Modelling Evolvable Component Systems

Part I: A logical framework

Howard Barringer* Dov Gabbay† David Rydeheard‡

Abstract

We develop a logical modelling approach to describe evolvable computational systems. In this account, evolvable systems are built hierarchically from components where each component may have an associated supervisory process. The supervisor's purpose is to monitor and possibly change its associated component. Evolutionary change may be determined purely internally from observations made by the supervisor or may be in response to external change. Supervisory processes may be present at any level in the component hierarchy allowing us to use evolutionary behaviour as an integral part of system design.

We model such systems in a revision-based first-order logical framework in which supervisors are modelled as theories which are at a logical meta-level to the theories of their components. This enables evolutionary change of the component to be induced by revision-based changes of the supervisor at the meta-level. In this way, the intervention required in evolutionary change is modelled purely logically.

The hierarchical component-based structure is fairly intricate so we present the basic ideas firstly in a simple setting, the well-known blocks world, before introducing tree-based structures to represent component hierarchies. We also introduce some techniques for establishing the behaviour of evolvable systems specified in this logical framework. The ideas and concepts are driven by example throughout. We conclude with a more substantial example, that of a simple model of an evolvable system of automated bank teller machines.

1 Introduction

Computational systems may be viewed as evolvable at various levels of abstraction, from the rather low-level computational-step evolution of a hardware system state, or of a program's computation state, to whole or partial system reconfiguration that may occur in the maintenance or installation of software or hardware updates. The execution of a program is evolution of its computation state. It is automatic and usually happens rather quickly, on a nanosecond timescale. On the other hand, system updates have largely been a non-automated process requiring explicit user action, although internet-based computing has changed this view, e.g. automated updates of virus detection software or security updates to operating systems. These updates are relatively infrequent, being on a timescale of weeks, months or years. Such timescales of change are studied as 'Software Evolution', see e.g. [16].

There are changes to computational systems on timescales between these two extremes that can also sensibly be considered as evolution, or even adaptation. These changes are typically invoked at runtime and may be aimed at ensuring, for example, that certain behavioural requirements are satisfied, or that performance is improved, or that the system adapts its behaviour to a changing environment. Systems displaying such behaviour are naturally structured as evolutionary in that we distinguish between normal computational steps and more radical change brought about by intervention with evolutionary steps. Consider, for example, supervisory control systems for, say, reactive planning [11], or systems for adaptive querying (evaluating queries over changing databases [7]), or responsive memory management (variable capacity memory allocation), or data structure repair [5], or hybrid systems which change their

*School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK. email: howard.barringer@manchester.ac.uk

†Department of Computer Science, Kings College London, The Strand, London, WC2R 2LS, UK. email: dov.gabbay@kcl.ac.uk

‡School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK. email: david.rydeheard@manchester.ac.uk

computational behaviour in response to environmental factors which they may themselves influence [15]. One major area where evolutionary behaviour is a prominent feature is in business processes, where adaptivity to both internal and external imperatives to change is important. Thus the full computational modelling of business processes and their supporting IT structure necessarily includes evolutionary behaviour [20, 1, 12, 13].

In fact, there is increasing focus on developing software systems that feature limited forms of autonomy, adaptation or evolution. This is quite natural given the ever expanding application of computer-based processes for supporting human endeavour. Not only is this important for adaptivity and autonomy of systems, but major companies in the computing industry have turned, over the last few years, to so-called 'autonomic computing' as a way of responding to the increasing difficulties encountered in software development for massively complex, distributed and diverse computing environments [14]. The demand is for software components that can adapt themselves so that behavioural requirements remain satisfied even in partially known and unpredictably changing computing environments.

A natural architectural model of an evolvable system consists of a hierarchy of components in which each component is monitored at runtime by a dedicated supervisory process. Various names may be used for these supervisory processes, such as supervisors, monitors, evolvers, managers, controllers etc. Their role is to monitor and respond to the behaviour of their supervisee components and invoke evolutionary change when particular conditions arise. Conditions requiring evolutionary action may arise through external influence on the supervisor as well as through monitoring of the computational behaviour of the supervisee. In fact, Warboys et al. [20, 1, 12, 13] in modelling business processes take this view to an extreme in that their system architectures are constructed in such a way that every component consists of a pair of an 'evolver' and its underlying component, which they call a 'producer'. Thus the fundamental building blocks of their systems are 'evolver/producer-pairs'. Moreover, evolvers can create their associated producers, and so the evolvers determine the entire behaviour of the system.

The dynamic structure of evolutionary change, that is, the way that an evolutionary step may be invoked in an already running system, needs careful consideration. For such a step to take place, normal computation must be suspended. This may be an interruption with immediate termination of an entire system. However, a less comprehensive termination may be appropriate, for example it may mean running until a 'quiescent' state is reached (as in [1, 12]) and then possibly just locally i.e. only those components affected by the evolutionary change need to run to termination. The system then evolves – the dynamics of this may be complex as an evolution may affect many components and may also introduce or change supervisory processes. Finally, the system is restarted from a suitable point in the computation and in a suitable state. This new state is usually a modified form of the state before the evolution, with parts of the state persisting through the evolutionary change. Clearly the mechanism by which evolutionary actions intervene in normal computation is important for implementing evolvable systems. However, it is not at a suitable level of abstraction for tractable specification, design and verification methods for evolvable systems.

It is with a view to introducing evolution at a higher level of abstraction in system description that we introduce a logical account of evolutionary systems. Mathematical theories of computation have largely ignored these types of evolutionary behaviours. Attention instead has focussed on developing models of computation to support effective reasoning about fixed sequential, parallel and distributed software and hardware. The issue for us is that as one changes the nature of systems so that adaptation and/or evolution becomes a dominant feature which is present at a high level of system organisation, how does one specify and formally reason about such system behaviour? The introduction of evolutionary behaviour appears to allow considerably more freedom in the way that systems may behave and it is not at all obvious that traditional methods for modelling the semantics, or for specifying and reasoning about systems, remain adequate. Our aim is to provide a logical foundation for modelling evolvable systems that allows considerable flexibility in the design of these systems, enabling us to distribute evolutionary behaviour amongst components at various levels in a component hierarchy and supporting effective reasoning about the behaviour of such evolvable component systems.

In this paper we introduce and develop an appropriate "model-oriented" specification framework, somewhat akin to VDM or Z but with different logical foundations, where the focus of attention is on the specification of system state and on actions (or operations) that change the system state. Component specifications are thus presented as first-order theories. In a subsequent paper (Part II of this account), we extend component specifications to include programs which constrain the possible computational paths of a system, and we describe the semantics of these programs. One approach to semantic modelling has been explored in [3] where we describe a temporal logic-based approach to specifying evolvable systems behaviour and give their models in terms of Kripke structures. The Kripke structures are themselves built from Kripke structures to provide two-level models in which we maintain a separation between evolutionary steps (between Kripke structures) and normal computation (within a Kripke structure). We do not pursue this approach to modelling here — explicit description in terms of Kripke structures becomes unwieldy for highly structured component-based systems. Instead, we introduce logical structures which capture the component hierarchy of a system and also model the logical relationship between supervisors and supervisees within a system. Kripke structures are then naturally associated with a transition-based operational semantics of programs in such a logical model and temporal specifications are associated with monitoring programs within supervisors. We consider these issues in the sequel paper.

Let us now turn to the form of component hierarchy that we have in mind. Component hierarchies are natural not only from a system-structuring perspective but also in terms of evolutionary behaviour. Supervisory processes are associated with components in a system, and these pairs of a supervisor and supervisee themselves form components, from which new components may be built. These new components, whose subcomponents are evolvable, may themselves be evolvable, i.e. have associated supervisory processes to monitor the sub-hierarchy of (evolvable) components and able to evolve both the subcomponents and their supervisors. In this way, we may introduce evolutionary behaviour both locally within components and at various levels in a component hierarchy. We consider a wide range of possible evolutionary changes, from local changes that involve modifying the structure within an individual component, or the replacement of a single component with a new one, through to entire reconfigurations of a component system.

Formalising the relationship between supervisors and their supervisees is the key to developing a mathematical account of evolvable component behaviour. To describe this logically, the logical theory of a supervisor needs access to a logical description of the object-level behaviour and be able to modify these descriptions as the system evolves. That is, the logic of a supervisor stands in relation to logic of the object-level system as a *meta-logic* in which the predicates, types, formulae and other aspects of the object-level logic are available. This observation itself is not new – there has been considerable interest in meta-level computational systems and the associated notion of reflection, especially in AI (see, for example, [17]). What is new here is the incorporation of this idea into the specification and modelling of component-structured systems and therefore using the idea of meta-level and reflective systems to provide a mathematical underpinning for the design of systems with distributed evolutionary capabilities and the logical analysis of the behaviour of such systems.

We express model-oriented component specifications in terms of a revision-based logic. A component's state is represented as a set of ground atomic formulae, i.e. formulae with no free variables built from a single predicate applied to arguments. These formulae correspond to observations made of the component. This has a natural correspondence with the logical description of a supervisor as monitoring and observing various 'facts' about the object-level system. These states, as sets of formulae, change as the system runs. We describe the changes in a 'belief revision' style and treat the actions that a component may undertake as revisions to the state, as is standard in revision-based logics [9, 18, 24, 8]. One advantage of this approach is the built-in persistency — formulae change only when specified by a revision action, all other formulae remain unchanged. For evolvable systems, this is an appropriate property of the logic as evolutionary steps explicitly change part of the computational structure and all other parts of the system should remain unchanged.

Supervisors themselves have states of the same form. These states record facts about the supervisor's current computational state, and, in addition, contain observations of the state of the supervisee. Of course, these observations must actually hold in the supervisee state. We will examine exactly what this means and define a suitable relationship between supervisor and supervisee states. Revision actions at the meta-level may induce change at the object-level, that is, if a supervisor undertakes an evolutionary action, the supervisee must change so that the supervisor and supervisee states remain correctly related. This is an induced action at the object-level as there is no explicit definition of the action, only that the state must change to agree with that of the meta-level. In this way we abstract away from the operational details of how a supervisor executes an evolutionary step. Moreover, treating the supervisor as a revision system in exactly the same form as the object-level component allows us to express directly the hierarchical nature of evolvable component-based systems. The state of such a hierarchical system is built from the states of the constituent components as a tree of states. By 'state' we mean not only the computational state but also the information determining the behaviour of the components and their subcomponents. We need to maintain this in its structured form so that the effect of component reconfigurations can be determined. Evolutionary steps for component-based systems therefore involve fairly intricate tree manipulations. This may sound a little complex but, declaratively and logically, it is fairly straightforward.

The presentation here is example-driven throughout. The next section introduces the basic idea of describing object-level systems with each action specified as a revision process and then reflecting the object-level system at a meta-level for monitoring and evolutionary purposes. The example we choose is a 'blocks world' [24] where sets of ground atomic formulae are a standard description of states, and revision is a natural way of describing state change. Section 3 takes the ideas forward into the world of components, component composition, development and evolution. We start with a simple example: buffers as components, and then progress, in Section 4, to a larger-scale example of an evolvable system — a simple model of an evolvable banking system of automated teller machines. As well as introducing a logical framework and developing examples within it, we also begin to explore an appropriate proof theory which allows us to link logical descriptions of systems to their behaviour. In Section 5 we reflect on the expressivity, structure and applications of this logical account.

In a sequel to this paper (Part II, in preparation), we consider how to extend this framework to incorporate programs for sequencing actions. We describe a prototype programming language and its semantics, showing how components may contain programs and how programs distributed amongst components in a hierarchical assembly are combined to provide a description of the computation of the overall system. Supervisors, as well as their supervisees, may contain programs. These allow us to describe monitoring processes and algorithms for evolutionary behaviour.

2 The Blocks World example

We start with the well-known blocks world [24] and present it in a first-order logic. The world consists of blocks which may be placed on a table or on other blocks. The state of a world at any time is represented by a finite set of formulae which describes observations on the positions of blocks. Actions may be performed to move blocks around and these are described as 'revisions' or 'updates' of the state.

We describe this basic model and then show how to extend it with capacities for blocks and for tables. Section 2.6 introduces the idea of a meta-level description to observe the blocks world and then in Section 2.7 we show how to evolve a blocks world using meta-level descriptions of the required evolutionary steps. This meta-level is that of the supervisor which monitors and invokes evolutionary change in the object-level system. The supervisor is presented as a revision system in the same form as the object-level blocks world.

2.1 An object-level logical description

We specify the blocks world in terms of states and actions. Consider a first-order language built from capital letters A, B, C, D, ... for individual constants, each letter denoting a distinct block; we use the letter T to denote a table. Let the variables x, y range over individual constants, i.e. blocks or the table. The predicate $on(x, y)$ describes the situation where the block named by x is directly on top of the block/table named by y, and $free(x)$ indicates that the block or table x is able to have a block placed on it.

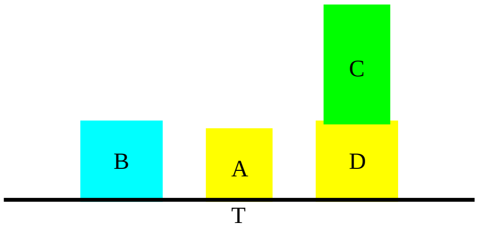


Figure 1: A blocks world situation.

The situation depicted in Figure 1 may be described by the set of ground atomic formulae

$$\Delta_0 = \{on(A, T), on(B, T), on(C, D), on(D, T), free(A), free(B), free(C)\}.$$

We view this set as a collection of known facts about the current state of the system. For the purposes of the modelling, this set records the entirety of our knowledge about the positions of blocks in the system. Of course, with just the observations $on(x, y)$ and $free(x)$, the situation where blocks A and B are swapped is indistinguishable from the current one.

Suppose block A is moved from the table onto block B. The situation, depicted in Figure 2, may be

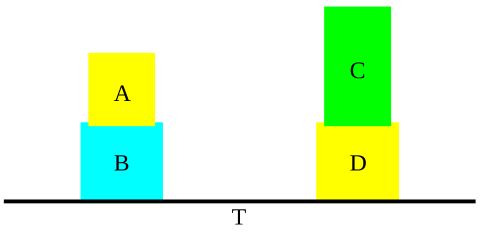


Figure 2: Another blocks world situation.

described by the set of formulae

$$\Delta_1 = \{on(A, B), on(B, T), on(C, D), on(D, T), free(A), free(C), free(T)\}.$$

We can specify such a *Move* action in several ways, for example, by pre- and post-conditions, or by a revision process on the state. One of the key requirements of the interpretation of actions is that of *persistence* — everything remains unchanged except for the blocks specifically moved. A revision-based approach (similar to STRIPS [8] and the Situation Calculus [18]) is naturally persistent and therefore an appropriate setting for describing blocks world actions. It also allows us to describe evolutionary actions as we shall demonstrate later. We thus specify an action by giving conditions under which it may be applied, i.e. a precondition set, together with a state revision. In this case, because states are simply sets of formulae, revisions take on a particularly simple form, consisting of a set of formulae to be added to a state and a set of formulae to be deleted from a state. We adopt the following schema notation to define an action to move x to y from z:

$Move(x, y, z)$	
pre	$\{on(x, z), free(x), free(y)\}$
add	$\{on(x, y), free(z)\}$
del	$\{on(x, z), free(y)\}$

We now define the state revisions determined by actions for the case of *ground* actions i.e. those obtained from schema by replacing the parameters with ground terms, for example $Move(A, B, T)$ (this keeps the definitions simple for the moment).

Definition 2.1 (Action revision - preliminary) *Let Δ be a set of ground atomic formulae (a state) and α a ground action with* pre-α, add-α *and* del-α *the precondition, addition and deletion sets for α. The revision of Δ by α, denoted by $\Delta * \alpha$ is defined when* pre-$\alpha \subseteq \Delta$ *and yields the state* $(\Delta \cup \text{add-}\alpha) \setminus \text{del-}\alpha$.

This preliminary definition requires that the preconditions occur explicitly as formulae in the state. As an example, consider the ground action $Move(A, B, T)$ and the above state Δ_0. Firstly, $\Delta_0 * Move(A, B, T)$ is defined since the precondition set $\{on(A, T), free(A), free(B)\}$ is a subset of Δ_0. Secondly, the revision adds the ground atoms $on(A, B)$ and $free(T)$ to Δ_0 and then $on(A, T)$ and $free(B)$ are deleted. Thus we have:

$$\Delta_0 * Move(A, B, T) = \Delta_1$$

The ground action $Move(A, C, B)$ is now defined for state Δ_1, in other words, we can move block A from B onto the top of block C. Indeed we have that

$$\Delta_1 * Move(A, C, B) = \{on(B, T), on(A, C), on(C, D), on(D, T), free(A), free(B), free(T)\} = \Delta_2.$$

Alternatively, we could have moved block A on top of block C directly from the initial state Δ_0, i.e. we have that

$$\Delta_0 * Move(A, C, T) = \Delta_2.$$

The account above is correct for the moves undertaken. However, there are several issues which are not addressed. One is the capacity of the table – when a block is removed from the table (as above), then the table is free to receive a block. However, when a block is placed on the table, we need to know the capacity of the table to determine whether the table is free to accommodate further blocks.

Another issue is that the *Move* action, as defined, allows us to reach states which we would consider inadmissible, e.g. $Move(A, A, T)$ moves block A onto itself. We now show how to address these issues in a logical framework.

2.2 Theories

State descriptions, in terms of sets of formulae, and transformations of state as revision actions, are defined, in general, in the presence of *logical theories*. These theories describe the properties and structure of the basic constituents of logical descriptions. As we shall see, specifying and building hierarchies of theories is the mechanism we use to describe computational systems including those that have evolutionary capabilities.

Theories have several roles in this revision-based description of computational systems. In terms of the validity of action application, whether or not a precondition holds is not simply a matter of the precondition formula occurring in the state, but is, in general, defined in terms of whether or not the formula is valid, i.e. holds in certain models of the theory. Moreover, the problem of specifying the capacity of the table in the blocks world is handled by axioms (or 'constraints') in a theory which specify the number of blocks that a table can hold. The precondition to actions which move blocks onto a table then requires that the table has free capacity.

Introducing a theory for the blocks world also addresses another problem, namely that certain forms of moves should not be accepted. For example, an action $Move(A, A, B)$ which moves A onto itself, is not valid. We can recognise these invalid moves as they lead to states that are *inconsistent* with respect to the theory.

Figure 3 specifies a suitable blocks world theory[1]. We present it as a typed first-order theory with built-in equality. The theory has TYPES introducing typed constants for individual blocks (A, B, etc.) and tables (just one, T) as enumeration types. PREDICATES are presented as two sets. We distinguish OBSERVATION predicates which are used to define the observable facts. In this case the predicate *on* is the only observation predicate. ABSTRACTION predicates are used to describe properties that may not be directly observable. In the case of the blocks world, we introduce two abstraction predicates, namely $free$, which we have already used, and *above*, which we use to describe properties of blocks world scenarios. As *on* is the only observation predicate, states contain only formulae of the form $on(b, o)$, where b is a constant denoting a block, and o a constant denoting an object (a block or table). This differs from the preliminary account above – theories allow us to link observations with abstractions such as freeness. We give a revised account of actions in the presence of a theory below.

The theory describes the CONSTRAINTS relevant to a blocks world as a collection of named constraints or axioms. For example, the formula named BWC ensures that no block can be on itself, that a block is on at most one object, etc. Towers of blocks must be acyclic, hence the introduction of the transitive *above* predicate. The constraint named $TableSize(T, 2)$ restricts the number of blocks that can be placed on table T to two and defines $free(T)$ in terms of *on*. Similarly, the constraint named $BlockSize(1)$ restricts the number of blocks that can be placed on another block to one and also defines freeness of a block. We later consider the evolution of such a blocks world system (for example changing the table size) and we will see the need to specify such constraints in a fully parametric fashion. To do this requires a metalanguage in which we may define formulae in which the number of quantified variables and the number of conjuncts and disjuncts in the formulae depend on numerical parameters.

The collection of ACTIONS of the blocks world theory consists of the single action *Move* which is defined as a revision of states, adding and deleting observations. The validity of applying this action is defined in terms of a precondition, which is a set of closed formulae constructed from both observation and abstraction predicates.

2.3 Logical foundations

It is not the purpose of this paper to set out the logical foundations of revision-based logical description. However, the notion of semantics for observation states in this approach to logical description needs to

[1] For an account of axiomatizing blocks worlds, see [4].

