Testing Refinements of State-based Formal Specifications

John Derrick and Eerke Boiten
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Phone: + 44 1227 764000, Email: J.Derrick@ukc.ac.uk.)

Abstract

A specification provides a concise description of a system, and can be used as both the benchmark against which any implementation is tested, and also as a means to generate tests. *Formal* specifications have potential advantages over informal descriptions because they offer the possibility of reducing the costs of testing by automating part of the testing process. This observation has led to considerable interest in developing test generation techniques from formal specifications, and a number of different methods have been derived for state based formalisms such as Z, B and VDM. However, after tests have been derived from a formal specification, the specification might be refined further before its implementation, and therefore a mechanism is needed to relate the abstract tests to the refined implementation.

The purpose of this paper is to provide such a method by exploring the relationship between testing and refinement. In this paper a model for test generation is used which constructs a finite state machine (FSM) from a Z specification by using a DNF partition analysis of the state and operations. The finite state machine is then used to derive suitable test suites. The paper describes a way of calculating a FSM for a refinement from an abstract FSM together with the information about the refinement embodied in the retrieve relation. This means that it is possible to test an implementation by generating a new concrete finite state machine from a set of abstract tests.

Keywords: Formal Specification; Z; Refinement; Finite State Machines; Partition analysis.

1 Introduction

A specification, whether formal or informal, acts as the benchmark against which any implementation is tested. A specification also provides a means by which tests can be generated. Formal methods are important because they offer a possibility of reducing the software development cost by automating part of the testing process.

This observation has led to considerable theoretical and practical work on how to automatically (or semi-automatically) generate test cases from formal specifications, and how scheduling of these tests can be achieved. Different types of formalisms have developed different ways to do this, and for state based languages such as Z [Spivey, 1989], B [Abrial, 1996] and VDM [Jones, 1989] a number of techniques have been developed, see for example [Scullard, 1988, Cusack and Wezeman, 1992, Dick and Faivre, 1993, Carrington and Stocks, 1994, Horcher, 1995, Stepney, 1995].

One elegant and simple method for generating and sequencing tests from state based languages has been developed by Dick and Faivre [Dick and Faivre, 1993]. The basic technique of test generation consists of a partition analysis, which reduces the specification of each operation into a Disjunctive

Normal Form (DNF). This is then used to construct a Finite State Machine (FSM) which can serve as a means to derive test suites. The approach was based on VDM, but has been applied to Z in [Horcher, 1995, Singh et al., 1997] and B in [van Aertryck et al., 1997], and benefits from tool support, which is described in [Dick and Faivre, 1993] and [van Aertryck et al., 1997]. In [Horcher, 1995] an industrial application of the method to an aircraft control system is described.

However, after tests have been derived from a formal specification, the specification might be developed or refined further before its implementation. Indeed any implementation can be viewed as a refinement of the original specification. The conditions under which a development is a correct refinement are encapsulated into two refinement rules: downward and upward simulations [Woodcock and Davies, 1996]. To verify a refinement the simulations use a retrieve relation which relates the concrete to abstract states.

The process of refinement changes the specification in a number of ways. For example, the concrete state space may change (e.g. sets may be implemented as lists) and non-determinism in the specification may be resolved. Because of this a finite state machine generated from the abstract specification *cannot* be used to test a concrete implementation except under the simplest of refinements, since the abstract tests may be insufficient or even incomparable to the concrete implementation. For example, the abstract tests may be defined in terms of sets whereas the concrete implementation uses lists, and in order to use the abstract tests to test the implementation it is necessary to relate the values in the state spaces (i.e. sets to lists). Similar but more complex situations arise when non-determinism in the abstract specification is resolved or moved.

Therefore tests generated from a specification will only be usable if the specification is the *implementation* specification (i.e. the one from which the coding is done directly), or if the refinements are extremely simple.

To deal effectively with these situations a method is needed to test an implementation based upon the tests generated from the abstract specification. The aim of this paper is to address this issue, which is achieved by describing a way of calculating a FSM for a refinement from an abstract FSM together with the information about the refinement embodied in the retrieve relation. This means a new concrete finite state machine can be generated in a simple manner from a set of abstract tests.

The structure of the paper is as follows. Section 2 describes how to generate finite state machines for Z specifications based on a DNF partition analysis, and Section 3 provides some background material on refinement in Z. Sections 4 and 5 describe how to calculate a concrete finite state machine for refinements which are downward simulations, and Section 6 for upward simulations. Conclusions are presented in Section 7.

2 Using Finite State Machines to Test Specifications

Different formal paradigms have associated methods for aiding the test generation process in an automatic, semi-automatic or manual fashion. The interest in this paper is in how to test specifications written in state based languages such as Z, B and VDM. The approach considered here is that of Dick and Faivre [Dick and Faivre, 1993], which describes a means to automate test generation and sequencing from VDM specifications, and has also been applied to Z specifications in [Horcher, 1995, Singh et al., 1997]. In [Horcher, 1995] Hörcher describes an application of this methodology to a portion of the Cabin Intercommunication Data System for the Airbus A330/340 aircraft. In [Singh et al., 1997] this methodology is combined with the classification tree method [Grochtmann and Grimm, 1993] and is used to test an adaptive control system.

Dick and Faivre consider the complete testing activity from test generation from individual operations, through the scheduling of tests, to the verification of test results. The basic technique of test case generation consists of a partition analysis, which reduces the specification of each operation into its Disjunctive Normal Form (DNF). Each element in the DNF represents an individual test case for the operation. From these test cases a partition of the system state is performed resulting in a set of disjoint states, each of which is either the before-state or after-state of at least one test to be performed. This partition then serves as a basis for the construction of a finite state machine which is then used to derive test suites (i.e. a structured sequence of test cases).

This paper is concerned with this construction of FSMs from state-based specifications, and the aim of the paper is to show how the FSM alters upon refinement. This will allow tests to be adapted to take account of implementation choices such as changes in the data structures and resolution of non-determinism. This paper builds upon earlier work described in [Derrick and Boiten, 1998b] which considered the partition analysis of individual operations. Although the results are described with reference to the construction of FSMs due to Dick and Faivre, it is equally applicable to other approaches, such as those described in [Murray et al., 1998, Hierons, 1997].

As an example of how to extract a FSM from a specification consider the specification of a system process scheduler adapted from [Dick and Faivre, 1993] (and rewritten in Z). The system consists of processes either *ready* to be scheduled or *waiting* to become ready and, optionally, a single *active* process. These processes are identified by a unique $Pid == \mathbb{N}$. A process cannot be both ready and waiting, and the active process is neither ready nor waiting. In addition, there must be an active process whenever there are processes ready to be scheduled. The *schedule* function describes the algorithm abstractly by picking any process from those which are ready. The system can be in two modes: *user* or *super*.

