Contracts in CML

Jim Woodcock[1], Ana Cavalcanti[1], John Fitzgerald[2],
Simon Foster[1], and Peter Gorm Larsen[3]

[1] University of York
[2] Newcastle University
[3] Aarhus University

Abstract. We describe the COMPASS Modelling Language (CML), which is used to model large-scale Systems of Systems and the contracts that bind them together. The language can be used to document the interfaces to constituent systems using formal, precise, and verifiable specifications including preconditions, postconditions, and invariants. The semantics of CML directly supports the use of these contracts for all language constructs, including the use of communication channels, parallel processes, and processes that run forever. Every process construct in CML has an associated contract, allowing clients and suppliers to check that the implementations of constituent systems conform to their interface specifications.

1 Introduction

The COMPASS Modelling Language (CML) is a formal language for describing "Systems of Systems" [29] (SoS). An SoS is a collaboration of smaller independent systems to achieve a goal that cannot be readily achieved by any of these constituents. Typical SoSs include traffic management systems, emergency response systems, and home automation systems, all of which include constituent systems over which there is no overall control. To achieve synergy, contracts must be negotiated to define the behaviour of the SoS and impose constraints on the constituents, which are otherwise free to behave independently. If such an SoS is to be dependable, it is necessary to ensure that all the constituents are capable of fulfilling their guarantees. We give a mathematically rigorous account of contracts and system models, so that we can verify their conformance.

The design-by-contract paradigm was originally by Meyer [33], and it needs adaptation to express SoS contracts. For SoSs, constituent system designers need to define formal, precise, and verifiable interface specifications for constituents with preconditions, postconditions, and invariants. The postcondition answers the question, "What can the client expect?" The precondition answers the question, "What can the supplier assume?" The invariant answers the question, "What is persistent?" These specifications can then act as contracts which inform the conditions and obligations imposed on a given constituent system.

Parnas calls for formal methods to be "really rethought" [38]; an example he gives is in rethinking the role of termination. Normally, we require programs to

T. Margaria and B. Steffen (Eds.): ISoLA 2014, Part II, LNCS 8803, pp. 54–73, 2014.

terminate to be considered (totally) correct; an extension of this is partial correctness, where it is only required that, if the program terminates, then the answer is correct; but many programs are designed to run indefinitely—specifically reactive systems and particularly SoS. Parnas draws the conclusion that the assertional technique is currently inadequate and that we need to find a good way to represent normal nontermination in correctness arguments.

CML extends the notion of a contract to language constructs not often dealt with in the design-by-contract paradigm, including the use of communication channels, parallel processes, and processes that run forever (addressing Parnas's concerns directly). In fact, every process construct in CML has an associated contract, and this allows clients and suppliers to check that dynamic behaviour conforms to interface specifications.

In the COMPASS project, CML is central to our approach to SoS engineering [17]. We base the approach on a combination of the Systems Modelling Language (SysML) with CML. The former brings facilities for describing system architecture and functionality in a largely graphical, multi-view modelling environment where contracts between constituent systems can be specified at a high-level of abstraction [5]. CML provides a formal basis for analysing SoS models based on SysML abstractions, and adds the rich description of data, functionality, and communication. The loose coupling between SysML and CML has enabled the use of well-established tools to construct CML models; these are translated to a CML version that can be subjected to static and dynamic analysis using the Symphony platform and its plug-ins. The viability of the approach has been demonstrated in diverse ways. For example, in COMPASS, SysML and CML, and tools supporting them, have been successfully deployed together to tackle complex problems in, applications such as home automation [4]. Patterns and profiles have been defined to aid SoS modelling for specific problems such as fault modelling [1].

Our contribution in this paper is to formalise an approach to modelling of contracts in CML, drawing on Meyer's approach to design by contract. In Section 2, we give an overview of CML. In Section 3, we describe the method used to define CML's semantics, Hoare and He's Unifying Theories of Programming, a framework for formalisation of heterogeneous semantics which has a mechanised foundation in Isabelle [18]. In Section 4, we go on to give an overview of CML's semantic domains. In Section 5, we give a series of examples of interface contracts for small fragments of CML processes. In Section 6, we give a complete example of a system in which contracts are used to specify emergent properties. Finally, in Section 7 we reflect on what has been achieved.

2 The COMPASS Modelling Language

The COMPASS Modelling Language (CML) has been developed for the modelling and analysis of Systems of Systems (SoSs), which typically are large-scale

systems composed of independent constituent systems [46][1]. CML is based on a combination of VDM [28,16] and CSP [24,42], in the spirit of *Circus* [44,35,36].

A CML model consists of a collection of types, functions, channels, and processes. The type system of CML is taken directly from VDM and includes support for numeric types, finite sets, finite maps, and records, all of which can be further constrained through type invariants. Functions are pure mathematical mappings between inputs and outputs, which can be specified explicitly via λ-calculus, or implicitly using pre and postconditions. In general we adopt the syntax of VDM-SL [14] for the functional and imperative parts of the language, whilst using *Circus* style syntax for the concurrent and reactive parts.

Channels and processes are used to model SoSs, their constituent systems, and the components of these systems. A channel is a medium though which processes can communicate with each other, optionally carrying data of a certain type. Each process encapsulates a number of local state variables, VDM style operations acting on the state (explicit or implicitly specified), and actions which specify the reactive behaviour. Actions are defined using CSP style process constructs, which enable a process to interact with its environment via synchronous communications. The main action constructs of the basic CML language with state, concurrency, and timing are described in Table 1. In addition to the standard CSP operators, such as prefix `a?v -> P` and external choice `P [] Q`, CML includes support for modelling real-time behaviour such as timeout `P [(n)> Q` which will behave like `Q` after `n` time units if `P` does not first interrupt.