BlocksWorld

TYPES

$Blocks \stackrel{dfn}{=} \{A, B, C, D, E, F\}$

$Tables \stackrel{dfn}{=} \{T\}$

$Objects \stackrel{dfn}{=} Blocks \cup Tables$

OBSERVATION PREDICATES

$on : Blocks \times Objects$

ABSTRACTION PREDICATES

$free : Objects$
$above : Blocks \times Objects$

CONSTRAINTS

$BWC \stackrel{dfn}{=}$

$$\begin{aligned}
&\forall b, b_1, b_2 : Blocks, o_1, o_2 : Objects \cdot \\
&\quad \neg on(b, b) \quad \wedge \\
&\quad on(b, o_1) \wedge on(b, o_2) \Rightarrow (o_1 = o_2) \quad \wedge \\
&\quad on(b_1, b_2) \Rightarrow (\exists o : Objects \cdot on(b_2, o)) \quad \wedge \\
&\quad on(b, o_1) \Rightarrow above(b, o_1) \quad \wedge \\
&\quad above(b, b_1) \wedge above(b_1, o_2) \Rightarrow above(b, o_2) \quad \wedge \\
&\quad above(b_1, b_2) \Rightarrow \neg above(b_2, b_1)
\end{aligned}$$

$TableSize(T, 2) \stackrel{dfn}{=}$

$$\begin{aligned}
&(\exists b_1, b_2 : Blocks \cdot on(b_1, T) \wedge on(b_2, T) \wedge (b_1 \neq b_2)) \Leftrightarrow \neg free(T) \quad \wedge \\
&\forall b_1, b_2, b_3 : Blocks \cdot on(b_1, T) \wedge on(b_2, T) \wedge on(b_3, T) \Rightarrow ((b_1 = b_2) \vee (b_2 = b_3) \vee (b_1 = b_3))
\end{aligned}$$

$BlockSize(1) \stackrel{dfn}{=}$

$$\begin{aligned}
&\forall b : Blocks \cdot (\exists b_1 : Blocks, o : Objects \cdot on(b_1, b) \wedge on(b, o)) \Leftrightarrow \neg free(b) \quad \wedge \\
&\forall b_1, b_2 : Blocks \cdot on(b_1, b) \wedge on(b_2, b) \Rightarrow (b_1 = b_2)
\end{aligned}$$

ACTIONS

$Move(x : Blocks, y, z : Objects)$	
pre	$\{on(x, z), free(x), free(y)\}$
add	$\{on(x, y)\}$
del	$\{on(x, z)\}$

Figure 3: A blocks world theory.

be considered in order to understand and interpret the following account of evolvable systems.

The syntax of logical descriptions used here is standard, except that we separate out certain predicate symbols as 'observation predicates'. Theories are collections of formulae (called 'axioms' or 'constraints'). Models of a theory consist of standard first-order set-based models. That is, a model allocates sets to types, functions to function symbols, and relations to predicate symbols in such a way that each axiom of the theory holds in the model.

The notion of satisfaction for observation states, however, is not standard. Consider a state Δ as a set of formulae which are ground atomic formulae built from observation predicates only. For an arbitrary closed formula ψ and theory W, it is usual to define ψ as following model-theoretically from Δ just when ψ is satisfied in all W-models which satisfy Δ. We write this form of satisfaction as $\Delta \mathrel{|\!\!\approx}_W \psi$ to distinguish it from a second form of satisfaction relation which we now introduce.

The form of satisfaction relation appropriate for observationally defined states and revision-based logic uses *minimum models*. The definition of minimum models is based upon observational validity. We write

$$\Delta \models_W \psi$$

when closed formula ψ is satisfied in all *minimum* W-models which satisfy Δ. In an appendix to this paper, we present a general account of observational states and define the appropriate minimum models and this satisfaction relation.

A state Δ is *inconsistent* with respect to theory W just when $\Delta \models_W \mathsf{false}$. Otherwise, we say Δ is *consistent* with respect to theory W.

As an example, consider the blocks world theory $W = \mathit{BlocksWorld}$ and a state

$$\Delta = \{on(A, B), on(B, T)\}.$$

We have $\Delta \models_W \neg on(A, T)$ as this follows from an axiom of the theory (no block may be on two different objects) and hence holds in all models of Δ. However, we also have $\Delta \models_W \neg on(C, T)$ because the observation $on(C, T)$ does not hold in all models of Δ. In fact, the state Δ is meant to describe the blocks world scenarios in which there are two blocks present, A and B, with A on B and B on the table T, and *no further blocks and no further observations except where they follow from those in* Δ. The proviso here is the key to modelling in this revision-based logic and is what is captured by the relation $\models_W$, which is the relation appropriate for the logical modelling in this paper.

2.4 Revision by actions

We now define the revision operation of actions in the presence of a first-order theory W.

Definition 2.2 (Action revision - constrained) *Let W be a typed first-order theory, Δ a set of ground atomic formulae, and α a ground action of W. Let* pre-α, add-α *and* del-α *denote the pre-condition, addition and deletion sets for the action α. The revision of Δ by the ground action α, written $\Delta * \alpha$, is defined when $\Delta \models_W \bigwedge$* pre-$\alpha$ *and when the resultant state* $(\Delta \cup \text{add-}\alpha) \setminus \text{del-}\alpha$ *is consistent with respect to the theory W.*

Using this definition of revision by actions, we present the general *Move* action as follows:

$Move(x, y, z)$	
pre	$\{on(x, z), free(x), free(y)\}$
add	$\{on(x, y)\}$
del	$\{on(x, z)\}$

Now freeness no longer occurs in the states and acts only through the capacity constraints for blocks and for tables.

Example 2.1 *In the BlocksWorld theory, consider a state*

$$\Delta_0 = \{on(A, B), on(B, C), on(C, T)\}.$$

The precondition of $Move(A, T, B)$ is the set $\{on(A, B), free(A), free(T)\}$. Clearly $on(A, B)$ holds in any model satisfying Δ_0 — the formula is present in Δ_0. The formula $free(A)$, however, is not present in Δ_0 but does hold by virtue of the theory constraint named $BlockSize(1)$. Similarly, the formula

$free(T)$, also not present in Δ_0 holds by virtue of the constraint named $TableSize(T, 2)$, which requires at least two different blocks to be directly on the table T for $free(T)$ not to hold. Thus the revision

$$\Delta_0 * Move(A, T, B)$$

is defined and yields the state

$$\Delta_1 = \{on(B, C), on(C, T), on(A, T)\}.$$

Note that the precondition of $Move(B, T, C)$ does not hold for Δ_1 — $free(T)$ fails to hold since there are two distinct blocks A and C both on the table T.

There is a further modification to the notion of revision by actions. Consider the *Move* action as defined above. The third argument z is, in fact, determined by the state as x has to be on an object z for a move to take place and this is a requirement in the precondition. We can thus replace the above specification of *Move* with the following as long as we redefine revision.

$Move(x, y)$	
pre	$\{on(x, z), free(x), free(y)\}$
add	$\{on(x, y)\}$
del	$\{on(x, z)\}$

To redefine revision by actions, we allow binding of variables occurring in preconditions. Notice that there may be more than one valid binding, so that revision becomes non-deterministic in general. This is indeed what we require, although in certain cases, because of constraints in the theory, bindings (if they exist) are unique (e.g. for the case of *Move* above). A term such as $Move(x, y)$, we call an *action term* of the theory.

Definition 2.3 (Action revision) *Let W be a typed first-order theory, Δ a set of ground atomic formulae, and α an action term of W with variables $\bar{x}$. Let* pre-α, add-α *and* del-α *denote the precondition, addition and deletion sets for α. Let $\alpha(\bar{t})$ be a ground instance of α. Consider a binding $[\bar{y} \mapsto \bar{u}]$ such that*

1. $\Delta \models_W \bigwedge \text{pre-}\alpha[\bar{t}/\bar{x}][\bar{u}/\bar{y}]$, *and*
2. $\Delta' = \Delta \cup \text{add-}\alpha[\bar{t}/\bar{x}][\bar{u}/\bar{y}]) \setminus \text{del-}\alpha[\bar{t}/\bar{x}][\bar{u}/\bar{y}]$ *is consistent with respect to the theory W.*

Then we say that Δ' is a revision of Δ by the ground action $\alpha(\bar{t})$, and write $\Delta \xrightarrow{\alpha(\bar{t})} \Delta'$.

The above account is not the only one possible for describing a revision-based logical system for the blocks world. We discuss here several possible variant formulations.

To determine whether preconditions of an action hold in a state will, in practice, need automated deduction tools. In general, this is computationally expensive and, initially, we considered an alternative where preconditions applied through set membership. For this, we need to cope with cases in which, for example, the freeness of a table is neither known nor known not to be the case. To formalise such states of knowledge, we introduced non-deterministic states, with states as sets of sets of ground atomic formulae including possible freeness. However, we rejected this approach in favour of the more complex precondition check with single state for at least three reasons. Firstly, this alternative required several different forms of consistency checking. Secondly, the approach led to a more intricate relationship between meta- and object-level systems. Thirdly, we ended up with a seemingly unnecessarily complex configuration structure for components. However, this alternative approach may have value when considering how to optimise an implementation of this logical foundations for evolvable systems.

Another possibility is to define states as sets of formulae which are deductively closed under a given theory. This makes revision under actions very costly to compute. Some Epistemic Logics adopt the position that inference itself is a revision action on states. This is not appropriate for the modelling we use here.

It should be noted that the use of abstraction predicates, rather than, say, the inclusion of capacity constraints directly as formulae in preconditions, is dictated by several considerations. Including capacity constraints as preconditions is not only poor design but also makes evolutionary steps difficult to define. Consider, for example, an evolution that increases the capacity of the table. For this to be definable, the table capacity needs to be sufficiently localised in the description so that the evolutionary step is itself a revision action. This is therefore a point where the 'evolvability' of a system influences the way that systems are structured.

2.5 Configurations

We now give a general formulation of systems such as the blocks world described above.

Let W be a first-order theory in a typed first-order language $\mathcal{L}$. The predicate symbols of the theory W are partitioned into OBSERVATIONS and ABSTRACTIONS. Let $GroundAtom$ be the set of atomic formulae with no free variables built from observation predicate symbols only. We define $State \triangleq 2^{GroundAtom}$ — the set of all states.

Let $Formula$ be the set of formulae of the first-order language $\mathcal{L}$ and $ObsFormula$ be the set of atomic formulae built from observation predicates only.

Let $ActId$ denote a finite set of action names, e.g. $Move$. Define the set

$$ActionDefs \triangleq ActId \to (Args \times 2^{Formula} \times 2^{ObsFormula} \times 2^{ObsFormula})$$

to contain the pre, add and del sets associated with each action. The set $Args$ is the set of argument lists.

Let $GroundTerm$ denote the set of ground terms of W and $GroundTerms$ the set of lists of ground terms (sometimes denoted $GroundTerm^*$), then define

$$GroundAction \triangleq ActID \times GroundTerms$$

of ground actions, for example $Move(A, T, B)$.

We now introduce the notion of a *configuration*. This is the logical structure around which we build an account of evolvable systems. A configuration describes the current state as well as the structure of the theory and the actions:

Definition 2.4 (Configuration) *In a first-order language $\mathcal{L}$, define a configuration as $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ where:*

Δ *is in State (i.e. is a subset of Ob),*

$\mathcal{C}$ *is a finite collection of parametrically named closed formulae of $\mathcal{L}$,*

$\mathcal{A}$ *is in $ActionDefs$ i.e. is s collection of action schema.*

A configuration $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ is said to be consistent iff the state Δ is consistent with respect to the first-order theory W defined by the constraints $\mathcal{C}$.

Let $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ be a configuration. For action name α, define pre-α as the first element of the triple $\mathcal{A}(\alpha)$, similarly, add-α and del-α name the second and third elements. The revision of states by actions is defined in Definition 2.3 where the theory W has the formulae named in $\mathcal{C}$ as constraints.

For a ground action $\alpha \in GroundAction$ and two consistent configurations $\mathbf{C} = (\Delta, \mathcal{C}, \mathcal{A})$ and $\mathbf{C}' = (\Delta', \mathcal{C}, \mathcal{A})$, we say that $\mathbf{C}'$ is a revision of $\mathbf{C}$ by α iff $\Delta \xrightarrow{\alpha} \Delta'$. In this case, we write $\mathbf{C} \xrightarrow{\alpha} \mathbf{C}'$.

2.6 Observing the Blocks World: Beginnings of a supervisor

Our goal is to describe a system that monitors the blocks world and provides a mechanism for changing the structure of this world. We refer to such a system as a *supervisor*. If we regard the blocks world to be object-level, then the supervisor is a meta-level system for this object-level. We describe the supervisor as a system in the same form as the object-level, that is, as a collection of revision actions on sets of formulae. Using meta-level revision actions, the supervisor may record observations about the blocks world as it changes through object-level actions. This can be thought of as the supervisor 'monitoring' the object-level system.

Let $c_0, c_1, c_2, \ldots$ be a list of distinct labels which the supervisor uses to name particular configurations at the object-level. We track the configurations through these names and introduce a successor function

s such that $c_1 = s(c_0)$, $c_2 = s(c_1)$, etc. At the supervisor level, we use the predicate $current(c_i)$ to indicate that the name c_i refers to the current object-level configuration that the supervisor is observing.

We reflect the object-level state observations at the meta-level using a *holds* predicate in the supervisor configuration. For example, if $holds(free(T), c_0)$ is present in a state of the supervisor, then this is interpreted as an observation of the object-level configuration named c_0 (by the meta-level) in which the object-level formula $free(T)$ holds in the state.

The supervisor, in fact, observes more than just the observation state at the object-level. It can observe all the structure of the object-level, including the actions and constraints. For example, we use the meta-level predicate $constraint(TableSize(T, 2))$ to indicate that the constraint named as $TableSize(T, 2)$ is present in the object-level configuration.

Monitoring actions of the supervisor are of the form $Observe(P)$, which record the observation that the set of formulae P holds in the blocks world configuration. These formulae may be observation formulae in the current object-level state or may be formulae constructed from observation and abstraction predicates and which hold in the current observation state. The effect of $Observe(P)$ at the meta-level is specified using the same approach as above, i.e. using revisions:

$Observe(P)$	
pre	$\{current(c)\}$
add	$\{holds(p, s(c)) \mid p \in P)\} \cup \{current(s(c))\}$
del	$\{current(c)\}$

Note that the supervisor is responsible for keeping the *current* state appropriately named[2].

We now give an example of a supervisor execution trace for a blocks world. Let us consider a blocks world theory with table size constraint of 3. Consider the following blocks world sequence of states.

$$\Delta_0 = \left\{ \begin{array}{l} on(A, B), \\ on(B, C), \\ on(C, D), \\ on(D, T) \end{array} \right\} \xrightarrow{Move(A,T,B)} \Delta_1 = \left\{ \begin{array}{l} on(A, T), \\ on(B, C), \\ on(C, D), \\ on(D, T) \end{array} \right\} \ldots$$

$$\xrightarrow{Move(B,T,C)} \ldots \Delta_2 = \left\{ \begin{array}{l} on(A, T), \\ on(B, T), \\ on(C, D), \\ on(D, T) \end{array} \right\} \xrightarrow{Move(C,B,D)} \Delta_3 = \left\{ \begin{array}{l} on(A, T), \\ on(B, T), \\ on(C, B), \\ on(D, T) \end{array} \right\}$$

An example of a corresponding supervisor trace of this blocks world is depicted below in terms of the supervisor's states. We use the notation $\Delta^{\mathtt{M}}$ to refer to states of the supervisor (at the meta-level).

$$\Delta_0^{\mathtt{M}} = \left\{ \begin{array}{l} current(c_0), \\ holds(free(T), c_0) \end{array} \right\} \xrightarrow{Observe(\{free(T)\})} \Delta_1^{\mathtt{M}} = \left\{ \begin{array}{l} current(s(c_0)), \\ holds(free(T), c_0), \\ holds(free(T), s(c_0)) \end{array} \right\} \xrightarrow{Observe(\{\})} \ldots$$

$$\ldots \Delta_2^{\mathtt{M}} = \left\{ \begin{array}{l} current(s(s(c_0))), \\ holds(free(T), c_0), \\ holds(free(T), s(c_0)) \end{array} \right\} \xrightarrow{Observe(\{\})} \Delta_3^{\mathtt{M}} = \left\{ \begin{array}{l} current(s(s(s(c_0)))), \\ holds(free(T), c_0), \\ holds(free(T), s(c_0)) \end{array} \right\}$$

In this example, the supervisor observes the freeness of the table, i.e. the validity of the object-level formula $free(T)$. Note that the truth of $free(T)$ is not immediately apparent from the blocks world state. It is derived from the state using the $TableSize$ constraints. In the above traces, the initial state of the supervisor $\Delta_0^{\mathtt{M}}$ reflects the initial state at the object-level (we define this relationship between states below). Later we give the supervisor further powers to view other parts of the object-level configurations, namely the constraints and the actions. The supervisor action $Observe(\{free(T)\})$ revises the supervisor state $\Delta_0^{\mathtt{M}}$ to yield the state $\Delta_1^{\mathtt{M}}$. We see that after the next supervisor observation, in the corresponding object-level state labelled $s(s(c_0))$, $free(T)$ no longer holds as there are three blocks on the table and so $holds(free(T), s(s(c_0)))$ is *not* a formula in the resulting supervisor state and $Observe(\{free(T)\})$ is not a valid revision action. Similarly for the observation after this.

We now define this relationship linking object-level and meta-level states, that is, we characterise formally the reflective arrow in Figure 4.

[2]In fact, for the presentation we make here we require the meta-level theory constraint $\forall c_1, c_2 \cdot current(c_1) \wedge current(c_2) \Rightarrow c_1 = c_2$ ensuring uniqueness of the current object-level state. We introduce such constraints later to define meta-level theories for the blocks world.

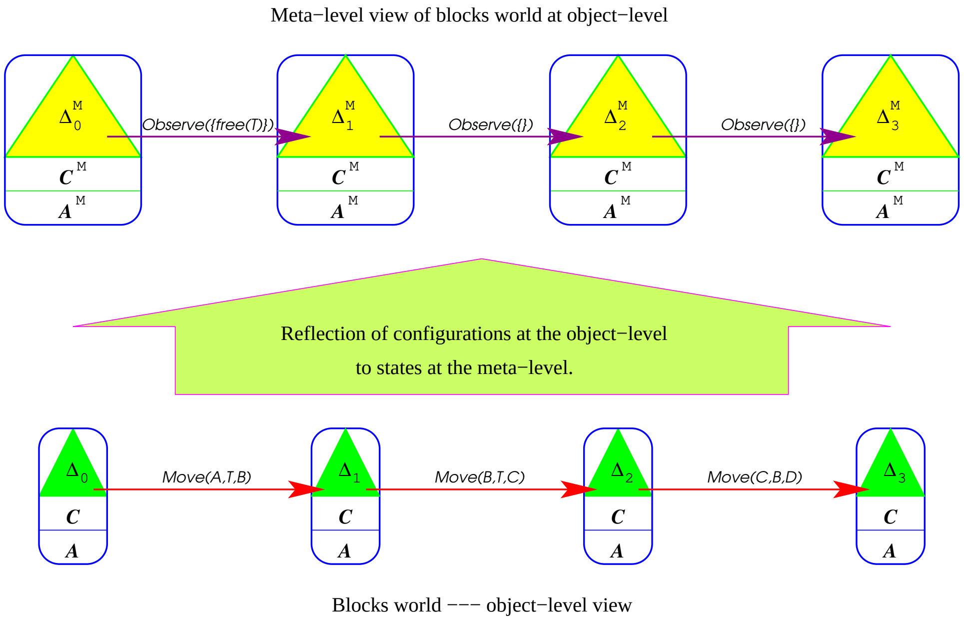


Figure 4: Object-level to meta-level reflection.

Definition 2.5 (State meta-view — first version) *Let W^M and W be first-order theories for meta-level and object-level systems respectively. We say that $\Delta^{\mathtt{M}}$ (from a configuration of W^M) is a* state meta-view *of Δ (of a configuration of W) when, for all formulae φ and configuration names c, if*

$$\{current(c), holds(\varphi, c)\} \subseteq \Delta^{\mathtt{M}}$$

then φ is a formula of W and $\Delta \models_W \varphi$.

For the above example of traces, each $\Delta_i^{\mathtt{M}}$ is a state meta-view of the blocks world Δ_i.

Note that the state of the supervisor maintains a history of observations. This is a consequence of the naming of the configurations and the persistence inherent in the revision process. It allows us to use historical observations to determine when meta-level actions should be applied — in particular, it encodes certain temporal information.