The specification describes *ready* and *waiting* as sets and contains four operations. *New* introduces another process, *Ready* puts a process into the ready state, and *Swap* changes the active process. The operation *Boot* enables a restart if the system is in *super* mode. (*nil* stands for an inactive process.)

$$
\begin{array}{|l}
schedule : \mathbb{P}\, Pid \rightarrow Pid \\
\hline
\forall\, s : \mathbb{P}\, Pid \bullet s \neq \varnothing \Rightarrow schedule(s) \in s
\end{array}
$$

State

$$
\begin{array}{|l}
active : Pid \\
ready : \mathbb{P}\, Pid \\
waiting : \mathbb{P}\, Pid \\
admin : user \mid super \\
\hline
ready \cap waiting = \varnothing \\
active \notin (ready \cup waiting) \\
nil \notin (ready \cup waiting) \\
active = nil \Rightarrow ready = \varnothing
\end{array}
$$

Init

$$
\begin{array}{|l}
State' \\
\hline
active' = nil \\
ready' \cup waiting' = \varnothing \\
admin' = user
\end{array}
$$

New

$$
\begin{array}{|l}
\Delta State \\
p? : Pid \\
\hline
p? \neq active \\
p? \notin (ready \cup waiting) \\
waiting' = waiting \cup \{p?\} \\
active' = active \\
ready' = ready \\
admin = admin' = user
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Ready} \\
\Delta \mathit{State} \\
q? : \mathit{Pid} \\
\hline
q? \in \mathit{waiting} \\
\mathit{waiting}' = \mathit{waiting} \setminus \{q?\} \\
\mathit{active} = \mathit{nil} \Rightarrow (\mathit{ready}' = \mathit{ready} \wedge \mathit{active}' = q?) \\
\mathit{active} \neq \mathit{nil} \Rightarrow (\mathit{ready}' = \mathit{ready} \cup \{q?\} \wedge \mathit{active}' = \mathit{active}) \\
\mathit{admin} = \mathit{admin}' = \mathit{user} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Swap} \\
\Delta \mathit{State} \\
\hline
\mathit{active} \neq \mathit{nil} \\
\mathit{waiting}' = \mathit{waiting} \cup \{\mathit{active}\} \\
\mathit{ready} = \varnothing \Rightarrow (\mathit{active}' = \mathit{nil} \wedge \mathit{ready}' = \varnothing) \\
\mathit{ready} \neq \varnothing \Rightarrow (\mathit{active}' = \mathit{schedule}(\mathit{ready}) \wedge \mathit{ready}' = \mathit{ready} \setminus \{\mathit{active}'\}) \\
\mathit{admin} = \mathit{admin}' = \mathit{user} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Boot} \\
\Delta \mathit{State} \\
\hline
\mathit{admin} = \mathit{super} \\
\mathit{active} = \mathit{nil} \\
\mathit{waiting} = \mathit{ready} = \varnothing \\
\mathit{admin}' = \mathit{user} \wedge \mathit{active}' \neq \mathit{nil} \\
(\mathit{ready}' = \varnothing \wedge \mathit{waiting}' \neq \varnothing) \vee (\mathit{ready}' \neq \varnothing \wedge \mathit{waiting}' = \varnothing) \\
\hline
\end{array}
$$

To generate and sequence tests Dick and Faivre build a FSM using the following procedure

1. Perform a partition analysis on all operations to generate the test cases. These are the transitions in the FSM.
2. From each test case obtain its before-state and after-state (by existentially quantifying variables not being considered).
3. Perform a DNF partition analysis on the states from step 2. This gives the states in the FSM.
4. Construct the FSM by resolving transitions against states.

In order to perform the partition analysis on the operations each operation is transformed into DNF. Each schema in this DNF then represents a single test case. Each test case will be disjoint, allowing them all to be treated separately. It should be noted that for any non-trivial specification there are many possible choices for DNF [Stocks, 1993], the choice taken depends upon the aspects that are considered important. Here the operations are partitioned by considering whether $\mathit{active} = \mathit{nil}$ and whether $\mathit{ready}, \mathit{waiting} = \varnothing$. The resulting test cases are given by *Init* plus the following eight tests

$$\begin{array}{|l} \multicolumn{1}{l}{New1} \\ \hline \Delta State \\ p? : Pid \\ \hline active' = active = nil \\ ready' = ready = \varnothing \\ p? \notin waiting \\ waiting' = waiting \cup \{p?\} \\ admin = admin' = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{New2} \\ \hline \Delta State \\ p? : Pid \\ \hline p? \neq active \\ p? \notin (ready \cup waiting) \\ waiting' = waiting \cup \{p?\} \\ active' = active \neq nil \\ ready' = ready \\ admin = admin' = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{Ready1} \\ \hline \Delta State \\ q? : Pid \\ \hline q? \in waiting \\ waiting' = waiting \setminus \{q?\} \\ active = nil \\ ready' = ready = \varnothing \\ active' = q? \\ admin = admin' = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{Ready2} \\ \hline \Delta State \\ q? : Pid \\ \hline q? \in waiting \\ waiting' = waiting \setminus \{q?\} \\ active \neq nil \\ ready' = ready \cup \{q?\} \\ active' = active \\ admin = admin' = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{Swap1} \\ \hline \Delta State \\ \hline active \neq nil \\ waiting' = waiting \cup \{active\} \\ ready = ready' = \varnothing \\ active' = nil \\ admin = admin' = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{Swap2} \\ \hline \Delta State \\ \hline active \neq nil \\ waiting' = waiting \cup \{active\} \\ active' \in ready \\ ready' = ready \setminus \{active'\} \\ admin = admin' = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{Boot1} \\ \hline \Delta State \\ \hline admin = super \\ active = nil \\ waiting = ready = \varnothing \\ admin' = user \wedge active' \neq nil \\ ready' = \varnothing \wedge waiting' \neq \varnothing \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{Boot2} \\ \hline \Delta State \\ \hline admin = super \\ active = nil \\ waiting = ready = \varnothing \\ admin' = user \wedge active' \neq nil \\ ready' \neq \varnothing \wedge waiting' = \varnothing \\ \hline \end{array}$$

This construction has two important properties: coverage and disjointness; that is, *New* equals the disjunction of its test cases (coverage) and these tests are disjoint. In general a collection of tests $\{AOp_i\}_i$ is said to cover an operation AOp acting on state space $Astate$ if

$$AOp = \bigvee_i AOp_i$$

and that the tests are disjoint, if, for all $i \neq j$

$$\neg \exists Astate; \ Astate' \bullet AOp_i \wedge AOp_j$$

It is easy to see that $\{New1, New2\}$ form a disjoint covering for *New*.

A distributed disjunction ($\bigvee$) has been used here, which although nonstandard Z, can be defined in the obvious manner (for example, by using existential quantification). Similarly, the equality

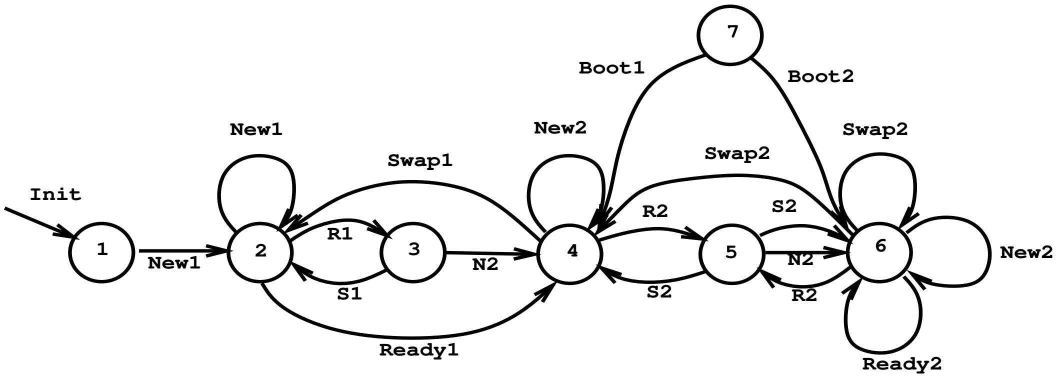

Figure 1: The FSM for the scheduler

sign between schemas should be viewed as schema equivalence. The symbols $\bigvee$ and $=$ are retained for the sake of clarity.

