it evolves. The process model is translated into VDM and standard VDM verification techniques are applied to show, among other things, that the behavioural properties are maintained by the process and that tools are invoked from the process model only when their preconditions are satisfied. These ideas are illustrated on a small case study. The process model is implemented in Merlin.

Keywords: process modelling, formal methods, VDM, configuration management

1 Introduction

The work reported in this paper has been motivated by the study of Software Configuration Management (SCM) requirements for systems supporting the use of Formal Methods. From the SCM perspective, a formal software development can be regarded as a configuration of its low-level components, such as specification components, proof obligations and formal proofs. The correctness and completeness of the software development configuration can be defined via configuration consistency checks [RL93]. In a large software development, in which components are frequently undergoing change, it can be difficult to maintain consistency across a whole configuration. Process support is part of the solution: a process model guides and controls changes to the development.

Of the different approaches to software process modelling, we concentrate here on the state-transition approach, in which development activities define transitions between various "states" of development components [DG90, PSW92, PS92]. We propose an approach in which the state-transition process model is annotated with "behavioural properties" which say how process states relate to properties of the underlying development configuration. We discuss verification techniques for showing that a process model satisfies its specified behavioural properties.

A small case study is used to illustrate the main points. The case study concerns the specification development phase, in which a specification is formulated and its "correctness" and "completeness" are verified by discharging proof obligations. Because of space limitations in this paper, the configuration is considered at a coarse level of granularity, where the atomic configuration items are the specification and the mathematical theory in which the proof obligations get discharged. We assume there are tools available for editing and checking the specification and the theory, and for bringing the theory into step with the specification (e.g. by modifying its axioms so that they reflect the definitions of the evolving specification). We define a process model which controls invocation of these tools.

The process model is defined graphically using statecharts and then translated into VDM [Jon90, VDM93]. The process model was implemented in Merlin [PSW92, PS92].

2 A Software Development Scenario

2.1 The Case Study

The case study concerns one phase in a formal development using VDM – namely, discharging the proof obligations associated with a formal specification. It is a cut-down version of a larger case study in [RL93] in which a VDM specification and its associated mathematical theory and proof obligations are modelled as a configuration of formal entities. The verification criteria required by the VDM methodology for these entities were defined as consistency checks on the configuration. In [RL93] the definition of the structure of the configuration and the configuration consistency relationships was described at a finer level of granularity. The techniques presented here apply equally well to the finer-grained system model, but space limitations means the results can only be presented for the coarser level.

The configuration consistency problem arises from the fact that the specification and the theory can be edited independently, and thus may get out of step. We shall assume there are tools for checking the syntactic correctness of the specification, for updating the theory to bring it back into step with the specification, and for checking that all the proof obligations have been correctly discharged. We have in mind a simplified and idealized version of the `mural` formal development support system [JJLM91]. Let us also suppose that some of the tools have preconditions to their use: e.g. the theory can be updated only if the specification is syntactically correct.

In Section 3 below we define a process model for controlling the invocation of tools. We show that semantic information concerning the configuration can be deduced from the state of the process, including (for certain states) the consistency and completeness of the configuration.

2.2 Configuration Structure

A development configuration is a representation of different *software configuration items* (SCIs), which are the components that fit together to describe the configured product. Each SCI is either an *atomic* or a *composite SCI*. Atomic SCIs are the finest grained components represented within a configuration. From an SCM perspective, an atomic SCI is a unified block of data: its internal structure is not relevant to the SCM support. Composite SCIs, on the other hand, are defined as a composition of subcomponent SCIs.

For the purposes of this case study, the development configuration is a composition of only two subcomponents: namely a specification SCI and a theory SCI, both of which are atomic (see Figure 1).

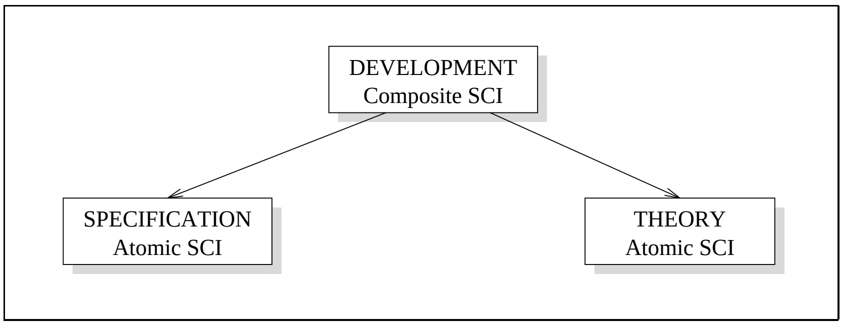

Figure 1: A VDM development configuration decomposed into atomic and composite SCIs

The specification of the structure of each SCI is called a *configuration template*. The collection of all SCI templates for a development configuration is called a (configuration) *system model*. Each SCI instance represented in a configuration conforms to a template described in the system model for the development configuration.

In this paper we use VDM type definitions to specify the configuration templates of the system model (see Figure 2). The template for an atomic SCI is a primitive (unanalysed) type, indicating the overall specification has no knowledge of an atomic SCI's internal structure. A configuration template is described by a VDM data type. In this case, a *DEVELOPMENT* is a simple composite (record) type with two fields: one for the specification and one for the theory.

$$\begin{aligned} \mathit{DEVELOPMENT} :: \ & \mathit{spec}\colon \mathit{SPECIFICATION} \\ & \mathit{theory}\colon \mathit{THEORY} \end{aligned}$$

$$\mathit{THEORY} = \ldots\ldots.$$

$$\mathit{SPECIFICATION} = \ldots\ldots.$$

Figure 2: The system model which describes the structure of the SCIs.

2.3 Configuration Consistency Relationships

Configuration consistency is defined in terms of relationships between SCIs. There are two basic kinds of configuration consistency relationships: *derived relationships*, which are defined in terms of other relationships; and *primitive (blackbox) relationships*, which can be checked or established by tools. In this paper both primitive and derived relationships are modelled in VDM as Boolean-valued functions.

For the case study, let us assume there are four primitive configuration consistency relationships (see Figure 3) [1]:

1. *spec_checked*(*spec*) which indicates that *spec* is syntactically and type correct;
2. *axioms_substantiated*(*spec*, *theory*) which indicates that the axioms of *theory* are only those which can be derived from *spec* (cf. Appendix A);
3. *obligations_stated*(*spec*, *theory*) which indicates that all the proof obligations of *spec* are recorded as conjectures or theorems within *theory*; and
4. *theory_checked*(*spec*) which indicates that all conjectures in *theory* have been proven.

$$\mathit{spec\_checked}\colon \mathit{SPECIFICATION} \to \mathbf{B}$$

$$\mathit{axioms\_substantiated}\colon \mathit{SPECIFICATION} \times \mathit{THEORY} \to \mathbf{B}$$

$$\mathit{obligations\_stated}\colon \mathit{SPECIFICATION} \times \mathit{THEORY} \to \mathbf{B}$$

$$\mathit{theory\_checked}\colon \mathit{THEORY} \to \mathbf{B}$$

Figure 3: Primitive relationships between SCIs have black box definitions.

A derived relationship *is_correct_development* checks that the theory part of a development correctly discharges the specification part (see Figure 4).

[1]See [RL93] for formal definitions.

$$is\_correct\_development : DEVELOPMENT \to \mathbf{B}$$

$$\begin{aligned} is\_correct\_development(dev) \;\triangleq\;& spec\_checked(dev.spec) \wedge \\ & axioms\_substantiated(dev.spec, dev.theory) \wedge \\ & obligations\_stated(dev.spec, dev.theory) \wedge \\ & theory\_checked(dev.theory) \end{aligned}$$

Figure 4: Derived relationships provide a complete description of a relationship between SCIs.

2.4 Development Tools

For the case study, let us suppose the following tools are available:

EDIT_SPEC, which allows the user to interactively edit the contents of the specification;

CHECK_SPEC, which determines if the specification is syntax and type correct;

EDIT_THEORY, which allows the user to edit the theory contents, e.g. to edit a proof;

CHECK_THEORY, which determines if all the conjectures in a theory have been proven; and

UPDATE, which brings the theory into step with the specification. The *UPDATE* tool has a precondition that the specification passed as input must be syntax and type correct before the tool can be invoked.

2.4.1 Tool Specifications

The effects of tools on SCIs will be defined via *tool specifications* consisting of a signature, a precondition, and a postcondition.

A tool signature defines the name of the tool and describes the input and result parameters. The input parameters may contain SCIs and/or user input. The results of a tool may represent either SCIs, which, for instance, may be used to replace an SCI in the current configuration, or other information which is passed on to the user interface or used for process control.

The precondition defines the necessary conditions which must exist before the tool can be invoked. The precondition is defined as a predicate over the input parameters. If there is no explicit precondition for a tool it has an implicit precondition of true: i.e. it is permissible to invoke the tool given any input parameters that satisfy the parameter typing criteria.

The postcondition describes the outcome of the tool in terms of a predicate defined over the input and result parameters. The execution of the tool, when the precondition has been satisfied, must provide a result based on the input parameters which meets the postcondition. Note that the definition of tools as described in this approach is purely functional: tools simply derive results from the input parameters.

Figure 5 provides partial specifications for the tools used in the case study. For the purposes of SCM, the pre- and postconditions describe how configuration consistency relationships are affected (only): they are simply one part of the complete specification of the tool. For example, a more complete specification of *EDIT_SPEC* would describe the relationship between the internal structure of *spec*, *changes* and *result*.

