Complete Behavioural Testing of Object-Oriented Systems using CCS-Augmented X-Machines

Mike Stannett and A. J. H. Simons

MOTIVE Project Team,
Verification and Testing Research Group, Department of Computer Science,
University of Sheffield, United Kingdom

{M.Stannett, A.Simons}@dcs.shef.ac.uk

http://www.dcs.shef.ac.uk/~mps

Abstract. We propose the combination of two established models of computation (*X*-machines and *CCS*) to generate a new behavioural specification and modelling language, *CCS-XM*. We show that such a language could be used to represent object-oriented systems, reason about their behavioural properties, and refine specifications into designs which are correct-by-construction. We delineate some of the semantics of such a language, and suggest areas where further research would be necessary.

1. Introduction

This paper describes approaches to deciding whether or not an object-oriented system is *correct* with respect to its specification. Answering this question is obviously important, but doing so is difficult because the concept of *correctness* is typically ambiguous for OO systems.

We develop our main theme in two directions. On the one hand, we acknowledge that functionality plays an important part in OO system specification. We therefore begin by asking whether it is always possible to find a finite test set which accurately distinguishes functionally correct implementations from functionally incorrect ones. We argue that this *can* be done, provided certain (realisable) constraints are imposed upon the implementations that are considered. In order to do this we represent OO systems as systems of *X-machines* [1, 2, 3, 4, 5, 6] a model-based formalism for which guaranteed testability results are already available [7, 8, 9, 10, 11].

At the same time, however, we uncover evidence that functionality cannot be the whole story. By adapting examples from concurrency theory, we show that two objects can be functionally equivalent, yet behaviourally distinct. The root of this discrepancy lies in the same computational features that inspired the creation the process calculus, *CCS* [12, 13] so it is natural to consider augmenting our *X-machine* model with *CCS* capabilities. The result is a joint formalism, *CCS-XM*, which inherits the full capabilities of both *CCS* and *X*-machines, just as multiple inheritance allows a derived class to inherit the full capabilities of two orthogonal parents.

By combining these two established formalisms within a single framework, *CCS-XM* allows us to represent not just the functional aspects of OO systems, but their non-functional conflicts and communications. We can, for example, model both synchronous and asynchronous interactions between objects (including demonic interference, where appropriate), and at the same time model the full computational effects of individual methods at all levels of refinement. Moreover, because every *CCS-XM* model is simultaneously a *CCS* model and an *X*-machine model, we can use existing tools from these domains to investigate *CCS-XM* models and the OO systems they represent.

Since *X*-machine concepts have rarely been applied in the OO literature, we begin with an explanatory summary.

1.1 Dual-refinement test strategy

We use the term *guaranteed testability* to mean that the correctness of a claimed implementation with respect to a known functional (later, behavioural) specification is decidable by observing the outcomes of a finite set of finite tests. The main claim of *X*-machine theory is that guaranteed testability can always be assured, but only if we're willing to pay a price. To see that some such price must be paid, we appeal to a basic tenet of recursive function theory, that no recursive algorithm is consistently able to decide correctly whether arbitrarily selected recursive functions are functionally equivalent to some fixed recursive test-function. When we say that an implementation is functionally correct, we are precisely asserting the equivalence of two recursive functions – we are saying that (some functional semantic interpretation of) the implementation is functionally equivalent to (some functional semantic interpretation of) its specification. Consequently, if we had an algorithm that could consistently determine OO correctness, we'd be able to circumvent the tenets of recursive function theory. Given any recursive function g, we'd first declare g to be a functional specification. Then, given some arbitrarily selected recursive function f, we'd construct an implementation of $Imp(f)$ in an OO language, whose functional semantic interpretation is precisely f. Our algorithm would then allow us to decide "Is $Imp(f)$ functionally correct with respect to g?" or in other words, "Is f functionally equivalent to g?"

It seems, then, that guaranteed testability ought to be a theoretical impossibility, but this is to misunderstand the possibilities inherent in the last paragraph. The conclusion we draw should not be "*guaranteed testability is impossible,*" but rather "*if testability is to be guaranteed, the assumptions of the last paragraph must be invalidated.*"

1.1.1 Functional descriptions are inadequate

One way to invalidate the argument is by showing that functions are incapable of specifying OO system behaviours (this is theme to which we will return later, but for now it is enough to illustrate the problem). This is because we appealed to the idea above that functions and behaviours are somehow interchangeable, and that a result about functions is 'really' a result about OO implementations. If functions aren't appropriate, this argument, that any system capable of testing arbitrary implementations against a specification is also capable of determining functional equivalence, loses its force.

In fact, arguments as to the inadequacy of functional descriptions are commonplace in concurrency theory; it was precisely this inadequacy that led to the development of early process languages in the first place. An easy counterexample is cited by Milner [13] in his introduction to *CCS*, and we re-apply it now in object-oriented form. Suppose we have two objects *obj1* and *obj2* belonging to the Java classes *Class1* and *Class2*, respectively, each defined by reference to some non-constant public static *int Static.Val* as shown in Fig. 1. From a functional point of view there is no difference between these two objects. Each has no members and no applicable methods and each was constructed by a default constructor whose functionality at completion is fully described as setting *Static.Val* to 10. From a concurrent point of view, however, the two classes are easily distinguishable, because they have the ability to interact with the object *demon* in different ways. Concurrent construction of *obj2* and *demon* can result in *Static.Val* taking the value 30, as the hypothetical interleaving in Table. 2 demonstrates, whereas no concurrent construction of *obj1* and *demon* can generate this outcome. On the principle that we should always be able to replace equals with equals, it is clear that *Class1* and *Class2* cannot be considered equivalent *as processes* even though they are functionally equivalent.

1.1.2 Specifications as design constraints

Another approach is to use the specification as an active constraint on the design process, because recursive undecidability of functional equivalence concerns the equivalence of *arbitrary* recursive functions.

```
public class Class1 {
  public Class1 () {
    Static.Val = 10;
  }
}
Class1 obj1 = new Class1();

public class Class2 {
  public Class2 () {
    Static.Val = 0;
    Static.Val += 10;
  }
}
Class2 obj2 = new Class2();

public class Demon {
  public Demon () {
    Static.Val = 20;
  }
}
```

Figure 1. Behavioural equivalence requires more than functional equivalence. In this system two objects (*obj1* and *obj2*) are functionally equivalent, both in terms of the methods available to them, and also in terms of construction effects. Nonetheless *obj2* is able to interact with a third object (*demon*) in a way that *obj1* cannot, as shown in Table 2.

Processor Ticks	*Active Statement In*		Value of *Static.Val*
	Class2	*Demon*	
0			*unknown*
1	Static.Val = 0;		0
2		Static.Val = 20;	20
3	Static.Val += 10;		30

Table 2. Concurrent execution of two constructors can cause side-effects that are not present when either constructor is considered in isolation. In this example, which continues an analysis of the classes *Class2* and *Demon* introduced in Fig. 1, interleaved executions of constructor commands results in *Static.Val* being set to 30 as a side-effect. Since *Demon* cannot interact with *Class1* to generate this side-effect we conclude that *Class1* and *Class2* are behaviourally distinct, even though they are functionally equivalent.

The undecidability of functional equivalence can be sidestepped if we transform a specification into an implementation one step at a time, and derive test cases as part of the same underlying procedure. No aspect of the implementation could then be considered arbitrary with respect to the specification, and the undecidability result would not apply. The idea that designs should be constrained directly by a high-level specification, and test-sets generated in tandem with those designs, is by no means new [14, 15]. Indeed, the designer's intent is always that the final implementation should conform to the initial specification. But we mean that the *design process itself* is constrained, so that designers don't have a completely free choice as to how they refine semi-abstract constructs at one level into more concrete code at the next. The feasibility of this approach rests entirely on the dualistic nature of specifications. On the one hand we can regard a specification as an implementation written in a high-

level programming language (for example, a functional specification might be regarded as a program in a functional language). On the other hand, we can regard a specification as a test-centred requirements document, because it tells us what outcomes should arise in response to which inputs and stimuli. This means we can refine a specification in two directions, both as a program-specification and as a test-specification, and provided we perform the two refinements in tandem we can arrange for tests at each stage of refinement to match the associated implementation as summarised is Fig. 3.