Table 1. The core of the CML language

deadlock	`Stop`	*termination*	`Skip`
divergence	`Chaos`	*assignment*	`(v := e)`
specification statement	`[frame w pre p post q]`		
simple prefix	`a -> Skip`	*prefixed action*	`a -> P`
guarded action	`[g] & P`	*sequential composition*	`P ; Q`
internal choice	`P |~| Q`	*external choice*	`P [] Q`
parallel composition	`P [|cs|] Q`	*interleaving*	`P ||| Q`
abstraction	`P \\ A`	*recursion*	`mu X @ P(X)`
wait	`Wait(n)`	*timeout*	`P [(n)> Q`
untimed timeout	`P [> Q`	*interrupt*	`P /\ Q`
timed interrupt	`P /(n)\ Q`	*starts by*	`P startsby(n)`
ends by	`P endsby(n)`	*while*	`while b do P`

An example CML process is shown in Figure 1, which models a stack. We first define the `Element` type for stacks, which in this case are integers constrained to values between 0 and 255 (i.e. bytes) through a type invariant. Next we define five channels with which to communicate with the stack, including an initialisation channel (`init`), channels which indicate whether the stack is empty or not (`empty`, `nonempty`), and finally channels to push and pop, which carry elements.

[1] The COMPASS project is described in detail at `http://www.compass-research.eu`.

The actual `Stack` process consists of a single state variable `stack` storing a sequence of elements. We define two operations, `Push` and `Pop`, contractually in terms of their pre and postconditions, which respectively add and remove an element from the stack. `Push` has no precondition, and has the postcondition that the new stack must have the pushed value at its head, and the previous value of the stack (denoted by a tilde) at its tail. `Pop` requires that that the stack be nonempty, returns the head of the stack, and suitably updates the state. We also define a simple function to check for emptiness of the stack.

```
types
  Element = int inv x == x >= 0 and x <= 255

channels
  init, empty, nonempty
  push, pop : Element

process Stack =
begin
  state stack : seq of Element

  operations
    Push(e : Element)
      post hd(stack) = e and tl(stack) = stack~

    Pop() e : Element
      pre stack <> []
      post stack = tl(stack~) and e = hd(stack)

  functions
    isEmpty : seq of Element -> bool
    isEmpty(s) == s = []

  actions
    Cycle =
      (  push?e -> Push(e)
      [] [not isEmpty(stack)] &
           (dcl v : Element @ v := Pop() ; pop!v -> Skip)
      [] [isEmpty(stack)] & empty -> Skip
      [] [not isEmpty(stack)] & nonempty -> Skip ) ; Cycle

  @ init -> stack := []; Cycle
end
```

Fig. 1. CML model of a stack

The main reactive cycle of the process is defined through the action `Cycle`, that consists of an external choice between four options. The options describe the following behaviour, respectively:

- wait for input over the `push` channel, and then call the `Push` operation;
- if the stack is not empty, create a local variable `v`, pop the top of the stack and assign it to `v`, and then offer this value over the `pop` channel;
- if the stack is empty, offer communication on the `empty` channel;
- if the stack is not empty, offer communication on the `nonempty` channel.

After each of these behaviours, `Cycle` recurses. The top-level behaviour of the process is then given by the main action, defined after the @ symbol. It waits for an input on `init`, empties the stack, and then enters `Cycle`.

Development of CML models is facilitated through *Symphony*[2], an Eclipse-based integrated development environment. Symphony provides a parser, type checker, simulator, model checker, and theorem prover for CML, all of which have been implemented based on a common semantic basis in the UTP.

The semantics of CML is specification oriented: there is a natural notion of contract for every process language construct and an intuitive refinement ordering. We next describe both notions and give examples of their use.

3 Unifying Theories of Programming

The semantics of CML is defined in Hoare & He's Unifying Theories of Programming (UTP), which is a long-term research agenda for computer science and software engineering [25,11,45]. It can be described as follows: researchers propose programming theories and practitioners use pragmatic programming paradigms; what is the relationship between them? UTP, based on predicative programming [23], gives three principal ways to study such relationships: (i) by computational paradigm, identifying common concepts; (ii) by level of abstraction, from requirements, through architectures and components, to platform-specific implementation technologies; and (iii) by method of presentation—namely, denotational, algebraic, and operational semantics—and their mutual embeddings.

UTP presents a theoretical foundation for understanding software and systems engineering. In its original presentation, it describes nondeterministic sequential programming, the refinement calculus, the algebra of programming, compilation of high-level languages, concurrency, communication, reactive processes, and higher-order programming [25]. Subsequently, it has been exploited in a diversity of areas such as component-based systems [49], hardware verification [40,39], and hardware/software co-design [2].

UTP can also be used in a more active way for constructing domain-specific languages, especially ones with heterogeneous semantics. Examples include the semantics for *Circus* [35,36,44] and Safety-Critical Java (SCJ) [8,10,9], both of which have been composed from individual, reusable theories of sequential and concurrent programming. SCJ additionally has real-time tasking and a sophisticated model of memory usage. The analogy for these kinds of compositional semantics is of a theory supermarket, where you shop for exactly those features you need, whilst being confident that the theories plug-and-play nicely together.

[2] Symphony can be obtained from `http://symphonytool.org/`

The semantic metalanguage for UTP is an alphabetised version of Tarski's relational calculus, presented in a pointwise predicative style that is reminiscent of the schema calculus in the Z notation [47]. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of a paradigm that contains three essential parts: an alphabet, a signature, and some healthiness conditions.