Next the states that result from these test cases are calculated. The before state for the *New1* test case is given by $\exists\, State' \bullet New1 \setminus \{p?\}$, and the after state will be given by $\exists\, State \bullet New1$. After performing a DNF partition analysis on the result the following seven states remain:

State1
$State$
$active = nil$
$ready = \varnothing$
$waiting = \varnothing$
$admin = user$

State2
$State$
$active = nil$
$ready = \varnothing$
$waiting \neq \varnothing$
$admin = user$

State3
$State$
$active \neq nil$
$ready = \varnothing$
$waiting = \varnothing$
$admin = user$

State4
$State$
$active \neq nil$
$ready = \varnothing$
$waiting \neq \varnothing$
$admin = user$

State5
$State$
$active \neq nil$
$ready \neq \varnothing$
$waiting = \varnothing$
$admin = user$

State6
$State$
$active \neq nil$
$ready \neq \varnothing$
$waiting \neq \varnothing$
$admin = user$

State7
$State$
$active = nil$
$ready = \varnothing$
$waiting = \varnothing$
$admin = super$

The FSM can now be constructed by calculating the individual transitions. For example, a transition of *New1* exists from state *State1* to *State2* precisely when the following evaluates to true: $\exists\, State;\ State';\ inputs;\ outputs \bullet State1 \wedge New1 \wedge State2'$. This produces the FSM shown in figure 1.

Although state 7 cannot be reached via transitions in the FSM this state is not pruned from the FSM for reasons discussed in Section 5.

3 Refinement

In addition to deriving tests from a formal specification, the specification might be *refined* further before its implementation. Such a refinement might typically weaken the precondition of an operation, remove some non-determinism or even alter the state space of the specification. The conditions under which a development is a correct refinement are encapsulated into two rules: downward and upward simulations [Woodcock and Davies, 1996]. These refinement rules are known to be sound and jointly complete, that is any upward or downward simulation is a valid refinement, and any refinement can be proved correct by application of appropriate upward and downward simulations [He, 1989, Woodcock and Morgan, 1990]. (Downward and upward simulations are sometimes also known as forward and backward simulations respectively.)

The downward simulation rules are more straightforward, and form the usual presentation of refinement (e.g. as in [Spivey, 1989]), however, upward simulations are occasionally necessary, for example when the resolution of non-determinism has been postponed [Woodcock and Davies, 1996]. Consider an abstract specification with state space *Astate* and initialisation schema *Ainit* being refined by a concrete specification with state space *Cstate* and initialisation schema *Cinit*. Downward and upward simulations are defined as follows.

Definition 1 *Downward simulation*
The concrete specification is a downward simulation of the abstract if there is a retrieve relation Ret such that every abstract operation AOp is recast into a concrete operation COp and the following hold.

DS.1 $\forall Astate;\ Cstate \bullet \mathrm{pre}\, AOp \wedge Ret \Longrightarrow \mathrm{pre}\, COp$

DS.2 $\forall Astate;\ Cstate;\ Cstate' \bullet Ret \wedge \mathrm{pre}\, AOp \wedge COp \Longrightarrow \exists Astate' \bullet Ret' \wedge AOp$

DS.3 $\forall Cstate' \bullet Cinit \Longrightarrow \exists Astate' \bullet Ainit \wedge Ret$

Definition 2 *Upward simulation*
The concrete specification is an upward simulation of the abstract if there is a retrieve relation Ret such that every abstract operation AOp is recast into a concrete operation COp and the following hold.

US.1 $\forall Cstate \bullet (\forall Astate \bullet Ret \Longrightarrow \mathrm{pre}\, AOp) \Longrightarrow \mathrm{pre}\, COp$

US.2 $\forall Astate';\ Cstate;\ Cstate' \bullet (\forall Astate \bullet Ret \Longrightarrow \mathrm{pre}\, AOp) \Longrightarrow (COp \wedge Ret' \Longrightarrow \exists Astate \bullet Ret \wedge AOp)$

US.3 $\forall Astate';\ Cstate' \bullet Cinit \wedge Ret' \Longrightarrow Ainit$

Downward and upward simulations are used to verify that a concrete specification is indeed a refinement of an abstract specification. They allow the state spaces to be different, but these must be linked by a suitable retrieve relation, and the concrete specification will typically be more deterministic than the abstract specification. For examples and discussion of the role of refinement see [Woodcock and Davies, 1996].

As an example, consider an implementation of the system process scheduler. The implementation uses sequences instead of sets to record the ready and waiting processes. It has also chosen a particular scheduling strategy, namely to take the process at the head of the ready queue. The implementation has therefore resolved some of the non-determinism in the abstract specification and also changed the state space. The specification is given as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{CState}\qquad\qquad\qquad\qquad}}\\
active : Pid \\
cready : \operatorname{seq} Pid \\
cwaiting : \operatorname{seq} Pid \\
admin : user \mid super \\
\hline
\operatorname{ran} cready \cap \operatorname{ran} cwaiting = \varnothing \\
active \notin (\operatorname{ran} cready \cup \operatorname{ran} cwaiting) \\
nil \notin (\operatorname{ran} cready \cup \operatorname{ran} cwaiting) \\
active = nil \Rightarrow cready = \langle\, \rangle \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{CInit}\qquad\qquad\qquad\qquad}}\\
CState' \\
\hline
active' = nil \\
cready' = cwaiting' = \langle\, \rangle \\
admin' = user \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{CNew}\qquad\qquad\qquad\qquad}}\\
\Delta CState \\
p? : Pid \\
\hline
p? \neq active \\
p? \notin (\operatorname{ran} cready \cup \operatorname{ran} cwaiting) \\
\operatorname{ran} cwaiting' = \operatorname{ran} cwaiting \cup \{p?\} \\
active' = active \\
cready' = cready \\
admin = admin' = user \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{CReady}\qquad\qquad\qquad\qquad}}\\
\Delta CState \\
q? : Pid \\
\hline
q? \in \operatorname{ran} cwaiting \\
\operatorname{ran} cwaiting' = \operatorname{ran} cwaiting \setminus \{q?\} \\
active = nil \Rightarrow (cready' = cready \land active' = q?) \\
active \neq nil \Rightarrow (\operatorname{ran} cready' = \operatorname{ran} cready \cup \{q?\} \;\land\; active' = active) \\
admin = admin' = user \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{CSwap}\qquad\qquad\qquad\qquad}}\\
\Delta CState \\
\hline
active \neq nil \\
\operatorname{ran} cwaiting' = \operatorname{ran} cwaiting \cup \{active\} \\
cready = \langle\, \rangle \Rightarrow (active' = nil \land cready' = \langle\, \rangle) \\
cready \neq \langle\, \rangle \Rightarrow (active' = head\ cready \land cready' = tail\ cready) \\
admin = admin' = user \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{CBoot}\qquad\qquad\qquad\qquad}}\\
\Delta CState \\
\hline
admin = super \\
active = nil \\
cwaiting = cready = \langle\, \rangle \\
admin' = user \land active' \neq nil \\
cready' = \langle\, \rangle \land cwaiting' \neq \langle\, \rangle \\
\hline
\end{array}
$$

This specification is a downward simulation of the abstract scheduler where the retrieve relation is given by

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Ret}} \\
\hline
\mathit{State} \\
\mathit{CState} \\
\hline
\mathit{ready} = \mathrm{ran}\ \mathit{cready} \\
\mathit{waiting} = \mathrm{ran}\ \mathit{cwaiting} \\
\hline
\end{array}
$$

It should be possible to test the concrete implementation against the abstract specification, by using the test cases and FSM calculated above. However, the transitions and states in the FSM are described in terms of the abstract state space (i.e. sets etc) and not the concrete realisation (e.g. sequences). Therefore *New1* or *New2* cannot be used to test *CNew* because the latter is an operation defined in terms of sequences whereas the tests are defined in terms of sets. In addition, the process of refinement potentially resolves or moves the non-determinism in the operations, for example the concrete operations might have weaker preconditions and stronger postconditions. For example, the operation *CSwap* has resolved non-determinism that was present in *Swap* (and so has a stronger postcondition), and in more complex situations (e.g. see [Derrick and Boiten, 1998b]) the non-determinism can be moved around the specification in quite subtle ways. Hence the FSM generated from the abstract specification cannot be used to test a concrete implementation except under the simplest of refinements. This can happen even in a specification close to implementation as the implementor can still implement any valid refinement and therefore change the specification in any of the ways mentioned above. Therefore it is necessary to provide a means to test such a refinement and instead of re-calculating the FSM from scratch, the abstract FSM is used together with the information about the refinement embodied in the retrieve relation. This means it is possible to generate a new FSM in a simple manner.