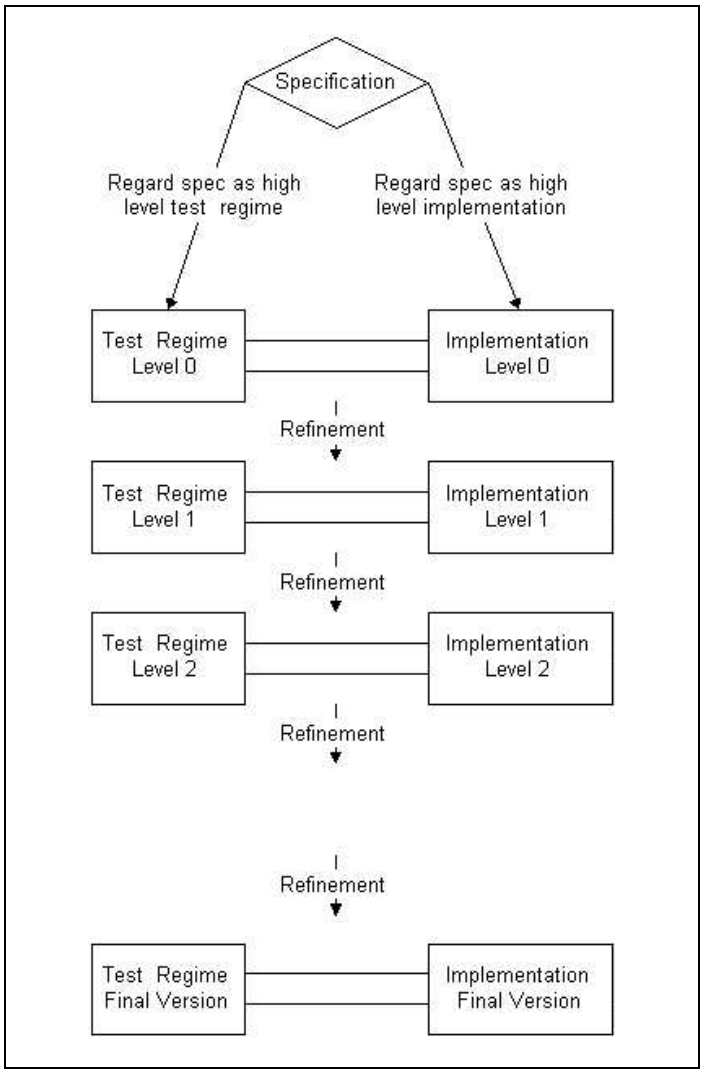

Figure 3. Dual-refinement strategy. A high-level description of the system, usually of a functional nature, can be regarded simultaneously as a system specification and a test-set specification. By progressively refining the specification in these two directions we ensure that the implemented system satisfies the tests, whence it is *correct-by-construction*

Similar considerations are involved every time we attempt to debug a program. Suppose, for example, that we want to develop algorithms to check whether or not any program we develop will eventually stop running, or instead run for ever. Theoreticians tell us that this *Halting Problem* is undecidable, and that no recursive-algorithm exists which can consistently tell us whether or not an arbitrarily presented program will eventually halt when supplied with some appropriate set of inputs. But this does not stop us noticing in Fig. 4 that the application *Halt* will definitely halt, while *RunForever* will definitely fail to do so.

```
public class Halt {
  public static void main (String[] args)
{
    System.out.println(“Stopped”);
  }
}

public class RunForever {
  public static void main (String[] args)
{
    while ( true )
      System.out.println(“Running”);
  }
}
```

Figure 4. The Halting Problem. Even though the Halting Problem is undecidable, we are still able to decide that the first of these classes, *Halt*, will halt while the second, *RunForever*, won't. Undecidability results concern arbitrarily presented entities, whereas systems developed by software engineers are subject to a welter of inter-relationships that can be used to help ensure testability

In daily practice we are rarely if ever concerned with "arbitrary" implementations; the objects, classes and systems we develop all bear some definite relationship to our original specification, and are developed subject to design rules, in languages, and inside development environments which conspire to limit the potential programs (and problems) that can arise. So even though arbitrary programs may be untestable relative to some given specification, it is reasonable to expect, and even to insist, that systems implemented under a dual-refinement strategy should be testable *by construction*.

1.2 X-Machines and SXM Testing

Dual-refinement has had particular success in the context of the *stream X-machines*. The stream *X*-machine (SXM) is a generalisation of the finite-state transducer, and was described briefly by Eilenberg in his comprehensive study of automata theory [1]; however, it is normally attributed to Laycock, who first investigated the model in detail [4, 16].

The SXM model has become increasingly important over the last ten years, because it has full recursive power (any recursive algorithm can be expressed), and yet its control theory is essentially identical to that of finite state transducers. It is consequently easy to extend standard transducer test strategies to the SXM with only minimal changes, and so obtain a test-strategy for general recursive algorithms. A comprehensive description of SXM testing is available as [8]. Of course, whenever one introduces a new model (whether of computation or anything else) one has to answer the obvious question: *why*? The development of SXM testing can be justified and motivated in several ways, and perhaps the simplest is by looking at the deficiencies of standard FSM models.

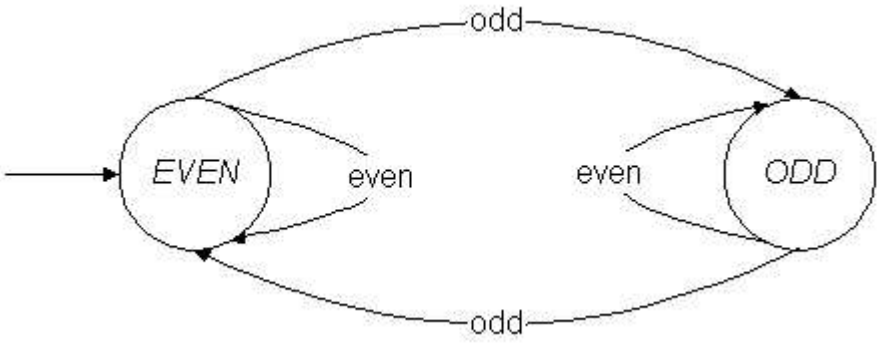

Figure 5. A simple finite state machine (FSM) model. This FSM models the transformation of parity during integer addition. The machine has two states (*EVEN* and *ODD*), and is initially *EVEN*, corresponding to a running total of 0L. The transitions indicate whether we are adding an *odd* or *even* value to the running total. If the total is currently *EVEN* and we add an *odd* number, the total becomes *ODD*, signified by following the transition labeled *odd* that leads from *EVEN* to *ODD*.

Fig. 5 shows a typical FSM model, in this case one that tracks the parity of a Java *long* to which various other odd or even *long*s can be added. Reasonable though FSM models may be from a basic point of view, consideration of this example reveals a serious problem. If we should happen to add 14 to 2017, for example, it is unlikely that knowing the answer is *ODD* will be good enough; we really need to know that it's 2031. However, modelling each potential value with a separate control state would result in unacceptable state explosion. An easy alternative is to annotate states with values, so that for example "the total is currently 2017, and is odd" would be modelled by the structured state, $ODD\{total \mapsto 2017\}$. The state has been split into two components, a *control state* (*ODD*) and an *internal state* ($\{total \mapsto 2017\}$).

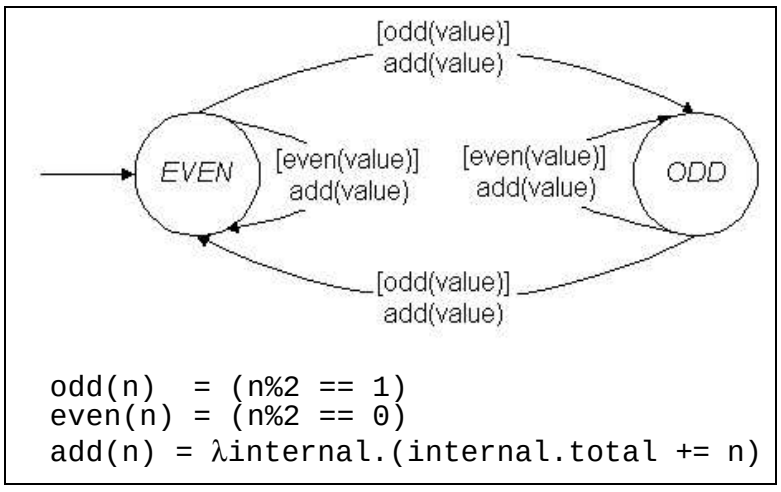

Figure 6. A simple structured-state model. If the total is currently *ODD* and we attempt to add 14, *i.e.*, apply *add(14)*, the total remains *ODD*, as signified by following the transition with satisfied guard *even(14)* that leads from *ODD* to *ODD*. At the same time the function *add(*14*)* is applied to the internal state, causing *total* to be incremented by 14. The control structure of this machine is identical to that of the FSM model shown in Fig. 5.