The alphabet is a set of variable names that gives the vocabulary for the theory being studied: names are chosen for any relevant external observations of behaviour. For instance, programming variables x, y, and z are normally part of the alphabet. Theories for particular programming paradigms require the observation of extra information. Some examples from existing theories are: a flag that says whether the program has started (*ok*); the current time (*clock*); the number of available resources (*res*); a trace of the events in the life of the program (*tr*); a set of refused events (*ref*); or a flag that says whether the program is waiting for interaction with its environment (*wait*).

The signature gives syntactic rules for denoting objects of the theory. Finally, healthiness conditions identify properties that characterise the predicates of the theory. They are often expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier, so ϕ must be idempotent: $P = \phi(\phi(P))$. The fixed-points of this equation are the healthy predicates of the theory.

An alphabetised predicate (P, Q, ..., ***true***) is an alphabet-predicate pair, such that the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet comprises plain variables (x, y, z, ...) and dashed variables (x', a', ...); the former represent initial observations, and the latter, intermediate or final observations. The alphabet of P is denoted αP. A *condition* (b, c, d, ..., *true*) has no dashed variables. Predicate calculus operators combine predicates in the obvious way, but with specified restrictions on the alphabets of the operands and specified resulting alphabet. For example, the alphabet of a conjunction is the union of the alphabets of its components, and disjunction is defined only for predicates with the same alphabet.

A distinguishing feature of UTP is the central role played by program correctness, which is defined in the same way in every programming paradigm in [25]. Informally, it is required that, in every state, the behaviour of an implementation implies its specification. If we suppose that for a predicate P, $\alpha P = \{a, b, a', b'\}$, then the universal closure of P is $\forall\, a, b, a', b' \bullet P$, denoted $[\,P\,]$. P is correct with respect to a specification S providing every observation of P is also an observation of S: $[\,P \Rightarrow S\,]$; this is described by $S \sqsubseteq P$ (read S is refined by P).

UTP has an infix syntax for the conditional, $P \lhd b \rhd Q$, and it is defined as $(b \wedge P) \vee (\neg\ b \wedge Q)$, if $\alpha b \subseteq \alpha P = \alpha Q$. Sequence is modelled as relational composition: two relations may be composed, providing that their alphabets match: $P(v')\ ;\ Q(v) \ \widehat{=}\ \exists\, v_0 \bullet P(v_0) \wedge Q(v_0)$, if $out\alpha P = in\alpha Q' = \{v'\}$. If $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, then the assignment $x :=_A e$ defined below, with expression vector e and variable vector x, changes only x's value.

$$x :=_A e \ \widehat{=}\ (x' = e \wedge y' = y \wedge \cdots \wedge z' = z)$$

There is a degenerate form of assignment that changes no variable; it is called "skip" defined as $\Pi_A \mathrel{\widehat{=}} (v' = v)$, if $A = \{v\}$. Nondeterminism can arise in one of two ways: either as the result of runtime factors, such as distributed processing or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is disjunction: $P \sqcap Q \mathrel{\widehat{=}} P \vee Q$.

The set of alphabetised predicates with a particular alphabet A forms a complete lattice under the refinement ordering (which is a partial order). The bottom element is denoted $\bot_A$, and is the weakest predicate ***true***; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted $\top^A$, and is the strongest predicate ***false***; this is the program that performs miracles and implements every specification. Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a complete lattice of fixed points. The weakest fixed-point of the function F is denoted by μF, and is simply the greatest lower bound (the *weakest*) of all the fixed-points of F. This is defined: $\mu F \mathrel{\widehat{=}} \bigsqcap\{ X \mid F(X) \sqsubseteq X \}$. The strongest fixed-point νF is the dual of the weakest fixed-point.

4 CML Semantics

Currently, CML contains several language paradigms. These are all presented through formalisation as UTP theories.

State-based description. The theory of designs provides a nondeterministic programming language with precondition and postcondition specifications as contracts, written $P \vdash Q$, for precondition P and postcondition Q. The concrete realisation is VDM.

Concurrency and communication. The theory of reactive processes provides process networks communicating by message passing. The concrete realisation is CSP_M with its rich collection of process combinators.

Object orientation. This theory is built on designs with state-based descriptions structured by sub-typing, inheritance, and dynamic binding, with object creation, type testing and casting, and state-component access [7].

References. The theory of heap storage and its manipulations supports a reference semantics based on separation logic.

Time. The theory of timed traces in UTP supports the observation of events in discrete time. It is used in a theory of Timed CSP.

As explained in the previous section, the semantic domains are each formalised as lattices of relations ordered by refinement. Mappings exist between the different semantic domains that can be used to translate a model from one lattice into a corresponding model in another lattice. For example, the lattice of designs is completely disjoint from the lattice of reactive processes, but a mapping ***R*** maps every design into a corresponding reactive process. Intuitively, the mapping equips the design with the crucial properties of a reactive process: that it has a trace variable (tr) that records the history of interactions with its environment and that it can wait for such interactions. A vital healthiness condition is that

this trace increases monotonically: this ensures that once an event has taken place it cannot be retracted—even when the process aborts.

Another mapping counteracts $\boldsymbol{R}$: it is called $\boldsymbol{H}$, and it is the function that characterises what it is to be a design. $\boldsymbol{H}$ puts requirements on the use of the observations ok and ok', and it is the former that concerns us here. It states that, until the operation has started properly (ok is true), no observation can be made of the operation's behaviour. So, if the operation's predecessor has aborted, nothing can be said about any of the operation's variables, not even the trace observation variable. This destroys the requirement of $\boldsymbol{R}$ that restricts the trace so that it only ever increases monotonically, even when ok is false.

This pair of mappings forms a Galois connection [13], and such pairs exist between all of CML's semantic domains. One purpose of a Galois connection is to embed one theory within another, and this is what gives the compositional flavour of UTP and CML, since Galois connections compose to form other Galois connections. For example, if we establish a Galois connection between reactive processes and timed reactive processes, then we can compose the connection between designs and reactive processes with this new Galois connection to form a connection between designs and timed reactive processes.