The idea of testing an implementation using a retrieve relation is not new. Dick and Faivre raise the problem in [Dick and Faivre, 1993]: "... the test-bed will have to be equipped with the means of converting between these abstract and concrete values. In ideal circumstances, retrieval functions could be implemented as part of the test-bed...", but provides no means by which to do this. The problem is also discussed in [Stocks, 1993], as it is in [Stepney, 1995]. However, the latter erroneously states that the abstract states can be used without conversion. Hörcher discusses the problem in more depth in [Horcher, 1995], and in particular comments that retrieve relations may be used in the production of test oracles. The purpose of this paper is to provide answers as to how exactly to achieve this and describe how properties of the retrieve relation effect the construction.

In order to do this it is necessary to *calculate* the most general refinement of an abstract specification together with a retrieve relation, a process now described.

3.1 Calculating Downward Simulations

Given an abstract specification, a concrete state space and a retrieve relation between the concrete and abstract state spaces, it is possible to calculate the weakest (most general) description of the concrete operations [Josephs, 1988, Woodcock and Davies, 1996, Derrick and Boiten, 1998a]. An operation *COp* is the weakest refinement of *AOp* whenever it describes precisely the same operation modulo the retrieve relation. If it is a refinement but not the weakest refinement, then it means it has made some further implementation choices such as resolving some of the non-determinism in *AOp*. Let *Astate* and *Cstate* be the abstract and concrete state spaces, *Ret* the retrieve relation and *AOp* an abstract operation. The weakest refinement *COp* of *AOp* is calculated by

$$\mathit{COp} \mathrel{\widehat=} \exists\, \mathit{Astate};\ \mathit{Astate}' \bullet \mathit{Ret} \land \mathit{AOp} \land \mathit{Ret}'$$

In general, if it is not known whether *Ret* defines a refinement, it is necessary to check the applicability. This is summarised in the following theorem (for a proof see [Derrick and Boiten, 1998a, Josephs, 1988]) which shows that COp is the weakest refinement of AOp, provided that one exists.

Theorem 1 *Let $\sqsubseteq_{DS}$ denote a downward simulation. Suppose that AOp specifies an operation over the abstract state space Astate. Let Cstate be a concrete state space, and Ret a retrieve relation between concrete and abstract. Let COp be defined as above. Then for every operation X*

$$AOp \sqsubseteq_{DS} X \text{ iff } \mathit{pre}\, AOp \land Ret \Rightarrow \mathit{pre}\, COp \text{ and } COp \sqsubseteq_{DS} X$$

In the context of this discussion it is known that applicability (pre $AOp \land Ret \Rightarrow$ pre COp) holds since tests are being generated for an existing development and therefore it is known that Ret defines a refinement. In these circumstances COp describes the most general concrete refinement of the operation AOp. It is also possible to calculate the most general concrete initialisation which will be given by

$$Cinit \mathrel{\widehat{=}} \exists\, Astate' \bullet Ainit \land Ret'$$

3.2 Calculating Upward Simulations

Some valid refinements can not be proved correct with a downwards simulation, and for these it is necessary to use an upwards simulation. An example of this is given below in Section 6.

For refinements that are upward simulations the following will define the weakest refinement of an abstract operation AOp (for a proof see [Derrick and Boiten, 1998a])

$$COp \mathrel{\widehat{=}} (\forall\, Astate \bullet Ret \Longrightarrow \mathrm{pre}\, AOp) \land \forall\, Astate' \bullet (Ret' \Longrightarrow \exists\, Astate \bullet Ret \land AOp)$$

Theorem 2 *Let $\sqsubseteq_{US}$ denote an upward simulation. Suppose that AOp specifies an operation over the abstract state space Astate. Let Cstate be a concrete state space, and Ret a retrieve relation between concrete and abstract. Let COp be defined as above. Then for every operation X*

$$AOp \sqsubseteq_{US} X \text{ iff } (\forall\, Astate \bullet Ret \Longrightarrow \mathit{pre}\, AOp) \Longrightarrow \mathit{pre}\, COp \text{ and } COp \sqsubseteq_{US} X$$

If the retrieve relation is a function, then the weakest refinement of AOp will be given by

$$COp \mathrel{\widehat{=}} \exists\, Astate;\ Astate' \bullet Ret \land AOp \land Ret'$$

a formula that is identical to the downward simulation case. For an arbitrary relation R it is still necessary to check applicability

$$\forall\, Cstate \bullet (\forall\, Astate \bullet R \Longrightarrow \mathrm{pre}\, AOp) \Longrightarrow \mathrm{pre}\, COp$$

However, if it is known that the retrieve relation does indeed define an upward simulation it is not necessary to check this.

3.3 Generating Tests

The technique employed to generate tests for a refinement is very simple. Given an abstract specification with operation AOp and a covering disjoint set of tests $\{AOp_i\}_i$; a concrete specification with operation COp which refines AOp, and a retrieve relation Ret, a set of tests $\{COp_i\}_i$ is generated where each test COp_i is the weakest refinement calculated from Ret and AOp_i. These new concrete tests will become the transitions in the new FSM. The states are generated using the retrieve relation in a similar fashion.

The remainder of the paper discusses the two cases of downward and upward simulations separately. In each case the following questions are explored:

- how are new transitions COp_i generated;
  - do these tests $\{COp_i\}_i$ cover COp;
  - are these tests $\{COp_i\}_i$ disjoint.
- how are new states and the FSM generated?

The following two sections discuss these questions when the refinement is a downward simulation, and Section 6 then looks at upward simulations.

4 Refining the Partition analysis

Downward simulations are perhaps the most common form of state based refinement, for example the concrete scheduler is a downward simulation of the abstract scheduler. This section discusses how the test cases of an operation change under such refinements.

Given an operation AOp with $AOp = \bigvee_i AOp_i$ being its disjoint set of tests, and a retrieve relation Ret, the concrete tests are given by

$$COp_i \mathrel{\widehat{=}} \exists\, Astate;\ Astate' \bullet Ret \land AOp_i \land Ret'$$

These will in some way represent test cases for the original concrete operation COp, and in fact the following result (for a proof see [Derrick and Boiten, 1998b]) holds.

Theorem 3 *Let AOp be an abstract operation with $AOp = \bigvee_i AOp_i$ being its disjoint set of tests. Let COp be a downward simulation of AOp. Let Ret be the retrieve relation. Let COp_i be the concrete tests given above. Then*

$$\bigvee_i COp_i \sqsubseteq_{DS} COp$$

and if COp is the weakest downward simulation of AOp then $COp = \bigvee_i COp_i$.

The practical consequences of this is that it is possible to use abstract tests (i.e. the partition analysis) together with the retrieve relation to calculate a new concrete partition analysis for the refinement. If the concrete is the weakest refinement of the abstract then these concrete tests exactly cover the concrete operations.