The *EDIT_SPEC* and *EDIT_THEORY* tools represent editing activities: they produce a new SCI based on editing a given SCI with a set of changes passed from the user interface.

The *CHECK_SPEC* and *CHECK_THEORY* tools are checking tools whose result will be used to determine flow of control in the process model. The specification says that *CHECK_SPEC*(*spec*)

$EDIT\_SPEC\ (spec\colon SPECIFICATION, changes\colon \Delta\text{-}SPECIFICATION)$
$\qquad result\colon SPECIFICATION$

post true

$CHECK\_SPEC\ (spec\colon SPECIFICATION)$
$\qquad result\colon \mathbf{B}$

post $result \ \Rightarrow\ spec\_checked(spec)$

$EDIT\_THEORY\ (theory\colon THEORY, changes\colon \Delta\text{-}THEORY)$
$\qquad result\colon THEORY$

post true

$CHECK\_THEORY\ (theory\colon THEORY)$
$\qquad result\colon \mathbf{B}$

post $result \ \Rightarrow\ theory\_checked(theory)$

$UPDATE\ (spec\colon SPECIFICATION, theory\colon THEORY)$
$\qquad result\colon THEORY$

pre $spec\_checked(spec)$

post $axioms\_substantiated(spec, result) \wedge obligations\_stated(spec, result)$

Figure 5: Tools used in the evolution of specifications and theories.

returns ‘true’ only if *spec* is syntax and type correct. *CHECK_THEORY*(*theory*) returns ‘true’ only if all conjectures within *theory* have been proven.

UPDATE is a derivation tool which brings a theory into step with a specification. The current theory is passed as input to *UPDATE*, so that as much previous work can be used as possible.[2] On completion of the tool invocation, the specification and resulting theory satisfy the *axioms_substantiated* and *obligations_stated* consistency relationships.

Most development tools fall into one of the three classes illustrated above: editing, derivation or checking.

2.4.2 Tool Theorems

Tool theorems are assumptions that are made about the effect of tools, derived directly from tool specifications (see Figure 6). For example, the `CHECK_SPEC-defn` says that if *CHECK_SPEC*(s) returns ‘true’ then s is syntax and type correct. These theorems will be used in the verification in Section 5 below. Note that the tool specifications for the editing tools (*EDIT_SPEC* and *EDIT_THEORY*) do not give rise to any interesting theorems.

$$\boxed{\text{CHECK\_SPEC-defn}}\ \frac{s\colon SPECIFICATION}{CHECK\_SPEC(s)\ \Rightarrow\ spec\_checked(s)}$$

$$\boxed{\text{UPDATE-defn}}\ \frac{\begin{array}{c} s\colon SPECIFICATION, \\ t\colon THEORY, \\ spec\_checked(s) \end{array}}{\begin{array}{c} axioms\_substantiated(s, UPDATE(s, t)) \wedge \\ obligations\_stated(s, UPDATE(s, t) \end{array}}$$

$$\boxed{\text{CHECK\_THEORY-defn}}\ \frac{t\colon THEORY}{CHECK\_THEORY(t)\ \Rightarrow\ theory\_checked(t)}$$

Figure 6: Theorems concerning the effects of development tools.

3 A Development Process

A process model will be used to manage the invocation of tools and to track the consistency of a configuration. The process model associates *status* information with certain SCIs in the configuration. Our innovation is to add *behavioural properties* which are statements which formally express relationships between the status of SCIs and properties of the configuration. In section 5 we introduce techniques for verifying that the process model achieves its goals. In Appendix B the process model in the following case study is implemented, with very little modification, in Merlin.

3.1 Process State

The process state is defined by annotating SCIs in the system model with status information. For the case study we defined the process state to consist of a specification and a theory:

- The status of a specification is either CHECKED or UNCHECKED. Intuitively, a status of CHECKED indicates that the specification has been syntax and type checked.

[2] `mural` falls short of this ideal, because although it generates axioms corresponding to the new specification, it does not delete axioms corresponding to the old specification, and hence does not satisfy *axioms_substantiated*(*spec*, *result*).

- The status of a theory is one of the following: Raw, Raw-Chkd, In_Step and Complete. Intuitively, a status of In_Step indicates that the theory is instep with the specification and a status of Complete in addition indicates that all conjectures have been proven.

In this paper the states of the process are described using a VDM state description (see Figure 7).

SPEC_STATUS = Unchecked | Checked
THEORY_STATUS = Raw | Raw-Chkd | In_Step | Complete

```
state Process_State of
        spec: SPECIFICATION
        theory: THEORY
        spec_status: SPEC_STATUS
        theory_status: THEORY_STATUS
end
```

Figure 7: The process state consists of both SCIs and corresponding statuses.

A simplified definition of the process state has been used here (Section 3.1) to simplify the exposition. More generally, the process state can be extended to support multiple developments. See Section 6 for more details.

3.2 Process Model

The dynamic behaviour of the process will be described using a statechart-like approach. The basis for the statechart approach is abstracted from the graphical development method for Merlin process models [Jun94], based on earlier development of statecharts [Har87].

A statechart is provided to define the transition for each document status. In this case study there is a statechart for both *spec_status* and *theory_status* (see Figure 8).

The main features of each statechart are *nodes*, *transitions* and *initialisation*. Nodes are represented for each status value. For example, the *spec_status* statechart has two nodes representing its two status values, Checked and Unchecked.

Transitions update status information as the development evolves. They are represented by a link between statechart nodes. We have annotated transitions with numbered labels for later cross-referencing. There are two types of transitions:

Tool activity transitions: which identify updates to the status information as a result of tool invocations. These transitions are further divided into two types: unconditional and conditional. Unconditional transitions have a single resulting status value, whereas conditional transitions may result in one of two statuses depending on the outcome of the tool applied.

Unconditional transitions include an *event/action* pair, which respectively designate the circumstances under which the transition is activated and the operation on the SCIs that occurs. For example, transition 5 in the *spec_status* statechart defines a transition that is enabled when the `do_spec_edit` event occurs. This may be sent from the user, via a selection from a menu for instance. The activation of the operation then assigns the *spec* component to the result of applying the *EDIT_SPEC* tool on the current value of *spec*, with changes provided through user input.

Conditional transitions are similar to unconditional transitions, with the addition of an implicit intermediate node. From this intermediate node the transition branches according to the result of the tool applied. For example, transition 7 of the *spec_status* statechart applies the

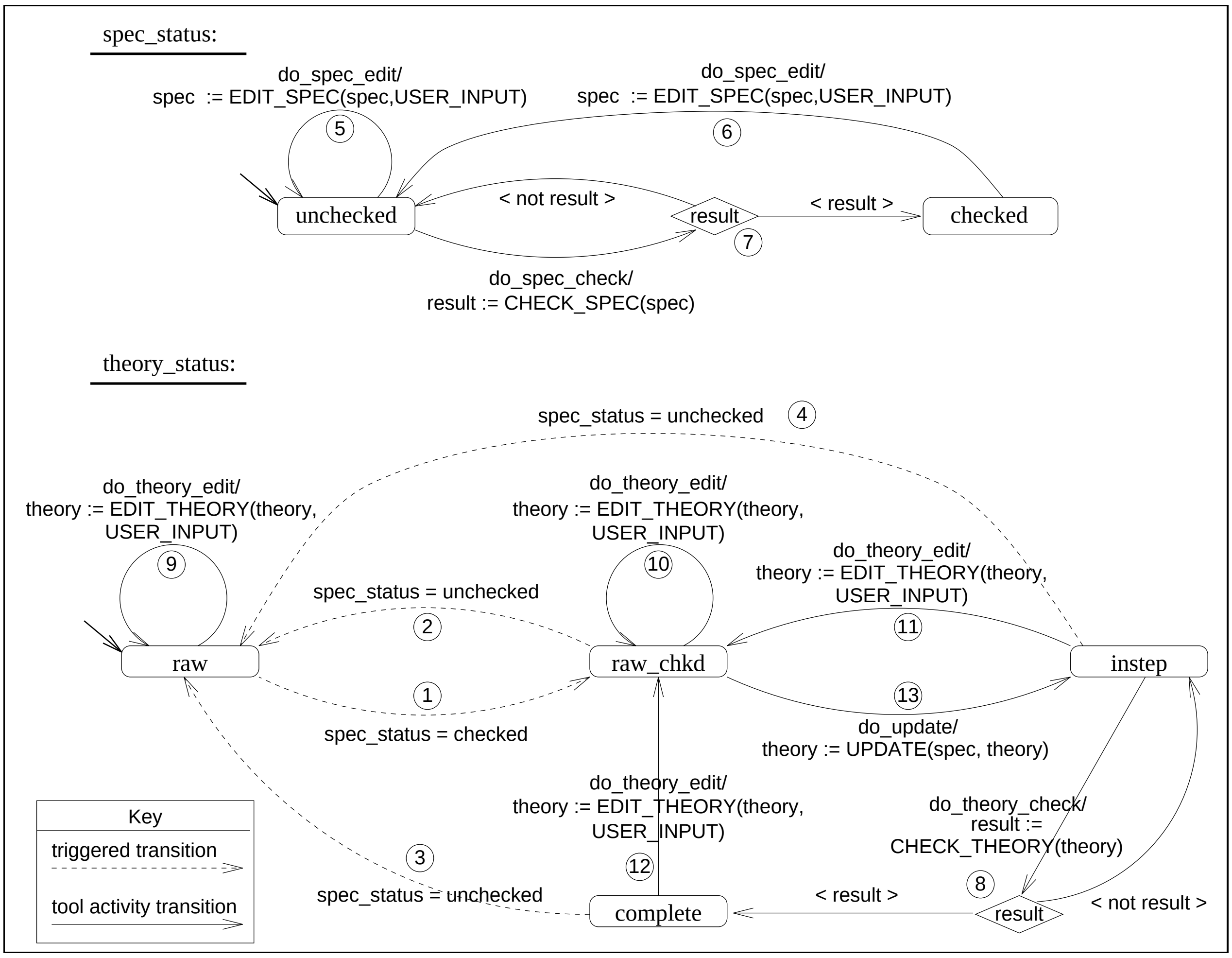

Figure 8: Specification and theory process transition systems.