The possibly obscure mathematical fact that there is a Galois connection between designs and reactive processes is of great practical value. One of the most important features of designs is assertional reasoning based on preconditions and postconditions, including the use of Hoare logic and weakest precondition calculus. Assertional reasoning, as defined in the theory of designs, can be incorporated into the theory of reactive processes by means of the mapping $\boldsymbol{R}$.

In the theory of designs, a Hoare triple $\{p\}\, Q\, \{r\}$, where p is a precondition, r is a postcondition, and Q is a reactive process, is given the meaning $(\boldsymbol{R}(p \vdash r') \sqsubseteq Q)$, which is a refinement assertion. In the specification $\boldsymbol{R}(p \vdash r')$ the precondition p and the postcondition r are assembled into a design, with r as a condition on the after-state; this design is then translated into a reactive process using $\boldsymbol{R}$. The semantics of the Hoare triple requires that this reactive specification is implemented correctly by the reactive process Q. Thus, reasoning with preconditions and postconditions is, in this way, extended from the state-based operations of the theory of designs to cover all operators of the reactive language, including non-terminating processes, concurrency, and communication.

This is the foundation of the contractual approach used in COMPASS: preconditions and postconditions (designs) are embedded in each of the semantic domains and this brings uniformity through a familiar reasoning technique.

5 Contracts in CML

In this section, we give a series of examples of the use of contracts in CML.

Example 1 (Single shot). The CML action $a \rightarrow \mathit{Skip}$ will perform one a event and then terminate. It never diverges, so it has precondition $\boldsymbol{true}$. Its postcondition depends on whether or not it is waiting, indicated by the observational variable $wait'$ being true. If it is waiting, then it has not performed any events

and the trace is unchanged ($tr' = tr$), but it is also not refusing to perform the a event: $a \notin ref'$, where ref' is the refusal set. Otherwise, it has performed exactly one a event ($tr' = tr \frown \langle a \rangle$). This precondition-postcondition pair forms a design that gives the contract for the action; of course, the contract must also insist on $\boldsymbol{R}$-healthiness. In full, the contract is as follows.

$$\boldsymbol{R}(\boldsymbol{true} \vdash (tr' = tr \land a \notin ref') \lhd wait' \rhd tr' = tr \frown \langle a \rangle)$$

We notice the use of observational variables: ok, ok', $wait$, $wait'$, tr, tr', ref, and ref'. These are "ghost variables", not code; that is, they are part of the underlying semantic model and cannot be manipulated at run time. Ghost variables provide a convenient way of forming contracts by allowing us explicitly to restrict possible reactive behaviours.

There is a technicality about any assertion involving the ghost variable ref'. If an action may refuse a set s, then it may refuse any subset of s. That is, if an action refuses the set $\{a, b\}$, then it will also refuse the sets $\{a\}$ and $\varnothing$, for example. For this reason, an assertion such as $a \in ref'$ is unsatisfiable. So if we really did want to assert that a is always refused, then we would instead say that it never occurs: it never appears in the trace, rather than restrict refusals.

The precise form of a CML contract is derived from the fact that every CML action can be expressed in the form $\boldsymbol{R}(P \vdash Q)$. We saw the syntactic form of a design above; its semantics depends on the two observations mentioned in Section 4, ok and ok': if the design is started (ok is true) in a state in which the precondition holds (P is true), then it must terminate (ok' will be true) and when it does, the postcondition must hold (Q must be true). This justifies the definition $P \vdash Q \;=\; ok \land P \Rightarrow ok' \land Q$ for designs.

Example 2 (Chocolate vending machine). We consider a grossly simplified model of a vending machine `VM`. A complete transaction with the machine involves inserting a coin and extracting a chocolate; the machine repeatedly engages in such transactions as specified by the action below.

```
VM = coin -> choc -> VM
```

This is a CML action, but what is the contract? We notice that there is no state, so the contract must be entirely in terms of the ghost variables ok, ok', $wait$, $wait'$, tr, tr', ref, and ref'. A reasonable contract for the machine comes in two parts: a requirement on the user and a requirement on the machine itself.

– The machine should not lose money: *every chocolate must be paid for.*

```
NOLOSS = freq(choc,tr'-tr) <= freq(coin,tr'-tr)
```

– The machine should be fair: *the machine should not build up too much credit.*

```
FAIR = freq(coin,tr'-tr) <= freq(choc,tr'-tr) + 1
```

The auxiliary function `freq` gives the frequency of an event in a trace. It is defined below. The specification of the vending machine is given by the conjunction of these requirements. It is defined below.

```
VMSPEC = NOLOSS and FAIR
```

The VDM (and, therefore, CML) definition of `freq` is as follows.

```
freq: Event * (seq of Event) -> nat
freq(e,s) =
  if s = [] then
    0
  else
    if hd(s) = e then
      1 + freq(e,tl(s))
    else
      freq(e,tl(s))
```

How can we check that `VM` satisfies this specification? There are four principal ways, two using theorem proving and two using model checking:

1. prove that $[\, VM \Rightarrow VMSPEC \,]$,
2. use the assertional technique (that is, Hoare logic),
3. use a refinement model checker, or
4. use a temporal logic model checker.

(1), (2), and (3) are essentially the same: they check the refinement relation. We may regard (1) as a full-frontal attack on the problem using a theorem prover. On the other hand, (2) is more subtle, using inference rules to match the structure of the implementation `VM` and check the assertion `VMSPEC`. For (3), the specification must be captured as a finite state CML action, just like the implementation. A model checker (such as [21] or [31]) is then used to check that the observable behaviours of the implementation are all behaviours of the specification.