Example 1 *Calculating tests for a refinement.*

Consider the *New* operation in the abstract scheduler. Its test cases were *New1* and *New2*, these are used to calculate test cases $CNew1 \mathrel{\widehat{=}} \exists\, State;\ State' \bullet Ret \land New1 \land Ret'$ etc, which give

CNew1
$\Delta CState$
$p? : Pid$
$active' = active = nil$
$cready' = cready = \langle\rangle$
$p? \notin \operatorname{ran} cwaiting$
$\operatorname{ran} cwaiting' = \operatorname{ran} cwaiting \cup \{p?\}$
$admin = admin' = user$

CNew2
$\Delta CState$
$p? : Pid$
$p? \neq active$
$p? \notin (\operatorname{ran} cready \cup \operatorname{ran} cwaiting)$
$\operatorname{ran} cwaiting' = \operatorname{ran} cwaiting \cup \{p?\}$
$active' = active \neq nil$
$\operatorname{ran} cready' = \operatorname{ran} cready$
$admin = admin' = user$

It can be seen that $\bigvee_i CNewi \sqsubseteq_{DS} CNew$, however, $CNew$ is not the weakest refinement because the weakest refinement only requires that $\operatorname{ran} cready' = \operatorname{ran} cready$ whereas $CNew$ resolved this non-determinism to require that $cready' = cready$ (i.e. the order in the ready sequence must be preserved). So the calculated tests contain additional detail not included in the concrete operation.

However, in this case an exact covering can be constructed by replacing the test $CNew2$ by the test $CNew2 \wedge CNew$. Indeed this is a general strategy which works whenever the concrete operation has failed to be the weakest refinement because it has resolved more non-determinism than formally necessary.

This strategy is also applied to the $CBoot$ operation. $CBoot$ has resolved non-determinism present in $Boot$, it is therefore not its weakest refinement. Calculating tests $CBoot1 \wedge CBoot$ and $CBoot2 \wedge CBoot$ results in the latter evaluating to false, leaving just one test for the concrete boot operation given by:

$$
\begin{array}{|l}
\multicolumn{1}{l}{CBoot1} \\
\hline
\Delta CState \\
\hline
admin = super \\
active = nil \\
cwaiting = cready = \langle\,\rangle \\
admin' = user \wedge active' \neq nil \\
cready' = \langle\,\rangle \wedge cwaiting' \neq \langle\,\rangle \\
\hline
\end{array}
$$

$\Box$

This provides a means to calculate tests which cover the concrete operation, are these tests disjoint? For a functional retrieve relation disjoint abstract tests will generate disjoint concrete tests.

Theorem 4 *Let $\{AOp_i\}_i$ be disjoint test cases, Ret a functional (from concrete to abstract) retrieve relation and $\{COp_i\}_i$ calculated from $\{AOp_i\}_i$. Then $\{COp_i\}_i$ are disjoint.*

For a proof see [Derrick and Boiten, 1998b]. Note that disjointness is not the same as inequality (two tests with false predicates are considered disjoint).

Example 2 *Refined tests are disjoint for a functional retrieve relation.*

The retrieve relation for the scheduler is a surjective function from concrete to abstract. All the calculated tests are therefore disjoint, and the remainder of these are given by the concrete initialisation together with

$$
\begin{array}{|l}
\multicolumn{1}{l}{CReady1} \\
\hline
\Delta CState \\
q? : Pid \\
\hline
q? \in \operatorname{ran} cwaiting \\
\operatorname{ran} cwaiting' = \operatorname{ran} cwaiting \setminus \{q?\} \\
active = nil \\
cready' = cready = \langle\,\rangle \\
active' = q? \\
admin = admin' = user \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{CReady2} \\
\hline
\Delta CState \\
q? : Pid \\
\hline
q? \in \operatorname{ran} cwaiting \\
\operatorname{ran} cwaiting' = \operatorname{ran} cwaiting \setminus \{q?\} \\
active \neq nil \\
\operatorname{ran} cready' = \operatorname{ran} cready \cup \{q?\} \\
active' = active \\
admin = admin' = user \\
\hline
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CSwap}1} \\
\Delta \textit{CState} \\
\hline
\textit{active} \neq \textit{nil} \\
\operatorname{ran} \textit{cwaiting}' = \operatorname{ran} \textit{cwaiting} \cup \{\textit{active}\} \\
\textit{cready} = \textit{cready}' = \langle\,\rangle \\
\textit{active}' = \textit{nil} \\
\textit{admin} = \textit{admin}' = \textit{user} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CSwap}2} \\
\Delta \textit{CState} \\
\hline
\textit{active} \neq \textit{nil} \\
\operatorname{ran} \textit{cwaiting}' = \operatorname{ran} \textit{cwaiting} \cup \{\textit{active}\} \\
\textit{active}' \in \operatorname{ran} \textit{cready} \\
\operatorname{ran} \textit{cready}' = \operatorname{ran} \textit{cready} \setminus \{\textit{active}'\} \\
\textit{admin} = \textit{admin}' = \textit{user} \\
\hline
\end{array}$$

From this it can be seen that *CSwap2* also contains more non-determinism than is in *CSwap*, it can be further constrained by taking the test to be $\textit{CSwap}2 \wedge \textit{CSwap}$. Seven new disjoint concrete tests have been produced, calculated from the eight original abstract tests. □

Although refined tests are disjoint for a functional retrieve relation, in general disjointedness fails. This can be seen from the following example.

Example 3 *Refined tests are not disjoint in general.*

Consider the abstract and concrete schedulers. Suppose both specifications are limited to the three operations *Ready*, *New* and *Swap*, and that the specification of *CNew* is modified to the following:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CNew}} \\
\Delta \textit{CState} \\
p? : \textit{Pid} \\
\hline
p? \neq \textit{active} \\
p? \notin (\operatorname{ran} \textit{cready} \cup \operatorname{ran} \textit{cwaiting}) \\
\operatorname{ran} \textit{cwaiting}' = \operatorname{ran} \textit{cwaiting} \cup \{p?\} \\
\textit{active}' = \textit{active} \\
\operatorname{ran} \textit{cready}' = \operatorname{ran} \textit{cready} \\
\textit{admin} = \textit{admin}' = \textit{user} \\
\hline
\end{array}$$

The set based scheduler is now a refinement of *this* specification with the same retrieve relation as before. However, viewed this way round the retrieve relation is not functional: each set *ready* has many (abstract) representations as a sequence *cready* with $\textit{ready} = \operatorname{ran} \textit{cready}$.

One possible partition analysis for *CNew* would contain one test for each permutation of *cready*; for example, two such tests would be

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CNew}_1} \\
\Delta \textit{CState} \\
p? : \textit{Pid} \\
\hline
p? \neq \textit{active} \\
p? \notin (\operatorname{ran} \textit{cready} \cup \operatorname{ran} \textit{cwaiting}) \\
\operatorname{ran} \textit{cwaiting}' = \operatorname{ran} \textit{cwaiting} \cup \{p?\} \\
\textit{active}' = \textit{active} \\
\textit{cready}' = \textit{cready} \\
\textit{admin} = \textit{admin}' = \textit{user} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CNew}_2} \\
\Delta \textit{CState} \\
p? : \textit{Pid} \\
\hline
p? \neq \textit{active} \\
p? \notin (\operatorname{ran} \textit{cready} \cup \operatorname{ran} \textit{cwaiting}) \\
\operatorname{ran} \textit{cwaiting}' = \operatorname{ran} \textit{cwaiting} \cup \{p?\} \\
\textit{active}' = \textit{active} \\
\textit{cready}' = \textit{rev}\ \textit{cready} \\
\textit{admin} = \textit{admin}' = \textit{user} \\
\hline
\end{array}$$

Calculating the refined tests for each one of these abstract tests produces the same test *New* in every case. So *all* the abstract tests are mapped onto the same concrete test, which are therefore not disjoint. □

5 Refining the Finite State Machine

The previous section has discussed how to take transitions in an abstract FSM and calculate corresponding transitions to test an implementation with a concrete FSM. To construct the concrete FSM it is also necessary to define the states of the FSM and then describe the transitions between the states. This section describes the process for downward simulations, and also discusses how new transitions can be enabled due to the refinement process.