CHECK_SPEC tool on the *spec* component. If this tool succeeds (i.e. returns 'true') then the transition updates *spec_status* to CHECKED; otherwise *spec_status* remains set to UNCHECKED.

Triggered transitions: which identify updates to status information that are applied automatically whenever a predetermined *trigger* condition is satisfied. In this paper triggered transitions are indicated by a dashed line. The transition is labelled with the condition under which the transition will occur.

Consider the triggered transition between COMPLETE and RAW in the *theory_status* statechart (transition 3). This transition will be applied automatically whenever *theory_status* = COMPLETE and the condition *spec_status* = UNCHECKED is satisfied. The transition results in the *theory_status* state being updated to RAW. In this case study the transition will typically occur each time the specification is edited when *theory_status* = COMPLETE because *spec_status* will become UNCHECKED.

Initialisation defines the status of SCIs when the process first commences. This is identified by a short arrow pointing to the appropriate node. For this case study, initial values of *spec_status* and *theory_status* are UNCHECKED and RAW respectively.

3.3 Behavioural Properties

The process model describes how the development evolves as the result of applying development tools and puts constraints on their use. The goal of the process is to construct a *DEVELOPMENT* SCI which satisfies the *is_correct_development* relationship. We claim that this consistency requirement is reached whenever the process model reaches a state where *spec_status* = CHECKED and *theory_status* = COMPLETE. This behavioural requirement can be formally described as a condition defined using information in the process state and configuration consistency relationships (see Figure 9).

$$\begin{aligned} &\mathit{spec\_status} = \textsc{Checked} \wedge \mathit{theory\_status} = \textsc{Complete} \\ &\qquad \Rightarrow \ \mathit{is\_correct\_development}(\mathsf{mk}\text{-}\mathit{DEVELOPMENT}(\mathit{spec}, \mathit{theory})) \end{aligned}$$

Figure 9: General requirement of the process behaviour.

A more detailed observation of the behaviour of the process would indicate consistency relationships for other states of the process, for example:

- *spec_status* is CHECKED only if *spec* is syntax and type correct;
- *theory_status* is COMPLETE only if all conjectures in *theory* have been proven; and
- *theory* is in step with *spec* if *spec_status* is CHECKED and *theory_status* is IN_STEP or COMPLETE.

These requirements have been collected together in the definition *Has_OK_Statuses* (see Figure 10). Note that the general requirement (in Figure 9) is a logical consequence of these requirements.

We introduce a new keyword '**behav**' to indicate *behavioural properties* of process models. The behavioural property records properties which are maintained by the process model. In section 5 below we define proof obligations which ensure that these properties are maintained when the process model is enacted.

The behavioural properties allow certain information about the dynamic behaviour of the process to be deduced without having to re-analyse the whole model. For example, whenever *spec_status* is CHECKED then *spec* is guaranteed to be syntax and type correct: there is no need to (re)invoke the *CHECK_SPEC* tool before applying *UPDATE*. Configuration consistency relationships can thus be monitored incrementally rather than having to be re-evaluated each time.

$$\mathsf{behav\ mk\text{-}}\mathit{Process\_State}(\mathit{spec}, \mathit{theory}, \mathit{spec\_status}, \mathit{theory\_status}) \ \triangleq$$
$$\mathit{Has\_OK\_Statuses}(\mathit{spec}, \mathit{theory}, \mathit{spec\_status}, \mathit{theory\_status})$$

$$\begin{aligned}
&\mathit{Has\_OK\_Statuses}(\mathit{spec}, \mathit{theory}, \mathit{spec\_status}, \mathit{theory\_status}) \ \triangleq \\
&\quad \mathit{spec\_status} = \textsc{Checked} \wedge (\mathit{theory\_status} = \textsc{Complete} \vee \\
&\qquad\qquad\qquad\qquad\qquad \mathit{theory\_status} = \textsc{In\_Step}) \\
&\qquad \Rightarrow \ \mathit{axioms\_substantiated}(\mathit{spec}, \mathit{theory}) \wedge \mathit{obligations\_stated}(\mathit{spec}, \mathit{theory}) \\
&\quad \wedge \mathit{spec\_status} = \textsc{Checked} \\
&\qquad \Rightarrow \ \mathit{spec\_checked}(\mathit{spec}) \\
&\quad \wedge \mathit{theory\_status} = \textsc{Complete} \\
&\qquad \Rightarrow \ \mathit{theory\_checked}(\mathit{theory})
\end{aligned}$$

Figure 10: The behavioural property documents assumptions for various states of the process.

4 Translation into VDM

Most of the definitions presented in this paper have been provided in VDM, with exception of the statecharts. By including the translation of the statecharts into VDM, a complete VDM specification is provided. This section describes the translation of the statecharts.

The translation of statecharts provides two forms of definition: initialisation and operations. Derivation of the VDM initialisation condition (see Figure 11) is a relatively simple translation, and provides a condition which is intuitive from the statechart.

$$\mathsf{init}\ \mathit{pm} \ \triangleq \ \mathit{pm.spec\_status} = \textsc{Unchecked} \wedge \mathit{pm.theory\_status} = \textsc{Raw}$$

Figure 11: A VDM initialisation condition is simply translated from the statecharts.

The translation of transitions into VDM operation definitions is more complex. Triggered transitions are translated first. Triggered transitions are automatically invoked whenever the trigger condition is satisfied. These conditions are represented as the preconditions of the corresponding VDM operation definitions (see Figure 12).

In the statechart, triggered transitions take precedence over tool activity transitions: the latter can only occur if no triggered transition can be invoked. Let us call a state in which no trigger condition is satisfied a "stable state" (see Figure 13). We note in passing that the following theorem is a logical consequence of the definition of a stable state:

$$\boxed{\text{stable\_state-prop}} \ \frac{\mathit{spec\_status}\colon \mathit{SPEC\_STATUS},\ \ \mathit{theory\_status}\colon \mathit{THEORY\_STATUS}}{\mathit{stable\_state}(\mathit{spec\_status}, \mathit{theory\_status}) = (\mathit{spec\_status} = \textsc{Unchecked} \Leftrightarrow \mathit{theory\_status} = \textsc{Raw})}$$

The translation of tool activity transitions into VDM operations is now straight-forward (see Figure 14). The precondition defines the statuses in which the transition is invoked, along with the stable state requirement. The postcondition defines the effect of the corresponding tool invocation along with status updates. Conditional transition postconditions are defined by an *if-then-else* constructor.

Translation from statecharts (as described in Section 3.2) into VDM is a mechanical procedure. Tool support for this procedure would clearly be feasible.

$TRANS\text{-}1$
ext wr $theory\_status$
rd $spec\_status$
pre $theory\_status = \textsc{Raw} \wedge$
$spec\_status = \textsc{Checked}$
post $theory\_status = \textsc{Raw-Chkd}$

$TRANS\text{-}2$
ext wr $theory\_status$
rd $spec\_status$
pre $theory\_status = \textsc{Raw-Chkd} \wedge$
$spec\_status = \textsc{Unchecked}$
post $theory\_status = \textsc{Raw}$

$TRANS\text{-}3$
ext wr $theory\_status$
rd $spec\_status$
pre $theory\_status = \textsc{Complete} \wedge$
$spec\_status = \textsc{Unchecked}$
post $theory\_status = \textsc{Raw}$

$TRANS\text{-}4$
ext wr $theory\_status$
rd $spec\_status$
pre $theory\_status = \textsc{In\_Step} \wedge$
$spec\_status = \textsc{Unchecked}$
post $theory\_status = \textsc{Raw}$

Figure 12: Triggered transitions are translated into VDM operations.