Of course, we cannot add structure to states and then simply forget about the transitions. If states contain values, then change of state typically involves changing those values, and the transitions have to tell us which changes occur. So when we add 14, instead of simply traversing an arrow labeled "*even*," we apply the function λ*state.(state.total+=14)* to the internal state, in this case adding 14 to the value of *Odd.total*. This basic structured model is shown in Fig. 6.

Each transition in a structured-state machine of this kind is labeled with a pre-condition and a function (or a relation if non-determinism is allowed), and the rule for firing a transition becomes, *if the precondition is met, apply the function to the internal state*. If the system state is *ODD*{*total* $\mapsto$ 2017} and we want to *add(14)*, the only transition that can be triggered is the one leading back to *ODD*. Applying *add(14)* causes the state to change to *ODD*{ *total* $\mapsto$ 2031 }.

More generally there might be several variables whose values conspire to delineate the current state, so the state might look more like this: *ODD*{*total* $\mapsto$ 2031, *author* $\mapsto$ "Mike", *previous* $\mapsto$ 2017}. If we were to program a simulation of this model, one way to represent the internal state would be as an object of the type *InternalState* defined in Fig 7.

```
class InternalState {
  long total;
  String author;
  long previous;
}
```

Figure 7. The general representation of internal state in an structured-state machine.

Each transition in the machine diagram would then be implemented as a function of type *InternalState* $\rightarrow$ *InternalState*. We call such a structured-state model of this kind an *InternalState*-machine (*i.e.*, a machine that manipulates values of type *InternalState*) and more generally, if the internal state can be represented by an object of type *X*, we call the model an *X-machine*. Everyday examples of *X*-machines include simple calculators (*Number*-machines), air conditioners (*Temp*-machines) and word processors (*Document*-machines).

This basic *X*-machine concept was introduced in [1], though its usefulness as a model of hierarchical refinement-derived systems was first identified in [2]. If we want to model reactive systems in which a stream of triggers is received and a stream of outcomes generated, the stream X-machine is more appropriate, because it focuses on step-by-step conversion of an input stream into an output stream. This isn't really an extension of the *X*-machine model so much as a special case, as it amounts to taking *X* to be of the form $X = Output^* \times Mem \times Input^*$ where *Mem* is some representation of internal memory, *Input* is the input alphabet and *Output* the output alphabet.

```
class InternalState {
  Output[] outStream;
  Mem      memory;
  Input[]  inStream;
}
```

Figure 8. The fundamental *InternalState* data type of a generic SXM, as it might be programmed in Java. The types *Output*, *Mem* and *Input* are assumed to be defined elsewhere.

Each transition acts by removing the front object from the input stream, using this to decide what processing function should be applied to *memory* (if more than one function can be applied the

behaviour becomes potentially nondeterministic), and then appending some appropriate outcome to the *outStream*.. That is, each transition function is essentially of the form ϕ: *Memory* $\times$ *Input* $\rightarrow$ *Output* $\times$ *Memory*. Because *X*-machine transition functions should formally be of type $X \rightarrow X$, we notionally encode the 'true' behaviour

$$\phi: (memory, input) \mapsto (output, memory') \quad (1)$$

in the relation

$$\phi: (outStream, memory, input::inStream) \mapsto (outStream::output, memory', inStream) \quad (2)$$

but in practice the abuse of notation (using ϕ to represent both functions) is tolerated.

The semantics of an *X*-machine is given in terms of the underlying FSM. The "meaning" of an FSM is usually taken to be the language it generates, in other words the set of all paths from valid initial to valid terminal states. Moving to the *X*-machine model each path now represents a sequence of basic operations to be applied to the internal state, and we take the meaning of the path to be the composition of these behaviours. The behaviour of the whole machine is then taken to be the union of all these *path relations*. Formally, we suppose the existence of a set $\Phi \subseteq [X \rightarrow X]$ of unary operators (more generally, relations) on a type *X*. An *X*-machine is a pair $M = (F, \Lambda)$ where *F* is a FSM over *A* and Λ: $A \rightarrow \Phi$ is some 1-1 relabelling of *F* which replaces symbols in *A* with functions from Φ. Writing $|F|$ for the regular language generated by *F*, the behaviour of *M* is defined to be the relation $|M| =_{def} \Lambda(|F|)$ where Λ is lifted to sequences (including the empty sequence, ε) and sets of sequences, in the usual way, *i.e.* $\Lambda(\varepsilon) = \mathbf{1}_X$, $\Lambda(a_1 \ldots a_n) = \Lambda(a_1) \ \ldots \ \Lambda(a_n)$, and $\Lambda(\cup \mathbf{word}_i) = \cup\Lambda(\mathbf{word}_i)$.

1.3 X-machines and Testability

X-machines and stream *X*-machines are essentially interchangeable as models, and each is at least as descriptive as standard recursion theory. This is because a Turing machine can be thought of as a *Tape*-machine, where *Tape* is some representation of the standard Turing tape – each of the standard Turing-computation steps (read a symbol, move the read/write head, etc) can easily be modelled as an operator manipulating *Tape* while changing control state. (Indeed we can actually compute *any* set-theoretic relation ϕ: $Y \quad Z$ between arbitrary types *Y* and *Z* by constructing a $(Y \times Z)$-machine based on the single-transition automaton " $\rightarrow^a$ " and mapping *a* to an encoding of ϕ, *e.g.*, $\phi'(y, anything) = \{(anything, z) \mid z \in \phi(y)\}$ [17, 18, 19]. In principle this can introduce serious theoretical problems, because we need to ensure at each stage that all constructs remain computable. However, the operations used in this paper have been chosen to ensure we remain clearly within recursive territory at all times, and we accordingly provide no formal demonstration of this fact.)

That *X*-machines can simulate Turing computations is remarkably fortuitous, for unlike Turing machines and push-down automata, with their notorious habit of confusing the control and processing aspects of almost every non-trivial algorithm, *X*-machines maintain a clear distinction between control and processing at all levels of representation. This makes them ideal for modelling and especially reasoning about objects, and it was with this in mind that the UK-EPSRC project *MOTIVE* (Method for Object Testing, Integration and Verification) was established at Sheffield in 1999 to extend results obtained for non-OO systems over the preceding decade. Holcombe [2] had observed that *X*-machines were well-suited to top-down refinement-based modelling, in which simple high-level machine descriptions could be progressively redefined, both by replacing transition functions with sub-machines that implement them, and by subdividing control states into families of sub-states (much as happens in the development of statechart models [20, 21, 22, 23, 24, 25]). Laycock's subsequent development of the stream *X*-machine opened the door to extensions of transducer-based testing. and case studies followed [26, 27, 28] which quickly demonstrated the feasibility of using refinement-based *X*-machine modelling for realistic design problems (and Vanak [29] has recently obtained promising results showing the applicability of SXM testing to realistic hardware problems). At the

same time, major developments occurred in the underlying theory of the model, both in terms of delineating its power relative to standard models of recursion [3, 19], and with specific regard to system testing. In particular, Holcombe and Ipate [5, 7, 30, 8] showed that the dual-refinement approach could be used to ensure *complete* functional testability of SXM systems, in the sense that test-sets can be generated with the property that if the system passes every test it is *guaranteed* to be correct with respect to its specification. (Because we will not be developing their techniques further, we do not include give a detailed exposition of SXM testing in this paper – we focus instead on the reasons *why* the method works. For further details of the method itself we recommend the later chapters, including the appendices, of [8].)