5.1 Refining the States

Suppose that an abstract FSM has n disjoint states $Astate_1, \ldots, Astate_n$, each one being a before or after state of an abstract transition AOp_i $(1 \leq i \leq m)$. The concrete states $Cstate_1, \ldots, Cstate_n$ are calculated by taking

$$Cstate_i \mathrel{\widehat=} \exists\, Astate \bullet Astate_i \land Ret$$

Each concrete state $Cstate_i$ will be a potential before or after state of a concrete transition. However, some abstract states might collapse (i.e. become identified) when their concrete counterparts are calculated. In fact it is not hard to see that the concrete states will be disjoint whenever the retrieve relation is a function, but that a general relation will not necessarily produce disjoint concrete states.

Example 4 *Calculating the concrete states.*

The concrete states can now be calculated for the scheduler, and since Ret is functional they will be disjoint. Upon calculation the states are:

$$\begin{array}{|l} \multicolumn{1}{l}{CState1} \\ \hline CState \\ \hline active = nil \\ cready = \langle\,\rangle \\ cwaiting = \langle\,\rangle \\ admin = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{CState2} \\ \hline CState \\ \hline active = nil \\ cready = \langle\,\rangle \\ cwaiting \neq \langle\,\rangle \\ admin = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{CState3} \\ \hline CState \\ \hline active \neq nil \\ cready = \langle\,\rangle \\ cwaiting = \langle\,\rangle \\ admin = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{CState4} \\ \hline CState \\ \hline active \neq nil \\ cready = \langle\,\rangle \\ cwaiting \neq \langle\,\rangle \\ admin = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{CState5} \\ \hline CState \\ \hline active \neq nil \\ cready \neq \langle\,\rangle \\ cwaiting = \langle\,\rangle \\ admin = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{CState6} \\ \hline CState \\ \hline active \neq nil \\ cready \neq \langle\,\rangle \\ cwaiting \neq \langle\,\rangle \\ admin = user \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{CState7} \\ \hline CState \\ \hline active = nil \\ cready = \langle\,\rangle \\ cwaiting = \langle\,\rangle \\ admin = super \\ \hline \end{array}$$

□

5.2 Building the FSM

A set of potential states $Cstate_j$ has been identified, as has a set of potential tests COp_i. It is now necessary to resolve the test cases against the states, and see if any new transitions are enabled due to the refinement process.

Because the concrete test calculations are given by

$$COp_i \mathrel{\widehat{=}} \exists\, Astate;\ Astate' \bullet Ret \wedge AOp_i \wedge Ret'$$

it is clear that in the weakest refinement there exists a transition COp_i between $Cstate_j$ and $Cstate_k$ precisely when there exists a transition AOp_i between $Astate_j$ and $Astate_k$.

Whether these are the only transitions depends on whether or not the concrete specification is the weakest refinement of the abstract with respect to the retrieve relation.

5.2.1 The concrete specification is the weakest refinement of the abstract

If the retrieve relation is functional, then disjoint abstract states and transitions produce disjoint concrete states and transitions. Because the concrete operations are the weakest refinement of the abstract ones, the concrete tests cover the concrete operations, no further non-determinism has been resolved and there exists a transition COp_i between $Cstate_j$ and $Cstate_k$ precisely when there exists a transition AOp_i between $Astate_j$ and $Astate_k$.

Therefore the concrete FSM will be isomorphic to the abstract FSM (isomorphic but not identical as the states and transitions are defined in terms of the concrete state space).

Example: in the concrete scheduler the operation *CReady* was the weakest refinement of *Ready*. Therefore the concrete tests and transitions for the *CReady* operation are identified precisely by direct reference to those due to *Ready* in the abstract FSM. □

If the retrieve relation is not functional, no new transitions can be enabled, but the concrete FSM potentially has fewer states and transitions than the abstract because disjoint states and transitions can be identified under refinement. However, this is the result of the calculation, and no further additions to the process are needed.

5.2.2 The concrete specification is not the weakest refinement of the abstract

When the concrete specification is not the weakest refinement, the process of refinement can potentially add new transitions or disable existing ones because refinement can both weaken an operation's precondition but also reduce any non-determinism in an operation by strengthening its postcondition.

When an operation's postcondition is strengthened, the set of potential outcomes is reduced. If one of these outcomes was a transition on its own, the concrete FSM can dispense with this transition. It was shown above that the actual transitions can be calculated by taking $COp_i \wedge COp$ where COp_i is the calculated test and COp the more deterministic concrete operation.

Example: in the concrete scheduler the postcondition in *CBoot* strengthens that of *Boot*, resulting in only one test for *CBoot* because $CBoot2 \wedge CBoot$ was false. The concrete FSM therefore does not have a transition *CBoot2*. □

Note that in general a transition is not always lost, whether it is depends on the partition analysis chosen. For example, in the concrete scheduler *CNew* and *CSwap* were not the weakest refinement

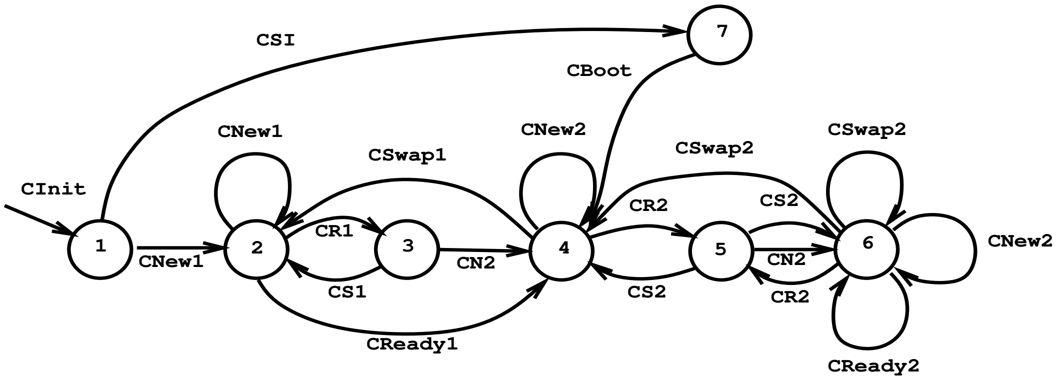

Figure 2: The FSM for the concrete scheduler.

of *New* and *Swap*. However, restricting the non-determinism in the tests $CNew2$ and $CSwap2$ did not disable them completely but just restricted their functionality.

When an operation's precondition is weakened under refinement, states and transitions have to be added in general to the concrete FSM, possibly having to recalculate new portions of the graph.

Example: suppose the concrete scheduler is refined further by replacing the *CSwap* operation by $CCS \mathrel{\widehat{=}} CSwap \lor CSI$ where CSI is given by

CSI

$$\begin{array}{|l}
\Delta CState \\
\hline
active = active' = nil \\
cready = cready' = cwaiting = cwaiting' = \langle\, \rangle \\
admin = user \\
admin' = super
\end{array}$$

This new swap operation has weakened the precondition by being enabled initially. Now tests $CSwap1$ and $CSwap2$ do not cover CCS exactly:

$$CSwap1 \lor CSwap2 \sqsubseteq CCS \text{ but } CSwap1 \lor CSwap2 \neq CCS$$

Under these circumstances it is necessary to include additional tests (in fact just one test here, CSI itself) and recalculate new states and transitions. New states might arise if the after-state of the additional test is not included in the existing concrete partition, which can happen if $\bigvee Cstate_i \neq Cstate$.

In fact, in this example this new test CSI adds only one new transition from $CState1$ to $CState7$, and the FSM can be adjusted accordingly. The final FSM is shown in figure 2. $\square$

Notice that by weakening an operation's precondition under refinement, portions of the graph which were unreachable have now become directly accessible. It was for this reason that unreachable states were not pruned in the initial abstract finite state machine.

6 Refinements due to Upward Simulations

As commented above, some valid refinements can not be proved correct with a downwards simulation, and for these it is necessary to use an upwards simulation. Here is an example.