2.7 Evolving the Blocks World

In the preceding section, we described an action at the meta-level that corresponds to observing a blocks world state. Here, we show how to define a supervisor action that can expand the capacity of the blocks world table. Note that in the blocks world theory (Figure 3), the capacity of the table is fixed and determined by a constraint named $TableSize(T, 2)$. The meta-level, which has access to all aspects of the blocks world, has the ability to change these object-level constraints. Recall that a configuration contains constraints as parametrically named formulae. Let the meta-level formula $constraint(TableSize(T, 2))$ mean that the object-level configuration has a formula named $TableSize(T, 2)$ amongst its current constraints.

Consider a supervisor action $Expand(n)$ which expands the table capacity to size n specified as follows.

$Expand(n : Int)$	
pre	$\{current(c), constraint(TableSize(T, m)),\ m < n\}$
add	$\{current(s(c)), holds(free(T), s(c)), constraint(TableSize(T, n))\}$
del	$\{current(c), constraint(TableSize(T, m))\}$

Consider an instance of this action, for example, $Expand(4)$ on a table of size 2. The action is defined if the blocks world configuration named in the supervisor as c is recorded as being current and that the current blocks world has a constraint named $TableSize(T, 2)$. The defined action then specifies that the constraint $TableSize(T, 2)$ is removed in the next blocks world configuration and replaced by the constraint of capacity of 4. Here we use the parameterised nature of the naming of constraints, which we define later. Notice also, that because this is a strict expansion of the table, we can assert at the meta-level that $holds(free(T), s(c))$, that is, the table is free to accept further blocks after a strict expansion. This relies on the consistency of the pre-expansion configuration, i.e. that no more than 2 blocks are on the table.

In the previous section, we gave a preliminary definition relating the meta-level and object-levels — the *state meta-view*. This definition now needs modification to take account of the *constraint* predicate.

Definition 2.6 (State meta-view — second version) *Let W^M and W be first-order theories for the meta-level and object-level respectively. We say that $\Delta^{\mathtt{M}}$ (from a configuration of W^M) is a* state meta-view *of a configuration $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ of theory W when*

1. *for all formulae φ and configuration names c, if $\{current(c), holds(\varphi, c)\} \subseteq \Delta^{\mathtt{M}}$ then φ is a formula of W and $\Delta \models_W \varphi$, and*
2. *for all ground constraint expressions cn, if $constraint(cn) \in \Delta^{\mathtt{M}}$ then $cn \in \mathcal{C}$.*

This definition relates meta-level states and object-level configurations, but, in itself, is not sufficient to enable us to define evolutionary actions which may be invoked by the supervisor. What we need is (1) new meta-level predicates to indicate how evolutionary actions can enforce change at the object-level, and (2) a relation between states before and after actions, relating states at the object-level to those at the meta-level. We now show how to define both of these. By doing so we can ensure the required persistency at the object-level (for example, an expand action should not change the arrangement of the blocks!), and that in this revision-based approach, we have a simple mechanism for induced evolutionary change.

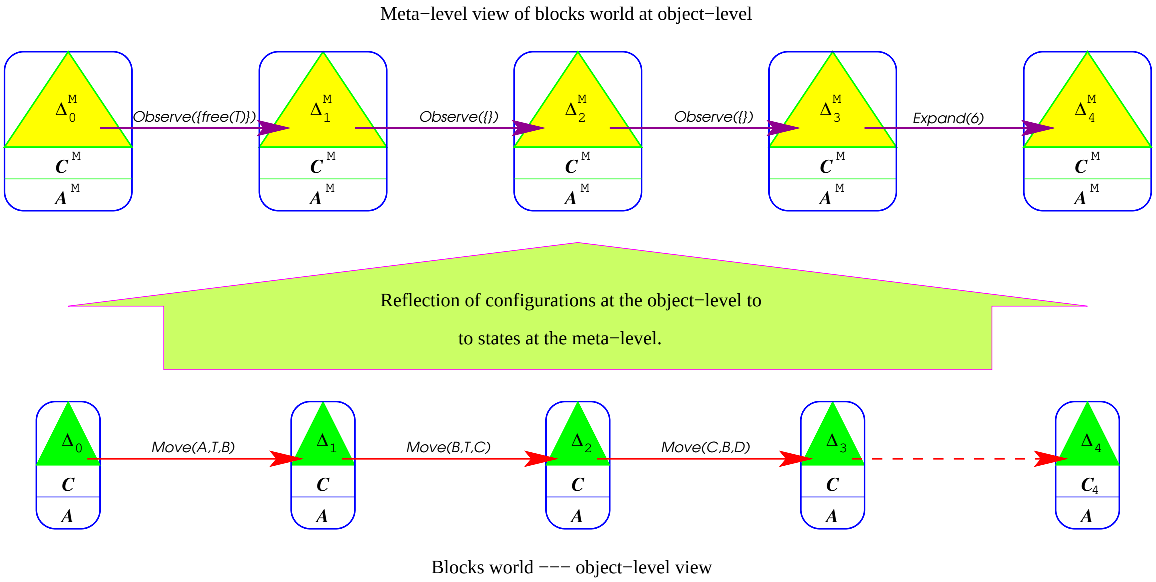
Firstly, we introduce a meta-level predicate $evolve(\delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, c)$ that corresponds to a revision of the object-level state which adds $\delta^+_{\mathcal{D}}$ and deletes $\delta^-_{\mathcal{D}}$. To ensure that this is correctly interpreted, we now define the relation of transition meta-view:

Definition 2.7 (Transition meta-view — first version) *Given meta-level states, $\Delta^{\mathtt{M}}$ and $\Delta^{\mathtt{M}\prime}$ of theory W^M, and object-level configurations, $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A} \rangle$ and $\mathbf{C}' = \langle \Delta', \mathcal{C}', \mathcal{A}' \rangle$ of theory W, such that $\Delta^{\mathtt{M}}$, $\Delta^{\mathtt{M}\prime}$ are state meta-views of $\mathbf{C}$, $\mathbf{C}'$ respectively, we say that the pair $\langle \Delta^{\mathtt{M}}, \Delta^{\mathtt{M}\prime} \rangle$ is a* transition meta-view *of $\langle \mathbf{C}, \mathbf{C}' \rangle$ when if $\{evolve(\delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, c), current(c)\} \subseteq \Delta^{\mathtt{M}\prime}$ and $\Delta' = \Delta \cup \delta^+_{\mathcal{D}} \backslash \delta^-_{\mathcal{D}}$ is consistent with W then $\mathbf{C}' = \langle \Delta', \mathcal{C}, \mathcal{A} \rangle$.*

We now modify the definition of the *Expand* action to include $evolve(\{\}, \{\}, s(c))$ as a formula added to the state. This ensures, through the transition meta-view, that *Expand* induces a change at the object-level that does not alter the object-level state i.e. the arrangement of blocks on the table.

$Expand(n : Int)$	
pre	$\{current(c), constraint(TableSize(T, m)),\ m < n\}$
add	$\{current(s(c)), holds(free(T), s(c)), evolve(\{\}, \{\}, s(c)),$ $constraint(TableSize(T, n))\}$
del	$\{current(c), constraint(TableSize(T, m))\}$

To illustrate this, let us extend the execution traces of the supervisor and the blocks world with an *Expand* action:



We require that each pair of consecutive states at the meta-level is a *transition meta-view* of the corresponding configurations at the object-level. Consider the pairs $\langle \Delta_3^{\mathtt{M}}, \Delta_4^{\mathtt{M}} \rangle$ and $\langle \mathbf{C}_3, \mathbf{C}_4 \rangle$ where

$$\Delta_3^{\mathtt{M}} = \left\{ \begin{array}{l} current(s(s(s(c_0)))), \\ holds(free(T), c_0), \\ holds(free(T), s(c_0)), \\ constraint(TableSize(T, 3)) \end{array} \right\}.$$

In this example, the object-level configuration starts with a constraint of $TableSize(T, 3)$, which is recorded via the *constraint* predicate in the meta-level state. The preconditions of the supervisor action $Expand(6)$ are then satisfied in the supervisor state $\Delta_3^{\mathtt{M}}$. A revision by $Expand(6)$ is therefore defined and gives rise to the supervisor state $\Delta_4^{\mathtt{M}}$, where:

$$\Delta_4^{\mathtt{M}} =$$
$$\left\{ \begin{array}{l} current(s(s(s(s(c_0))))), \\ holds(free(T), c_0), holds(free(T), s(c_0)), holds(free(T), s(s(s(s(c_0))))), \\ evolve(\{\}, \{\}, s(s(s(s(c_0))))), \\ constraint(TableSize(T, 6)) \end{array} \right\}$$

Now consider the object-level trace. Assuming that $TableSize(T, 3)$ is indeed the name of one of the constraints in $\mathcal{C}$, then $\Delta_3^{\mathtt{M}}$ is indeed a state meta-view of the corresponding object-level configuration, where

$$\Delta_3 = \left\{ \begin{array}{l} on(A, T), \\ on(B, T), \\ on(C, B), \\ on(D, T) \end{array} \right\}.$$

After the *Expand* action, the state meta-view requirement is that $TableSize(T, 6)$ a constraint of $\mathbf{C}_4$. Moreover if

$$\Delta_4 = \{on(D, T)\}$$

then $\Delta_4^{\mathtt{M}}$ is a state meta-view of $\mathbf{C}_4$. However, the pair of (object-level) configurations $\langle \mathbf{C}_3, \mathbf{C}_4 \rangle$ do NOT satisfy the requirements for $\langle \Delta_3^{\mathtt{M}}, \Delta_4^{\mathtt{M}} \rangle$ to be a transition meta-view of $\langle \mathbf{C}_3, \mathbf{C}_4 \rangle$. The blocks world configuration has not been preserved. The observations noted at the meta-level require

$$\Delta_4 = \left\{ \begin{array}{l} on(A, T), \\ on(B, T), \\ on(C, B), \\ on(D, T) \end{array} \right\}.$$

in which there is no change in the blocks world state. Secondly, however, because the set of constraints in the object-level has changed, the transition meta-view relation also fails. We must therefore modify the previous *evolve* predicate to include additions and deletions that may simultaneously occur in the state and in the constraint set, i.e. the predicate is of the form

$$evolve(\delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, \delta^+_{\mathcal{C}}, \delta^-_{\mathcal{C}}, c)$$

where $\delta^+_{\mathcal{D}}$ are the additions to the state, $\delta^-_{\mathcal{D}}$ are the deletions to the state, $\delta^+_{\mathcal{C}}$ are the additions to the set of constraints of the configuration and theory, and $\delta^-_{\mathcal{C}}$ are the deletions to this constraint set.

The definition of the *transition meta-view* relation is thus modified:

Definition 2.8 (Transition meta-view — second version) *Given meta-level states,* $\Delta^{\mathtt{M}}$ *and* $\Delta^{\mathtt{M}\prime}$ *of theory* W^M*, and object-level configurations,* $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A}\rangle$ *and* $\mathbf{C}' = \langle \Delta', \mathcal{C}', \mathcal{A}'\rangle$ *of theory* W*, such that* $\Delta^{\mathtt{M}}$*,* $\Delta^{\mathtt{M}\prime}$ *are state meta-views of* $\mathbf{C}$*,* $\mathbf{C}'$ *respectively, we say that the pair* $\langle \Delta^{\mathtt{M}}, \Delta^{\mathtt{M}\prime}\rangle$ *is a* transition meta-view *of* $\langle \mathbf{C}, \mathbf{C}'\rangle$ *when if* $\{evolve(\delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, \delta^+_{\mathcal{C}}, \delta^-_{\mathcal{C}}, c), current(c)\} \subseteq \Delta^{\mathtt{M}\prime}$ *and* $\Delta' = \Delta \cup \delta^+_{\mathcal{D}} \backslash \delta^-_{\mathcal{D}}$ *is consistent in theory* W'*, where* W' *is the theory* W *with constraint set* $\mathcal{C}$ *revised to* $\mathcal{C}' = (\mathcal{C} \cup \delta^+_{\mathcal{C}} \backslash \delta^-_{\mathcal{C}})$*, then* $\mathbf{C}' = \langle \Delta', \mathcal{C}', \mathcal{A}\rangle$*.*

This definition requires that the revised object-level state $\Delta \cup \delta^+_{\mathcal{D}} \backslash \delta^-_{\mathcal{D}}$ is consistent with respect to the revised set of object-level theory constraints, $\mathcal{C} \cup \delta^+_{\mathcal{C}} \backslash \delta^-_{\mathcal{C}}$. In light of this change, we modify the previous specification of the supervisor *Expand* action to become the following:

$Expand(n : Int)$	
pre	$\{current(c), constraint(TableSize(T, m)),\ m < n\}$
add	$\{current(s(c)), holds(free(T), s(c)),$ $evolve(\{\}, \{\}, \{TableSize(T, n)\}, \{TableSize(T, m)\}, s(c)),$ $constraint(TableSize(T, n))\}$
del	$\{current(c), constraint(TableSize(T, m))\}$

With appropriate definitions of the formulae $TableSize(T, 3)$ and $TableSize(T, 6)$, in the example above, $\langle \Delta^{\mathtt{M}}_3, \Delta^{\mathtt{M}}_4\rangle$ is indeed a transition meta-view of $\langle \mathbf{C}_3, \mathbf{C}_4\rangle$. The meta-level action $Expand(6)$ on a table of size 3 therefore changes the blocks world to *evolve* to a new system whose table T has doubled its previous capacity.

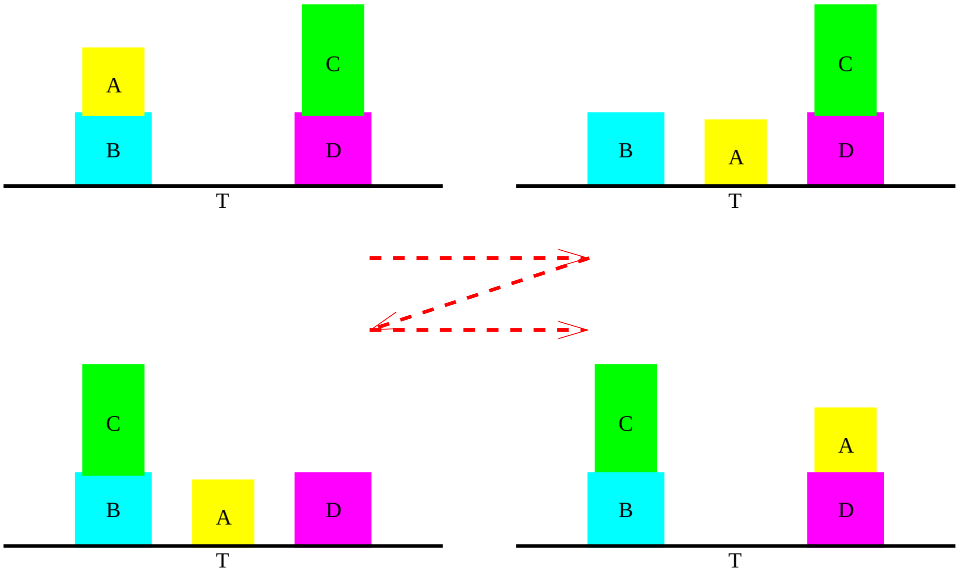
Example 2.2 (Contraction) *As another example of a simple blocks world evolution, consider contraction of the table size. This again is a supervisor action, which, in this case, we define to reduce the size by one. Notice that we cannot assert freeness of the table at the meta-level after contraction, as we could for strict expansion. Notice also that, if the table already has a maximum number of blocks on it, expansion is permitted but contraction does not lead to a pair of states related as a transition meta-view, since consistency of the resulting object-level state is required, and thus contraction is not permitted in this case.*

$Contract()$	
pre	$\{current(c), constraint(TableSize(T, m)), m > 0\}$
add	$\{current(s(c)),$ $evolve(\{\}, \{\}, \{TableSize(T, m-1)\}, \{TableSize(T, m)\}, s(c)),$ $constraint(TableSize(T, m-1))\}$
del	$\{current(c), constraint(TableSize(T, m))\}$

2.8 Changing Blocks World actions

The preceding section considered an evolution of the blocks world in which the object-level table T of the blocks world had its capacity changed. In this section, we show how a supervisor can introduce, delete and modify actions of the blocks world.

Imagine a situation where the blocks world has the following series of moves made quite often.



The table is used as intermediate location in order to perform a swap of blocks A and C from the state $\{on(A,B), on(C,D), \ldots\}$ to $\{on(C,B), on(A,D), \ldots\}$. The supervisor observes this common pattern of behaviour and introduces a two-armed robot that is able to swap the blocks directly without using the table! In other words, the system evolves so that the following action is introduced at the object-level.

$Swap(x, y, u, v)$	
pre	$\{on(x,u), free(x), on(y,v), free(y)\}$
add	$\{on(x,v), on(y,u)\}$
del	$\{on(x,u), on(y,v)\}$

The supervisor must therefore add this *Swap* action to the blocks world configuration. To do this, we extend the meta-level predicate *evolve* to include the addition and deletion of actions. Thus, in

$$evolve(\delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, \delta^+_{\mathcal{C}}, \delta^-_{\mathcal{C}}, \delta^+_{\mathcal{A}}, \delta^-_{\mathcal{A}}, c),$$

the first four arguments are as before, and $\delta^+_{\mathcal{A}}$ and $\delta^-_{\mathcal{A}}$ are the additions and deletions to the set of actions. For example, the presence of

$$evolve(\{\}, \{\}, \{\}, \{\}, \{[action\_name \mapsto \langle vars, pre, add, del\rangle]\}, \{\}, c),$$

at the meta-level is used to add an action *action_name* at the object-level specified by

$$[action\_name \mapsto \langle vars, pre, add, del\rangle]$$

for sets *vars*, *pre*, *add*, and *del*. The supervisor might therefore have the following action defined.

$addAction(name, vars,$ $pre, add, del)$	
pre	$\{current(c)\}$
add	$\{evolve(\{\}, \{\}, \{\}, \{\}, \{[name \mapsto \langle vars, pre, add, del\rangle]\}, \{\}, s(c)),$ $current(s(c))\}$
del	$\{current(c)\}$

We need to amend the definition of *transition meta-view* to take account of changing the actions in the configuration.

Definition 2.9 (Transition meta-view — third version) *Given meta-level states, $\Delta^{\mathtt{M}}$ and $\Delta^{\mathtt{M}\prime}$ of theory W^M, and object-level configurations, $\mathbf{C} = \langle \Delta, \mathcal{C}, \mathcal{A}\rangle$ and $\mathbf{C}' = \langle \Delta', \mathcal{C}', \mathcal{A}'\rangle$ of theory W, such that $\Delta^{\mathtt{M}}$, $\Delta^{\mathtt{M}\prime}$ are state meta-views of $\mathbf{C}$, $\mathbf{C}'$ respectively, we say that the pair $\langle \Delta^{\mathtt{M}}, \Delta^{\mathtt{M}\prime}\rangle$ is a* transition meta-view *of $\langle \mathbf{C}, \mathbf{C}'\rangle$ if $\{evolve(\delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, \delta^+_{\mathcal{C}}, \delta^-_{\mathcal{C}}, \delta^+_{\mathcal{A}}, \delta^-_{\mathcal{A}}, c), current(c)\} \subseteq \Delta^{\mathtt{M}\prime}$ and $\Delta' = \Delta \cup \delta^+_{\mathcal{D}} \backslash \delta^-_{\mathcal{D}}$ is theory W' consistent, where W' is the theory W with set $\mathcal{C}$ revised to $\mathcal{C}' = (\mathcal{C} \cup \delta^+_{\mathcal{C}} \backslash \delta^-_{\mathcal{C}})$, then $\mathbf{C}' = \langle \Delta', \mathcal{C}', \mathcal{A} \cup \delta^+_{\mathcal{A}} \backslash \delta^-_{\mathcal{A}}\rangle$.*

2.9 A Blocks World supervisor theory

We now bring together the elements of a supervisor of the blocks world and present it as a typed first-order theory which is at a *meta-level* to the blocks world theory (Figure 3). To be a meta-level theory, the theory is equipped with a number of built-in types for the names of entities of the object-level theory, these include the *predicates*, *variables*, *formulae* and the other syntactic classes of the object-level. A formula $free(T)$ of the object-level theory becomes the term "$free(T)$", with appropriate conditions to ensure compositionality of quoting, e.g. "$free(T)$"="$free$"("T"). See [2] for an exposition of meta-level structure, in this case for the executable temporal logic MetateM.