For (4), the specification must be captured as an expression in temporal logic; LTL is commonly used [3], with its operators such as "eventually" and "henceforth". A model checker, such as PAT [31] or the CML model checker [34], is then used to test whether the system satisfies the temporal logic specification. Roscoe [43] and Lowe [32] have each studied the relationship between refinement-based checking and temporal-logic checking. An account of that in the UTP can justify its use for CML.

To illustrate the use of the refinement model checker, we construct actions to embody the two parts of the specification. First, for `NOLOSS`, we construct an action parametrised by the number of coins and chocolates already dispensed. `NOLOSS` is always willing to accept further coins, since that cannot contribute to a financial loss, but dispenses a chocolate only if there is outstanding credit.

```
NoLossProc =
  coins, chocs: nat @
    coin -> NoLossProc(coins+1,chocs)
    []
    [chocs < coins] & choc -> NoLossProc(coins,chocs+1)
```

The action embodying FAIR complements NOLOSS: it is always willing to dispense chocolates, since that cannot be unfair to a customer, but accepts a coin only if at least as many chocolates have been dispensed as paid for.

```
FairProc = coins, chocs: nat @
  [coins <= chocs] & coin -> NoLossProc(coins+1,chocs)
  []
  choc -> FairProc(coins,chocs+1)
```

A vending machine that does nothing makes no loss and is trivially fair, so NOLOSS and FAIR is not a very good specification; so how do we say something stronger? There is a liveness aspect to fairness: if the customer has paid for a chocolate, then the machine should not refuse to dispense it.

```
FAIR1 = (freq(choc,tr'-tr) < freq(coin,tr'-tr) => choc not in ref')
```

Similarly, there is a liveness aspect to profit making: if every chocolate that has been paid for has been dispensed, then the machine should not refuse a coin.

```
PROFIT1 = (freq(choc,tr'-tr) = freq(coin,tr'-tr) => coin not in ref')
```

The two actions embodying our specification already have these two properties.

Specifications involving ref' can be used to assert deadlock freedom. If the event alphabet for an action is A, then deadlock freedom can be specified as NONSTOP = ref' <> A. This states that the action can never reject, that is, refuse, the entire alphabet of events, and so is never deadlocked.

When checking with a model checker such as FDR[3], it is sufficient to check satisfaction of each part of the specification independently. So to check that the vending machine does not make a loss and that it is fair to its customers, we can use the two separate assertions below.

```
assert NoLossProc [= VM

assert FairProc [= VM
```

Equivalently, we can check the two properties simultaneously. To do this, we need to assemble the NoLossProc and FairProc actions in parallel, synchronising on the choc and coin events as shown below.

```
assert NoLossProc [|{choc, coin}|] FairProc [= VM
```

The process *Chaos* misbehaves badly, like your worst nightmare: it melts down the reactor; it switches off the in-flight computer; it transfers all your funds into my bank account. The next examples illustrate its use.

Example 3 (Deferred chaos). Consider the process:

```
a -> Chaos
```

[3] http://www.fsel.com.

This process is perfectly safe, providing its environment never engages in the event a. So what is the contract? The precondition must record the assumption that the a event never occurs, which it does as a relation: $\neg\,(tr \frown \langle a \rangle \leq tr')$. The precondition is describing a protocol in terms of the trace, a kind of rely-condition in the sense of Jones [27]. Now, if we assume that the precondition holds, then the postcondition is straightforward: the action is forever waiting ($wait'$), and the trace never changes ($tr' = tr$), but a is not refused ($a \notin ref'$).

$$\boldsymbol{R}(\neg\,(tr \frown \langle a \rangle \leq tr') \vdash wait' \wedge tr' = tr \wedge a \notin ref')$$

Example 4 (Badly behaved vending machine). We now consider the following badly-behaved vending machine `VMC`.

```
VMC =
  in2 -> ( large -> VMC
           []
           small -> out1 -> VMC )
  []
  in1 -> ( small -> VMC
           []
           in1 -> ( large -> VMC
                    []
                    in1 -> Chaos ) )
```

Initially, `VMC` is prepared to accept either a £1 coin or a £2 coin. If the £2 coin is inserted, then the customer has a choice between a large and a small chocolate bar. If the small bar is selected, then the machine offers change of £1. Alternatively, if the £1 coin is inserted, then a choice is offered between extracting a small chocolate bar or inserting a further £1 coin. If another £1 coin is inserted, the choice then becomes between extracting a large chocolate bar or inserting yet another £1 coin, whereupon the machine behaves chaotically. The precondition here is, therefore, $\neg\,(tr \frown [in1, in1, in1] \leq tr')$.

6 Mini-Mondex

Mondex[4] is an electronic purse hosted on a smart card and developed about fifteen years ago to the high-assurance standard ITSEC Level E6 [26] by a consortium led by NatWest, a UK high-street bank. Eight years ago, a community effort was launched to mechanically verify the original models of Mondex in a variety of different notations, in order to compare and contrast their effectiveness [48,41,22,6,30,20,19]; the problem has now become a benchmark for formal verification. In this section, we describe a simplified version of the problem: mini-Mondex, where we ignore faulty behaviour and focus on specifying functional requirements.

[4] `http://www.mondex.com`.

Purses interact using a communications device, and strong guarantees are needed that transactions are secure in spite of power failures and mischievous attacks. These guarantees ensure that electronic cash cannot be counterfeited, although transactions are completely distributed. There is no centralised control: all security measures are locally implemented, with no real-time external audit logging or monitoring; key properties *emerge* from local behaviour.