Example 5 *An upward simulation.*

Consider the following two simple specifications. The abstract specification is:

$$\begin{array}{|l} \textit{Astate} \\ \hline x : 0..5 \\ \hline \end{array} \qquad \begin{array}{|l} \textit{Ainit} \\ \hline \textit{Astate}' \\ \hline x' = 0 \\ \hline \end{array} \qquad \begin{array}{|l} A \\ \hline \Delta \textit{Astate} \\ \hline x = 0 \land x' \in \{1, 2\} \\ \hline \end{array}$$

$$\begin{array}{|l} B \\ \hline \Delta \textit{Astate} \\ \hline (x = 1 \land x' = 3) \lor (x = 2 \land x' = 4) \\ \hline \end{array} \qquad \begin{array}{|l} D \\ \hline \Delta \textit{Astate} \\ \hline (x = 1 \land x' = 3) \lor (x = 0 \land x' = 5) \\ \hline \end{array}$$

The concrete specification is

$$\begin{array}{|l} \textit{Cstate} \\ \hline y : \{0, 1, 2, 3, 5\} \\ \hline \end{array} \qquad \begin{array}{|l} \textit{Cinit} \\ \hline \textit{Cstate}' \\ \hline y' = 0 \\ \hline \end{array} \qquad \begin{array}{|l} A \\ \hline \Delta \textit{Cstate} \\ \hline y = 0 \land y' = 1 \\ \hline \end{array}$$

$$\begin{array}{|l} B \\ \hline \Delta \textit{Cstate} \\ \hline y = 1 \land y' \in \{2, 3\} \\ \hline \end{array} \qquad \begin{array}{|l} D \\ \hline \Delta \textit{Cstate} \\ \hline y = 0 \land y' = 5 \\ \hline \end{array}$$

The concrete specification is an upward simulation (but not a downward simulation) of the abstract, where the retrieve relation is given by

$$\begin{array}{|l} \textit{Ret} \\ \hline \textit{Astate} \\ \textit{Cstate} \\ \hline x \in \{0, 3, 5\} \Rightarrow x = y \\ x = 4 \text{ iff } y = 2 \\ x \in \{1, 2\} \text{ iff } y = 1 \\ \hline \end{array}$$

□

In order to generate a FSM for an upward simulation the same methodology is adopted as before, that is, use the weakest refinement calculation to generate the transitions and the states in the concrete FSM. The exact means to do this depends on whether the retrieve relation is functional or not, and these cases are discussed separately.

When the retrieve relation is a function, the calculation of concrete tests is given by

$$\textit{COp}_i \mathrel{\widehat{=}} \exists\, \textit{Astate};\ \textit{Astate}' \bullet \textit{Ret} \land \textit{AOp}_i \land \textit{Ret}'$$

In this case all the calculations about states and transitions carry through from the downward simulation case, and the comments made before apply here in their entirety.

For a non-functional retrieve relation in order to generate concrete tests from the abstract test cases $\{AOp_i\}_i$ the following formula is used

$$COp_i \mathrel{\widehat{=}} (\forall\, Astate \bullet Ret \implies \mathrm{pre}\, AOp_i) \land \forall\, Astate' \bullet (Ret' \implies \exists\, Astate \bullet Ret \land AOp_i)$$

Since it is known that Ret defines a refinement (no need to check applicability), each COp_i is a refinement of AOp_i.

To see that the general formula is necessary, consider the operation D in the abstract specification given above. The *weakest* refinement of this is the refinement given in the concrete specification, however, notice that this refinement does not require a concrete transition that corresponds to the abstract one between 1 and 3. If the simpler calculation $COp \mathrel{\widehat{=}} \exists\, Astate;\ Astate' \bullet Ret \land AOp \land Ret'$ had been used, then the concrete specification of D would have included the option $(y = 1 \land y' = 3)$. The simpler calculation therefore constrains the operation too much, and is thus not the weakest refinement.

Although it is necessary to use a more complex formula for an upward simulation, the coverage and disjointedness results are similar to those for downward simulations, and we have the following [Derrick and Boiten, 1998b].

Theorem 5 *Let AOp be an abstract operation with $AOp = \bigvee_i AOp_i$ being its disjoint set of tests. Let COp be an upward simulation of AOp. Let Ret be the retrieve relation. Let COp_i be the concrete tests given above. Then*

$$\bigvee_i COp_i \sqsubseteq_{US} COp$$

and if COp is the weakest upward simulation of AOp then $COp = \bigvee_i COp_i$.

The disjointedness properties are also symmetric. When the retrieve relation is a function, the formulae for calculating tests are the same as for downward simulations. Therefore, as was the case then, disjoint abstract tests will produce disjoint concrete tests. However, in general refined tests are not disjoint.

Example 6 *Calculating concrete tests from an upward simulation.*

A FSM can be calculated for the above abstract specification which partitions the state into five states: $\{0\}, \ldots, \{5\}$ with the FSM shown in figure 3.

Following the procedure outlined above this is used to calculate a concrete FSM, which contains the following tests: A, $D2$ ($\mathrel{\widehat{=}} D$) and

$B1$
$\Delta Cstate$
$y = 1 \land y' = 3$

$B2$
$\Delta Cstate$
$y = 1 \land y' = 2$

Notice also that the concrete tests cover the operations (i.e. $B1 \lor B2 = B$). However, the concrete tests are not disjoint illustrating that functionality of the retrieve relation really is needed for disjointedness. To see this, note that the abstract tests $A1$ and $A2$ become the same concrete test A upon calculational refinement. □

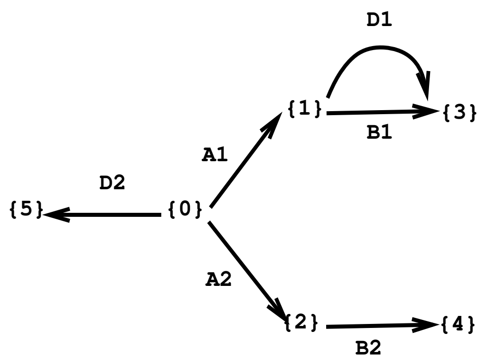

Figure 3: The FSM for the abstract specification.

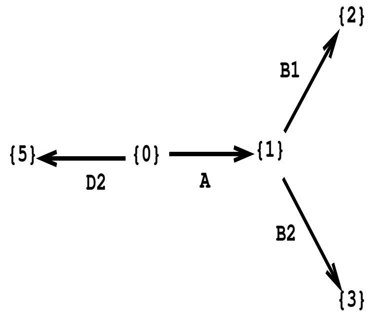

Figure 4: The FSM for the concrete specification.

Finally, having calculated the test cases it is necessary to calculate the states for a concrete FSM. To do so calculate $Cstate_i \mathrel{\widehat{=}} \exists\, Astate \bullet Astate_i \wedge Ret$ for each state $Astate_i$ in the abstract FSM. This gives, for a weakest refinement, all the possible concrete states in the new FSM. However, in general some of these states might be redundant because a transition has been disabled under the upward simulation. The example given above illustrates the situation where after calculation the test $D1$ does not appear in the refinement, and on calculating the states and checking the transitions we arrive at the concrete FSM shown in figure 4. Notice as well that the non-functionality of the retrieve relation has collapsed some of the states to produce non-disjoint concrete states. □

As a parting word a comment is in order concerning the changes in the partitioning of the operations under a refinement. Dick and Faivre posed the question: *does refining a specification create a super-set of the partitions of the previous level?* The answer is no in general for both types of simulations. For the upward case it has just been seen that the tests for operation A are a subset rather than a super-set of those on the previous level. In general there is no relation between the number of tests in a partition and those in a subsequent refinement: tests can be identified by a non-functional retrieve relation, on the other hand weakening the precondition can enable more tests to be included in the FSM.

Even for a functional retrieve relation and without weakening the preconditions, there is no relation between the number of tests in a partition and those in a subsequent refinement. This is because refinement can move non-determinism through a specification while preserving its observable behaviour. See [Derrick and Boiten, 1998b] for a discussion and illustration of this issue.