For the presentation here, we consider meta-theories that include a type obsFormula that corresponds to the set of atomic observation formulae of the object-theory (with obsFormulae as the powerset of obsFormula), a type groundAtom of ground atomic observation formulae, a type formula that corresponds to the set of formulae of the object theory, and so on. For notational convenience, we omit quotation marks from object-level entities quoted at the meta-level when the context is apparent, e.g. the formula $constraint(TableSize(T, 2))$ used in the blocks world supervisor theory is actually here denoting the quoted blocks world level constraint name denoted by the same string of symbols. We abuse notation even further by allowing meta-level variables to be embedded within quoted object level entities, such as $constraint(TableSize(T, m))$, that uses the meta-level variable m within the tacitly quoted parametric constraint name.

One crucial aspect of this meta-level to object-level linkage is that when we define a theory to be meta to another then, if there is a change to the object-theory, that change is directly reflected in the meta-theory through the types representing the object-level entities. For example, if a blocks world theory is revised to contain a new predicate then any theory that has been defined meta to the blocks world theory will automatically be revised to reflect the updated sets of predicate names, atomic formulae, formulae, etc., that arise through the addition of the new predicate.

The blocks world supervisor theory is presented below. It is prefixed with the keyword Meta to indicate that this is a theory at a meta-level to an object-level and so has access to the logic and state of an object-level configuration. Thus, this meta-level theory is to be combined with an object-level theory to form a joint theory of a supervisor and its supervisee. Actions of the meta-level theory are run in conjunction with object-level actions in combinations which we explore later. We do not specify an object-level theory as part of the meta-level theory. In general, meta-levels can combine with a range of appropriate object-level theories, so that, as computation proceeds, the meta-level theory may change the object-level theory that is under its supervision.

Meta *BlocksWorldSupervisor*

Types

$ConfigName$

Functions

$s : ConfigName \to ConfigName$

$c_0 : ConfigName$

Observation Predicates

$current : ConfigName$

$holds :$ formula $\times\, ConfigName$

$constraint :$ constraintName

$evolve :$ groundAtoms × groundAtoms × constraintNames × constraintNames × actionNames × actionNames × $ConfigName$

Constraints

$$BWEC \stackrel{dfn}{=}$$
$$\forall c_1, c_2 : ConfigName \cdot current(c_1) \wedge current(c_2) \Rightarrow (c_1 = c_2) \quad \wedge$$
$$\forall\, \delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, \delta^{+\prime}_{\mathcal{D}}, \delta^{-\prime}_{\mathcal{D}} : \text{GROUNDATOMS},$$
$$\delta^+_{\mathcal{C}}, \delta^-_{\mathcal{C}}, \delta^{+\prime}_{\mathcal{C}}, \delta^{-\prime}_{\mathcal{C}} : \text{CONSTRAINTNAMES},$$
$$\delta^+_{\mathcal{A}}, \delta^-_{\mathcal{A}}, \delta^{+\prime}_{\mathcal{A}}, \delta^{-\prime}_{\mathcal{A}} : \text{ACTIONNAMES},$$
$$c : ConfigName \cdot$$
$$(evolve(\delta^+_{\mathcal{D}}, \delta^-_{\mathcal{D}}, \delta^+_{\mathcal{C}}, \delta^-_{\mathcal{C}}, \delta^+_{\mathcal{A}}, \delta^-_{\mathcal{A}}, c) \wedge evolve(\delta^{+\prime}_{\mathcal{D}}, \delta^{-\prime}_{\mathcal{D}}, \delta^{+\prime}_{\mathcal{C}}, \delta^{-\prime}_{\mathcal{C}}, \delta^{+\prime}_{\mathcal{A}}, \delta^{-\prime}_{\mathcal{A}}, c)) \Rightarrow$$
$$(\delta^+_{\mathcal{D}} = \delta^{+\prime}_{\mathcal{D}}) \wedge (\delta^-_{\mathcal{D}} = \delta^{-\prime}_{\mathcal{D}}) \wedge (\delta^+_{\mathcal{C}} = \delta^{+\prime}_{\mathcal{C}}) \wedge (\delta^-_{\mathcal{C}} = \delta^{-\prime}_{\mathcal{C}}) \wedge (\delta^+_{\mathcal{A}} = \delta^{+\prime}_{\mathcal{A}}) \wedge (\delta^-_{\mathcal{A}} = \delta^{-\prime}_{\mathcal{A}})$$

Actions

$Observe(P : \text{FORMULAE})$	
pre	$\{current(c)\}$
add	$\{holds(p, s(c)) \mid p \in P\} \cup \{current(s(c))\}$
del	$\{current(c)\}$

$Expand(n : Int)$	
pre	$\{current(c), constraint(TableSize(T, m)), m < n\}$
add	$\{current(s(c)), holds(free(T), s(c)),$ $evolve(\{\}, \{\},$ $\{TableSize(T, n)\}, \{TableSize(T, m)\},$ $\{\}, \{\}, s(c)),$ $constraint(TableSize(T, n))\}$
del	$\{current(c), constraint(TableSize(T, m))\}$

$Contract()$	
pre	$\{current(c)), constraint(TableSize(T, m)), m > 0\}$
add	$\{current(s(c)),$ $evolve(\{\}, \{\},$ $\{TableSize(T, m-1)\}, \{TableSize(T, m)\},$ $\{\}, \{\}, s(c)),$ $constraint(TableSize(T, m-1))\}$
del	$\{current(c), constraint(TableSize(T, m))\}$

$addAction($ $name : \text{ACTIONNAME},$ $vars : \text{VARNAMES},$ $pre : \text{FORMULAE},$ $add : \text{ATOMS},$ $del : \text{ATOMS})$	
pre	$\{current(c)\}$
add	$\{evolve(\{\}, \{\}, \{\}, \{\},$ $\{[name \mapsto \langle vars, pre, add, del\rangle]\}, \{\}, s(c)),$ $current(s(c))\}$
del	$\{current(c)\}$

The *BlocksWorldSupervisor* theory has the observation predicates we have introduced above, namely *current*, *holds*, *constraint* and *evolve*, together with associated consistency constraints for *current* and *evolve*. The equality in the definition of uniqueness of the *evolve* predicate at a configuration is that of set equality. More generally, as we shall see later, the equality here is the extensional equality of state transformations, which here takes this particularly simple form. The type $ConfigName$ is introduced for the supervisor's names for blocks world configurations together with an initial value c_0 and a successor function s. The actions in this theory are those that we have already introduced.

3 Introducing components

So far, we have introduced a basic notion of configuration for describing a global view of a system at an object-level and also at a meta-level which allows us to express the monitoring and evolving of the object-level system.

In this section, we turn to a more structured view of computational systems, extending the framework to systems built hierarchically from components. To do so, we need to revisit the logical structure we have presented and modify it to incorporate hierarchically-structured logical theories. In particular, the notion of a *configuration* is redefined in terms of tree-structured elements. As a consequence, definitions of action revision and the meta-view relations are modified. Because we consider this as an extension of the ideas of the previous section, we retain the terminology of configurations, meta-views, etc. but redefine them in this new context.

We motivate the development through an example. We consider various examples of buffers, i.e. finite storage devices operating on a FIFO principle (First-in, First-out), with operations of 'Send' which removes an element (the first-in) from the buffer and 'Receive' which adds an element to the buffer. Later in this section, we consider examples of buffers with evolutionary capabilities.

3.1 Buffers as components

At a basic level, we treat a component as a collection of predicates, named constraint formulae and actions, just as we did for the logical theory of the blocks world. As an example, we present a specification of a theory for a FIFO buffer of fixed but unknown capacity.

Buffer$(N : Int)$

OBSERVATION PREDICATES

$content : Value\text{-}list$

ABSTRACTION PREDICATES

$free$

CONSTRAINTS

$$Uniqueness \stackrel{dfn}{=} \forall l_1, l_2 : Value\text{-}list \cdot content(l_1) \wedge content(l_2) \Rightarrow l_1 = l_2$$

$$Size(M : Int \text{ INITIALLY } N) \stackrel{dfn}{=} (\exists l : Value\text{-}list \cdot content(l) \wedge (|l| < M)) \Leftrightarrow free \wedge \forall l : Value\text{-}list \cdot content(l) \Rightarrow (|l| \leq M)$$

ACTIONS

$Send(v)$	
pre	$\{content(l :: v)\}$
add	$\{content(l)\}$
del	$\{content(l :: v)\}$

$Receive(v)$	
pre	$\{free, content(l)\}$
add	$\{content(v :: l)\}$
del	$\{content(l)\}$

The *Buffer* component is parameterized by an integer N which represents the initial capacity of an instance of a *Buffer* theory. This capacity may be changed by a buffer supervisor, as we shall. The schema presents two actions $Send(v)$ and $Receive(v)$, together with two predicates, *content* and $free$. The formula $content(l)$ means that the content of the buffer is the list of values l, and $free$ means that the buffer can accept more input. The predicate *content* is treated as an observation and thus can be present in a state of the buffer, whereas $free$ is an abstraction of a buffer state. Two constraints are specified: the first characterizing uniqueness of the buffer contents; the second characterizing the freeness and the capacity of the buffer. In this constraint, we set an initial value (using 'INITIALLY') for the capacity so that each instance of this theory has an initial buffer size.

Our aim is to use component schema to create instances of components within other components. For example, below we specify a schema *No_Buffering* that introduces no new predicates or constraints, but instead specifies how a new component is built from two instances of buffer subcomponents. Incorporating subcomponents in this way means that we need to modify the notion of a configuration, which we do in the next subsection.

No_Buffering

COMPONENTS

$B_1, B_2 : Buffer(2)$

ACTIONS

$$Send(v) \stackrel{dfn}{=} B_2.Send(v)$$

$$Receive(v) \stackrel{dfn}{=} B_1.Receive(v)$$

The *No_Buffering* schema is also specified to have two actions, $Send(v)$ and $Receive(v)$, however, in this case they have been identified, respectively, as the send action of the buffer B_2 and the receive action of B_1. The actions of the subcomponents of *No_Buffering* are named using the component instance names B_1 and B_2. At the moment, these are the *only* actions that an instance of *No_Buffering* may undertake, and thus there is no communication between the two component buffers. As it stands, the *No_Buffering* schema is not very useful! Let us introduce internal communication between the two buffers. We specify this as a *joint* action, an action that will require $B_1.Send(v)$ to be undertaken synchronously with a $B_2.Receive(v)$ action. For simplicity, we will name the new joint action $Internal$. It is described as a choice over all values $v \in Value$. each time a particular value is transferred, that element of the choice is invoked.

Buffering
COMPONENTS
$\quad B_1, B_2 : Buffer(2)$
ACTIONS
$\quad Send(v) \stackrel{dfn}{=} B_2.Send(v)$
$\quad Receive(v) \stackrel{dfn}{=} B_1.Receive(v)$
$\quad Internal \stackrel{dfn}{=} \vert_{v \in Value}\ B_1.Send(v) \vert\vert B_2.Receive(v)$

An instance of *Buffering* acts like a buffer, in the sense that it can receive values, retain them, and send them out. The action named $Internal$ will simultaneously revise the configuration associated with the *Buffering* instance by $B_1.Send(v)$ and $B_2.Receive(v)$ for any value v. We will give the revision semantics for joint and choice actions later. This configuration contains associated sub-configurations for the *Buffer* subcomponents B_1 and B_2, and hence the $B_1.Send(v)$ action revises the B_1 sub-configuration, similarly for B_2. Component hierarchies require a more complex configuration structure than the flat, global configuration structure that we have used so far. Moreover, evolutionary steps for systems built hierarchically from components may involve the reconfiguration of components, such as changing existing components, adding new components or reconfiguring the system hierarchy. Thus revision actions no longer operate simply on sets of formulae, but in addition on the tree structure of the components. We begin to develop this formally in the next section.

As another example, suppose the above component schema is modified to contain a local variable through which the subcomponents B_1 and B_2 may pass a single value. This is sometimes called a 'latch'. To describe this, we can either specify a separate latch component, embedded and connected appropriately with the other buffer subcomponents, or we can specify the latch actions directly within the component schema as below.

BufferingLatch
OBSERVATION PREDICATES
$\quad transfer : Value,$
$\quad ready$
COMPONENTS
$\quad B_1, B_2 : Buffer(2)$
CONSTRAINTS
$\quad Uniqueness \stackrel{dfn}{=} \forall v_1, v_2 : Value \cdot transfer(v_1) \wedge transfer(v_2) \Rightarrow v_1 = v_2$
ACTIONS
$\quad Send(v) \stackrel{dfn}{=} B_2.Send(v)$
$\quad Receive(v) \stackrel{dfn}{=} B_1.Receive(v)$
$\quad Internal \stackrel{dfn}{=} \vert_{v \in Value}\ B_1.Send(v) \vert\vert TransferIn(v) \mid TransferOut(v) \vert\vert B_2.Receive(v)$
$\quad\quad$ WHERE

$TransferOut(v)$	
pre	$\{transfer(v)\}$
add	$\{ready\}$
del	$\{transfer(v)\}$

$TransferIn(v)$	
pre	$\{ready\}$
add	$\{transfer(v)\}$
del	$\{ready\}$

The actions of *BufferingLatch* are either a B_2 send action, a B_1 receive action, or an internal action of two kinds (separated by the disjunction '|'), either a B_1 send action in synchrony with a $TransferIn$ action, or a B_2 receive action in synchrony with a $TransferOut$ action. Note that the $TransferIn(v)$ and $TransferOut(v)$ actions can be performed only in these synchronous combinations.

Through these examples, we have presented the essence of how we adapt the previous logical descriptions to systems built from components, using the notion of joint action to allow communication between components. Now we begin the formal treatment of these ideas, starting with a revised notion of configuration which incorporates component hierarchies.

3.2 A configuration structure for components

Previously, in the blocks world, we defined a configuration as having three elements: a global state (a set of formulae), a set of constraints and a collection of action definitions. Such a global configuration is no longer appropriate for hierarchically (and potentially dynamically) structured systems. There are two possible approaches to describing configurations for hierarchical systems. Each uses a global environment that maintains the definitions of component schema, together with either

1. a global state containing the observations of all the components — the component hierarchy is present in the naming scheme for the predicates, prefixing them by the path amongst the components, or

2. a hierarchical structure, using a tree of local states, one for each component in the hierarchy; likewise for constraints and actions, each localised at nodes of a tree.

It turns out that it is easier to maintain the separation and linkage of the meta-levels and the object-levels using the second of these two (equivalent) representations.

We thus define the key notion of a *configuration* for a component system. This replaces the previous definition of a configuration for systems not built in terms of components. Each element of a configuration for a component system is a tree (or forest) representing the hierarchical nature of component systems.

Definition 3.1 *A* configuration *for a component system is a triple of the form:*

$$Configuration \;\triangleq\; \mathit{ObservationState} \times \mathit{ComponentMap} \times \mathit{SchemaDefs}$$

We define below the constituents of a configuration, that is $\mathit{ObservationState}$, $\mathit{ComponentMap}$ and $\mathit{SchemaDefs}$ for component systems.

To ease the presentation, we omit types (e.g. of variables) where possible. The following basic types are used:

- sets of identifiers - e.g. $\mathit{SchemaID}$, $\mathit{ComponentID}$, $\mathit{ConstraintID}$, $\mathit{ActionID}$, etc.
- $Predicates$ — the set of sets of predicates;
- $Formula$ — the set of formulae of the relevant logic;
- $Formulae$ — the set of subsets of formulae of the relevant logic;
- $ObsFormula$ — the set of atomic formulae built from observation predicates of the logic;
- $\mathit{GroundAtom}$ — the subset of $ObsFormula$ with no free variables;
- $Args$ — the set of lists of formal arguments;
- $Terms$ — the set of lists of terms, each term built from functions and variables in accordance with arities and types.

We now give a formal description of the schema used to define instances of components. Here $A \rightharpoonup B$ denotes the set of partial functions from A to B.

Definition 3.2

$$
\begin{array}{lcl}
\mathit{SchemaDefs} & \triangleq & \mathit{SchemaID} \rightharpoonup \mathit{ComponentSchema} \\
\mathit{ComponentSchema} & \triangleq & \mathit{Args} \times \mathit{Types} \times \mathit{Functions} \times \\
 & & \mathit{Predicates} \times \mathit{Predicates} \times \\
 & & \mathit{ComponentSchemaMap} \times \\
 & & \mathit{ConstraintSchemaDefs} \times \\
 & & \mathit{ActionDefs} \\
\mathit{ComponentSchemaMap} & \triangleq & \mathit{ComponentID} \rightharpoonup \mathit{ComponentSchemaInstance} \\
\mathit{ComponentSchemaInstance} & \triangleq & \mathit{SchemaID} \times \mathit{Terms} \mid \\
 & & \mathit{ComponentID} \times \mathit{SchemaID} \times \mathit{Terms} \times \\
 & & \qquad \mathit{ComponentID} \times \mathit{ComponentSchemaInstance} \\
\mathit{ConstraintSchemaDefs} & \triangleq & \mathit{ConstraintSchema}^* \\
\mathit{ConstraintSchema} & \triangleq & \mathit{ConstraintID} \times \mathit{Args} \times \mathit{Terms} \times \mathit{Formula} \\
\mathit{ActionDefs} & \triangleq & \mathit{ActionDef}^* \\
\mathit{ActionDef} & \triangleq & \mathit{ActionID} \times \mathit{Args} \times \mathit{ActionBody} \\
\mathit{ActionBody} & \triangleq & \mathit{BasicAction} \mid \mathit{PairedActions} \mid \mathit{JointActions} \mid \mathit{ChoiceActions} \\
\mathit{BasicAction} & \triangleq & \mathit{Pre\text{-}set} \times \mathit{Add\text{-}set} \times \mathit{Del\text{-}set} \\
\mathit{Pre\text{-}set} & \triangleq & 2^{Formula} \\
\mathit{Add\text{-}set} & \triangleq & 2^{ObsFormula} \\
\mathit{Del\text{-}set} & \triangleq & 2^{ObsFormula} \\
\mathit{ActionName} & \triangleq & \mathit{ComponentIDs} \times \mathit{ActionID} \\
\mathit{ComponentIDs} & \triangleq & \mathit{ComponentID}^* \\
\mathit{Action} & \triangleq & \mathit{ActionName} \times \mathit{Terms} \\
\mathit{PairedActions} & \triangleq & \mathit{M\_Action} \mid \mathit{MO\_Action} \\
\mathit{M\_Action} & \triangleq & \mathit{Action} \\
\mathit{MO\_Action} & \triangleq & \mathit{Action} \times \mathit{Action} \\
\mathit{JointActions} & \triangleq & \mathit{Action}^n, n \geq 2 \\
\mathit{ChoiceActions} & \triangleq & (\mathit{Action} \mid \mathit{PairedActions} \mid \mathit{JointActions})^n, \;\; n \geq 2
\end{array}
$$

In Definition 3.2 above, we define *SchemaDefs*, which form part of a configuration. Schema definitions are described in terms of *ComponentSchema* which capture the formal structure of the specifications of components that we have presented in this section, consisting of a list of formal argument names, two sets of predicates, possible subcomponents (given by a mapping of type *ComponentSchemaMap* from component identifiers to their associated schema identifiers and actual arguments, or to a schema for a supervised component), a set of constraint schema definitions and a set of action definitions. This structure takes into account supervised component pairings, which remain to be discussed (Section 3.3). Actions are of various forms including joint actions for the synchronous combination of actions, and choice actions which represent a (possibly infinite) disjunctive choice of actions.

In Definition 3.3 below, we define *ComponentMap* which is the second element of a configuration and consists of a labelled hierarchy of components. For a basic component, i.e. one given without an associated supervisor, the structure records the component identifier, the schema identifier and the actual arguments applied to create the instance, together with the component constraints (derived from the relevant schema definition) and any subcomponent instances. For an instance of a supervised component, consisting of a supervisor and a supervisee, the structure records the identifier and component instance of the supervisor, together with the identifier and component instance of its supervisee.

Definition 3.3

$$
\begin{array}{lcl}
ComponentMap & \triangleq & ComponentID \rightharpoonup ComponentInstance \\
ComponentInstance & \triangleq & BasicComponent \mid SupervisedComponent \\
BasicComponent & \triangleq & SchemaID \times Terms \times \\
 & & Constraints \times \\
 & & ComponentMap \\
SupervisedComponent & \triangleq & ComponentID \times BasicComponent \times \\
 & & ComponentID \times ComponentInstance \\
Constraint & \triangleq & ConstraintID \times Terms \\
Constraints & \triangleq & Constraint^*
\end{array}
$$

Finally, in Definition 3.4 below, we define *ObservationState* – the first element of a configuration – as a labelled hierarchy built from sets of ground atomic formulae. A node in this hierarchy is a state of either a basic component or a supervised component: for the former, the state is a set of ground atomic formulae (the local state) together with states of each of the subcomponents; for the latter, the state is a pair, the first element being the state of the supervisor, and the second being the state for the supervisee.