$trigger\_state : SPEC\_STATUS \times THEORY\_STATUS \to \mathbf{B}$

$$\begin{aligned} &trigger\_state(spec\_status, theory\_status) \quad \triangleq \\ &\quad (theory\_status = \textsc{Raw} \wedge spec\_status = \textsc{Checked}) \vee \\ &\quad (theory\_status = \textsc{Raw-Chkd} \wedge spec\_status = \textsc{Unchecked}) \vee \\ &\quad (theory\_status = \textsc{Complete} \wedge spec\_status = \textsc{Unchecked}) \vee \\ &\quad (theory\_status = \textsc{In\_Step} \wedge spec\_status = \textsc{Unchecked}) \end{aligned}$$

$stable\_state : SPEC\_STATUS \times THEORY\_STATUS \to \mathbf{B}$

$$stable\_state(spec\_status, theory\_status) \quad \triangleq \quad \neg trigger\_state(spec\_status, theory\_status)$$

Figure 13: Trigger and stable state conditions are provided as VDM auxiliary functions.

```
$TRANS$-5 ($input$: $\Delta$-$SPECIFICATION$)
ext wr $spec\_status$, $spec$
    rd $theory\_status$
pre $spec\_status = \text{UNCHECKED} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post $spec = EDIT\_SPEC(\overleftarrow{spec}, input)$
     $\wedge\ spec\_status = \text{UNCHECKED}$
```

```
$TRANS$-6 ($input$: $\Delta$-$SPECIFICATION$)
ext wr $spec\_status$, $spec$
    rd $theory\_status$
pre $spec\_status = \text{CHECKED} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post $spec = EDIT\_SPEC(\overleftarrow{spec}, input)$
     $\wedge\ spec\_status = \text{UNCHECKED}$
```

```
$TRANS$-7
ext wr $spec\_status$
    rd $spec$, $theory\_status$
pre $spec\_status = \text{UNCHECKED} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post if $CHECK\_SPEC(spec)$
     then $spec\_status = \text{CHECKED}$
     else $spec\_status = \text{UNCHECKED}$
```

```
$TRANS$-8
ext wr $theory\_status$
    rd $theory$, $spec\_status$
pre $theory\_status = \text{IN\_STEP} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post if $CHECK\_THEORY(theory)$
     then $theory\_status = \text{COMPLETE}$
     else $theory\_status = \text{IN\_STEP}$
```

```
$TRANS$-9 ($input$: $\Delta$-$THEORY$)
ext wr $theory\_status$, $theory$
    rd $spec\_status$
pre $theory\_status = \text{RAW} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post $theory = EDIT\_THEORY(\overleftarrow{theory}, input)$
     $\wedge\ theory\_status = \text{RAW}$
```

```
$TRANS$-10 ($input$: $\Delta$-$THEORY$)
ext wr $theory\_status$, $theory$
    rd $spec\_status$
pre $theory\_status = \text{RAW-CHKD} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post $theory = EDIT\_THEORY(\overleftarrow{theory}, input)$
     $\wedge\ theory\_status = \text{RAW-CHKD}$
```

```
$TRANS$-11 ($input$: $\Delta$-$THEORY$)
ext wr $theory\_status$, $theory$
    rd $spec\_status$
pre $theory\_status = \text{IN\_STEP} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post $theory = EDIT\_THEORY(\overleftarrow{theory}, input)$
     $\wedge\ theory\_status = \text{RAW-CHKD}$
```

```
$TRANS$-12 ($input$: $\Delta$-$THEORY$)
ext wr $theory\_status$, $theory$
    rd $spec\_status$
pre $theory\_status = \text{COMPLETE} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post $theory = EDIT\_THEORY(\overleftarrow{theory}, input)$
     $\wedge\ theory\_status = \text{RAW-CHKD}$
```

```
$TRANS$-13
ext wr $theory\_status$, $theory$
    rd $spec\_status$, $spec$
pre $theory\_status = \text{RAW-CHKD} \wedge$
    $stable\_state(spec\_status, theory\_status)$
post $theory = UPDATE(spec, \overleftarrow{theory})$
     $\wedge\ theory\_status = \text{IN\_STEP}$
```

Figure 14: Tool activity transitions are translated into VDM operations.

5 Verification of Process Model

VDM provides facilities for formal verification [Jon90, BFL$^+$94] which check, amongst other properties, the consistency of VDM specifications. By translating the process model into a VDM representation we show that such checks can provide valuable feedback to the process model designer.

The following identifies some of the checks that are carried out on the process model through formal verification of the VDM representation:

1. Behavioural properties are maintained by the process;
2. Tool preconditions are satisfied each time a tool is invoked by the process; and
3. Tool activity transitions are invocable at some state representation.

In this section we give rigorous proofs of these properties. Appendix A gives a partial axiomatization of VDM specifications with behavioural properties, together with corresponding proof obligations: see [BFL$^+$94, LvK94] for more details.

5.1 Behaviour Maintained

This check indicates that behavioural properties are maintained through the dynamic behaviour of the process. This is done by showing behavioural properties are true initially and are preserved by all operations. See [LvK94] for a justification of this verification technique.

5.1.1 Maintained by Initialisation

The first part of the check is discharging that the initialisation satisfies the behavioural property. This is indicated by the `init-maint` proof obligation:

$$\boxed{\text{Process\_State-init-maint}}\ \frac{}{\begin{array}{c}\forall pm\colon \mathit{Process\_State}\cdot\\ pm.\mathit{spec\_status} = \textsc{Unchecked} \land pm.\mathit{theory\_status} = \textsc{Raw}\\ \Rightarrow\ \mathit{Has\_OK\_Statuses}(pm.\mathit{spec}, pm.\mathit{theory},\\ pm.\mathit{spec\_status}, pm.\mathit{theory\_status})\end{array}}$$

The proof of this property is quite simple as there are no assumptions for *spec* and *theory* when the specification status and theory status is UNCHECKED and RAW respectively.

5.1.2 Maintained by Operations

The second phase of checking the behavioural property is maintained involves showing that the property is preserved by all transitions. This is indicated by the `OP-maint` proof obligation for every operation definition `OP`. It involves proving that the postcondition implies the behavioural property in the after state, assuming the behavioural property and the precondition hold in the before state.

For example, the transition in which *UPDATE* is invoked requires satisfying the following requirement:

$$\boxed{\text{TRANS-13-maint}}\ \frac{\begin{array}{c}\mathsf{mk}\text{-}\mathit{Process\_State}(\mathit{spec}, \overleftarrow{\mathit{theory}}, \mathit{spec\_status}, \mathit{theory\_status})\colon \mathit{Process\_State},\\ \mathit{Has\_OK\_Statuses}(\mathit{spec}, \overleftarrow{\mathit{theory}}, \mathit{spec\_status}, \mathit{theory\_status}),\\ \overleftarrow{\mathit{theory\_status}} = \textsc{Raw-Chkd} \land \mathit{stable\_state}(\mathit{spec\_status}, \overleftarrow{\mathit{theory\_status}})\end{array}}{\begin{array}{c}\forall \mathit{theory}\colon \mathit{THEORY}, \mathit{theory\_status}\colon \mathit{THEORY\_STATUS}\cdot\\ (\mathit{theory} = \mathit{UPDATE}(\mathit{spec}, \overleftarrow{\mathit{theory}}) \land \mathit{theory\_status} = \textsc{In\_Step})\\ \Rightarrow\ \mathit{Has\_OK\_Statuses}(\mathit{spec}, \mathit{theory}, \mathit{spec\_status}, \mathit{theory\_status})\end{array}}$$

To satisfy this proof obligation it is sufficient to show

$$\mathit{Has\_OK\_Statuses}(\mathit{spec}, \mathit{UPDATE}(\mathit{spec}, \overleftarrow{\mathit{theory}}), \mathit{spec\_status}, \textsc{In\_Step}).$$

This in turn reduces to showing

$$\mathit{axioms\_substantiated}(\mathit{spec}, \mathit{UPDATE}(\mathit{spec}, \overleftarrow{\mathit{theory}}) \wedge \mathit{obligations\_stated}(\mathit{spec}, \mathit{UPDATE}(\mathit{spec}, \overleftarrow{\mathit{theory}}))$$

when $\mathit{spec\_status} = \textsc{Checked}$, and this in turn follows from the tool theorem `UPDATE-defn`.

5.2 Tool Preconditions Satisfied

This check determines whether preconditions to tools are satisfied when they are invoked by the process. Here, tool preconditions should be interpreted in a broad sense as meaning the assumptions that the user of that tool can make about the state of the configuration at the time the tool is invoked. The check is achieved by verifying that the postcondition of the corresponding process model operation(s) is a well-formed formula (i.e. the `OP-post-wff` proof obligation).

For example, the corresponding proof obligation for *TRANS*-13 is:

$$\boxed{\text{TRANS-13-post-wff}} \quad \frac{\begin{array}{c}\mathsf{mk}\text{-}\mathit{Process\_State}(\mathit{spec}, \overleftarrow{\mathit{theory}}, \mathit{spec\_status}, \mathit{theory\_status})\colon \mathit{Process\_State},\\ \mathsf{mk}\text{-}\mathit{Process\_State}(\mathit{spec}, \mathit{theory}, \mathit{spec\_status}, \mathit{theory\_status})\colon \mathit{Process\_State},\\ \mathit{Has\_OK\_Statuses}(\mathit{spec}, \overleftarrow{\mathit{theory}}, \mathit{spec\_status}, \mathit{theory\_status}),\\ \mathit{Has\_OK\_Statuses}(\mathit{spec}, \mathit{theory}, \mathit{spec\_status}, \mathit{theory\_status}),\\ \overleftarrow{\mathit{theory\_status}} = \textsc{Raw-Chkd} \wedge \mathit{stable\_state}(\mathit{spec\_status}, \overleftarrow{\mathit{theory\_status}})\end{array}}{(\mathit{theory} = \mathit{UPDATE}(\mathit{spec}, \overleftarrow{\mathit{theory}}) \wedge \mathit{theory\_status} = \textsc{In\_Step})\colon \mathbf{B}}$$

In order to show the formula is well-formed we must show, amongst other requirements, that the precondition to $\mathit{UPDATE}(\mathit{spec}, \overleftarrow{\mathit{theory}})$ is satisfied, i.e. $\mathit{spec\_checked}(\mathit{spec})$.