7 Conclusions

This paper has provided a means to calculate concrete FSMs from abstract ones for both upward and downward simulations. For retrieve relations which are functions the calculations simplified considerably, and in this case the calculations needed for upward and downward simulations coincide.

This can be used as a basis for a methodology for determining the correct concrete FSM calculation. Given abstract and concrete state spaces, a retrieve relation and abstract operations, generate a concrete FSM by

- generating concrete transitions COp_i from abstract transitions;
- generating concrete states $Cstate_j$ from abstract states $Astate_j$;
- resolving the concrete transitions against the states using the abstract FSM and properties of the retrieve relation.

To do this proceed as follows:

1. Determine whether Ret is a function. If it is, then the concrete tests are given by

   $$COp_i \mathrel{\widehat=} \exists\, Astate;\ Astate' \bullet Ret \land AOp_i \land Ret'$$

2. If Ret is not a function determine whether it defines a downward or upward simulation. Do this by determining if

   $$\mathrm{pre}\, AOp \land Ret \Rightarrow \mathrm{pre}\, COp$$

   If this is the case, then the refinement is a downward simulation, and therefore the concrete tests are still given by

   $$COp_i \mathrel{\widehat=} \exists\, Astate;\ Astate' \bullet Ret \land AOp_i \land Ret'$$

3. If Ret does not define a downward simulation, then the refinement must be an upward simulation. In this case the concrete tests are given by

   $$COp_i \mathrel{\widehat=} (\forall\, Astate \bullet Ret \Longrightarrow \mathrm{pre}\, AOp_i) \land \forall\, Astate' \bullet (Ret' \Longrightarrow \exists\, Astate \bullet Ret \land AOp_i)$$

4. Generate concrete states by taking

   $$Cstate_i \mathrel{\widehat=} \exists\, Astate \bullet Astate_i \land Ret$$

5. Build the concrete FSM. First check whether each concrete operation COp was in fact the weakest refinement, do this by determining if

   $$\bigvee_i COp_i = COp$$

   (a) If the concrete specification is the weakest refinement of the abstract and the retrieve relation is functional, then the concrete FSM will be isomorphic to the abstract FSM.

   (b) If the concrete specification is the weakest refinement of the abstract and the retrieve relation is not functional, then the concrete FSM potentially has fewer states and transitions than the abstract because disjoint states and transitions can be identified under refinement.

(c) If the concrete specification is not the weakest refinement of the abstract, then
   i. When an operation's postcondition is strengthened the actual transitions can be calculated by taking the tests to be $\{COp_i \wedge COp\}_i$.
   ii. When an operation's precondition is weakened it is necessary, in general, to add states and transitions to the concrete FSM, possibly having to recalculate new portions of the graph.

If COp is the weakest refinement of AOp then the set of tests $\{COp_i\}_i$ cover COp. If Ret is a function then the concrete tests and states will be disjoint whenever the abstract tests and states are disjoint.

One issue not considered in this paper is the sequencing of tests where a path through the FSM is found, preferably involving the minimum of duplication. Some of these issues are discussed in [Dick and Faivre, 1993], and it would be interesting to see whether paths can be refined in a similar way to the refinement of FSMs derived above. One issue noted above, however, was that when an operation's precondition is weakened upon refinement it was possible that previously unreachable portions of the FSM were now reachable. Any approach to refining test sequencing would have to address this problem.

References

[Abrial, 1996] Abrial, J. R. (1996). *The B-Book: Assigning programs to meanings.* CUP.

[Bowen et al., 1998] Bowen, J. P., Fett, A., and Hinchey, M. G., editors (1998). *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, 24–26 September 1998*, volume 1493 of *LNCS*. Springer-Verlag.

[Carrington and Stocks, 1994] Carrington, D. and Stocks, P. (1994). A tale of two paradigms: Formal methods and software testing. In Bowen, J. and Hall, J., editors, *ZUM'94, Z User Workshop*, pages 51–68, Cambridge, United Kingdom.

[Cusack and Wezeman, 1992] Cusack, E. and Wezeman, C. (1992). Deriving tests for objects specified in Z. In Bowen, J. P. and Nicholls, J. E., editors, *Seventh Annual Z User Workshop*, pages 180–195, London. Springer-Verlag.

[Derrick and Boiten, 1998a] Derrick, J. and Boiten, E. (1998a). Calculating upward and downward simulations of state-based specifications. Submitted for publication.

[Derrick and Boiten, 1998b] Derrick, J. and Boiten, E. (1998b). Testing refinements by refining tests. In [Bowen et al., 1998], pages 265–283.

[Dick and Faivre, 1993] Dick, J. and Faivre, A. (1993). Automating the generation and sequencing of test cases from model-based specifications. In Woodcock, J. C. P. and Larsen, P. G., editors, *FME'93: Industrial-Strength Formal Methods*, pages 268–284. Formal Methods Europe, Springer-Verlag. Lecture Notes in Computer Science 670.

[Grochtmann and Grimm, 1993] Grochtmann, M. and Grimm, K. (1993). Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3:63–82.

[He, 1989] He, J. (1989). Process refinement. In McDermid, J., editor, *The Theory and Practice of Refinement.* Butterworths.

[Hierons, 1997] Hierons, R. (1997). Testing from a Z specification. *Software Testing, Verification and Reliability*, 7(1):19–33.

[Horcher, 1995] Horcher, H.-M. (1995). Improving software tests using Z specifications. In Bowen, J. P. and Hinchey, M. G., editors, *Ninth Annual Z User Workshop*, LNCS 967, pages 152–166, Limerick. Springer-Verlag.

[Jones, 1989] Jones, C. B. (1989). *Systematic Software Development using VDM.* Prentice Hall.

[Josephs, 1988] Josephs, M. B. (1988). The data refinement calculator for Z specifications. *Information Processing Letters*, 27:29–33.

[Murray et al., 1998] Murray, L., Carrington, D., MacColl, I., McDonald, J., and Strooper, P. (1998). Formal derivation of finite state machines for class testing. In [Bowen et al., 1998], pages 42–59.

[Scullard, 1988] Scullard, G. (1988). Test case selection using VDM. In *VDM '88 VDM – The Way Ahead*, pages 178–186.

[Singh et al., 1997] Singh, H., Conrad, M., and Sadeghipour, S. (1997). Test case design based on Z and the classification-tree method. In Hinchey, M. and Liu, S., editors, *First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, pages 81–90, Hiroshima, Japan. IEEE Computer Society.

[Spivey, 1989] Spivey, J. M. (1989). *The Z notation: A reference manual*. Prentice Hall.

[Stepney, 1995] Stepney, S. (1995). Testing as Abstraction. In Bowen, J. P. and Hinchey, M. G., editors, *Ninth Annual Z User Workshop*, LNCS 967, pages 137–151, Limerick. Springer-Verlag.

[Stocks, 1993] Stocks, P. (1993). *Applying Formal Methods to Software Testing*. PhD thesis, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia.

[van Aertryck et al., 1997] van Aertryck, L., Benveniste, M., and Metayer, D. L. (1997). Casting: a formally based software test generation method. In Hinchey, M. and Liu, S., editors, *First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, pages 101–110, Hiroshima, Japan. IEEE Computer Society.

[Woodcock and Davies, 1996] Woodcock, J. and Davies, J. (1996). *Using Z: Specification, Refinement, and Proof*. Prentice Hall.

[Woodcock and Morgan, 1990] Woodcock, J. C. P. and Morgan, C. C. (1990). Refinement of state-based concurrent systems. In Bjorner, D., Hoare, C. A. R., and Langmaack, H., editors, *VDM '90 VDM and Z - Formal Methods in Software Development*, LNCS 428, pages 340–351, Kiel, FRG. Springer-Verlag.