Definition 3.4

$$
\begin{array}{lcl}
LocalState & \triangleq & 2^{GroundAtom} \\
ComponentState & \triangleq & O\_State \mid MO\_State \\
O\_State & \triangleq & LocalState \times ObservationState \\
MO\_State & \triangleq & ComponentID \times ComponentState \times \\
 & & ComponentID \times ComponentState \\
ObservationState & \triangleq & ComponentID \rightharpoonup ComponentState
\end{array}
$$

Finite trees in the recursively-defined set of trees *ComponentState* are built from basic components which have no subcomponents, i.e. leaves of the tree correspond to the *ObservationState* being the empty partial function.

A well-formedness condition on configurations is required:

Definition 3.5 (Well-formedness of configurations) *Let* $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$ *be a configuration.* Γ *is a well-formed configuration if and only if*

- Δ *and* Θ *have the same component identifier labelling, a labelling which is in accord with the schema definitions* Σ*;*
- *constraint schema in* Σ *are instantiated as constraints in* Θ*;*
- *predicate names appearing in local state descriptions within* Δ*, for any (sub)component of* Θ *are present in the associated schema definitions.*

The definition of schemas above (Definition 3.2) describes four forms of action definitions. We here consider only revision by a basic action definitions. We discuss and define paired action revision in Section 3.3, then consider joint and choice actions in the section after that. Revision operates on trees rather than sets. To define this, we introduce the following functions:

> $getState : ComponentIDs \times ObservationState \rightarrow ComponentState$ extracts the component state for a (sub)component located by the path given as the first argument from the state given as second argument;
>
> $update : ComponentIDs \times ObservationState \times ComponentState \rightarrow ObservationState$ updates the component state for the (sub)component located by the path given as the first argument in the state given as second argument by the component state given as the third argument;
>
> $getActionBody : ActionID \times ComponentIDs \times Configuration$ extracts the action body whose action identifier is the first argument from the component instance located by the path given as the second argument from the configuration given by the third argument. For action body β, name the projections pre-β, add-β and del-β for the precondition, add and del sets for body β.

We also introduce the 'flattening' operations on tree-structured states, yielding a set of path-prefixed ground atomic formulae:

$$\downarrow : ObservationState \rightarrow LocalState$$
$$\downarrow \Delta \stackrel{dfn}{=} \bigcup\{c.(\downarrow \Delta(c)) \mid c \in \mathbf{dom}(\Delta)\}$$

Component states are of two forms, object-level states O_State, and a combination of a meta-level and object-level state, MO_State. We define the flattening of these as follows:

$$\begin{array}{l} \downarrow : ComponentState \rightarrow LocalState \\ \downarrow \langle \sigma, \mu \rangle \stackrel{dfn}{=} \sigma \cup \downarrow \mu \quad \text{for the } O\_State \text{ case,} \\ \downarrow \langle c_1, \sigma_1, c_2, \sigma_2 \rangle \stackrel{dfn}{=} c_1.(\downarrow \sigma_1) \cup c_2.(\downarrow \sigma_2) \quad \text{for the } MO\_State \text{ case.} \end{array}$$

Definition 3.6 (Local Basic Action Revision) *Let*

> $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$ *be a well-formed configuration,*
>
> $\delta = [c \mapsto \langle \sigma, \mu \rangle]$ *be a component state of type O_State within Δ for a component instance named by the path $p.c$ of component schema sid,*
>
> $\alpha = \langle aid, \bar{t} \rangle$ *be an applied basic action of component instance located at $p.c$ of schema sid, and* $\beta = getActionBody(aid, p.c, \Gamma)$ *be the action body associated with α.*

Consider a binding $[\bar{y} \mapsto \bar{u}]$ such that

1. $\downarrow \delta \models_{\Gamma_{p.c}} \bigwedge \text{pre-}\beta[\bar{t}/\bar{x}][\bar{u}/\bar{y}]$*, and*
2. $\delta' = [c \mapsto \langle \sigma', \mu \rangle]$ *is consistent for Γ, where* $\sigma' = (\sigma \cup \text{add-}\beta[\bar{t}/\bar{x}][\bar{u}/\bar{y}]) \setminus \text{del-}\beta[\bar{t}/\bar{x}][\bar{u}/\bar{y}]$.

Then we say δ' is a revision of δ by α.

Definition 3.7 (Basic Action State Revision) *Let $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$ be a well-formed configuration, $a = \langle p.c, aid \rangle$ a well-formed action name denoting a basic action of the component located in Θ by $p.c$ and $\alpha = \langle a, \bar{t} \rangle$ a well-formed application to ground terms. Consider $\Delta' = update(p.c, \Delta, \delta')$ where δ' is a revision of $getState(p.c, \Delta)$ by $\langle aid, \bar{t} \rangle$ and Δ' is consistent for Γ. We say that Δ' is a revision of Δ by α, and we write $\Delta \xrightarrow{\alpha} \Delta'$.*

Extend basic action revision to configurations as follows: If $\Delta \xrightarrow{\alpha} \Delta'$, then we write $\langle \Delta, \Theta, \Sigma \rangle \xrightarrow{\alpha} \langle \Delta', \Theta, \Sigma \rangle$.

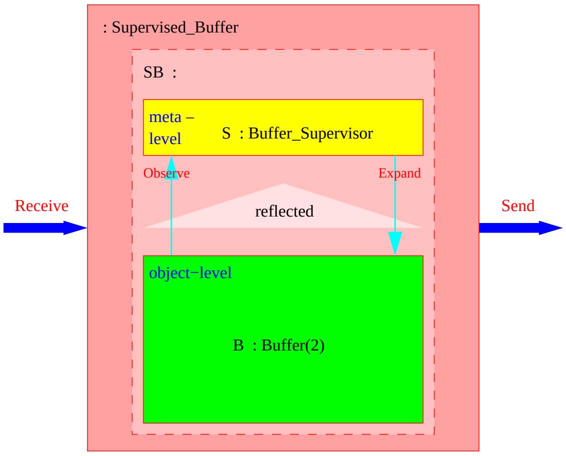
3.3 Evolvable components

The configuration structure just outlined is fairly complex, combining, as it does, multiple logical theories in a tree structure and incorporating a naming mechanism for locating elements in the tree. In the previous blocks world, we had a global object-level system and a related meta-level system. This separation of the two systems is no longer appropriate as we wish to couple together supervisors and their supervisees to form a supervised component (using 'evolver/producer' pairs in the Warboys architectural sense [12]). Supervised components may then be combined, possibly with additional supervisors. In all, a system consists of a hierarchy of components, some of which may be supervised and each component is either a basic component or consists of a hierarchy of the same kind.

Supervisors are components which stand in a suitable relationship to the object-level system with which it is paired. The ability of a supervisor not only to change the basic internal structure of a component, as with the blocks world example, but to add new components and to effect entire reconfigurations of the supervised system means that we have to reconsider the issue of how supervisors are related to object-level systems. Notice that whatever object-level system is provided initially, as computation progresses through supervisor actions, the supervisor is at a meta-level to a changing object-level, and what this object-level is at any instance depends on the history of the supervisory actions. This means that, although we may provide an initial object-level system, whenever a supervisory action is invoked, it needs to access the current structure of the object-level system. This access is provided through two mechanisms – one is the meta-level access to the object-level logical structure – the predicates, formulae etc, the other mechanism records in the meta-level state the changes made to the object-level. We have seen both of these in the blocks world example, but now the changes can affect the whole subcomponent hierarchy and so additional predicates are required to record the effect of these changes.

The ability of the supervisor to make considerable structural changes to the object-level system also raises issues of naming and description. In describing a component system, the natural way that a component can access another is through a component name. However, in the case of a supervisor's access to the object-level, named components may be created and destroyed and, an any stage, the names of the existing components depend upon the history of supervisory actions. Moreover, this evolving landscape of components suggests that the idea of a schema, as introduced above for components of a fixed structure, may not be as relevant for supervised structures.

We begin with an example, that of an evolvable buffer. In this case, the evolutionary actions adjust the buffer's capacity according to usage. The picture below outlines the main component structure, pairing an instance of a *Buffer_Supervisor* component (whose schema is described below), with a component that is initially (and thereafter too) an instance of the *Buffer* schema (from Section 3.1) to create a schema for supervised buffers.



We now describe a schema for such a buffer supervisor presented in a way similar to that of the blocks world supervisor of the previous section.

Meta *Buffer_Supervisor*

Types

$ConfigName$

Functions

$s : ConfigName \rightarrow ConfigName$
$c_0 : ConfigName$

Observation Predicates

$holds$: formula $\times$ $ConfigName$
$component$: componentMap
$evolve$: stateTransformer $\times$ componentMap $\times$ schemaDefs $\times$ $ConfigName$
$current : ConfigName$

Constraints

$$
\begin{aligned}
&Uniqueness \stackrel{dfn}{=} \\
&\quad \forall c_1, c_2 : ConfigName \cdot current(c_1) \wedge current(c_2) \Rightarrow (c_1 = c_2) \\
&\quad \forall\, \delta, \delta' : \textsc{stateTransformer}, \\
&\quad\quad \delta_M, {\delta_M}' : \textsc{componentMap}, \\
&\quad\quad \delta_S, {\delta_S}' : \textsc{schemaDefs}, \\
&\quad\quad c : ConfigName \cdot \\
&\quad\quad\quad (evolve(\delta, \delta_M, \delta_S, c) \wedge evolve(\delta', {\delta_M}', {\delta_S}', c)) \Rightarrow \\
&\quad\quad\quad\quad ((\delta = \delta') \wedge (\delta_M = {\delta_M}') \wedge (\delta_S = {\delta_S}'))
\end{aligned}
$$

Actions

$Observe(Q$: formulae$)$	
pre	$\{current(c)\}$
add	$\{holds(q, s(c)) \mid q \in Q)\} \cup \{current(s(c))\}$
del	$\{current(c)\}$

$Expand$	
pre	$\{current(c),$ $component([B \mapsto bc]), bc = \langle sid, ts, cs, cm\rangle,$ $Size(m) \in cs\}$
add	$\{component([B \mapsto bc[(cs \cup \{Size(2 * m)\} \setminus \{Size(m)\})/cs]]),$ $evolve(\lambda\Delta \cdot \Delta, [B \mapsto bc[(cs \cup \{Size(2 * m)\} \setminus \{Size(m)\})/cs]], [], s(c)),$ $current(s(c))\}$
del	$\{component([B \mapsto bc]), current(c)\}$

The two actions specified are an *Observe* and an *Expand* action. The predicate $component(x)$, for a component map x, reflects in the meta-level state the fact that the component map x is a part of the component instance map of the object-level configuration (in this case for the buffer component). Thus, in the precondition of the *Expand* action, the formulae $component(\{[B \mapsto bc]\})$, $bc = \langle sid, ts, cs, cm\rangle$ and $Size(m) \in cs$ hold if there are appropriate bindings for the variables such that $Size(m)$ with the variable m appropriately substituted, is a constraint of the component instance named by B. A configuration for an instance B of the *Buffer* schema might, for example, have a component instance map

$$\{[B \mapsto \langle Buffer, \langle 2\rangle, \langle(Uniqueness, \langle\rangle), (Size, \langle 4\rangle)\rangle, []\rangle]\}$$

so that B is of schema type *Buffer* instantiated with argument value 2, containing constraints named $Uniqueness$ and $Size(4)$, and has no subcomponent instances.

The predicate *evolve* has the same purpose as that in the blocks world supervisor although its arguments have changed to reflect the more complex configuration structure. The first argument is a transformation of object-level states, i.e. it defines the revision process for the object-level tree-structured state. Thus state transformers are functions from *ObservationState* to *ObservationState*. The second and third arguments provide revisions, respectively, to the component instance map and the schema definition map. The final argument is the meta-level name for the current object-level configuration. These changes in meta-level predicates and component configurations require new definitions of the two meta-view relations.

Definition 3.8 (State meta-view — for components) *Let W^M and W be the typed first-order theories for meta-level and object-level systems respectively. We say that $\Delta^{\mathtt{M}}$ (from a configuration $\Gamma^{\mathtt{M}}$ of W^M) is a* state meta-view *of a configuration $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$ of theory W when, for any valid non-empty path of basic component identifiers p in $\Delta^{\mathtt{M}}$*

- *for all formulae φ and configuration names c, if $p.\{current(c), holds(\varphi, c)\} \subseteq \downarrow \Delta^{\mathtt{M}}$, then φ is a formula of W and $\downarrow \Delta \models_W \varphi$;*
- *for all component instance maps θ, if $p.component(\theta) \in \downarrow \Delta^{\mathtt{M}}$, then $\theta \subseteq \Theta$;*
- *for all schema definition maps σ, if $p.schema(\sigma) \in \downarrow \Delta^{\mathtt{M}}$, then $\sigma \subseteq \Sigma$.*

When this holds, we say that Γ^M is a meta-configuration *for Γ.*

As components may be hierarchically structured, with supervisor/supervisee pairs themselves as components, we extend the above definition to such situations. We define the notion of *state meta-consistency* for components.

Definition 3.9 (State meta-consistency — for components) *A well-formed component configuration $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$ is said to be* state meta-consistent *if the component map Θ refers*

- *to a basic component containing no subcomponents;*
- *to a basic component and the extracted configurations for each subcomponent are state meta-consistent;*
- *to a supervised component (supervisor/supervisee pair) such that the supervisor configuration is a meta-configuration for the supervisee configuration, and both the supervisor and supervisee configurations are each state meta-consistent.*

Definition 3.10 (Transition meta-view — for components) *Given meta-level configurations, $\Gamma^{\mathtt{M}} = \langle \Delta^{\mathtt{M}}, \Theta^{\mathtt{M}}, \Sigma^{\mathtt{M}} \rangle$ and $\Gamma^{\mathtt{M}\prime} = \langle \Delta^{\mathtt{M}\prime}, \Theta^{\mathtt{M}\prime}, \Sigma^{\mathtt{M}\prime} \rangle$ in theory W^M, and object-level configurations, $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$ and $\Gamma' = \langle \Delta', \Theta', \Sigma' \rangle$ of theory W, such that $\Delta^{\mathtt{M}}$, $\Delta^{\mathtt{M}\prime}$ are, respectively, state meta-views of Γ, Γ', we say that the pair $\langle \Delta^{\mathtt{M}}, \Delta^{\mathtt{M}\prime} \rangle$ is a* transition meta-view *of $\langle \Gamma, \Gamma' \rangle$ when*

> *for any valid non-empty path of basic component identifiers p in $\Delta^{\mathtt{M}}$, if*
>
> $$p.\{evolve(\delta, \theta, \sigma, c), current(c)\} \subseteq \downarrow \Delta^{\mathtt{M}\prime}$$
>
> *and $\Delta' = \delta(\Delta)$ is theory W' consistent, where W' is the theory W with component instance map Θ updated to $\Theta' = \Theta \dagger \theta$ and component schema definitions Σ updated to $\Sigma' = \Sigma \dagger \sigma$,*
>
> *then $\Gamma' = \langle \Delta', \Theta', \Sigma' \rangle$.*

When this holds, we say that the configuration pair $\langle \Gamma^{\mathtt{M}}, \Gamma^{\mathtt{M}\prime} \rangle$ is a transition meta-configuration pair *for $\langle \Gamma, \Gamma' \rangle$.*

Let us now define a theory, *Supervised_Buffer*, as a paired component built from an instance of *Buffer_Supervisor* which is meta to an instance of a *Buffer*(2) theory.

Supervised_Buffer

COMPONENTS

$SB : (S : \mathit{Buffer\_Supervisor}$ META TO $B : \mathit{Buffer}(2))$

ACTIONS

$$Send(v) \stackrel{dfn}{=} |_{Q \in Formulae}\ SB.\langle S.Observe(Q), B.Send(v) \rangle$$
$$Receive(v) \stackrel{dfn}{=} |_{Q \in Formulae}\ SB.\langle S.Observe(Q), B.Receive(v) \rangle$$
$$Internal \stackrel{dfn}{=} SB.\langle S.Expand, \rangle$$

The infix operator 'META TO' creates a pair of an object-level system and a system at a meta-level to it. In this example, the operator creates a pair of a supervisor instance named S of schema type

Buffer_Supervisor and its supervisee instance named B of schema type *Buffer*(2). The enclosing component schema *Supervised_Buffer* has actions *Send*, *Receive* and *Internal*. The *Send* and *Receive* actions are the paired actions of an *Observe* action of the S instance of *Buffer_Supervisor* paired with a *Send* (or *Receive*) action of the B instance of *Buffer*(2). The *Internal* action is an evolutionary *Expand* action of the supervisor inducing change in the *Buffer* instance B.

We now define composition and decomposition of configurations for such pairs.

Definition 3.11 (Supervisor-supervisee composition) *Given configurations $\Gamma_M = \langle \Delta_M, \Theta_M, \Sigma_M \rangle$ built from theory W_M and $\Gamma_O = \langle \Delta_O, \Theta_O, \Sigma_O \rangle$ built from W_O such that Δ_M is a state meta-view of Γ_O with theory W_M meta to W_O, we define the configuration pairing constructor*

$$\begin{aligned}\mathcal{MO}(mo, \Gamma_M, \Gamma_O) = \langle\, & [mo \mapsto \langle m, \Delta_M(m), o, \Delta_O(o)\rangle],\\ & [mo \mapsto \langle m, \Theta_M(m), o, \Theta_O(o)\rangle],\\ & \Sigma_M \cup \Sigma_O \rangle\end{aligned}$$

for the configuration of the component instance named mo consisting of a supervisor component instance m paired with a supervisee instance o.

Definition 3.12 (Supervisor-supervisee decomposition) *Let $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$ be a configuration for a supervised component, i.e.*

$$\begin{aligned}\Delta &= [mo \mapsto \langle m, \Delta_m, o, \Delta_o \rangle] \text{ and}\\ \Theta &= [mo \mapsto \langle m, \Theta_m, o, \Theta_o \rangle],\end{aligned}$$

for appropriate component identifiers mo, m and o, define the supervisor and supervisee configuration projections of Γ by

$$\mathcal{M}(\Gamma) = \langle [m \mapsto \Delta_m], [m \mapsto \Theta_m], \Sigma \rangle$$

$$\mathcal{O}(\Gamma) = \langle [o \mapsto \Delta_o], [o \mapsto \Theta_o], \Sigma \rangle.$$

We now give the revision semantics of these paired actions as follows.

Definition 3.13 (Local paired action revision - I) *Consider a well-formed configuration Γ for a supervised component instance mo, an action name $a = \langle \langle mo \rangle, aid \rangle$ referring to a paired action body $\beta = \langle \alpha_M, \alpha_O \rangle$ and a ground instance $\alpha = \langle a, \bar{t} \rangle$. Let*

$$\mathcal{M}(\Gamma) \xrightarrow{\alpha_M[\bar{t}/\bar{x}]} \Gamma'_M \quad \text{and} \quad \mathcal{O}(\Gamma) \xrightarrow{\alpha_O[\bar{t}/\bar{x}]} \Gamma'_O.$$

If Γ'_M is a state meta-configuration for Γ'_O, then we say $\mathcal{MO}(mo, \Gamma'_M, \Gamma'_O)$ is a revision of Γ by α, and we write $\Gamma \xrightarrow{\alpha} \mathcal{MO}(mo, \Gamma'_M, \Gamma'_O)$.

Thus, for a paired action comprising a supervisor action in synchrony with a supervisee action, a revision of a configuration by the action can be determined by simply deconstructing the configuration, revising by the separate supervisor and supervisee actions and then reconstructing the configuration. More interesting are the evolutionary actions, which use transition meta-views to define revision:

Definition 3.14 (Local paired action revision - II) *Consider a well-formed configuration Γ for a supervised component instance mo, an action name $a = \langle \langle mo \rangle, aid \rangle$ referring to a paired action body $\beta = \langle \alpha_M \rangle$ and a ground instance $\alpha = \langle a, \bar{t} \rangle$. Let*

$$\mathcal{M}(\Gamma) \xrightarrow{\alpha_M[\bar{t}/\bar{x}]} \Gamma'_M$$

and let Γ'_O be a configuration such that the pair of supervisor configurations $\langle \mathcal{M}(\Gamma), \Gamma'_M \rangle$ is a transition meta-configuration pair for the supervisee configurations $\langle \mathcal{O}(\Gamma), \Gamma'_O \rangle$, then we say $\mathcal{MO}(mo, \Gamma'_M, \Gamma'_O)$ is a revision of Γ by α, and write $\Gamma \xrightarrow{\alpha} \mathcal{MO}(mo, \Gamma'_M, \Gamma'_O)$.