Our model of Mondex has the following constant values: `N`, the number of cards in the system; `V`, the maximum value that may be held by a card; and `M`, the total money supply. These are specified in CML as follows.

```
values
  N: nat = 10
  V: nat = 10
  M: nat = N*V
```

There are two types related to these constants: the `Index` set for cards; and the `Money`. The relationship with `N` and `M` is made explicit through two invariants.

```
types
  Index = nat
  inv i == i in set {1,...,N}

  Money = nat
  inv m == m in set {0,...,M}
```

We also specify three functions which are needed for our contract. `initseq(n)` builds a sequence of numbers from 0 to `n`. `subtseq(xs, i, n)` subtracts `n` from the `i`th item of sequence `xs`. `addseq(xs, i, n)` adds `n` to the `i`th item of `xs`.

```
functions
  initseq: nat -> seq of nat
  initseq(n) == [i | i in set {0,...,n}]

  subtseq: seq of nat * nat * nat -> seq of nat
  subtseq(xs, i, n) == xs ++ {i |-> xs(i) - n}
    pre len(xs) > i and xs(i) >= n

  addseq: seq of nat * nat * nat -> seq of nat
  addseq(xs, i, n) == xs ++ {i |-> xs(i) + n}
    pre len(xs) > i
```

There are a number of channels that connect cards with each other and with the environment. A user can instruct one card to pay another with the event `pay.i.j.n`, which corresponds to instructing card `i` to pay card `j` the sum of `n` money units. The attempted transfer of money is made between cards using the `transfer` channel. The transaction may be `accepted` or `rejected`.

```
channels
  pay, transfer: Index * Index * Money
  accept, reject: Index
```

Each card is modelled by an indexed CML process with its encapsulated state. The `state` of the process consists of a single component `value`, which is a natural number recording the balance in the purse. There are three operations: (i) `Init`, which sets the initial value to `V`; (ii) `Credit`, which increments the `value` by the parameter `n`; (iii) and `Debit`, which decrements the `value` by the parameter `n`. There are also three actions: (i) `Transfer` accepts a `pay` communication and analyses it to see if there are sufficient funds to honour the debit, replying appropriately with an `accept` or `reject` communication; if there are sufficient funds, then a `transfer` message is sent to the receiving card and debits its local state. (ii) `Receive` accepts a `receive` message and credits its local state appropriately. (iii) `Cycle` repeatedly offers the `Transfer` and `Receive` actions. The `@`-symbol marks the main action for the process.

```
process Card = val i: Index @
  begin
    state value: nat
    operations
      Init: () ==> ()
      Init() == value := V

      Credit: nat ==> ()
      Credit(n) == value := value + n

      Debit: nat ==> ()
      Debit(n) == value := value - n
      pre n <= value

    actions
      Transfer =
        pay.i?j?n ->
          ( [n > value] & reject!i -> Skip
            []
            [n <= value] & transfer.i.j!n -> accept!i -> Debit(n) )

      Receive = transfer?j.i?n -> Credit(n)

      Cycle = ( Transfer [] Receive ); Cycle
  @
      Init(); Cycle
  end
```

The network is defined by the parallel composition of all the indexed cards. We need to specify the interface for each card to specify its interaction with the rest of the network. As defined above, `Card(i)` participates in the following events.

- Every event of the form `pay.i.j.n`, for any card `j` and amount `n`.
- Every event of the form `transfer.i.j.n`, for any card `j` and amount `n`. These represent the money leaving the card.

- Every event of the form `transfer.j.i.n`, for any card `j` and amount `n`. These represent the money entering the card.
- The events `accept.i` and `reject.i`.

In the construction of the network, we identify this alphabet of events for each of the `Card(i)` processes, and assemble the `N` cards in parallel.

```
process Cards =
  || i in set {1,...,N} @
    [{| pay.i,transfer.i,accept.i, reject.i|} union
      {| transfer.j.i.n | j in set {1,...,N}, n in set {0,...,M}|}
    ] Card(i)
```

Cards that share the same event in their alphabet need to synchronise on that event. The network, therefore, ensures that transfers between cards `i` and `j` are achieved when both cards cooperate.

Finally, we need to hide the internal channels that connect the cards: these do not form part of the extensional behaviour of the network:

```
process Network = Cards \ {|transfer|}
```

We identify the following properties that are required of mini-Mondex.

No counterfeiting: There must be no increase in the total value in the system.
Fairness: There must be no loss of value.
Usefulness: If we demand a transfer from a card that has the required funds, then the transfer should take place.

We notice that the first two properties above are emergent global properties, but the system has only local behaviour. In what follows, we describe these properties using the CML contract `Spec` below.

This contract models the network as a single process with an Olympian view of the state of all cards; it has only one state component, `valueseq`, which is a sequence of numbers, indexed by card indexes. An invariant requires there to be an element in the sequence for every card. The `valueseq` is initialised to correspond with the initialisation of each card: each containing the value `V`.

```
process Spec =
  begin
    state
      valueseq: seq of nat
    inv
      len(valueseq) = N
    operations
      Init: () ==> ()
      Init() == valueseq := initseq(N)
```

There are two actions. The first, `Pay` is parametrised by source and destination cards and an amount to be paid. Its behaviour starts with the communication

`pay.i.j.n`. Following this, there is an analysis of whether the card paying the amount can afford it. If it cannot, then the transaction is rejected. If it can, then the payer's and payee's values are updated accordingly to reflect the transfer of money, and the transaction is accepted.

The second action is a repetitive cycle; on each step, a nondeterministic choice is made of a payer, a payee, and an amount to be paid. Thus, `Cycle` represents all possible financially correct transactions.

```
  actions
    Pay = i,j: Index, n: Money @
      pay.i.j.n ->
        if n > valueseq(i) then
          reject.i -> Skip
        else
          ( valueseq := subtseq(valueseq,i,n);
            valueseq := addseq(valueseq,j,n);
            accept.i -> Skip )

    Cycle =
      ( |~| i,j: Index, n: Money @ Pay(i,j,n) );
      Cycle
@
  Cycle
end
```

`Spec` gives us an arena in which to specify the correctness of mini-Mondex. The properties identified above can be described as follows.