The Holcombe/Ipate approach is based on Chow's *W-method* [31] for verifying whether or not two finite state transducers implement the same input-output relation. Given two such transducers we suppose that one of them, the specification (*Spec*), is fully observable; we are free to examine its internal structure, its transitions, its state set and so forth. Since we can always use this information to minimise the machine if necessary, it is assumed that the specification machine is already minimal, and hence deterministic. The second machine is some implementation (*Imp*), and as this may have been generated by third party vendors we assume nothing except what can be determined by supplying inputs and observing the associated output behaviour. For technical reasons we assume that the two machines are defined over the same alphabet, *A*, and writing $m = \#Spec$ and $n = \#Imp$ for the number of states in each of these machines, we assume further that an upper bound on $|n - m|$ can be determined (or is independently specified). A basic theoretical result [*see e.g.*, 1, p. 34] guarantees under these conditions that $|Imp| \subseteq |Spec|$ whenever the two machines behave identically for all strings of length less than $2^m n$. If we can further assume that *Imp* is also deterministic this bound can be reduced to *mn*, but even so this represents an unfeasibly difficult task for all but the smallest systems, because it requires us to perform $O(exp(mn))$ tests in order to demonstrate functional equivalence of the two transducers. Chow's method is important, not because it establishes that test sets can be constructed, but because it reduces the size of the test set to tractable proportions.

Given that the behaviour of a stream *X*-machine *SXM* can be expressed as $|SXM| = \Lambda(|F|)$, where F is the underlying transducer, Chow's method clearly gives us a method for testing the functional equivalence of SXM behaviours. For suppose *Spec* and *Imp* happen to be the transducers underlying the stream *X*-machines $\underline{Spec}$ and $\underline{Imp}$, so that $\underline{Spec} = (Spec, \Lambda_S)$ and $\underline{Imp} = (Imp, \Lambda_I)$. In order to demonstrate that the two stream *X*-machines implement the same functional behaviour we have to show that $|\underline{Spec}| = |\underline{Imp}|$, or in other words, that $\Lambda_S(|Spec|) = \Lambda_I(|Imp|)$. Since Chow's method tells us how to check whether $|Spec| = |Imp|$, all that remains is to ensure that the relabellings Λ_S and Λ_I behave identically, and this is handled by imposing sufficient constraints on the designer to ensure constructive provability of the statement "$\Lambda_S = \Lambda_I$." For example, we typically impose an *output distinguishability* constraint that requires of the designer that we can distinguish labels *a* and *b* in *A* simply by examining the outputs we obtain on applying the relations $\Lambda(a)$ and $\Lambda(b)$ to some suitable input. We also require that the refinement process be well-founded, in the sense that all refinements eventually result in components being used that are inherently trustworthy — for example, when refining a design into a *C++* program we insist that the program eventually reach a stage where commands are expressed entirely in terms of primitive, or at least trusted, classes and objects. Design-for-test (DFT) constraints of this kind are commonly encountered in hardware design [14, 15, 32], where they reduce construction costs by allowing us to generate systems that are inherently testable. By importing the concept into software design, *X*-machine models allow us to translate statements about observed output sequences at the SXM level into deductions about the paths that must (or alternatively cannot) exist at the underlying transducer level, thereby enabling us to affirm or refute functional correctness.

1.4 Summary

SXM test-generation techniques provide an established means of establishing the functional correctness of an implementation relative to its specification, subject to the design process provably satisfying well-defined DFT constraints. This suggests that SXM techniques might offer a useful

platform on which to base a similar DFT approach for objects and object-oriented systems, provided certain obstacles can be overcome.

There are many contributing factors at work here. For example, SXM testing is based on the idea that the system is constructed top-down from a functional system specification, with machine functionality and test-sets being refined in tandem. But we have already observed that functional descriptions are not always capable of distinguishing objects which are known to be behaviourally distinct. Moreover, requiring the design to evolve by top-down functional refinement encourages the development of components that are optimised for the specific system under construction, and this flatly contradicts the central OO principle of component re-use. The biggest omissions, however, are nondeterminism and concurrency. *X*-machine semantics are based on the idea that each run of a system causes the traversal of precisely one path through the underlying automaton, and SXM design-for-test conditions (which are imposed precisely to simplify scenarios) typically block nondeterminism at the earliest opportunity. This situation is changing, albeit slowly. In addition to various attempts to introduce communication [6, 33] between *X*-machines (these are discussed in more detail in the next section), the theoretical analog- and timed- variants of the *X*-machine [3, 19] allow multiple execution paths to exist simultaneously, but they do not yet support explicit communication, nor have they ever been applied to testing. Moreover, researchers have recently found ways to start relaxing the design-for-test conditions [34] for SXM-based models. Hierons and Harman in particular [9, 11] have started moving the focus away from functional *equivalence* (the idea that two systems must be exactly equivalent) towards functional *conformance* (the implementation should not violate the terms of the specification), and in so doing have developed a significant generalisation of Chow's method (their technique can be applied more widely than Chow-based SXM testing, and collapses into Chow's method for the more constrained systems). Even so, this work relaxes only the conditions on nondeterminism – concurrency is not addressed.

We have discussed *X*-machines at length because we believe passionately that they have the potential to play a significant role in the modelling and semantics of OO systems, and indeed the work we describe in this paper represents just one approach of many to extending SXM techniques into the OO system domain. We do not claim that *X*-machines are ideally suited to the task as they currently stand, but that they are "good enough" to warrant the small amount of effort needed to improve their relevance as tools. Some of important features of *X*-machines include:

- *clean separation of control from processing*.

  No matter how complex the operation modelled by an *X*-machine, its underlying control mechanism is just a finite state machine.

- *the ability to model any recursive behaviour*

  X-machines have full representational power. An *X*-machine's processing power comes from its transition functions, not from its control structure.

- *complete functional testing subject to design-for-test rules*

  Because stream *X*-machines are essentially homomorphic images (under relabellings, Λ) of finite state transducers, standard FSM test methods can be extended easily to allow testing of SXM functional correctness, provided we agree to constrain the designer by applying design-for-test rules.

Equally, however, *X*-machines display certain glaring deficiencies, not the least of which is the need to incorporate

- message handling (both synchronous and asynchronous)
- inheritance

We deal with these omissions next, by showing how *X*-machine transition functions have an intuitively natural interpretation within the paradigm of stimulus-response behaviours. But first we should explain briefly why we feel a new approach is warranted, when several formal approaches to object-level concurrency are already established in the literature.

Various object-oriented formal methods approaches have been summarised as part of the ground work for the ESPRIT DeVa (Design for Validation) project [35], and we take this as our point of reference. The DeVa team observed that formal OO specification languages could be divided into four categories, according to their underlying formalism:

1. first-order logic and set theory
2. algebras and algebraic specification
3. Petri nets and high-level nets
4. temporal logic

The language we describe below cannot easily be assigned to any of the four DeVa categories, although it is related to all four. Our language, *CCS-XM*, uses the process calculus *CCS* to annotate *X*-machine transitions, and these in turn have a semantics defined in functional set-theoretic terms. Our language has no direct connection to Petri nets, and yet the net-based OO specification language CO-OPN/2 [36] incorporates descriptions expressed in *Hennessey-Milner Logic* (HML) [13, 37, 38], a modal logic which is associated with CCS process specifications, and which displays certain structural similarities to temporal logics.

We should also remark that CCS is already associated with a theory of testing. If we suppose the existence of an action σ (*success*) in which our system might engage, then correctness of the system is in some sense established as soon as we demonstrate that σ *must* eventually occur, no matter how the system evolves during execution. HML specifically includes constructs which enable us to assert or deduce statements like "σ may occur" and "σ must occur," so there is a strong argument for thinking of *CCS* as a natural framework in which to examine the satisfaction of test criteria. On the other hand, this approach to testing is far more theoretical in nature than a typical OO programmer might envisage. Her task is rarely "tell me whether success will eventually occur", so much as "tell me what constitutes success in the first place."