To define the revision of a component configuration by a paired action within a component hierarchy, we introduce the following two functions on configuration structures:

> $getConfiguration : ComponentIDs \times Configuration \rightarrow Configuration$ extracts a configuration for the component instance located by the path given as the first argument within the configuration which is the second argument.

$updateConfiguration : ComponentIDs \times Configuration \times Configuration \rightarrow Configuration$ updates the third argument configuration by replacing the configuration located by path given as the first argument with the configuration given by the second argument.

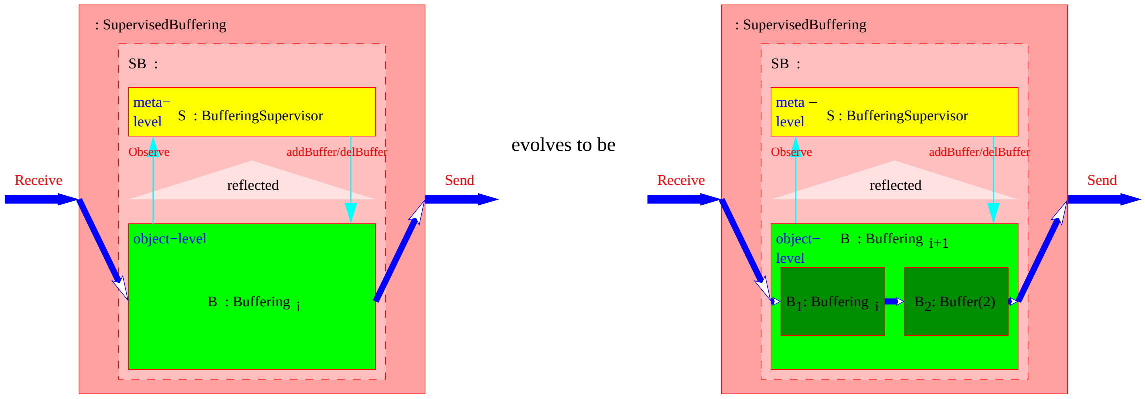
Definition 3.15 (Paired action revision) *Given a well-formed configuration* Γ*, an action name* $a = \langle cs, aid\rangle$ *referring to a paired action body and a ground instance* $\alpha = \langle a, \bar{t}\rangle$*. Let* $\alpha' = \langle\langle last(cs), aid\rangle, \bar{t}\rangle$ *and suppose*

$$getConfiguration(cs, \Gamma) \xrightarrow{\alpha'} \Gamma'_{cs}$$

with $\Gamma' = updateConfiguration(cs, \Gamma'_{cs}, \Gamma)$ *consistent, then* Γ' *is a revision of* Γ *by* α*, and we write* $\Gamma \xrightarrow{\alpha} \Gamma'$*.*

3.4 Adding components to a system

As we have indicated, there are many different ways in which a system may evolve. In the blocks world, we demonstrated evolution via changing theory constraints and by adding actions. Now that we are in the world of component hierarchies, we are able to consider evolutions of a system which not only modify existing components, as in the case of buffer expansion in the previous section, but also add new components into the system. More generally, an evolution may be an entire reconfiguration of a system of components. For example, an alternative way to expand a buffer, and perhaps a more natural one in the component world, is to create a new buffer component and couple it to the existing buffer in such a way that the composite appears like a buffer. In pictorial terms, we want a supervisor to achieve the following structural change.



There are several subtle points involved in this form of evolution, though at first sight it appears rather straightforward. Firstly, before the evolutionary change the supervisor instance S oversees a component named B. Indeed B is the handle that S has for the component instance — the name was passed to S on the creation of the supervisor-supervisee pair. It is a principle of this approach that a component does not evolve itself, thus the name of the component that S oversees must remain B although the type of the component B, i.e. its schema, may change — indeed this is usually the case. If the name of the component were to change then such a change would need to be induced by a supervisor meta to the evolvable buffer. Here the buffering component name B is kept the same but its schema changes. The schema definition for the evolved component must already exist at the time of evolution, or be created at that time. We also want an *addBuffer* supervisor action that can be repeatedly applied, i.e. it should be generic. In the picture, we assume an indexed family of schema names $Buffering_i$, and indicate that the evolved buffer has schema identifier $Buffering_{i+1}$. This new component schema has two subcomponents, the previous buffering instance (renamed) of schema type $Buffering_i$ and a basic *Buffer* component. The subcomponents must be appropriately linked, i.e. the new schema must specify the internal joint action, effecting communication between the $Buffering_i$ and *Buffer* components, as well as appropriate external actions. Firstly, we present the object-level schema for the case when $i = 1$, then we show how a $\mathit{Buffering}_{i+1}$ schema can be generated from $\mathit{Buffering}_i$ using a supervisor.

$\mathit{Buffering}_1(N : \mathit{Int})$

ABSTRACTION PREDICATES

$free$

COMPONENTS

$B1 : \mathit{Buffer}(N)$

CONSTRAINTS

$freeDef \overset{dfn}{=} free \Leftrightarrow B1.free$

ACTIONS

$\mathit{Send}(v : Value) \overset{dfn}{=} B1.\mathit{Send}(v)$

$\mathit{Receive}(v : Value) \overset{dfn}{=} B1.\mathit{Receive}(v)$

$Internal \overset{dfn}{=} \textsc{identity}$

Thus a $\mathit{Buffering}_1$ component is formed directly from a $\mathit{Buffer}(N)$ subcomponent instance $B1$. The predicate $free$ is defined to be $B1$'s $free$ predicate. The *Send* and *Receive* actions are those of $B1$. For the demonstration here it has not been necessary to introduce other predicates, such as *content*. However, should some future action require such knowledge, it can easily be incorporated.

We now show how to define a supervisor action that achieves the desired goal of extending a buffer with an additional buffer of fixed capacity.

In the following, to simplify the presentation, we have introduced some additional notation. To help distinguish between meta-level variables and object-level ones, we have used a plain roman font to refer to the object-level names. Finally, we use the notation $id\hat{\ }\mathrm{i}$ for the identifier *id* subscripted by the number i, e.g. if *id* is the name C, then $id\hat{\ }2$ is the component identifier C_2. We allow arithmetic in labels as in $C\hat{\ }i+1$, for $i = 2$, yields C_3.

Not only does the supervisor need to create the new extended component and name its subcomponents, but also it needs to create a schema for the extended component from the schema of the unextended component. To do this, it accesses the object-level schema using the predicate *schema* to retrieve the definition for $\mathit{Buffering}_i$ that must be present in the object-level component system for this action to be defined. The supervisor's state is revised through the inclusion of

the new current configuration name for the object-level component,

an *evolve* predicate detailing the object-level schema addition, and

a new *schema* predicate instance.

The schema for $\mathit{Buffering}_{i+1}$ is created with two subcomponents, the first is the existing $\mathit{Buffering}_i$ schema and the second is the new *Buffer*. It is endowed with the constraint named $freeDef$ that defines the abstraction $free$ of the new schema to be the object-level formula $C_1.free \vee C_2.free$ where C_1 and C_2 are the subcomponent identifiers. The actions of the new schema are

linking the new *Send* action to that of the C_2 instance of *Buffer*,

linking the new *Receive* action to that of the C_1 instance of $\mathit{Buffering}_i$, and

connecting the *Send* action of $\mathit{Buffering}_i$ with the *Receive* action of *Buffer* and making it one of the choices of $\mathit{Buffering}_{i+1}$'s $Internal$ action, the other choice being $\mathit{Buffering}_i$'s own $Internal$ actions.

This then is the definition of the supervisor's action:

$$
\begin{array}{l|l}
\multicolumn{2}{l}{\mathit{addBuffer}(N : \mathit{Int})} \\
\hline
\text{pre} & \{\mathit{current}(c), \\
& \;\; \mathit{component}([C \mapsto \mathit{ci} \text{ as } \langle \text{Buffering\^{}i}, \mathit{ts}, \mathit{cs}, \mathit{scm} \rangle]), \\
& \;\; \mathit{schema}([\text{Buffering\^{}i} \mapsto \langle \mathit{vs}, \mathit{ps}, \mathit{cm}, \mathit{csd}, \mathit{asd} \rangle])\} \\
\hline
\text{add} & \{\mathit{current}(s(c)) \\
& \;\; \mathit{evolve}(\mathit{stateTransformer}, \mathit{newComponent}, \mathit{newBuffer}, s(c)), \\
& \;\; \mathit{component}(\mathit{newComponent}), \\
& \;\; \mathit{schema}(\mathit{newBuffer})\} \\
& \text{where} \\
& \quad \mathit{stateTransformer} \text{ is} \\
& \qquad \lambda\Delta \cdot [C \mapsto \langle \{\}, [\, C\text{\^{}}1 \mapsto \Delta(C), \\
& \qquad\qquad\qquad\qquad C\text{\^{}}2 \mapsto \langle \{\text{content}([])\}, [] \rangle] \rangle] \\
& \quad \mathit{newComponent} \text{ is} \\
& \qquad [C \mapsto \langle\, \text{Buffering\^{}i+1}, \mathit{ts} :: \langle N \rangle, \langle \text{freeDef}, \langle\rangle \rangle, \\
& \qquad\qquad [C\text{\^{}}1 \mapsto \mathit{ci}, \\
& \qquad\qquad\; C\text{\^{}}2 \mapsto \langle \text{Buffer}, \langle N \rangle, \langle \langle \text{Size}, \langle N \rangle \rangle, \langle \langle \text{Uniqueness} \rangle, \langle\rangle \rangle \rangle \rangle] \rangle] \\
& \quad \mathit{newBuffer} \text{ is} \\
& \qquad [\text{Buffering\^{}i+1} \mapsto \langle\;\; \mathit{vs} :: \langle \text{N} \rangle, \{\}, \{\}, \{\}, \{\text{free}\}, \\
& \qquad\qquad\qquad\qquad [C\text{\^{}}1 \mapsto \langle \text{Buffering\^{}i}, \mathit{vs} \rangle, \\
& \qquad\qquad\qquad\qquad\; C\text{\^{}}2 \mapsto \langle \text{Buffer}, \langle \text{N} \rangle], \\
& \qquad\qquad\qquad\qquad \langle \text{freeDef}, \langle\rangle, \langle\rangle, \text{free} \Leftrightarrow C\text{\^{}}1.\text{free} \vee C\text{\^{}}2.\text{free} \rangle, \\
& \qquad\qquad\qquad\qquad \mathit{actions} \rangle] \\
& \quad \text{where} \\
& \quad \mathit{actions} \text{ is} \\
& \qquad \langle\, \langle \text{Send}, \langle \text{v : Value} \rangle, \langle \langle \langle C\text{\^{}}2 \rangle, \text{Send} \rangle, \langle \text{v} \rangle \rangle \rangle, \\
& \qquad\;\; \langle \text{Receive}, \langle \text{v : Value} \rangle, \langle \langle \langle C\text{\^{}}1 \rangle, \text{Receive} \rangle, \langle \text{v} \rangle \rangle \rangle, \\
& \qquad\;\; \langle \text{Internal}, \langle\rangle, \;\; \langle \langle \langle C\text{\^{}}1 \rangle, \text{Internal} \rangle, \langle\rangle \rangle \mid \\
& \qquad\qquad\qquad (|_{\text{v} \in \text{Value}} (\langle \langle \langle C\text{\^{}}1 \rangle, \text{Send} \rangle, \langle \text{v} \rangle \rangle || \langle \langle \langle C\text{\^{}}2 \rangle, \text{Receive} \rangle, \langle \text{v} \rangle \rangle)) \rangle \rangle \\
\text{del} & \{\mathit{current}(c), \\
& \;\; \mathit{component}([C \mapsto \mathit{ci}])\}
\end{array}
$$

The precondition requires that sufficient of the component instance tree (containing actual arguments to schemas and instances of constraints, etc.) is available as meta-level observations to enable the evolve action to specify the de-construction and re-construction of the state and component instance trees. The component instance tree is rebuilt so that the top-level component is now of schema type $\mathit{Buffering}_{i+1}$, with subcomponents named C_1 of $\mathit{Buffering}_i$ schema and a C_2 instance of $\mathit{Buffer}(N)$. The latter subcomponent, i.e. the new $\mathit{Buffer}(N)$ instance, is created with capacity N and its local state is required to be empty. The local state of the $\mathit{Buffering}_i$ component, however, is preserved. We present the induced revision to the component state as a state transformer.

For completeness, we present appropriate $BufferingSupervisor$ and $SupervisedBuffering$ component schemas.

META $BufferingSupervisor$

TYPES

$ConfigName$

FUNCTIONS

$s : ConfigName \rightarrow ConfigName$
$c_0 : ConfigName$

OBSERVATION PREDICATES

$holds : \text{FORMULA} \times ConfigName$
$component : \text{COMPONENTMAP}$
$schema : \text{SCHEMADEFS}$
$evolve : \text{STATETRANSFORMER} \times \text{COMPONENTMAP} \times \text{SCHEMADEFS} \times ConfigName$
$current : ConfigName$

CONSTRAINTS

$Uniqueness \stackrel{dfn}{=}$

$\forall c_1, c_2 : ConfigName \cdot current(c_1) \wedge current(c_2) \Rightarrow (c_1 = c_2)$
$\forall cm_1, cm_2 : \text{COMPONENTMAP} \cdot component(cm_1) \wedge component(cm_2) \Rightarrow$
$\quad \neg conflict(cm_1, cm_2)$
$\forall sd_1, sd_2 : \text{SCHEMADEFS} \cdot schema(sd_1) \wedge schema(sd_2) \Rightarrow \neg conflict(sd_1, sd_2)$
$\forall \delta, \delta' : \text{STATETRANSFORMER},$
$\quad \delta_M, {\delta_M}' : \text{COMPONENTMAP},$
$\quad \delta_S, {\delta_S}' : \text{SCHEMADEFS},$
$\quad c : ConfigName \cdot$
$\qquad (evolve(\delta, \delta_M, \delta_S, c) \wedge evolve(\delta', {\delta_M}', {\delta_S}', c)) \Rightarrow$
$\qquad\quad ((\delta = \delta') \wedge (\delta_M = {\delta_M}') \wedge (\delta_S = {\delta_S}'))$

where

$$conflict(m_1, m_2 : \text{MAP}) \stackrel{dfn}{=} \exists x \cdot (x \in \mathbf{dom}\, m_1 \wedge x \in \mathbf{dom}\, m_2 \wedge m_1(x) \neq m_2(x))$$

ACTIONS

$Observe(Q : \text{FORMULAE})$	
pre	$\{current(c)\}$
add	$\{holds(q, s(c)) \mid q \in Q)\} \cup \{current(s(c))\}$
del	$\{current(c)\}$

$addBuffer(N : Int)$	
pre	$\{current(c),$ $component([C \mapsto ci$ as $\langle \text{Buffering\^{}i}, ts, cs, scm \rangle]),$ $schema([\text{Buffering\^{}i} \mapsto \langle vs, ps, cm, csd, asd \rangle])\}$
⋮	⋮

$delBuffer$	
⋮	⋮

The $addBuffer$ supervisor action is defined previously. The $delBuffer$ action may be defined similarly. Finally, we create a $SupervisedBuffering$ component schema by combining the above supervisor with a buffering component.

$$
\begin{array}{|l}
\textit{SupervisedBuffering} \\
\hline
\textsc{Components} \\
\quad SB : (S : \textit{BufferingSupervisor}\ \textsc{meta to}\ B : \textit{Buffering}_1(4)) \\
\textsc{Actions} \\
\quad Send(v) \overset{dfn}{=} |_{Q \in Formulae}\ SB.\langle S.Observe(Q), B.Send(v)\rangle \\
\quad Receive(v) \overset{dfn}{=} |_{Q \in Formulae}\ SB.\langle S.Observe(Q), B.Receive(v)\rangle \\
\quad Internal \overset{dfn}{=}\ SB.\langle S.addBuffer(2), \rangle\ | \\
\qquad\qquad\quad SB.\langle S.delBuffer, \rangle\ | \\
\qquad\qquad\quad |_{Q \in Formulae}\ SB.\langle S.Observe(Q), B.Internal\rangle
\end{array}
$$

This component is thus the pairing of a *BufferingSupervisor* component S and a $\textit{Buffering}_1$ component B together with actions defined on the paired component.

3.5 Joint and choice actions

We have introduced above joint, or shared, actions to allow communication between components. The participating actions of a joint action revise the local states of two distinct components in synchrony. We now define this revision (for two actions, the extension to more is straightforward).

Definition 3.16 (Local joint action revision) *Given a well-formed configuration Γ for a component instance c and a ground action instance $\alpha = \langle\langle\langle c\rangle, aid\rangle, \bar{t}\rangle$ with joint action body consisting of a left and right action:*

$$\beta = \langle\langle\langle p_l, aid_l\rangle, \bar{t_l}\rangle, \langle\langle p_r, aid_r\rangle, \bar{t_r}\rangle\rangle.$$

Let Γ_{p_l} denote $getConfiguration(p_l, \Gamma)$, similarly for Γ_{p_r}. Assume α_{p_l} denotes the left action of β relative to the component located by p_l, similarly for α_{p_r}. Let

$$\Gamma_{p_l} \overset{\alpha_{p_l}[\bar{t_l}/\bar{x}]}{\longrightarrow} \Gamma'_{p_l} \quad \textit{and} \quad \Gamma_{p_r} \overset{\alpha_{p_r}[\bar{t_r}/\bar{x}]}{\longrightarrow} \Gamma'_{p_r}.$$

Then $updateConfiguration(p_r, \Gamma'_{p_r}, updateConfiguration(p_l, \Gamma'_{p_l}, \Gamma))$ is a revision of Γ by α, and we write $\Gamma \overset{\alpha}{\longrightarrow} updateConfiguration(p_r, \Gamma'_{p_r}, updateConfiguration(p_l, \Gamma'_{p_l}, \Gamma))$.

The independent locality of the revisions allows us to revise the main configuration Γ sequentially in either order. We re-use definition 3.15 for joint action revision:

Definition 3.17 (Joint action revision) *Let Γ be a well-formed configuration and $\alpha = \langle\langle cs, aid\rangle, \bar{t}\rangle$ a ground action instance referring to a joint action body. Let $\alpha' = \langle\langle last(cs), aid\rangle, \bar{t}\rangle$ and suppose*

$$getConfiguration(cs, \Gamma) \overset{\alpha'}{\longrightarrow} \Gamma'_{cs}$$

with $\Gamma' = updateConfiguration(cs, \Gamma'_{cs}, \Gamma)$ consistent for Γ, then Γ' is a revision of Γ by α, and we write $\Gamma \overset{\alpha}{\longrightarrow} \Gamma'$.

We have also introduced choice actions, specifically for gathering together possible 'internal' actions into a single action. We define the revision semantics for a choice of two actions (the extension to more than two, including the possibility of infinite choices, is straightforward):

Definition 3.18 (Local choice action revision) *Given a well-formed configuration Γ for a component instance c and a ground action instance $\alpha = \langle\langle\langle c\rangle, aid\rangle, \bar{t}\rangle$ referring to the choice action body $\beta = \langle\langle\langle p_1, aid_1\rangle, \bar{t_1}\rangle, \langle\langle p_2, aid_2\rangle, \bar{t_2}\rangle\rangle$. Let $\Gamma_{p_i} = getConfiguration(p_i, \Gamma)$, for $i = 1, 2$. Assume α_{p_i} denotes the i-th action of β relative to the component located by p_i. Let*

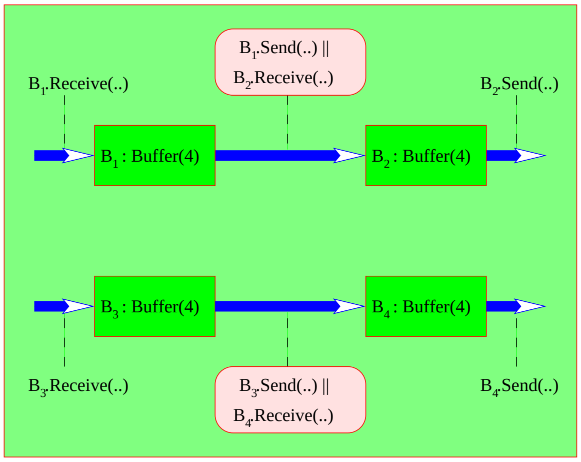
$$\Gamma_{p_i} \overset{\alpha_{p_i}[\bar{t_i}/\bar{x}]}{\longrightarrow} \Gamma'_{p_i}.$$

Then, for $i = 1$ and $i = 2$, $updateConfiguration(p_i, \Gamma'_{p_i}, \Gamma)$ is a revision of Γ by α, and we write $\Gamma \overset{\alpha}{\longrightarrow} updateConfiguration(p_i, \Gamma'_{p_i}, \Gamma)$, for $i = 1$ and $i = 2$.