This follows from the hypothesis that

$$\overleftarrow{\mathit{theory\_status}} = \textsc{Raw-Chkd} \wedge \mathit{stable\_state}(\mathit{spec\_status}, \overleftarrow{\mathit{theory\_status}}),$$

the theorem `stable_state-prop` above (which shows that $\mathit{spec\_status} = \textsc{Checked}$), and that fact that $\mathit{Has\_OK\_Statuses}(\mathit{spec}, \overleftarrow{\mathit{theory}}, \textsc{Checked}, \mathit{theory\_status})$ implies $\mathit{spec\_checked}(\mathit{spec})$.

5.3 Transitions Invocable

This check determines that for any state in which a tool is invoked it must be possible for the process to be stable in this state. If this check were not satisfied then there would be tool activity transitions in the process model which can never be invoked.

This check is achieved through showing that the precondition for each tool activity transition is satisfiable (`OP-pre-satis`). [3] For example, for the transition which invokes *UPDATE* the requirement is:

$$\boxed{\text{TRANS-13-pre-satis}} \quad \frac{}{\begin{array}{c}\exists pm\colon \mathit{Process\_State} \cdot pm.\mathit{theory\_status} = \textsc{Raw-Chkd} \wedge\\ \mathit{stable\_state}(pm.\mathit{spec\_status}, pm.\mathit{theory\_status})\end{array}}$$

In other words, there is a stable state in which $\mathit{theory\_status} = \textsc{Raw-Chkd}$. By the `stable_state_prop` theorem, it suffices to exhibit a system configuration in which *spec* is syntax and type checked (so $\mathit{spec\_status} = \textsc{Checked}$), and the theory is untouched (so $\mathit{theory\_status} = \textsc{Raw-Chkd}$ as a result of a triggered transition).

Note that this simple check does not establish absence of livelock or absence of deadlock: more sophisticated checks are required for these.

[3] Strictly speaking `OP-pre-satis` is not a proof obligation of the VDM methodology: that is, there is no formal requirement that an operation's precondition need ever be satisfiable. But, since it is probably not the specifier's intention to define an operation which could never be invoked, there is a "proof opportunity" here.

6 Implementing the Process Model in Merlin

This section considers a minor generalization of the process model, defined in Section 3 above, which allows it to be implemented in Merlin more effectively. The full implementation is given in Appendix B.

6.1 Multiple Developments

Section 3 defined a process model for a configuration system corresponding to a formal software development with a single specification and a single theory. Since Merlin process models are defined in terms of instances of document types and relationships between them, it is a straightforward matter to extend the process model of Section 3 to one that supports multiple simultaneous formal developments. Multiple developments are supported by supporting multiple instances of the each document, i.e. specifications and theories. They are grouped into development configurations by connection through a *discharges* relationship.

6.1.1 Process State

The process state (see Figure 15) allows storage of multiple document instances of each document type. This is supported through the provision of mappings (e.g. *SPEC_MAP* and *SPEC_STATUS_MAP*) from the document identifiers (e.g. *SPECID*) to the document's contents (e.g. *SPECIFICATION*) and to the document's status (e.g. *SPEC_STATUS_MAP*). The *DISCHARGES_RELN* represents the one-to-one relationship between (identifiers for) specifications and the theories which discharge them. The process invariant says:

- every specification identified in the system ($\mathsf{dom}\, spec$) has an associated status and an associated theory which discharges it.
- every theory identified in the system ($\mathsf{dom}\, theory$) has an associated status and discharges some specification in the system.

6.1.2 Process Model

The statecharts from Section 3.2 are extended by making specifications and theories parameters of the statecharts and explicitly including the inter-document relationships in transition criteria (see Figure 16).

6.1.3 Behavioural Properties

Figure 17 shows the behavioural property from Section 3.3 for the revised process model.

6.2 Translation into VDM

The translation of the extended process model into VDM is very similar to that discussed in Section 4. The major difference is in the increased complexities resulting primarily from the inclusion of quantification over the multiple instances represented within the process. In this section we briefly sketch the translation of the extended process model.

The statecharts indicate that all specifications have an initial status of UNCHECKED and all theories have an initial status of RAW. This is defined as a VDM initialisation condition through quantification over all the specification and theory instances in the process state (see Figure 18).

$\mathit{SPEC\_STATUS} = \textsc{Unchecked} \mid \textsc{Checked}$
$\mathit{THEORY\_STATUS} = \textsc{Raw} \mid \textsc{Raw-Chkd} \mid \textsc{In\_Step} \mid \textsc{Complete}$
$\mathit{SPEC\_MAP} = \mathit{SPECID} \xrightarrow{m} \mathit{SPECIFICATION}$
$\mathit{SPEC\_STATUS\_MAP} = \mathit{SPECID} \xrightarrow{m} \mathit{SPEC\_STATUS}$
$\mathit{THEORY\_MAP} = \mathit{THEORYID} \xrightarrow{m} \mathit{THEORY}$
$\mathit{THEORY\_STATUS\_MAP} = \mathit{THEORYID} \xrightarrow{m} \mathit{THEORY\_STATUS}$
$\mathit{DISCHARGES\_RELN} = \mathit{THEORYID} \xleftrightarrow{m} \mathit{SPECID}$

state Merlin of
    $\mathit{spec}\colon \mathit{SPEC\_MAP}$
    $\mathit{spec\_status}\colon \mathit{SPEC\_STATUS\_MAP}$
    $\mathit{theory}\colon \mathit{THEORY\_MAP}$
    $\mathit{theory\_status}\colon \mathit{THEORY\_STATUS\_MAP}$
    $\mathit{discharges}\colon \mathit{DISCHARGES\_RELN}$
end

inv $\mathit{mk\text{-}Merlin}(\mathit{spec}, \mathit{spec\_status}, \mathit{theory}, \mathit{theory\_status}, \mathit{discharges}) \quad \triangleq$
    $\mathsf{dom}\, \mathit{spec} = \mathsf{dom}\, \mathit{spec\_status} = \mathsf{rng}\, \mathit{discharges} \wedge$
    $\mathsf{dom}\, \mathit{theory} = \mathsf{dom}\, \mathit{theory\_status} = \mathsf{dom}\, \mathit{discharges} \wedge$

Figure 15: The process state extended to support multiple developments

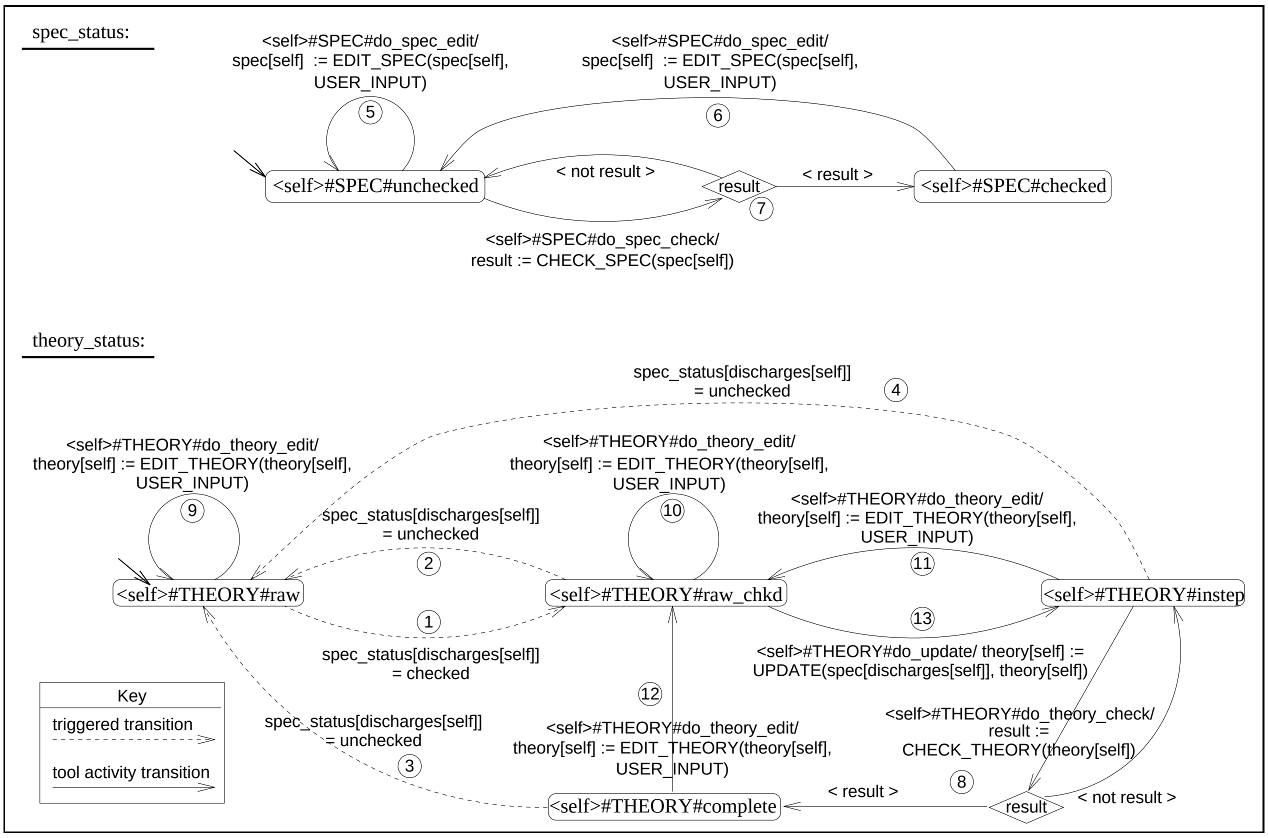
spec_status:
<self>#SPEC#do_spec_edit/ spec[self] := EDIT_SPEC(spec[self], USER_INPUT)
5
<self>#SPEC#do_spec_edit/ spec[self] := EDIT_SPEC(spec[self], USER_INPUT)
6
<self>#SPEC#unchecked
< not result >
result
7
< result >
<self>#SPEC#checked
<self>#SPEC#do_spec_check/ result := CHECK_SPEC(spec[self])
theory_status:
spec_status[discharges[self]] = unchecked
4
<self>#THEORY#do_theory_edit/ theory[self] := EDIT_THEORY(theory[self], USER_INPUT)
9
<self>#THEORY#do_theory_edit/ theory[self] := EDIT_THEORY(theory[self], USER_INPUT)
10
<self>#THEORY#do_theory_edit/ theory[self] := EDIT_THEORY(theory[self], USER_INPUT)
11
spec_status[discharges[self]] = unchecked
2
<self>#THEORY#raw
<self>#THEORY#raw_chkd
<self>#THEORY#instep
1
13
spec_status[discharges[self]] = checked
<self>#THEORY#do_update/ theory[self] := UPDATE(spec[discharges[self]], theory[self])
Key
triggered transition
tool activity transition
spec_status[discharges[self]] = unchecked
3
12
<self>#THEORY#do_theory_edit/ theory[self] := EDIT_THEORY(theory[self], USER_INPUT)
<self>#THEORY#do_theory_check/ result := CHECK_THEORY(theory[self])
8
< result >
<self>#THEORY#complete
result
< not result >