2. CCS-augmented X-Machines

CCS (or *process calculus*) is essentially a mathematical toolkit for analyzing aspects of concurrent computation; it has its origins in the world of Artificial Intelligence *circa* 1972 [12, 13] when the realisation first became widespread that concurrent systems cannot always be completely described by functional specifications. We provide here a basic description of the constructs available in *CCS*, but cannot include a comprehensive introduction. Equally important for our purposes, though not included in these main *CCS* references, is the existence of the *Concurrency Workbench* (CWB) [37], a freely available tool for simulating *CCS* processes and testing them against statements written in Hennessey-Milner Logic (HML) and its extension the modal μ-calculus [38]. For those familiar with *CCS*, a word of warning. Standard expositions of *CCS* make use of typographic conventions that are not available when constructing files for use with the CWB, so we avoid them and stick to CWB format throughout. The version of CWB used in our work is the *Concurrency Workbench of the New Century* (CWB-NC), which may be obtained free of charge from `http://www.cs.sunysb.edu/~cwb`.

2.1 Constructs available in CCS

Left to themselves component processes evolve by performing one action (possibly one of several different actions) at a time, so we can model processes as labeled transition systems. Communication between processes is modelled by requiring labels to come in two complementary flavours (the

complement of a will be denoted a', so that $a'' = a$). Two components can communicate by engaging in complementary actions a and a'. Labels are normally interpreted as the names of communications channels, so that the action $a(x)$ indicates receipt of a value through the channel a, which is then bound to the variable x, and $a'(v)$ represents the export of the value v over the same channel. Frequently, however, value-passing is absent and we focus on the act of communication itself. One label, τ, plays a special role. It denotes *silent*, or *internal*, activity which is unobservable outside the process. It is the only label which is deemed self-complementary, so that $\tau = \tau'$.

Nondeterministic choice between two behaviours P and Q is denoted $P+Q$. This process is free to choose whether it behaves like P or like Q. Notice that this is not the same as randomly choosing between the two processes – an outcome can be nondeterministic even when it is inevitable, as long as we have no means of determining what that inevitable outcome will be. Where large numbers of alternative behaviours are possible we can also use summation notation, *e.g.* ΣP_i is a process which chooses one of the behaviours P_i. Choice is deceptively subtle, and almost all unexpected features of *CCS* arise through the interaction of choice with silent actions. In "functional specification" terms, for example, the processes "$\tau.(a + b)$" and "$\tau.a + \tau.b$" are identical, because they both satisfy the post-condition "precisely one of a or b is observed" (with pre-condition *true*). Nonetheless they are distinct as processes because we can perform an experiment that distinguishes between them. Imagine a black box with two buttons, labeled a and b. Each button can be depressed if and only if the process hidden within the box is still capable of performing the associated action. If the box contains "$\tau.(a + b)$", then as long as we haven't observed an a or a b, both are still available. On the other hand, if the box contains "$\tau.a + \tau.b$" then the process can silently choose (say) "$\tau.a$" to be its behaviour, and now it is no longer possible to press b even though nothing appears to have happened to preclude the choice.

If P is a process, and A is a set of actions, we write $P\backslash A$ ("P with A restricted") for the process that behaves exactly like P, except that displaying any of the behaviours cited in A is illegal. For example, consider the process *drink.Drunk*, which represents the behaviour "receive a *drink*, and by doing so become *Drunk*." If we restrict *drink*ing, you cannot become *Drunk*, because (*drink.Drunk*)\{*drink*} cannot legally perform any action – the only activity it could potentially perform initially is *drink*, but this is illegal. A process like this which is incapable of performing any action is denoted 0 or *NIL*. We typically ignore such processes (as we have above) when writing out behaviours, and write, *e.g.*, "*a.b*" in place of "*a.b*.0". Restriction plays an important role in defining object boundaries.

Communication can occur either synchronously or asynchronously. Synchronous communication is like shaking someone's hand – it happens for both the sender and the receiver at the same time, and if you know the message was sent, you automatically know it arrived. In OO terminology a slightly looser interpretation is allowed, and we usually consider *method calling* to be synchronous activity provided the caller has to wait for a response before continuing with other autonomous business. Asynchronous communication, on the other hand, is like posting a letter, and typically describes *message passing*, in which the message may arrive a substantial time after it was sent. The sender will typically have moved onto other business in the mean time, and cannot normally know whether the message was eventually received unless the recipient acknowledges the message with a receipt of some kind.

The special nature of synchronous communication shows up in the way communication interacts with restriction. Suppose there are two components whose behaviours can be described as P' and Q'. We write $P' \mid Q'$ for the system obtained by allowing P' and Q' to run simultaneously. If P' and Q' have no channels in common, they cannot influence one another, but if they share one or more channels they might be able to do so by communicating with one another. Suppose, then, that $P' = a.P$ and $Q' = a'.Q$, where a is the only channel the two processes have in common. Then $P' \mid Q'$ is the same thing as $(a.P) \mid (a'.Q)$, and because the two components of this combined process can do complementary actions they are capable of communicating with one another. They can do so asynchronously in two ways,

$$(a.P) \mid (a'.Q) \quad \rightarrow^{a} \quad P \mid (a'.Q) \qquad \rightarrow^{a'} \quad P \mid Q \tag{3}$$

$$(a.P) \mid (a'.Q) \quad \rightarrow^{a'} \quad (a.P) \mid Q \qquad \rightarrow^{a} \quad P \mid Q \tag{4}$$

and synchronously in one way

$$(a.P) \mid (a'.Q) \quad \rightarrow^{\tau} \quad P \mid Q \tag{5}$$

where we regard the shared synchronous communication event as being internal to the combined system. If we now restrict on $\{a\}$, both of the asynchronous behaviours become illegal, because they refer to a directly, whereas the synchronous communication is unaffected. The synchronous communication is hidden from view and not covered by the restriction, so we deduce that the only legal transition is

$$(\,(a.P) \mid (a'.Q)\,)\backslash\{a\} \quad \rightarrow^{\tau} \quad (P \mid Q)\backslash\{a\} \tag{6}$$

When two components A and B are allowed to run in parallel, and we then restrict on all their internal communications channels so that they become invisible to the outside world, we write the combined system as $A\|B$. Thus the above derivation could also be written

$$(a.P) \,\|\, (a'.Q) \quad \rightarrow^{\tau} \quad P \,\|Q \tag{7}$$

2.2 CCS-augmented X-Machines (Process X-Machines)

For the sake of brevity we shall call a *CCS*-augmented *X*-machine a *process X-machine* (PXM). The relationship between process X-machines, stream X-machines and *CCS* is straightforward – in the absence of communication process X-machines and stream X-machines are behaviourally equivalent; and moreover all *CCS* processes can be modelled as PXMs all of whose transition relations are taken to be the identity relation on X.

A process X-machine is defined to be a (typically disconnected) finite state machine structure, but with three-part labels (*3-actions*) of the form $\langle a \mid \phi \mid b' \rangle$, where a and b' are *CCS* channels and ϕ is a relation on X. Traversing an arc with this label can loosely be interpreted as executing the behaviour

1 Accept a signal on a

2 Perform the computation defined by ϕ

3 Generate a signal on b'

Again speaking loosely, we can interpret the *CCS* components of a 3-action as pre- and post-conditions determined through communication. Before ϕ can be initiated the 'pre-condition' a must be satisfied (a signal must be received over a), and for the post-condition to be asserted (before a signal can be sent over b') the computation must succeed. In particular, then, if ϕ cannot be applied to the current internal state, the computation of ϕ is usually deemed to have failed, and no output signal is generated.

In order to simulate pure X-machine transitions observationally, we set a and b' to τ, since the 3-action $\langle\tau|\phi|\tau\rangle$ always results in exactly the observable effect as simply applying ϕ while engaging in no communication. Similarly, the 'pure' *CCS* actions a and b' are simulated by the 3-actions $\langle a|1|\tau\rangle$ and $\langle\tau|1|b'\rangle$, where 1 is the identity function on X. From an observational viewpoint, a successful transition $\langle a \mid \phi \mid b' \rangle$ can always be factorised as a chain of three successive actions

$$\langle a \mid \phi \mid b' \rangle = \langle a \mid 1 \mid \tau \rangle \langle \tau \mid \phi \mid \tau \rangle \langle \tau \mid 1 \mid b' \rangle \tag{8}$$

and this allows us to refine single transitions into machines which generate the same overall behaviour, because our embedding of X-machines within *CCS-XM* is faithful: if we replace every transition ϕ in a X-machine M with the equivalent 3-action $\langle\tau|\phi|\tau\rangle$, the result is a PXM whose overall

behaviour is precisely $\langle 1| \Phi |\tau\rangle$, where $\Phi = |M|$. However, if the composite transition is unsuccessful, typically because ϕ cannot be applied in the current system state, the decomposition reduces to

$$\langle a \mid \phi \mid b' \rangle = \langle a \mid 1 \mid \tau \rangle \langle \tau \mid \varnothing \mid \tau \rangle \quad (9)$$

because the final action, *b*', is never enabled. The difference between these two situations must often be determined dynamically, because the ability to apply ϕ depends on the changes to internal state caused by binding any values received over *a*.