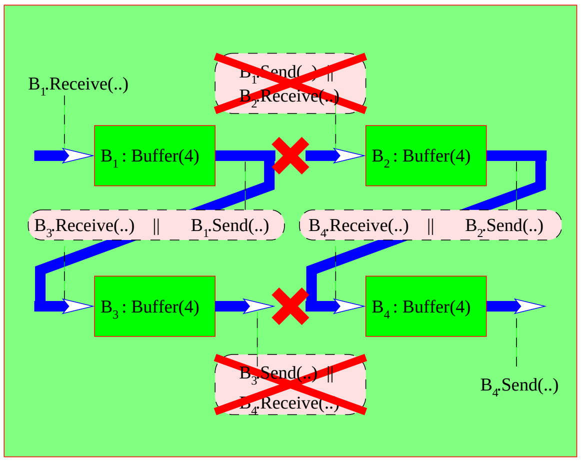
Extension of this definition to choice actions at any position in a hierarchy is as for joint actions.

3.6 Adding and modifying actions of components

The diagram below depicts a component consisting of a network of connected buffers, in which the buffer B_1 has its *Send* action joint with B_2's *Receive* action, i.e. establishing a connection from B_1 to B_2, and similarly for buffers B_3 and B_4.



A supervisor responsible for this network may change the network configuration to be that depicted below.



Thus the connection from B_1 to B_2, i.e. the joint action $B1.Send(v)||B2.Receive(v)$ as a choice over all $v \in Value.$, is to be deleted and replaced with a new connection from B_1 to B_3, i.e. a new joint action $B1.Send(v)||B3.Receive(v)$. Similarly for the other connection. Action schema definitions are part of a component schema. Thus, in order to evolve the network as desired, a new component schema is required. Let the component schema associated with the first network of buffers be $Network1$. The supervisor action therefore creates a new schema, $Network2$, built from $Network1$ but with the appropriate change in joint actions, and then ensures the instance name of the first network becomes of schema type $Network2$. Apart from this change, the component instance tree and state remain the same.

We do not give details of this supervisor. We have already presented all of the evolutionary changes needed to specify it — revision of actions, revision of schema, and naming of components.

In summary, we have seen how we can model a wide range of evolutionary actions in this framework, from simple changes of constraints, such as the capacity of a table or a buffer, to changes of actions that systems may execute, through to complex changes of network connectivity and the reconfiguration of whole systems with new and existing components.

3.7 Evolvable supervisors and hierarchy

The component structure we have introduced supports the concept of a supervisor-supervisee pair being treated as a component itself. This means that the framework supports hierarchies of supervisory processes. Consider a simple example depicted in Figure 5. It consists of a component built from two evolvable buffers, for example those specified earlier in Section 3.3, coupled to an encoder. The buffers are used to smooth out the variable data rates of devices to which the component connects.

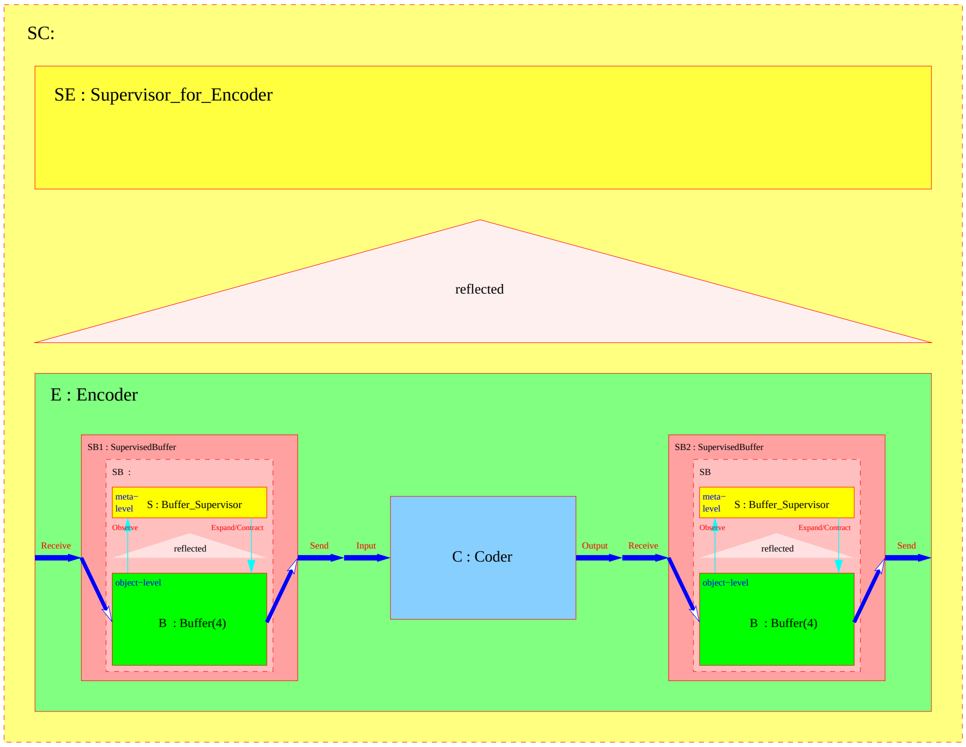


Figure 5: An example of a hierarchy of supervisors.

The buffers, being evolvable, are able to expand and contract according to demand. The way the buffers expand or contract, however, is fixed and controlled by the evolvable buffer components' own internal supervisors. An observer of these evolvable buffers may detect an undesirably high frequency of expansion and contraction and consequently 'tune' the evolvable buffers. In this situation, we may introduce a supervisor (SE in the diagram) which *evolves* supervised subcomponents.

From this simple example, we see that tiered hierarchies of evolvable components with supervisors at various levels of the hierarchy are readily definable in the framework and they enable us to model fairly complex evolvable structures. Notice that a special case of the supervising of supervised components is that where the supervisors at the two levels do not interfere and therefore both access only the underlying component. We are thus able to model the common situation of multiple monitors monitoring a common component with each individually able to change this component.

To show how complex specifications of evolvable systems may be developed, we present an extended example in Section 4. Before this, we turn now to some preliminary ideas towards a proof theory for the logical framework.

3.8 Component equivalence: Beginning an analysis of system behaviour

We now consider some aspects of a proof theory associated with this logical framework, examining how we may reason about the behaviour of evolvable systems and establish results about system equivalence. We introduce mechanisms for establishing certain forms of equivalence based on those developed in process algebra (see [19]). As an example of a proof of equivalence, we show in what sense expandable buffers, introduced above, behave as unbounded buffers.

Equivalence results are particularly appropriate in an evolutionary setting as we may require supervisors to modify networks of components by replacing components only with those of a similar behaviour so as to preserve some overall behavioural requirements. This is a major topic, covering correctness and the preservation of aspects of the behaviour of systems through evolutionary change. It is related to many of the issues in system refinement and development which have been well-studied in the literature.

Recall that we write $\Gamma \xrightarrow{\alpha} \Gamma'$ for configurations Γ and Γ' when Γ' is a revision of Γ' by action α. We consider pairs of component schemas (X, Y), and a relation ('correspondence') $\sim$ between the actions of X and actions of Y. We say that actions α and β *correspond* if $\alpha \sim \beta$, and refer to $\sim$ as an *action correspondence relation* for X and Y.

Definition 3.19 (Component strong similarity and bisimilarity) *Let A and B be component instances of schemas X and Y, and let Γ_A, etc., be a configuration for component instance A, etc. Let R be a binary relation over configurations. We call R a* strong simulation *relation if for any Γ_A and Γ_B, and any pair of corresponding actions α and β of A and B, if $(\Gamma_A, \Gamma_B) \in R$ then*

$$\forall {\Gamma_A}' \cdot \Gamma_A \xrightarrow{\alpha} {\Gamma_A}' \Rightarrow \exists {\Gamma_B}' \cdot \Gamma_B \xrightarrow{\beta} {\Gamma_B}' \wedge ({\Gamma_A}', {\Gamma_B}') \in R.$$

Furthermore, we say that configuration Γ_B strongly simulates *Γ_A, denoted by $\Gamma_A \preceq_S \Gamma_B$, if there is a strong simulation relation R such that $(\Gamma_A, \Gamma_B) \in R$. We extend to components by defining $A \preceq_S B$ if and only if for any configuration Γ_A there is a configuration Γ_B such that $\Gamma_A \preceq_S \Gamma_B$.*

For a strong simulation relation R, if we have in addition that for any pair of corresponding actions α and β of A and B, if $(\Gamma_A, \Gamma_B) \in R$ then

$$\forall {\Gamma_B}' \cdot \Gamma_B \xrightarrow{\beta} {\Gamma_B}' \Rightarrow \exists {\Gamma_A}' \cdot \Gamma_A \xrightarrow{\alpha} {\Gamma_A}' \wedge ({\Gamma_A}', {\Gamma_B}') \in R$$

then we say R is a strong bisimulation *over configurations. We extend to components to say A is* strongly bisimilar *to B, denoted by $A \sim_S B$.*

These are adapted from standard definitions in process algebra.

Example 3.1 *Consider a component instance A : $Buffer(4)$ based on $Buffer(N)$ schema presented at the start of Section 3 and an instance B : $DoubleBuffer(2)$ of the component schema below.*

$DoubleBuffer(N)$
COMPONENTS
$\quad B_1 : Buffer(N)$
$\quad B_2 : Buffer(N)$
ACTIONS
$\quad Receive(v) \stackrel{dfn}{=} B_1.Receive(v)$
$\quad Send(v) \stackrel{dfn}{=} B_2.Send(v)$
$\quad Internal \stackrel{dfn}{=} \vert_{v \in Value}\ (B_1.Send(v) \vert\vert B_2.Receive(v))$

Using the obvious correspondence between actions of A and B, namely $A.Send(v) \sim B.Send(v)$, etc., it is relatively straightforward to show that $B \preceq_S A$ but $A \not\preceq_S B$.

We introduce a formula Φ_R over the joint theories of A and B to characterize a relation R on configurations of A and B, namely:

$$\forall l : Value\text{-}list \cdot A.content(l) \Leftrightarrow \exists l_1, l_2 : Value\text{-}list \cdot (l = l_1 :: l_2) \wedge B.B1.content(l_1) \wedge B.B2.content(l_2).$$

More formally, configuration pair $(\Gamma_A, \Gamma_B) \in R$ iff $\downarrow \Delta(\Gamma_A), \downarrow \Delta(\Gamma_B) \models_{\Gamma_A, \Gamma_B} \Phi_R$.

It remains to show that Φ_R is a strong simulation relation. This is straightforward for the $B \preceq_S A$, but fails the other way around. The problem arises from the internal move that the double buffer can make to shift a value from the first buffer to the second. Consider the flattened double buffer state $\{B.B1.content([1]), B.B2.content([])\}$. The flattened state for the single buffer $\{A.content([1])\}$ is clearly related via Φ_R. The single buffer can perform the action $A.Send(1)$. However, revision by the corresponding action on the double buffer, i.e. $B.Send(1)$, is not defined since its precondition cannot be established — it requires the element 1 to be present, but its content is empty.

The example illustrates that a weaker form of simulation relation is required, which takes into account internal actions. In fact we can use the standard process-theoretic notion of weak simulation. Assume that component actions are classified as external or internal (in fact, we have been using *Internal* to denote such internal actions).

Definition 3.20 (Weak revision relation) *Given configurations* Γ *and* Γ' *for a component with an external action* α *and internal action* τ, *we say* Γ weakly revises *to* Γ' *under* α, *denoted by* $\Gamma \stackrel{\alpha}{\Longrightarrow} \Gamma'$, *if and only if there exist configurations* Γ_i *and* ${\Gamma_i}'$ *of the component such that* $\Gamma \stackrel{\tau^*}{\longrightarrow} \Gamma_i \stackrel{\alpha}{\longrightarrow} {\Gamma_i}' \stackrel{\tau^*}{\longrightarrow} \Gamma'$ *where each* τ^* *is a, possibly empty, finite sequence of* τ *actions.*

Definition 3.21 (Component weak similarity and bisimilarity) *Let* A *and* B *be component instances of schemas* X *and* Y, *and let* Γ_A, *etc., be a configuration for component instance* A, *etc. Let* R *be a binary relation over configurations. We call* R *a* weak simulation *relation if for any* Γ_A *and* Γ_B, *and any pair of corresponding actions* α *and* β *of* A *and* B, *if* $(\Gamma_A, \Gamma_B) \in R$ *then*

$$\forall {\Gamma_A}' \cdot (\Gamma_A \stackrel{\alpha}{\Longrightarrow} {\Gamma_A}') \ \Rightarrow\ \exists {\Gamma_B}' \cdot (\Gamma_B \stackrel{\beta}{\Longrightarrow} {\Gamma_B}') \wedge ({\Gamma_A}', {\Gamma_B}') \in R.$$

Furthermore, we say that configuration Γ_B weakly simulates Γ_A, *denoted by* $\Gamma_A \preceq_W \Gamma_B$, *if there is a weak simulation relation* R *such that* $(\Gamma_A, \Gamma_B) \in R$. *We extend to components by defining* $A \ll B$ *if and only if for any configuration* Γ_A *there is a configuration* Γ_B *for which we have* $\Gamma_A \preceq_W \Gamma_B$.

For a weak simulation relation R, *if we have in addition that for any pair of corresponding actions* α *and* β *of* A *and* B, *if* $(\Gamma_A, \Gamma_B) \in R$ *then*

$$\forall {\Gamma_B}' \cdot (\Gamma_B \stackrel{\beta}{\Longrightarrow} {\Gamma_B}') \ \Rightarrow\ \exists {\Gamma_A}' \cdot (\Gamma_A \stackrel{\alpha}{\Longrightarrow} {\Gamma_A}') \wedge ({\Gamma_A}', {\Gamma_B}') \in R$$

then we say R *is a* weak bisimulation *relation over configurations. We extend to components to say* A *is* weakly bisimilar *to* B, *denoted by* $A \sim_W B$.

In the above example of single and double buffers, the two systems are weakly bisimilar.

These notions of equivalence are important, not only for establishing behavioural properties, but also because some models of evolutionary behaviour (e.g. [12]) require processes to be in particular types of state before evolutionary actions can take place. We now examine one such notion of appropriate state for evolution actions, that of a 'quiescent' state.

Definition 3.22 (Quiescent states) *Given a component configuration* $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$, *its state* Δ *is said to be* quiescent *if and only if no internal action* τ *of the component is defined for* Δ, *i.e. there is no* τ *such that* $\downarrow \Delta \models_\Gamma$ *pre-*τ, *where pre-*τ *is the precondition in the* τ *action definition.. A configuration* $\Gamma = \langle \Delta, \Theta, \Sigma \rangle$ *is said to be quiescent if* Δ *is a quiescent state.*

Definition 3.23 (Reachable quiescent states) *Given a state* Δ *of a configuration* Γ, *we define* $Q_\Gamma(\Delta)$ *to be the set of quiescent states reachable from* Δ *by internal actions* τ, *i.e.*

$$Q_\Gamma(\Delta) = \{\Delta' \mid \Delta \stackrel{\tau^*}{\longrightarrow} \Delta',\ \Delta' \textit{ quiescent}\}$$

where τ^* *is a, possibly empty, finite sequence of* τ *actions. This definition is extended to configurations: for configuration* Γ, $Q(\Gamma) = \{\Gamma' \mid \Gamma \stackrel{\tau^*}{\longrightarrow} \Gamma',\ \Gamma' \textit{ quiescent}\}$.

Definition 3.24 (Quiescent action revision relation) *Given configurations* Γ *and* Γ' *for a component with an external action* α *and internal action* τ, *we say* Γ revises to quiescent *configuration* Γ' *under* α, *denoted by* $\Gamma \stackrel{\alpha}{\Longrightarrow}_Q \Gamma'$, *if and only if there exist configurations* ${\Gamma_i}'$ *of the component such that* $\Gamma \stackrel{\alpha}{\longrightarrow} {\Gamma_i}' \stackrel{\tau^*}{\longrightarrow} \Gamma'$ *and* Γ' *is a quiescent configuration.*

Definition 3.25 (Quiescent simulation and bisimulation) *Let* A *and* B *be component instances of schemas* X *and* Y, *and let* Γ_A, *etc., denote a configuration for component instance* A, *etc. Let* R *be a binary relation over configurations. We say that* R *is a* quiescent simulation *relation if for any* Γ_A *and* Γ_B, *and any pair of corresponding actions* α *and* β *of* A *and* B, *if* $(\Gamma_A, \Gamma_B) \in R$ *then*

$$\forall {\Gamma_A}' \cdot (\Gamma_A \stackrel{\alpha}{\Longrightarrow}_Q {\Gamma_A}') \ \Rightarrow\ \exists {\Gamma_B}' \cdot (\Gamma_B \stackrel{\beta}{\Longrightarrow}_Q {\Gamma_B}') \wedge ({\Gamma_A}', {\Gamma_B}') \in R.$$

Furthermore, we say that configuration Γ_B quiescently simulates Γ_A, *denoted by* $\Gamma_A \preceq_Q \Gamma_B$, *if there is a quiescent simulation relation* R *such that* $(\Gamma_A, \Gamma_B) \in R$. *We extend to components by defining* $A \preceq_Q B$ *if and only if for any configuration* Γ_A *there is a configuration* Γ_B *for which* $\Gamma_A \preceq_Q \Gamma_B$.

For such a quiescent simulation relation R, if in addition we have that for any pair of corresponding actions α and β of A and B, if $(\Gamma_A, \Gamma_B) \in R$ then

$$\forall \Gamma_B{}' \cdot (\Gamma_B \overset{\beta}{\Longrightarrow}_Q \Gamma_B{}') \;\Rightarrow\; \exists \Gamma_A{}' \cdot (\Gamma_A \overset{\alpha}{\Longrightarrow}_Q \Gamma_A{}') \wedge (\Gamma_A{}', \Gamma_B{}') \in R$$

then we say R is a quiescent bisimulation *relation over configurations. We extend to components to say A is* quiescently bisimilar *to B, denoted by $A \sim_Q B$.*

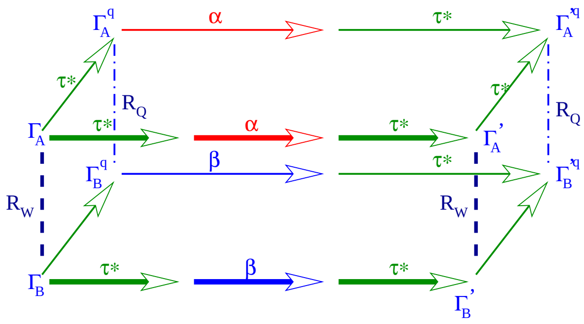
We have the following results:

Theorem 3.1 *For any component instances A and B,*

1. *$A \preceq_S B \Rightarrow A \preceq_W B$*
2. *$A \preceq_S B \Rightarrow A \preceq_Q B$.*

Quiescent states provide a natural way to relate states of different components. Are there conditions under which one can conclude that quiescent simulation and weak simulation define the same relation, i.e. under what conditions on A and B does $A \preceq_W B \Rightarrow A \preceq_Q B$ hold, and what are the conditions for $A \preceq_Q B \Rightarrow A \preceq_W B$?

Consider the case of $A \preceq_W B \Rightarrow A \preceq_Q B$. If the situation depicted below can be established then the result will follow.

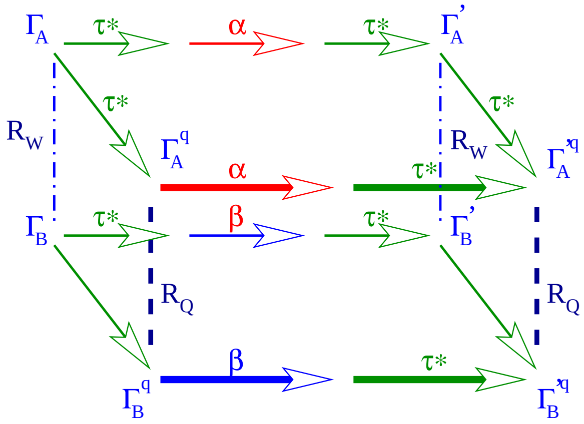


Let R_W be a weak similarity relation between component instances A and B. Consider any weakly similar configurations Γ_A and Γ_B and quiescent configurations Γ_A^q and Γ'^q_A reachable as in the diagram through action α. By definition of weak similarity, there exists a corresponding action β and configuration Γ'_B that completes the front face rectangle of the diagram. Assume the source and target configurations of β are non-quiescent. If it can be shown that by allowing the configuration Γ_B to reach a quiescent state first, revision by β is still defined and leads to a quiescent configuration Γ'^q_B, also a quiescent configuration from Γ'_B, then clearly a quiescent similarity relation R_Q can be constructed using the q superscripted configurations in the diagram. The maximality of R_W implies the maximality of R_Q. We thus need the following definition in order to establish the result.