Figure 16: Specification and theory process transition systems for the extended model.

```
behav $mk\text{-}Merlin(spec, spec\_status, theory, theory\_status, discharges) \;\triangleq$
$\forall th: THEORYID \mid th \in \mathsf{dom}\; theory \cdot$
    let $sp = discharges(th)$ in
        $(spec\_status(sp) = \textsc{Checked} \wedge$
        $(theory\_status(th) = \textsc{Complete} \vee theory\_status(th) = \textsc{In\_Step}))$
            $\Rightarrow\; axioms\_substantiated(spec(sp), theory(th)) \wedge$
                    $obligations\_stated(spec(sp), theory(th)))$
        $\wedge\; spec\_status(sp) = \textsc{Checked} \;\Rightarrow\; spec\_checked(spec(sp))$
        $\wedge\; theory\_status(th) = \textsc{Complete} \;\Rightarrow\; theory\_checked(theory(th))$
```

Figure 17: A behavioural property for the revised process model.

```
init $pm \;\triangleq\;$ rng $pm.spec\_status = \{\textsc{Unchecked}\} \wedge pm.theory\_status = \{\textsc{Raw}\}$
```

Figure 18: VDM initialisation defined for multiple developments

```
$TRANS\text{-}3\;(self: THEORYID)$
ext wr $theory\_status$
    rd $spec\_status, discharges$
pre $self \in \mathsf{dom}\, theory\_status \wedge$
    $theory\_status(self) = \textsc{Complete} \wedge$
    $spec\_status(discharges(self)) = \textsc{Unchecked}$
post $theory\_status = \overleftarrow{theory\_status} \dagger \{self \mapsto \textsc{Raw}\}$
```

Figure 19: VDM translation of one of the triggered transitions of the extended process model.

The triggered transitions of the extended process model (see Figure 19) are parameterized by *self*. A trigger transition occurs whenever there exists a *self* which satisfies its precondition.

As in the previous process model, the trigger state is generated from the triggered transition preconditions. However triggered transitions are parameterized, and the definition of the trigger state requires quantification over these parameters (see Figure 20). A precondition has been included to ensure well-formedness of the definition (i.e. to ensure that the arguments to the *discharges*, *spec_status* and *theory_status* functions are within their respective domains). This precondition follows from the process state invariant, and consequently has no effect on the process model as the function is only called on the pre-state of the tool activity transitions which satisfies the invariant.

$$\begin{array}{l}
\mathit{trigger\_state} : \mathit{SPEC\_STATUS\_MAP} \times \mathit{THEORY\_STATUS\_MAP} \times \mathit{DISCHARGES\_RELN} \\
\qquad\qquad \to \mathbf{B} \\[4pt]
\mathit{trigger\_state}(\mathit{spec\_status}, \mathit{theory\_status}, \mathit{discharges}) \quad \triangleq \\
\quad \exists \mathit{th} \colon \mathit{THEORYID} \mid \mathit{th} \in \mathsf{dom}\ \mathit{theory\_status} \cdot \\
\qquad \mathsf{let}\ \mathit{sp} = \mathit{discharges}(\mathit{th})\ \mathsf{in} \\
\qquad (\mathit{theory\_status}(\mathit{th}) = \textsc{Raw} \wedge \mathit{spec\_status}(\mathit{sp}) = \textsc{Checked}) \vee \\
\qquad (\mathit{theory\_status}(\mathit{th}) = \textsc{Raw-Chkd} \wedge \mathit{spec\_status}(\mathit{sp}) = \textsc{Unchecked}) \vee \\
\qquad (\mathit{theory\_status}(\mathit{th}) = \textsc{Complete} \wedge \mathit{spec\_status}(\mathit{sp}) = \textsc{Unchecked}) \vee \\
\qquad (\mathit{theory\_status}(\mathit{th}) = \textsc{In\_Step} \wedge \mathit{spec\_status}(\mathit{sp}) = \textsc{Unchecked}) \\[4pt]
\mathsf{pre}\ \mathsf{dom}\ \mathit{spec\_status} = \mathsf{rng}\ \mathit{discharges} \wedge \\
\quad \mathsf{dom}\ \mathit{theory\_status} = \mathsf{dom}\ \mathit{discharges} \\[12pt]
\mathit{stable\_state} : \mathit{SPEC\_STATUS\_MAP} \times \mathit{THEORY\_STATUS\_MAP} \times \mathit{DISCHARGES\_RELN} \\
\qquad\qquad \to \mathbf{B} \\[4pt]
\mathit{stable\_state}(\mathit{spec\_status}, \mathit{theory\_status}, \mathit{discharges}) \quad \triangleq \\
\quad \neg \mathit{trigger\_state}(\mathit{spec\_status}, \mathit{theory\_status}, \mathit{discharges}) \\[4pt]
\mathsf{pre}\ \mathsf{dom}\ \mathit{spec\_status} = \mathsf{rng}\ \mathit{discharges} \wedge \\
\quad \mathsf{dom}\ \mathit{theory\_status} = \mathsf{dom}\ \mathit{discharges}
\end{array}$$

Figure 20: Trigger and stable state conditions are provided as VDM auxiliary functions.

Notice that the trigger states definition has an explicit dependency on the *discharges* relationship. In the previous process model this dependency was implicit as there was only one instance of each document.

As in the previous example we can deduce a theorem which defines the stable states of the process model:

$$\boxed{\text{stable\_state-prop}}\ \frac{\begin{array}{c}\mathit{spec\_status} \colon \mathit{SPEC\_STATUS\_MAP}, \\ \mathit{theory\_status} \colon \mathit{THEORY\_STATUS\_MAP}, \\ \mathit{discharges} \colon \mathit{DISCHARGES\_RELN}, \mathsf{dom}\ \mathit{spec\_status} = \mathsf{rng}\ \mathit{discharges} \wedge \\ \mathsf{dom}\ \mathit{theory\_status} = \mathsf{dom}\ \mathit{discharges}\end{array}}{\begin{array}{c}\mathit{stable\_state}(\mathit{spec\_status}, \mathit{theory\_status}, \mathit{discharges}) = \\ \forall \mathit{th} \colon \mathit{THEORYID} \mid \mathit{th} \in \mathsf{dom}\ \mathit{discharges} \cdot \\ \mathit{spec\_status}(\mathit{discharges}(\mathit{th})) = \textsc{Unchecked} \ \Leftrightarrow\ \mathit{theory\_status}(\mathit{th}) = \textsc{Raw}\end{array}}$$

As with triggered transitions, tool activity transitions include *self* as a parameter to the operation (see Figure 21). As indicated by the postcondition of the operations, document contents and statuses are updated by overwriting the mappings defined in the process state.