2.3 Process X-machines vs. Communicating X-machines

Process *X*-machines can be regarded as a form of "communicating X-machines" [6, 39], in the sense that distinct machine components are able to exchange information, and so modify their neighbours' behaviour. However, we need to use this term with care, because the terminology "*communicating stream X-machine*" (CSXM) refers to a formally defined member of the *X*-machine family. Technically, a system of CSXMs communicates via a shared memory, which we can think of as a square array of pigeonholes. If CSXM#4 wishes to send a message to CSXM#11, for example, it places the message in the pigeonhole with coordinates (4,11). It is then barred from sending further messages to CSXM#11 until the first message is removed from the pigeonhole (only CSXM#11 can do this). It is possible to show a CSXM system of this kind can be fully simulated by a single monolithic SXM, as indeed one would expect, because the communication structure possible in such a system is known *a priori* – each component can communicate with every other component across precisely one channel, and the channel can hold at most one message at a time.

We can easily model CSXM systems using process X-machines, simply by including a model of the shared memory. Modelling the actual processing relations is straightforward, so we focus here on the communication events. We know that messages from CSXM#*m* to CSXM#*n* are shunted via the (*m,n*)-pigeonhole, and that only one message can be in the pigeonhole at a time. Accordingly, if PXM#*i* denotes out PXM model of CSXM#*i*, wherever CSXM#*m* exports *info* to the (*m,n*)-pigeonhole we model this in PXM#*m* as the *CCS* export action *setmem*[*m,n*]'(*info*). The corresponding import action performed by PXM#*n* is likewise modelled as a *CCS* import action *getmem*[*m,n*](*val*). We model the corresponding behaviour of the (*m,n*)-pigeonhole as a process *Mem*[*m,n*] defined recursively by

$$Mem[m,n] = setmem[m,n](x).BusyMem[m,n]\ (x) \quad (10)$$

$$BusyMem[m,n]\ (x) = getmem[m,n]'(x).\ Mem[m,n] \quad (11)$$

Given these definitions it is easy to check that (PXM#*m* | *Mem*[*m,n*] | PXM#*n*) is a process which exactly simulates the "send via pigeonhole" protocol of CSXM systems.

The converse is not true, however, because multiple PXMs can share the same channel names. Consequently PXM models inherently support the modelling of channel conflicts, whereas CSXM models do not because each location in shared memory can be written to by only one component, and likewise read-from by only one component.

3. Worked Examples

So far the material we've presented has been rather theoretical, because we needed to explain that the methods illustrated here are well-founded. However, the best way to demonstrate how these methods work is by example. Accordingly we'll start simple and gradually complicate matters.

3.1 Static variables

Consider Fig. 9, in which the value of a static C++ variable is assigned a new value. The stages involved in modelling the assignment "`t.value = 100`" are

1 *t* sends the request to *T*

2 *T* receives the request, then performs the request, then confirms completion

3 *t* receives the confirmation message

In PXM terms, synchronisation occurs between the following two arcs, one in the *t*-machine, the other in the *T*-machine, where *SetValue(x)* is a relation-label in the *T*-machine whose effect is to set *value* to *x*.

```
struct T {static int value; }
T t;
t.value = 100;
```

Figure 9. Changing the value of a static value

We consider this example in detail, for once this example is understood, the rest will follow easily. Figure 10 shows fragments of two distinct PXMs. One represents an *object*, the other a *class*, and the required effect is achieved by synchronizing over the shared channels *setvalue* and *ack_t*. By restricting the synchronized system (*i.e.* constructing (*object-machine* || *class-machine*) we ensure that the interactions remain hidden within the relevant class and object boundaries. This is standard in PXM representations. Regardless of the level at which an entity is defined, if it can appropriately be regarded as the *owner* of data, we model it as a PXM, because any attempts to access that data will involve it in communications.

The *object*-machine contains various transitions, of which two consecutive ones are shown. One of these ends by exporting the values 100 and *this* on the channel *setvalue*; the next can proceed only if a signal is received along *ack_t*.

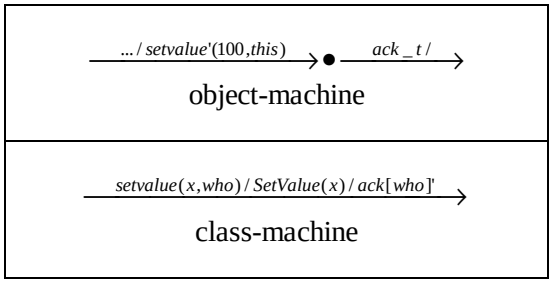

Figure 10. PXM assignment to a static class variable by an object

Only one transition is shown in the *class*-machine. This can only fire only if a signal is received on the channel *setvalue*. We know that a signal is available, so let's follow the action once it is received. The object-machine exported the values *100* and *this*, or in other words, "*t*", its own identity. So the class-machine binds these values to *x* and *who*, respectively. It then performs the computation *SetValue(x)*, and as *x* is bound to the value 100, this results in the static variable being set to the required value. Now the class-machine generates a signal, and we've used special notation here to indicate that the name of the export channel was supplied as data at the beginning of the transition. Thus *ack[who]* is interpreted as meaning *ack_t*, because *who* has the value "*t*", and an acknowledgement is sent to *t* as intended.

3.2 Assignments

All objects can be modelled as processes – this is a standard feature of the CCS approach. For example, a simple integer variable might be modelled as the process

$$Int = \sum_n Int_n \quad \text{where} \quad Int_n = \overline{n}\,.Int_n + \sum_m m.Int_m \tag{12}$$

This says that a general integer can behave in any of several ways, depending on its current value. If the value is *n*, and the value is queried, the variable simply exports the required information. If an assignment is made it, so that the value is now *m*, the behaviour changes to Int_m accordingly.

3.3 General class and object systems

Fig. 11 shows a typical class definition in Java. The class *BoundedIntegerStack* re-uses code by inheriting from *IntegerArray*, and has a *pop()* method which throws an exception if the stack is already empty. We show how to model three basic features from this example:

1. construction of a new *BoundedIntegerStack*
2. inheritance from *IntegerArray*
3. the behaviour of *pop()*, including the exceptional behaviour

3.3.1 Constructors

Constructors play an anomalous role as methods, because when they are called no object exists to which they can be attached. Since we model methods as transition labels we need to identify the machine model to which the transition belongs. The obvious solution is to attach the constructor not to any particular object machine, but to the associated class machine. Accordingly we regard all constructors as belonging to their class, and never to an object. In this respect they are treated in much the same way as static variables. A class machine rarely contains more than a single state, since it is not normal for the gross behaviour of a class to change throughout the lifetime of an application, and this is shown in Fig. 12 which shows a generic Class with constructor transition. A message is received over the *construct* channel including details of the *caller* (the name of the channel over which the result should be returned), and any parameters, *params*, required for construction. The class-machine then invokes a transition relation whose effect is to evaluate *ref = new Class(params)*, where *ref* is a locally held temporary reference to the newly constructed return value. Finally, the newly instantiated *ref* is returned over the channel *caller*'.

3.3.2 Inheritance

Inheritance presents unique problems for OO modelling languages, not least because of the ability of programmers to override methods in derived classes, and the need to allow dynamic binding [40]. In *CCS-XM* we overcome this problem by insisting that all levels of definition are explicitly represented whenever a new object is constructed. Fig. 13 shows a *BoundedIntegerStack* object. Each method available to the object is associated with a channel, this being the channel over which the "invoke this method" message is received. Notice that the constructor is not represented, as it belongs not to the object but to its class.