No increase in value: `sum(valueseq) <= M`. (`sum` returns the sum of the elements in a sequence.)

No loss in value: `sum(valueseq) >= M`.

Usefulness If we demand a transfer, and we have got the funds, then the transfer should take place:

```
  forall i, j: Index; n: Money @
    tr'-tr <> []
    and last(tr'-tr) = transfer.i.j.n
    and n >= valueseq(i)
      =>
        accept.i not in ref'
```

Finally, we need an invariant that relates the stored state, `valueseq`, to the history of transactions. For card `i`, the communications `transfer.i.j` represent outgoing payments; the communications `transfer.j.i` represent incoming payments; and the value `V` represents the initial value in the card.

```
forall i: {1,...,N} @
  valueseq(i) =
```

```
V + transum((tr'-tr) filter { transfer.i.j | j in set {1,...,N} })
  + transum((tr'-tr) filter { transfer.j.i | j in set {1,...,N} })
```

where

```
transum(s) =
  if s = [] then
    0
  else
    amount(hd(s)) + transum(tl(s))

forall i, j: Index; n: Money @ amount(transum.i.j.n) = n
```

7 Conclusions

We have presented a series of examples of the use of contracts in CML. It is based on the semantic embedding of a theory of total correctness based on preconditions and postconditions into the theory that defines the semantic model of CML. With that, we have a characterisation of preconditions and postconditions of reactive constructs, including communication, choice, and parallelism.

De Boer describes the postconditions of nonterminating processes as equivalent to *false* [15], since he considers their states to be unobservable; this is of little use in reasoning about reactive processes that run forever. As we mentioned in Section 1, Parnas calls for the development of assertional techniques to handle the normal nontermination of reactive processes [38]. Our work explicitly considers stable intermediate states (those satisfying the ghost expression $ok' \wedge wait'$), and provides these states with postconditions.

Jones has defined rely and guarantee conditions for assertional reasoning [27] in the presence of concurrency. These conditions are concerned with properties of interleaved atomic steps: guarantee conditions describe postconditions for atomic steps of the process, and rely conditions describe postconditions for atomic steps of the environment. Both rely and guarantee conditions are relations and, therefore, regarded as postconditions. Our postconditions are analogous to guarantee conditions, except that they relate initial states to intermediate states, rather than describing the postcondition of an arbitrary atomic step. Similarly, our preconditions are relational and are analogous to Jones's rely conditions, except that, again, they relate initial and intermediate states, rather than describing the postcondition of an atomic step of the environment.

Future work includes exploring the relationship between our contractual techniques and Jones's atomic-step semantics for rely and guarantee thinking. The development of tools to support assertional reasoning in CML is also essential for its practical relevance and scalability. Symphony does not yet support ghost variables.

The CML semantics has been partially mechanised in Isabelle, through a semantic embedding of UTP called Isabelle/UTP [18]. In particular we have mechanised the theory of designs, the theory of reactive processes, and have preliminary support for a Hoare calculus, which together provide the building blocks for

formal verification of contracts as shown in this paper. We have already used Isabelle/UTP to construct a CML theorem prover [12], and a verification-condition generator is a natural next step in this effort. Moreover we are currently working on a refinement tool for Symphony which provides calculational support for the CML refinement calculus, building on previous work [37]. This will provide tool support for showing conformance between a given CML contract and an underlying SoS, such as the Mondex example.

References

1. Andrews, Z., Fitzgerald, J., Payne, R., Romanovsky, A.: Fault Modelling for Systems of Systems. In: Proceedings of the 11th International Symposium on Autonomous Decentralised Systems (ISADS 2013), pp. 59–66 (March 2013)
2. Beg, A., Butterfield, A.: Linking a state-rich process algebra to a state-free algebra to verify software/hardware implementation. In: FIT 2010, 8th Intl Conf. on Frontiers of Information Technology, Islamabad, p. 47. ACM (2010)
3. Ben-Ari, M., Manna, Z., Pnueli, A.: The temporal logic of branching time. In: White, J., Lipton, R.J., Goldberg, P.C. (eds.) 8th Ann. ACM Symp. on Principles of Programming Languages, Williamsburg, pp. 164–176. ACM Press (1981)
4. Bryans, J., Fitzgerald, J., Payne, R., Kristensen, K.: Maintaining emergence in systems of systems integration: a contractual approach using SysML. In: INCOSE International Symposium (to appear, 2014)
5. Bryans, J., Fitzgerald, J., Payne, R., Miyazawa, A., Kristensen, K.: SysML Contracts for Systems of Systems. In: 9th Intl Conf. on Systems of Systems Engineering (SoSE). IEEE (June 2014)
6. Butler, M., Yadav, D.: An incremental development of the Mondex system in Event-B. Formal Asp. Comput. 20(1), 61–77 (2008)
7. Cavalcanti, A., Sampaio, A., Woodcock, J.: Unifying classes and processes. Software and System Modeling 4(3), 277–296 (2005)
8. Cavalcanti, A., Wellings, A., Woodcock, J.: The Safety-Critical Java memory model: A formal account. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 246–261. Springer, Heidelberg (2011)
9. Cavalcanti, A., Wellings, A.J., Woodcock, J.: The Safety-Critical Java memory model formalised. Formal Asp. Comput. 25(1), 37–57 (2013)
10. Cavalcanti, A., Wellings, A.J., Woodcock, J., Wei, K., Zeyda, F.: Safety-critical Java in Circus. In: Wellings, A.J., Ravn, A.P. (eds.) The 9th Intl Workshop on Java Technologies for Real-time and Embedded Systems, JTRES 2011, York, pp. 20–29. ACM (2011)
11. Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in Unifying Theories of Programming. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 220–268. Springer, Heidelberg (2006)
12. Couto, L., Foster, S., Payne, R.: Towards verification of constituent systems through automated proof. In: Proc. Workshop on Engineering Dependable Systems of Systems (EDSoS). ACM CoRR (2014)
13. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order, 2nd edn. Cambridge University Press (2002)
14. Dawes, J.: The VDM-SL Reference Guide. Pitman (1991) ISBN 0-273-03151-1
15. de Boer, F.S., Hannemann, U., de Roever, W.-P.: Hoare-style compositional proof systems for reactive shared variable concurrency. In: Ramesh, S., Sivakumar, G. (eds.) FST TCS 1997. LNCS, vol. 1346, pp. 267–283. Springer, Heidelberg (1997)

16. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer (2005)
17. Fitzgerald, J., Larsen, P.G., Woodcock, J.: Foundations for Model-based Engineering of Systems of Systems. In: Aiguier, M., et al. (eds.) Complex Systems Design and Management, pp. 1–19. Springer (January 2014)
18. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: 5th International Symposium on Unifying Theories of Programming (to appear, 2014)
19. Freitas, L., Woodcock, J.: Mechanising Mondex with Z/Eves. Formal Asp. Comput. 20(1), 117–139 (2008)
20. George, C., Haxthausen, A.E.: Specification, proof, and model checking of the Mondex electronic purse using RAISE. Formal Asp. Comput. 20(1), 101–116 (2008)
21. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — A modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 187–201. Springer, Heidelberg (2014)
22. Haneberg, D., Schellhorn, G., Grandy, H., Reif, W.: Verification of Mondex electronic purses with KIV: from transactions to a security protocol. Formal Asp. Comput. 20(1), 41–59 (2008)
23. Hehner, E.C.R.: Retrospective and prospective for Unifying Theories of Programming. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 1–17. Springer, Heidelberg (2006)
24. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
25. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall (1998)
26. ITSEC. Information Technology Security Evaluation Criteria (ITSEC): Preliminary harmonised criteria. Technical Report Document COM(90) 314, Version 1.2, Commission of the European Communities (1991)
27. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
28. Jones, C.B.: Systematic Software Development using VDM, 2nd edn. Prentice Hall International (1990)
29. Kopetz, H.: System-of-Systems complexity. In: Proc. 1st Workshop on Advances in Systems of Systems, pp. 35–39 (2013)
30. Kuhlmann, M., Gogolla, M.: Modeling and validating Mondex scenarios described in UML and OCL with USE. Formal Asp. Comput. 20(1), 79–100 (2008)
31. Liu, Y., Sun, J., Dong, J.S.: PAT 3: An extensible architecture for building multi-domain model checkers. In: Dohi, T., Cukic, B. (eds.) IEEE 22nd Intl Symp. on Software Reliability Engineering, ISSRE 2011, Hiroshima, pp. 190–199. IEEE (2011)
32. Lowe, G.: Specification of communicating processes: temporal logic versus refusals-based refinement. Formal Asp. Comput. 20(3), 277–294 (2008)
33. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)
34. Mota, A., Farias, A., Didier, A., Woodcock, J.: Rapid prototyping of a semantically well founded Circus model checker. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 235–249. Springer, Heidelberg (2014)
35. Oliveira, M., Cavalcanti, A., Woodcock, J.: A denotational semantics for Circus. Electr. Notes Theor. Comput. Sci. 187, 107–123 (2007)
36. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Asp. Comput. 21(1-2), 3–32 (2009)
37. Oliveira, M., Gurgel, A.C., Castro, C.G.: CRefine: Support for the Circus refinement calculus. In: 6th Intl. Conf. on Software Engineering and Formal Methods (SEFM 2008), pp. 281–290. IEEE Computer Society (November 2008)

38. Parnas, D.L.: Really rethinking 'formal methods'. IEEE Computer 43(1), 28–34 (2010)
39. Perna, J.I., Woodcock, J.: Mechanised wire-wise verification of Handel-C synthesis. Sci. Comput. Program. 77(4), 424–443 (2012)
40. Perna, J.I., Woodcock, J., Sampaio, A., Iyoda, J.: Correct hardware synthesis—an algebraic approach. Acta Inf. 48(7-8), 363–396 (2011)
41. Ramananandro, T.: Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. Formal Asp. Comput. 20(1), 21–39 (2008)
42. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall International (1997)
43. Roscoe, A.W.: On the expressive power of CSP refinement. Formal Asp. Comput. 17(2), 93–112 (2005)
44. Woodcock, J., Cavalcanti, A.: The semantics of Circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)
45. Woodcock, J., Cavalcanti, A.: A tutorial introduction to designs in Unifying Theories of Programming. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 40–66. Springer, Heidelberg (2004)
46. Woodcock, J., Cavalcanti, A., Fitzgerald, J.S., Larsen, P.G., Miyazawa, A., Perry, S.: Features of CML: A formal modelling language for systems of systems. In: 7th Intl Conf. on Systems of Systems Engineering, SoSE 2012, Genova, pp. 445–450. IEEE (2012)
47. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Inc. (1996)
48. Woodcock, J., Stepney, S., Cooper, D., Clark, J.A., Jacob, J.: The certification of the Mondex electronic purse to ITSEC Level E6. Formal Asp. Comput. 20(1), 5–19 (2008)
49. Zhan, N., Kang, E.Y., Liu, Z.: Component publications and compositions. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 238–257. Springer, Heidelberg (2010)