6.3 Verification of Process Model

Verification of the revised process model proceeds along precisely the same lines as in Section 5. The statement of the proof obligations, and the proofs themselves, are more complex because of the need

$$
\begin{array}{l}
\mathit{TRANS}\text{-}9\ (\mathit{self}\colon \mathit{THEORYID}, \mathit{input}\colon \mathit{\Delta}\text{-}\mathit{THEORY}) \\
\mathsf{ext\ wr}\ \mathit{theory\_status}, \mathit{theory} \\
\quad \mathsf{rd}\ \mathit{spec\_status}, \mathit{discharges} \\
\mathsf{pre}\ \mathit{self} \in \mathsf{dom}\,\mathit{theory\_status} \wedge \\
\quad \mathit{theory\_status}(\mathit{self}) = \textsc{Raw} \wedge \\
\quad \mathit{stable\_state}(\mathit{spec\_status}, \mathit{theory\_status}, \mathit{discharges}) \\
\mathsf{post}\ \mathit{theory} = \overleftarrow{\mathit{theory}} \dagger \{\mathit{self} \mapsto \mathit{EDIT\_THEORY}(\overleftarrow{\mathit{theory}}(\mathit{self}), \mathit{input})\} \wedge \\
\qquad \mathit{theory\_status} = \overleftarrow{\mathit{theory\_status}} \dagger \{\mathit{self} \mapsto \textsc{Raw}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{TRANS}\text{-}13\ (\mathit{self}\colon \mathit{THEORYID}) \\
\mathsf{ext\ wr}\ \mathit{theory\_status}, \mathit{theory} \\
\quad \mathsf{rd}\ \mathit{spec\_status}, \mathit{discharges}, \mathit{spec} \\
\mathsf{pre}\ \mathit{self} \in \mathsf{dom}\,\mathit{theory\_status} \wedge \\
\quad \mathit{theory\_status}(\mathit{self}) = \textsc{Raw-Chkd} \wedge \\
\quad \mathit{stable\_state}(\mathit{spec\_status}, \mathit{theory\_status}, \mathit{discharges}) \\
\mathsf{post}\ \mathit{theory} = \overleftarrow{\mathit{theory}} \dagger \{\mathit{self} \mapsto \mathit{UPDATE}(\mathit{spec}(\mathit{discharges}(\mathit{self})), \overleftarrow{\mathit{theory}}(\mathit{self}))\} \\
\qquad \wedge\ \mathit{theory\_status} = \overleftarrow{\mathit{theory\_status}} \dagger \{\mathit{self} \mapsto \textsc{In\_Step}\}
\end{array}
$$

Figure 21: VDM translation of two of the tool activity transitions of the extended process model.

to handle multiple instances of documents, but the analysis they perform is essentially the same.

For example, the `TRANS-13-maint` proof obligation of the extended process model requires that the behavioural property is satisfied for all theory instances. Since theories other than *self* are not affected by the operation, the behavioural property need only be checked for *self*. Ultimately the modification made by invoking *UPDATE* must ensure that the assignment of IN_STEP status to *self* is consistent. This is the same requirement that was identified in the proof of `TRANS-13-maint` in the previous process model.

7 Conclusions & Further Work

This paper provides verification and validation techniques for state-transition process models. It uses behavioural properties to capture the intended semantics of states of a process model by relating process states to properties of the underlying configuration. This implies that the meaning of the status of a software development can be determined directly from the process model states, rather than having to be evaluated dynamically by executing the whole process model. It also means that the correctness of the process model can be verified, at least in the following sense: whenever a development tool is invoked by the process model, the preconditions of that tool are satisfied.

The inclusion of behavioural properties within process models offers benefits in a number of areas:

Design: by capturing requirements of the process model, and providing guidance in the provision of the activities to meet those requirements;

Verification: for example, by showing that the tools are invoked correctly;

Readability: by providing an alternative view of the process, from which users of the process may determine properties of the underlying configuration without knowledge of the entire process;

Maintenance: by capturing important properties of the underlying configuration which should be preserved by new activities, or in comparing different versions of process descriptions.

In this paper we translated the process model into VDM and showed how the application of certain standard VDM verification techniques provide a valuable cross-check on the correctness of the process model. Eventually, however, we would like to remove the intermediate VDM representation and pass directly from the process model to appropriate proof obligations, so that the process model developer need know nothing about VDM.

Merlin offers support for concurrency control which enables teams of developer to work on the development at the same time [JPSW93]. This is supported by a transaction management concept, which regulates the concurrent updates to the process state made by the transitions. The verification of models executed under this approach has not been considered.

Acknowledgments

The authors would like to thank a number of colleagues for their fruitful contribution in the form of discussion and review. In particular Erik van Keulen from the Software Verification Research Centre for comments on early drafts of the paper, and Burkhard Peuschel and Gerald Junkermann from the University of Dortmund for discussions regarding the operational semantics of Merlin process descriptions.

References

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

[DG90] W. Deiters and V. Gruhn. Managing software processes in the environment MELMAC. In R. N. Taylor, editor, *Proc. 4th ACM SIGSOFT Symp. on Software Development Environments*, vol. 15, *SIGSOFT Software Engineering Notes.* ACM Press, Dec. 1990.

[Har87] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. D. Moore. *Mural: A Formal Development Support System.* Springer-Verlag, London, 1991.

[Jon90] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990.

[JPSW93] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment. Tech. Rep. 70, Univ. of Dortmund, Germany, September 1993.

[Jun94] G. Junkermann. How to improve process programming in merlin. Tech. Rep., To Appear, Univ. of Dortmund, Germany, 1994. To appear.

[LvK94] P. A. Lindsay and E. van Keulen. Verification Case Studies in Z and VDM. Tech. Rep. TR 94-3, Software Verification Research Centre, Dept. of Comp. Sci., Univ. of Queensland, Australia, 1994.

[PS92] B. Peuschel and W. Schäfer. Concepts and implementation of a rule-based process engine. In *Proc. 14th Int. Conf. on Software Engineering*, Melbourne, Australia, May 1992.

[PSW92] B. Peuschel, W. Schäfer, and S. Wolf. A knowledge-based software development environment supporting cooperative work. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):79–106, 1992.

[RL93] K. J. Ross and P. A. Lindsay. Maintaining consistency under changes to formal specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *Proceedings of Formal Methods Europe - 1993*, LNCS 670, pages 558–577. Springer-Verlag, April 1993.

[VDM93] VDM specification language proto-standard: Draft. British Standards Institute, Working Group IST/5/19, November 1993. ISO/IEC JTC1/SC22/WG19 N-20.

A Theory Generation

This section describes the construction of the mathematical theory which which corresponds to a given VDM specification. Each component of the VDM specification gives rise to a set of axioms and proof obligations in the theory.

A.1 Explicit Functions

Pattern

$f : A \times B \to R$

$f(x, y) \quad \triangleq \quad fdef(x, y)$

pre $fpre(x, y)$

Axioms

$$\boxed{\text{f-defn}} \ \frac{x: A,\ y: B,\quad fpre(x, y),\quad fdef(x, y): R}{f(x, y) = fdef(x, y)}$$

Proof obligations

$$\boxed{\text{f-pre-wff}} \ \frac{x: A,\ y: B}{fpre(x, y): \mathbf{B}} \qquad \boxed{\text{f-wf}} \ \frac{x: A,\ y: B,\quad fpre(x, y)}{fdef(x, y): R} \qquad \boxed{\text{f-pre-satis}} \ \frac{}{\exists x: A,\ y: B\cdot\ fpre(x, y)}$$

A.2 Implicit Functions

Pattern

$g\ (x: A, y: B)\ r: R$

pre $gpre(x, y)$

post $gpost(x, y, r)$

Axioms

$$\boxed{\text{g-wf}} \ \frac{x: A,\ y: B,\quad gpre(x, y),\quad \exists r: R \cdot gpost(x, y, r)}{g(x, y): R} \qquad \boxed{\text{g-defn}} \ \frac{x: A,\ y: B,\quad gpre(x, y),\quad \exists r: R \cdot gpost(x, y, r)}{gpost(x, y, g(x, y))}$$

Proof obligations

$$\boxed{\text{g-pre-wff}} \ \frac{x: A,\ y: B}{gpre(x, y): \mathbf{B}} \qquad \boxed{\text{g-post-wff}} \ \frac{x: A,\ y: B,\ r: R,\quad gpre(x, y)}{gpost(x, y, r): \mathbf{B}}$$

$$\boxed{\text{g-pre-satis}} \ \frac{}{\exists x: A,\ y: B \cdot gpre(x, y)} \qquad \boxed{\text{g-satis}} \ \frac{x: A,\ y: B,\quad gpre(x, y)}{\exists r: R \cdot gpost(x, y, r)}$$

A.3 State Definitions

Pattern

state S of
    $a\colon A$
    $b\colon B$
    $c\colon C$
end

$$\begin{array}{l} \textsf{inv mk-}S(x,y,z) \;\;\triangleq\;\; Sinv(x,y,z) \\ \textsf{init } s \;\;\triangleq\;\; Sinit(s) \\ \textsf{behav mk-}S(x,y,z) \;\;\triangleq\;\; Sbehav(x,y,z) \end{array}$$

Axioms

$$\boxed{\text{S-form}}\ \frac{x\colon A,\ y\colon B,\ z\colon C,\\ Sinv(x,y,z)}{\textsf{mk-}S(x,y,z)\colon S} \qquad \boxed{\text{S-inv-I}}\ \frac{\textsf{mk-}S(x,y,z)\colon S}{Sinv(x,y,z)}$$

$$\boxed{\text{S-defn}}\ \frac{s\colon S}{\textsf{mk-}S(S.a,S.b,S.c)=s} \qquad \boxed{\text{S-a-wf}}\ \frac{\textsf{mk-}S(x,y,z)\colon S}{\textsf{mk-}S(x,y,z).a\colon A}$$

$$\boxed{\text{S-a-defn}}\ \frac{\textsf{mk-}S(x,y,z)\colon S}{\textsf{mk-}S(x,y,z).a=x}$$

Same for b and c component.