When a new *BoundedIntegerStack* is constructed, we insist that a new *IntegerArray* object is constructed at the same time, and that this is then bound to the *BoundedIntegerStack* by hiding the two behind a restriction boundary (Fig. 13). Each time a method is accessed at the *BoundedIntegerStack* level the stack communicates with its hidden partner to retrieve the required result. The internal channels are restricted and cannot be accessed by external objects. However, if we access the stack *as-if* it were an array we do so via the standard method-channels for the array itself. These are still accessible, as they are not included in the restriction set.

```
class PopException extends Exception { ... }

class IntegerArray {
  IntegerArray (int capacity) { ... }
  int size() { ... }
  int elementAt(int loc) { ... }
  int removeElementAt(int loc) { ... }
  ...
}

class BoundedIntegerStack extends IntegerArray
{
  public BoundedIntegerStack(int maxsize) {
    super(maxsize); ...
  }
  public int pop() {
    if (size() > 0) {
      int loc = size()-1;
      int element = elementAt(loc);
      removeElementAt(loc);
    }
    else throw new PopException();
  }
  ...
```

Figure 11. An example of a class defined by inheritance and with exceptions

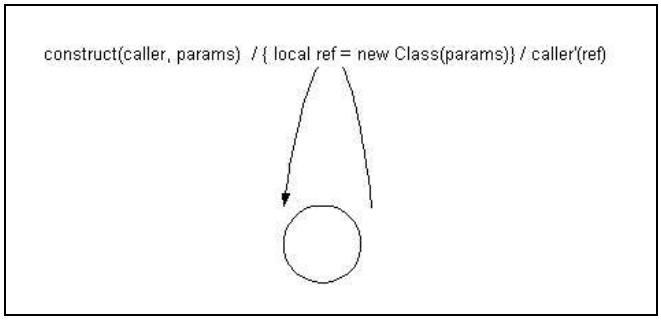

Figure 12. A generic class-machine, showing the constructor transition.

3.3.3 Method

Each method is modelled by the same translation mechanism. If a method has the signature

ReturnType methodname (*params*) (13)

we represent it via a transition in each associated object-machine with 3-label

$\langle methodname(caller, params) \mid MethodName(ref,params) \mid caller'(ref) \rangle$ (14)

That is, the method is invoked by sending a message over the channel of the same name. This message contains the two standard information chunks: the name of the channel over which the result is to be returned and the parameters to be supplied to the method. The transition function *MethodName(ref,params)* is defined as an operator on *X* which sets the local *ref* variable to *this.methodname(params)*, and the result is returned over the channel *caller*.

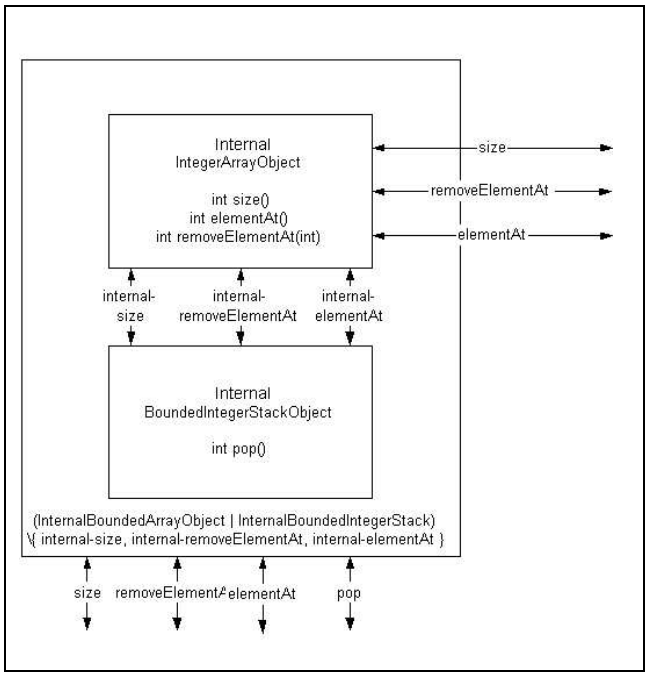

Figure 13. Inheritance in CCS-XM. When a new object is constructed, all ancestral versions of itself are represented internally as objects to which the current version has access over private channels. Casting the object to an ancestral type is equivalent to invoking methods over sub-object channels.

These transitions are inserted wherever a successful call to the method could be made – we assume that control states have been chosen as memory or protocol states, so that successful and unsuccessful transitions go to different terminal states. For example, *pop()* succeeds if and only if the stack is non-empty, so we expect the control space to partition the range of internal states accordingly into *EMPTY* and non-*EMPTY* states. Consequently the call to *throw new PopException()* can be modelled directly as the variant of the *pop* transition that leaves the offending state *EMPTY*. Given this proviso, the object-machine for a bounded integer stack contains states and transitions as shown in Fig. 14.

4. Conclusions and Further Research

We have illustrated how a combination of *CCS* and *X*-machines can be used to generate a natural representation of object-oriented systems. The functional nature of *X*-machine semantics exactly captures the work that goes on within methods, while the communication and concurrency characteristics of *CCS* allow us to model invocation of methods and constructors, as well as basic features like inheritance and polymorphism (the *size()* method of *BoundedIntegerStack* can be invoked as an *IntegerArray* method by invoking the method along a non-private channel). Because we can model OO systems using *X*-machines, and can apply top-down refinement to 3-actions, the results of SXM refinement theory can be expected to apply, whence the language should support proofs of guaranteed testability. Moreover, because each *CCS-XM* model is inherently a *CCS* model, we can examine such features as deadlock-freeness using publicly available tools like the *Concurrency Workbench*.

A great deal still needs to be done. We have touched on the semantics of *CCS-XM* by showing how certain factorizations can take place. But we do not know in detail which important algebraic properties, if any, can be proven, nor do we know how to weaken our basic DFT constraints (which amount to little more than saying “concurrency shouldn’t complicate matters”) to enable useful results to be proven. Moreover, it may yet prove advantageous to move to some later *CCS*-like formalism, most notably the π-calculus, as much of the semantics clearly relies on the ability to pass channel names as parameters.

Nonetheless, we recommend the amalgamation of *X*-machine theory and process algebra as a valuable tool in the development of fully expressive specification languages for OO systems, for which extensive theoretical background material is already available.

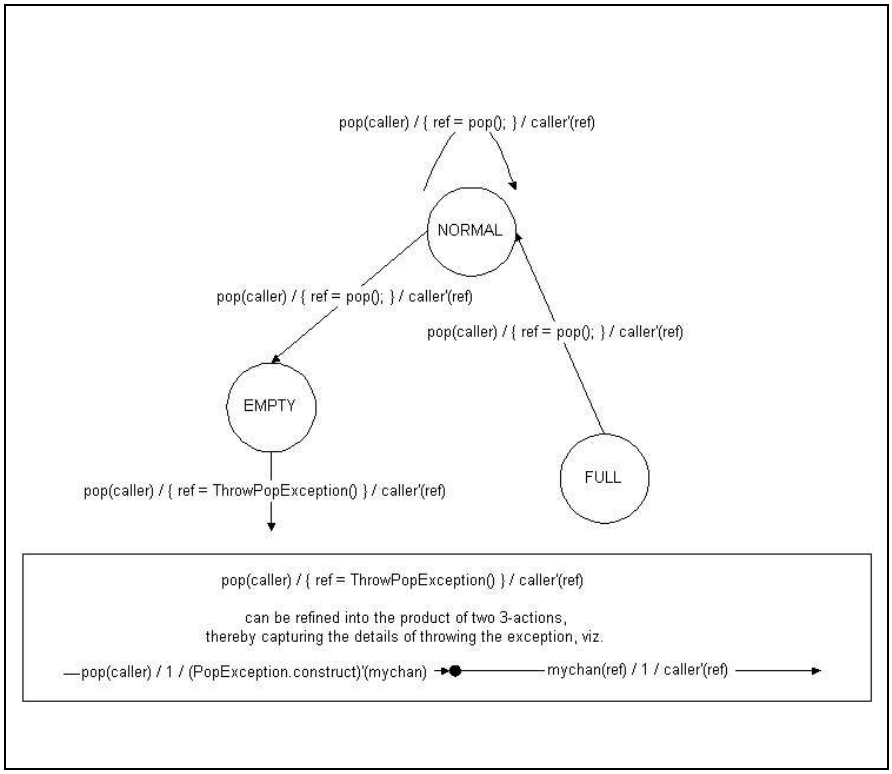

Figure 14. In this example, calling *pop()* in *EMPTY* raises an exception. This can be modelled as a single 3-action, or refined into a pair of 3-actions, depending on the level of detail required.