Proof obligations

$$\boxed{\text{S-inv-wff}}\ \frac{x\colon A,\ y\colon B,\ z\colon C}{Sinv(x,y,z)\colon \mathbf{B}} \qquad \boxed{\text{S-init-wff}}\ \frac{s\colon S}{Sinit(s)\colon \mathbf{B}}$$

$$\boxed{\text{S-inv-satis}}\ \frac{}{\exists x\colon A,\ y\colon B,\ z\colon C\cdot\\ Sinv(x,y,z)} \qquad \boxed{\text{S-init-satis}}\ \frac{}{\exists s\colon S\cdot Sinit(s)}$$

$$\boxed{\text{S-behav-wff}}\ \frac{\textsf{mk-}S(x,y,z)\colon S}{Sbehav(x,y,z)\colon \mathbf{B}} \qquad \boxed{\text{S-init-maint}}\ \frac{}{\forall s\colon S\cdot Sinit(s)\\ \Rightarrow\ Sbehav(S.a,S.b,S.c)}$$

A.4 Operations

Pattern

$OP\ (i\colon I)\ o\colon O$
ext rd $a\colon A$
    wr $b\colon B$
pre $OPpre(i, a, b)$
post $OPpost(i, o, a, \overleftarrow{b}, b)$

Proof obligations

$$\boxed{\text{OP-pre-wff}}\ \frac{i\colon I, \mathsf{mk}\text{-}S(x, y, z)\colon S, Sbehav(x, y, z)}{OPpre(i, x, y)\colon \mathbf{B}}$$

$$\boxed{\text{OP-post-wff}}\ \frac{\begin{array}{c} i\colon I, o\colon O, \\ \mathsf{mk}\text{-}S(x, \overleftarrow{y}, z)\colon S, \mathsf{mk}\text{-}S(x, y, z)\colon S, \\ Sbehav(x, \overleftarrow{y}, z), Sbehav(x, y, z), \\ OPpre(i, x, \overleftarrow{y}) \end{array}}{OPpost(i, o, x, \overleftarrow{y}, y)\colon \mathbf{B}}$$

$$\boxed{\text{OP-satis}}\ \frac{\begin{array}{c} i\colon I, \mathsf{mk}\text{-}S(x, \overleftarrow{y}, z)\colon S, \\ Sbehav(x, \overleftarrow{y}, z), \\ OPpre(i, x, \overleftarrow{y}) \end{array}}{\begin{array}{c} \exists o\colon O, y\colon B \cdot Sinv(x, y, z) \\ \wedge OPpost(i, o, x, \overleftarrow{y}, y) \end{array}}$$

$$\boxed{\text{OP-maint}}\ \frac{\begin{array}{c} i\colon I, \mathsf{mk}\text{-}S(x, \overleftarrow{y}, z)\colon S, \\ Sbehav(x, \overleftarrow{y}, z), \\ OPpre(i, x, \overleftarrow{y}) \end{array}}{\begin{array}{c} \forall o\colon O, y\colon B \cdot Sinv(x, y, z) \\ \wedge OPpost(i, o, x, \overleftarrow{y}, y) \\ \Rightarrow\ Sbehav(x, y, z) \end{array}}$$

Proof opportunities

$$\boxed{\text{OP-pre-satis}}\ \frac{}{\exists i\colon I, s\colon S \cdot OPpre(i, S.a, S.b)}$$

B Merlin Implementation

This section provides a set of Merlin process model implementation *facts*, which are interpreted by the Merlin *kernel* to provide the enactment of the process model described in Section 6.

The facts shown in this section collectively provide a program which can be executed within Merlin. For explanation of these process facts the reader should refer to the user guide [PS92].

B.1 Document Information

The Merlin facts defined in this section describe the process state by declaring instances of documents and the relations between them.

B.1.1 Document Definitions

Firstly the document structure, its icon representation to be displayed in the merlin user interface, and possible states are to be defined. Here we have declared both the specification and theory to be ascii files, with defined icons, and with the appropriate statuses as states.

```
document_type_structure( specification, ascii_file).
document_type_icon( specification, icons/vdm_spec).
document_type_states( specification, [unchecked, checked]).

document_type_structure( theory, ascii_file).
document_type_icon( theory, icons/vdm_theory).
document_type_states( theory, [raw, raw_ckd, in_step, complete]).
```

B.1.2 Document Relations

Only one document relation type is defined in this case study: a theory `discharges` a specification.

```
document_relation_type(discharges, theory, specification).
```

B.1.3 Instances

To populate the model we shall define two document instances: `dev_spec` which is a `specification` and is physically located in the file `unix_src/dev#spec`; and `dev_theory` which is a `theory` and is physically located in the file `unix_src/dev#theory`. The theory `dev_theory` discharges the specification `dev_spec`.

```
document(dev_spec, specification, unix_src/dev#spec).
document(dev_theory, theory, unix_src/dev#theory).

document_relation(discharges, dev_theory, dev_spec).
```

B.2 State Transitions

The statecharts in Figure 16 define the initialisation and different transitions that occur during the process. The statecharts are translated into Merlin facts as shown below.

B.2.1 Initial Document States

Initialisation information is given as `document_state` facts. Initially `dev_spec` has status `unchecked` and `dev_theory` has status `raw`.

```
document_state(dev_spec, unchecked).
document_state(dev_theory, raw).
```

B.2.2 Triggered Transitions

Triggered transitions are given as `next_state_or_cond` facts.[4]

The consistency condition facts below correspond to transitions 1-4 of Figure 16 respectively. For example, transition 1 changes the status of a theory from `raw` to `raw_ckd` if the theory is related by the `discharges` relation to a specification document with status `checked`. The corresponding fact indicates that the theory is the source of the `discharges` relation and the other document is the destination.

```
next_state_or_cond( theory, raw, raw_ckd,
                             [[destination, discharges, checked]]).
next_state_or_cond( theory, in_step, raw,
                             [[destination, discharges, unchecked]]).
next_state_or_cond( theory, complete, raw,
                             [[destination, discharges, unchecked]]).
next_state_or_cond( theory, raw_ckd, raw,
                             [[destination, discharges, unchecked]]).
```

B.2.3 Tool Activity Transitions

Tool activity transitions correspond to a change in process state when a tool activity is applied on a document. The tool invocation facts below correspond to transitions 5-13 of Figure 16 respectively.

A *tool invocation* fact consists of a description of the type of document on which the tool is invoked, the tool that is applied, the access privilege required, and the status changes that occur. For example, the third fact below indicates that the specification checker (`check_spec`) can be applied to a specification document whose status is `unchecked`; read access to the specification is required; on completion of the checker the resulting status is either `unchecked` or `checked`.

[4]See [PS92] for an explanation of the different kinds of triggered transitions supported by Merlin. We could have equally as well used `next_state_and_cond` here.

```
document_type_tools( specification, edit_spec, writeable,
                            checked, [unchecked]).
document_type_tools( specification, edit_spec, writeable,
                            unchecked, [unchecked]).
document_type_tools( specification, check_spec, readable,
                            unchecked, [unchecked, checked]).

document_type_tools( theory, check_theory, readable,
                            in_step, [in_step, complete]).
document_type_tools( theory, edit_theory, writeable,
                            raw, [raw]).
document_type_tools( theory, edit_theory, writeable,
                            raw_ckd, [raw_ckd]).
document_type_tools( theory, edit_theory, writeable,
                            in_step, [raw_ckd]).
document_type_tools( theory, edit_theory, writeable,
                            complete, [raw_ckd]).
document_type_tools( theory, update, writeable,
                            raw_ckd, [in_step]).
```

The tools are synonymous with unix commands which are passed references to documents to be analysed or modified, and which implement the functional definition provided in the statechart. For instance, `edit_spec` carries out the operation `spec[self] := EDIT_SPEC(spec[self], USER_INPUT)`.

In implementing the process model all batch tools (i.e. those without `USER_INPUT` - namely `update`, `check_spec` and `check_theory`) were simulated. Merlin provides support for batch tools, however we simulated them as interactive tools, in which the user browsed or edited the documents to reflect the functionality of the activity. The choice to simulate batch tools as interactive tools resulted in two alternative status results for transitions 7 and 8 of Figure 16. Process enactment requires the user to choose between the two alternative statuses based on their interactive analysis of the documents at run-time.

B.3 Auxiliary Information

For completeness, we include here the remaining facts that Merlin requires, describing roles and responsibilities in the software development process.

B.3.1 Project Developers

A project called `vdm_development` will have its development controlled by the process program described in the previous section. Let us suppose the developers `miller` and `smith` are assigned to the project.

```
project(vdm_development, [miller,smith]).
```

B.3.2 Developer Roles

Let us assume `miller` is a `specifier` and `smith` is a `verifier`.

```
has_roles(miller, [specifier]).
has_roles(smith, [verifier]).
```

B.3.3 Role Access

The role access defines when the different documents may be manipulated and accessed by the developers of a particular role. In this case, let us assume a specifier can modify a specification with any status and a verifier can modify a theory with any status. The other fields represent the related documents to be presented to the user performing this role.

```
roletype_document_work_on(specifier, specification, unchecked, [], [], []).
roletype_document_work_on(specifier, specification, checked, [], [], []).

roletype_document_work_on(verifier, theory, raw, [], [], []).
roletype_document_work_on(verifier, theory, raw_ckd, [], [], []).
roletype_document_work_on(verifier, theory, in_step, [], [], []).
roletype_document_work_on(verifier, theory, complete, [], [], []).
```

B.4 Developer Document Responsibilities

Finally, let us assume `miller`, when acting as a specifier, can access the specification `dev_spec`, and `smith` when acting as verifier can access the theory `dev_theory`.

```
responsibilities(miller,specifier, [dev_spec]).
responsibilities(smith,verifier, [dev_theory]).
```