Acknowledgements

This research is supported by the UK Engineering and Physical Sciences Research Council, grant number GR/M56777 (MOTIVE: Method for Object Testing, Integration and Verification).

References

[1] Eilenberg, S.: *Automata, Languages and Machines, vol. A*. Academic Press, 1974.

[2] Holcombe, M.: "X-machines as a basis for dynamic system specification." *Softw. Eng. J.*, **3**(2), 69-76 (1988).

[3] Stannett, M.: "X-Machines and the Halting Problem: Building a super-Turing Machine." *Formal Aspects of Computing* **2**, 331-41 (1990).

[4] Laycock, G.: "The Theory and Practice of Specification Based Software Testing." PhD Thesis, Dept of Computer Science, Sheffield University, 1993.

[5] Ipate, F.: "Theory of X-machines with Applications in Specification and Testing." PhD Thesis, Dept of Computer Science, Sheffield University, 1995.

[6] Balanescu, T., Cowling, T., Georgescu, H., Gheorghe, M., Holcombe M. and Vertan, C.: "Communicating stream X-machines are no more than X-machines." *J. Universal Comp. Sci.*, **5**(9), 494-507 (1999).

[7] Ipate, F. and Holcombe, M.: "An integration testing method that is proven to find all faults." *Intern. J. Computer Math*, **68**, 159-178 (1997).

[8] Holcombe, M. and Ipate, F.: *Correct Systems: Building a Business Process Solution*. Springer-Verlag, 1998.

[9] Hierons, R.M. and Harman, M. "Testing conformance to a quasi-nondeterministic stream X-machine." *Formal Aspects of Computing*, **12**, 423-442 (2000).

[10] Kehris, E., Eleftherakis, G. and Kefalas, P.: "Using X-Machines to Model and Test Discrete Event Simulation Programs." In *Systems and Control: Theory and Applications*, N. Mastorakis (ed), World Scientific and Engineering Society Press, 163-168, July 2000.

[11] Hierons, R.M. and Harman, M. "Testing conformance of a deterministic implementation against a non-deterministic stream X-machine." Preprint, Dept. of Computer Science, Brunel University, 30 October, 2001.

[12] Milner, R.: *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[13] Milner, R.: *Communication and Concurrency*. Prentice-Hall International, 1989.

[14] Holcombe, M., Stannett, M. and Rathore, S.: "Very High Level functional testing of VLSI – preliminary results." In *Fourth Technical Workshop on New Directions for IC Testing*, D.M. Miller (ed), Vancouver University, 1989.

[15] Stannett M.: "An abstract approach to VLSI test." *Digest 1991/102 Design for Testability* **10**/1-3, IEE Electronics Division, 16 May 1991.

[16] Laycock, G.: "Introduction to X-machines." *Tech. Report* CS-93-13, Dept of Computer Science, Sheffield University, United Kingdom.

[17] Stannett, M.: "An Introduction to post-Newtonian and non-Turing Computation." *Tech. Report* CS-91-02, Dept. of Computer Science, Sheffield University (reprint available online at http://noisefactory.co.uk/download/) 1991.

[18] Laycock, G. and Stannett, M.: "X-machine Workshop '92." *Tech. Report* CS-92-08, Dept of Computer Science, Sheffield University, 1992.

[19] Stannett, M.: "Computation over Arbitrary Models of Time." *Tech. Report* CS-01-08, Dept. of Computer Science, Sheffield University, 2001.

[20] Harel, D.: "Statecharts: a visual formalism for complex systems." *Sci. Comp. Prog.*, **8**, 231-274 (1987).

[21] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R. Shtull-Trauring, A. and Trakhtenbrot, M.: "STATEMATE: A working environment for the development of complex reactive systems." *IEEE Trans. Softw. Eng.*, **16**(4) , 403-414 (1990).

[22] Harel, D. and Naamad, A.: "The STATEMATE semantics of statecharts." *ACM Trans. Softw. Eng. Method.*, **5**(4), 293-333 (1996).

[23] Object Management Group: "Part 9: Statechart Diagrams." In *UML 1.3 Reference Manual, Section 3: Notation Guide*, pp. 3.131-3.150, 1999.

[24] Bogdanov, K., Holcombe, M. and Singh, H.: "Automated test set generation for statecharts." In *Applied Formal Methods – FM-Trends 98*, D. Hutter, W, Stephan, P. Traverso and M. Ullmann (eds.), *Lecture Notes in Computer Science*, vol. 1641, pp. 107-121, Springer-Verlag, 1999.

[25] Bogdanov, K. and Holcombe, M.: "Statechart testing method for aircraft control systems." *Softw. Test. Verif. Reliab.*, **11**, 39-54 (2001).

[26] Fairtlough M., Holcombe M., Ipate F., Jordan C., Laycock G. and Duan, Z. "Using an X-machine to model a Video Cassette Recorder." *Current issues in Electronic Modelling*, **3**, 141-161 (1995).

[27] Bogdanov, K., Fairtlough M., Holcombe M., Ipate, F. and Jordan, C.: "X-machine specification and refinement of digital devices." Preprint, Dept. of Computer Science, Sheffield University (available online at http://www.dcs.shef.ac.uk/~kirill/master7.ps) 1997.

[28] Ipate, F. and Holcombe, M.: "Specification and Testing using Generalised Machines: a Presentation and a Case Study." *Softw. Test. Verif. Reliab.*, **8**, 61-81 (1998).

[29] Vanak, S.K.: "Complete Functional Testing of Hardware Designs." *Preliminary Report*, Dept of Computer Science, Sheffield University, 7 June 2001.

[30] Ipate, F. and Holcombe, M.: "A method for refining and testing generalised machine specifications." *Intern. J. Computer Math*. **69**, 197-219 (1998).

[31] Chow, T.: "Testing software design modelled by finite state machines." *IEEE Trans. Softw. Eng.*, **SE-4**(3), 178-187 (1978).

[32] Eichelberger, E. and Williams, T.: "A Logic Design Structure for LSI Testing." *14th Design Automation Conference*, pp. 462-468 (June 1977).

[33] Aguado, J., Balanescu, T., Cowling, T., Gheorghe, M. and Ipate, F.: "P-systems with replicated rewriting and stream X-machines." *Workshop on Membrane Computing*, Curtea de Arges, August, 2001 (available online at http://www.dcs.shef.ac.uk/~joaquin) 2001.

[34] Ipate, F. and Holcombe, M.: "Generating test sets from non-deterministic stream X-machines." *Formal Aspects of Computing*, **12**, 443-458 (2000).

[35] Guelfi, N., Biberstein, O., Buchs, D., Gaudel, M-C., Canver, E., von Henke, F. and Schwier, D.: "Comparison of Object-Oriented Formal Methods." DeVa TR No. 27, Swiss Federal Institute of Technology, Software Engineering Laboratory, CH-1015 Lausanne, Switzerland (available online at http://www.newcastle.research.ec.org/deva/trs/index.html) 1977.

[36] Péraire, C., Barbey, S. and Buchs, D.: "Test Selection for Object-Oriented Software Based on Formal Specifications." DeVa TR No. 50, Swiss Federal Institute of Technology, Software Engineering Laboratory, CH-1015 Lausanne, Switzerland (available online at http://www.newcastle.research.ec.org/deva/trs/index.html) 1997.

[37] Walker, D.J.: "Automated analysis of mutual exclusion algorithms using CCS." *Formal Aspects of Computing*, **1**(3), 273-292 (1989).

[38] Stirling, C.: "Modal and Temporal Logics for Processes." *Tech. Report* ECS-LFCS-92-221, Dept. of Computer Science, Edinburgh University, 1992.

[39] Kefalas, P., Eleftherakis G. and Kehris, E.: "Communicating X-Machines: A practical approach for modular specification of large systems." *Tech Report* CS-09/00, Dept of Computer Science, CITY Liberal Studies, Thessaloniki, Greece, 2000.

[40] McGregor, J.D.: "Constructing functional test cases using incrementally-derived state machines." *Proc. 11th Int. Conf. Testing Computer Software*, Washington: USPDI, 1994.