Definition 3.26 (Internal interference freedom) *If for any component configurations Γ, Γ', Γ^q, Γ'^q and external component action α such that $\Gamma \overset{\alpha}{\Longrightarrow} \Gamma'$, $\Gamma^q \overset{\alpha}{\longrightarrow}\overset{\tau^*}{\longrightarrow} \Gamma'^q$ and $\Gamma^q \in Q(\Gamma)$ we have that $\Gamma'^q \in Q(\Gamma')$ then we say that α is* free of internal interference. *A component is said to be free of internal interference if all its external actions are free of internal interference.*

Theorem 3.2 *Given component instances A and B such that $A \preceq_W B$ and all external actions α in A and their corresponding actions in B are free from internal interference, and A and B have no infinite internal chatter, then $A \preceq_Q B$.*

Consider now the other direction $A \preceq_Q B \Rightarrow A \preceq_W B$. The relevant diagram is:



By the definition of quiescent similarity, a maximal relation R_Q exists such that the q superscripted configurations in the commuting cube are all quiescent. If action α is free from internal interference, then we can add the top face of the cube. Similarly for the bottom face. We then claim that the relation obtained by associating configurations (Γ_A, Γ_B) and (Γ'_A, Γ'_B) is a weak simulation relation.

Theorem 3.3 *Given component instances A and B such that $A \preceq_Q B$ and all actions α in A and their corresponding actions in B are free from internal interference, and A and B have no infinite internal chatter, then $A \preceq_W B$.*

Example 3.2 *Consider again the buffers $B : DoubleBuffer(2)$ and $A : Buffer(4)$ from Example 3.1. The double buffer component has internal moves corresponding to the transfer of values from the first buffer to the second. Quiescent states of $B : DoubleBuffer(2)$ can be characterized by the formula*

$$B.B_1.content([]) \vee \neg B.B_2.free.$$

For all states that do not satisfy the above formula, the internal (joint) action $B_1.Send(v)||B_2.Receive(v)$ will be defined for some value v, and for all states that satisfy the formula, the internal action is not defined.

As in the previous example, we use a formula over the joint theories for components A and B to characterize a quiescent (bi)simulation relation between the components. Indeed, the same formula as before will suffice, namely

$$\forall l : Value\text{-}list \cdot A.content(l) \Leftrightarrow \exists l_1, l_2 : Value\text{-}list \cdot (l = l_1 :: l_2) \wedge B.B1.content(l_1) \wedge B.B2.content(l_2).$$

Furthermore, since one can show that the Send and Receive actions of component A are free of interference by the action $Internal$ (representing an internal τ action), we have established that the components are weakly bisimilar.

As a final example in this section, we consider the original form of evolvable buffer, the theory $Supervised\_Buffer$ of Section 3.3. There is clearly a sense in which this behaves as an *unbounded buffer*, for whilst any instance of it is a bounded buffer, it always has the capability of expanding its capacity when required. We now formulate this result:

Example 3.3 *We begin by defining the theory of unbounded buffers by modifying that for bounded buffers, removing the size constraint and the freeness predicate:*

$Unbounded\_Buffer$

OBSERVATION PREDICATES

$content : Value\text{-}list$

CONSTRAINTS

$Uniqueness \overset{dfn}{=} \forall l_1, l_2 : Value\text{-}list \cdot content(l_1) \wedge content(l_2) \Rightarrow l_1 = l_2$

ACTIONS

$Send(v)$	
pre	$\{content(l :: v)\}$
add	$\{content(l)\}$
del	$\{content(l :: v)\}$

$Receive(v)$	
pre	$\{content(l)\}$
add	$\{content(v :: l)\}$
del	$\{content(l)\}$

Consider the correspondence of the Send and Receive actions between the two component theories Supervised_Buffer and Unbounded_Buffer, and the internal action of Expand in Supervised_Buffer. Using this correspondence and internal action, the theory Supervised_Buffer is weakly bisimilar *to the theory Unbounded_Buffer.*

Here the weak bisimilarity may be established directly, as the internal actions are not those of interaction between components (as above), but are evolutionary expansion actions which leave the contents unchanged.

In summary, we have begun, in this section, the development of techniques to reason about evolvable systems, introducing some of the key ideas for establishing equivalence results. Clearly, there is much more to be done in this direction in order to develop a full account of reasoning techniques for evolvable systems in this framework.

4 Banks, ATMs, cards, chips, PINs and all that jazz

We now present an extended example of an evolvable system within the logical framework to illustrate how such systems may be developed and how they may be structured around evolutionary capabilities.

The example is that of an imaginary bank, the Bank of New Island, with its extensive network of twentieth century automated teller machines (ATMs). Recently it has lost its previously highly-valued customers' trust and confidence through a series of major attacks of debit card fraud. The Board of Governors agreed to fund a major overhaul of the bank's ATM network. The Board members, being only too aware of the high cost of increasing security on such a massive scale, believed a novel adaptive or evolutionary approach may be appropriate, and sponsored the University of NoMoreDisasters to collaborate with the bank on the design of the new system.

In this section, we use a highly simplified model of the proposed system to illustrate the revision-based logical modelling of evolvable systems.

4.1 ATMs old and the new

The bank's old form of ATM, although comprising distinct hardware components, such as magnetic strip readers, note counters, keypads, displays, etc., had its local software built in a somewhat unstructured, monolithic, fashion. Only limited security checks were programmed and certainly not easily changed (indeed the whole ATM network would need to be shutdown to perform even minor upgrades). The design of the proposed system is such that each individual ATM will monitor, adapt and evolve its behaviour, including its security checking, to fit best with the bank's and its customers' desires and expectations. The individual (software) components used in the ATM will themselves also be evolvable and the network of ATMs will naturally support dynamic evolution.

4.2 The old system

The original banking system consists of a non-evolvable version which we now describe. The banking system is modelled as a component consisting of two types of component, a bank records component and a series of automated teller machine components (ATMs). Figure 6 shows a number of ATMs linked to a central bank records component. In the formal model we simplify to just two ATM subcomponents:

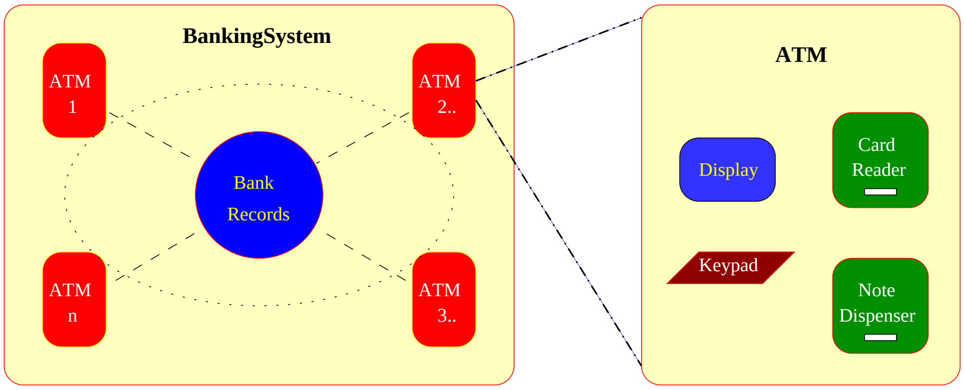


Figure 6: The banking system component structure

BankingSystem

COMPONENTS

$bank : BankRecords$
$atm1 : ATM$
$atm2 : ATM$

ACTIONS

$checkBalance(acc : Account, bal : Int)$

$$\stackrel{dfn}{=} \langle bank.checkBalance(acc, bal) \,||\, atm1.checkBalance(acc, bal)\rangle \mid$$
$$\langle bank.checkBalance(acc, bal) \,||\, atm2.checkBalance(acc, bal)\rangle \mid$$

$debitAccount(acc : Account, n : Int)$

$$\stackrel{dfn}{=} \langle bank.debitAccount(acc, n) \,||\, atm1.debitAccount(acc, n)\rangle \mid$$
$$\langle bank.debitAccount(acc, n) \,||\, atm2.debitAccount(acc, n)\rangle \mid$$

We model two actions of the banking network – checking an account balance and debiting an account. These actions are joint between an ATM component and the $BankRecords$ component. The $BankRecords$ component is responsible for maintaining customer account details. We model this below via the *accounts* predicate. Note that the *checkBalance* action has no effect on the (specified) observation state.

BankRecords

OBSERVATION PREDICATES

$accounts : Account \times Int$

CONSTRAINTS

$unique \stackrel{dfn}{=} \forall a : Account, i, j : Int \cdot ((account(a, i) \wedge accounts(a, j)) \Rightarrow i = j)$

ACTIONS

$checkBalance(acc : Account, bal : Int)$	
pre	$\{accounts(acc, bal)\}$
add	$\{\}$
del	$\{\}$

$debitAccount(acc : Account, n : Int)$	
pre	$\{accounts(acc, bal)\}$
add	$\{accounts(acc, bal - n)\}$
del	$\{accounts(acc, bal)\}$

We describe an automated teller machine, ATM, with four components, a card reader, a note dispenser, a keypad and a display screen.

ATM

OBSERVATION PREDICATES

COMPONENTS

$CR : CardReader_0$
$ND : NoteDispenser$
$KP : KeyPad$
$DS : DisplayScreen$

ACTIONS

$checkNotesAvailable(n : Int)$

pre	$\{ND.noteStore(m), m \geq n\}$
add	$\{\}$
del	$\{\}$

$checkBalance(acc : Account, bal : Int)$

pre	$\{\exists pin : PIN \cdot CR.currentCard(acc, pin)\}$
add	$\{\}$
del	$\{\}$

$debitAccount(acc : Account, n : Int)$

pre	$\{\exists pin : PIN \cdot CR.currentCard(acc, pin)\}$
add	$\{\}$
del	$\{\}$

Following insertion of a card into the ATM's card reader, and its subsequent validation, the customer may request withdrawal of funds. The ATM will then check that the customer's account has sufficient funds (a *checkBalance* enquiry with the *BankRecords* component), check availability of notes in the *NoteDispenser*, debit the account (a joint *debitAccount* action with the *BankRecords* component) and serve the customer the appropriate number of notes (via the *NoteDispenser* component).

$CardReader_0$

OBSERVATION PREDICATES

$currentCard : Account \times PIN$
$cardAccepted, cardRejected$
$swallowedCard : Account \times PIN$

CONSTRAINTS

$$unique \stackrel{dfn}{=} \forall a_1, a_2 : Account, p_1, p_2 : PIN \cdot ((currentCard(a_1, p_1) \wedge currentCard(a_2, p_2)) \Rightarrow (a_1 = a_2 \wedge p_1 = p_2)) \wedge$$
$$\neg(cardAccepted \wedge cardRejected) \wedge$$
$$\forall a : Account, p_1, p_2 : PIN \cdot ((swallowedCard(a, p_1) \wedge swallowedCard(a, p_2)) \Rightarrow (p_1 = p_2))$$

ACTIONS

$cardIn(acc : Account, pin : PIN)$

pre	$\{\neg\exists a : Account, p : PIN \cdot currentCard(a, p)\}$
add	$\{currentCard(acc, pin)\}$
del	$\{cardAccepted, cardRejected\}$

$validateCard(userPin : PIN)$

pre	$\{currentCard(acc, userPin)\}$
add	$\{cardAccepted\}$
del	$\{\}$

cardOut

pre	$\{currentCard(acc, pin)\}$
add	$\{\}$
del	$\{currentCard(acc, pin)\}$

swallowCard

pre	$\{currentCard(acc, pin), \neg cardAccepted\}$
add	$\{swallowedCard(acc, pin), cardRejected\}$
del	$\{currentCard(acc, pin)\}$

The $CardReader$ component (for security level 0) is abstracted to holding the account and PIN (Personal Indentification Number) for the card currently in the ATM card reader and any cards that have not been returned to the customer. The $cardIn$ action is defined when no card is present (a $currentCard(acc, pin)$ atom will not be in the state) and makes the account number of the card (modelled as an argument to the action) a state observation. Validation of the current card is noted by the $validateCard$ action. This action is defined when the user-supplied PIN is the same as that stored on the card and the effect of the action is to record as an observation $cardAccepted$. The $cardOut$ action simply removes the observation from the state. The $swallowCard$ action removes the $currentCard$ observation and adds the fact that the card is swallowed, $swallowedCard(acc, pin)$, as well as its rejection. Although strictly not necessary, we have ensured that the action can only be defined when the current card has not been accepted.

NoteDispenser

Observation Predicates

$noteStore : Int$

Constraints

$unique \stackrel{dfn}{=} \forall i, j : Int \cdot ((noteStore(i) \wedge noteStore(j)) \Rightarrow i = j)$

Actions

dispenseNotes$(n : Int)$

pre	$\{noteStore(m), m \geq n\}$
add	$\{noteStore(m - n)\}$
del	$\{noteStore(m)\}$

The $NoteDispenser$ component maintains a count of the number of the notes held and is made observable so that the ATM can check availability of notes requested by a customer. The $NoteDispenser$ component's own action $dispenseNotes$ reduces the count by the given argument.

4.3 The new system

Let us now investigate a more refined version of the above ATM banking system, one in particular that has been structured to support adaptive validation and security checking. Firstly, though, we add a supervisor to the note dispenser component in order to handle its re-stocking.

Meta *NoteDispenserSupervisor*

Types

$ConfigName$

Functions

$s : ConfigName \rightarrow ConfigName$

$c_0 : ConfigName$

Observation Predicates

$holds : \text{FORMULA} \times ConfigName$

$current : ConfigName$

$evolve : \text{STATETRANSFORMER} \times \text{COMPONENTMAP} \times \text{SCHEMADEFS} \times ConfigName$

Constraints

...

Actions

$Observe(Q : \text{FORMULAE})$

pre	$\{current(c)\}$
add	$\{holds(q, s(c)), current(s(c)) \mid q \in Q\}$
del	$\{current(c)\}$

$stockNotes(n : Int)$

pre	$\{current(c), holds(noteStore(m), c)\}$
add	$\{evolve(\ \lambda\Delta \cdot [C \mapsto \langle\ local(\Delta(C)) \cup \{noteStore(n)\} \backslash \{noteStore(m)\}, subStates(\Delta(C))\rangle], [], [], s(c)), current(s(c)), holds(noteStore(n, s(c))\}$
del	$\{current(c)\}$

Two actions have been initially specified, a standard observation action that can be used to track the quantity of notes in the dispenser, together with an action to restock the dispenser. This latter action induces change in the associated note dispenser component through the presence of the *evolve* predicate as a meta-level observation. Of course, we could have incorporated such a straightforward change as an action directly in the note dispenser component. We believe, however, it is better software engineering practice to maintain a clear separation of concerns, i.e. let a supervisor be responsible for ensuring good (and timely) stock levels. Although not specified here, the supervisor may have been programmed to have been more adaptive and replenish with differing levels of stock according to expected demand, based on previous usage patterns.

We define a composite evolvable note dispenser below.

SupervisedNoteDispenser

Observation Predicates

$noteStore : Int$

Components

$SND : (S : NoteDispenserSupervisor \text{ META TO } (ND : NoteDispenser))$

Constraints

$alias \stackrel{dfn}{=} \forall m : Int \cdot noteStore(m) \Leftrightarrow ND.noteStore(m)$

Actions

$dispenseNotes(n : Int) \stackrel{dfn}{=} |_{Q \in Formulae}\ SND.\langle S.Observe(Q), ND.dispenseNotes(n)\rangle$

$internal \stackrel{dfn}{=} SND.\langle S.stockNotes(1000), \rangle$

This component schema replaces $NoteDispenser$ in the previous ATM schema. We could develop further levels of monitoring and evolutionary change for the note dispenser, for example, enabling changes to note denominations, currency being served, etc. Instead, we focus on aspects of the card reader componentry.

Firstly, we specify some generic monitoring aspects of a supervisor for a card reader component. The currently specified card reader component records whether a card is rejected or accepted. Based just on this, there can then be a number of different types of temporal criteria that may be monitored. For example, the system may be programmed to monitor the ratio of rejected to accepted cards over a rolling 24-hour basis, or just on a daily basis, or over a fixed number of evening/night hours, etc. We introduce a meta-level observation predicate *criterion* parameterized by a criterion type and value; the value may represent series data (necessary to compute rolling ratios, etc.), and use the function *updateCriterion* to update the criterion value associated with a particular criterion type. We also assume an observation predicate *clock* that provides time information.

Meta *CardReaderSupervisor*

Types

$ConfigName$

$CriterionType \stackrel{dfn}{=} \ldots$

$CriterionValue \stackrel{dfn}{=} \ldots$

$\vdots$

Functions

$s : ConfigName \rightarrow ConfigName$

$c_0 : ConfigName$

$MaxKnownSecurityLevel : Int$

$updateCriterion : CriterionType \times CriterionValue \times Int \times Time \rightarrow CriterionValue$

Observation Predicates

$clock : Time$

$criterion : CriterionType \times CriterionValue$

$securityUpgrade : Int \times \textsc{stateTransformer} \times \textsc{componentTransformer} \times \textsc{schemaTransformer}$

$holds : \textsc{formula} \times ConfigName$

$current : ConfigName$

$evolve : \textsc{stateTransformer} \times \textsc{componentMap} \times \textsc{schemaDefs} \times ConfigName$

Constraints

$\ldots$

Actions

idle

pre	$\{\}$
add	$\{\}$
del	$\{\}$

$observeAccept(X : 2^{CriterionType})$

pre	$\{current(c), clock(t), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$
add	$\{holds(cardAccepted, s(c)), current(s(c)), \bigwedge_{ct \in X} criterion(ct, updateCriterion(ct, cv_{ct}, 1, t))\}$
del	$\{current(c), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$

$observeReject(X : 2^{CriterionType})$

pre	$\{current(c), clock(t), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$
add	$\{holds(cardRejected, s(c)), current(s(c)), \bigwedge_{ct \in X} criterion(ct, updateCriterion(ct, cv_{ct}, 0, t))\}$
del	$\{current(c), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$

So far the supervisor has only monitoring actions specified, the principal actions being *observeAccept* and *observeReject*, which record results of desired statistical analyses. For simplicity's sake, let us assume that the card reader supervisor has pre-programmed transformations that it can apply to the card reader in order to increase the level of security checking it uses when validating a card. We have specified a basic level of security checking (level 0) as being a simple check of user supplied PIN against the PIN stored on the card. A higher level might be for the ATM to check with the bank on the card's recent transaction history to determine whether its current use is out of the norm, and then, if so, to proceed through further security checks, for example via questions agreed previously with the customer. It may also be possible to invoke other forms of unique customer identification, e.g. finger prints, iris prints, etc., depending upon hardware capability and information stored on chip. The following action schema abstracts the update via three transformations that are stored by the supervisor. The supervisor observation predicate $securityUpgrade(level, st, ct, cs)$ records the fact that st, ct and cs are, respectively, state, component instance and component schema transformers to yield a card reader at security level $level$. The $upgradeSecurityChecking$ action applies these transformers in the appropriate way to the observation state, component instance map and schema map of the object-level configuration for the card reader component by the addition of a suitably instantiated $evolve$ predicate in the supervisor's observation state.

$upgradeSecurityChecking(level : int)$

pre	$\{current(c), securityUpgrade(level, st, ct, cs),$ $component(thisComp \text{ as } [C \mapsto \langle CardReader_{SL}, \_, \_, \_\rangle]),$ $level > SL\}$
add	$\{current(s(c)), component(ct(thisComp)),$ $evolve(st, ct(thisComp), cs(CardReader_{SL}), s(c))\}$
del	$\{current(c), component(thisComp)\}$

The supervisor will determine when to apply the above upgrade action. We have not specified this but its application will be dependent on a combination of security criterion values (computed through the monitoring aspect of the supervisor) passing certain thresholds. A downgrade action could be defined in a similar way.

A relevant question is then what happens if a security upgrade is determined desirable but none available locally to the supervisor. In principle, such situations should be observed by a higher-level supervisor, which may then be able to supply an appropriate upgrade. However, the lack of upgrade may proceed to the top of the component hierarchy, in which case no upgrade of the required kind has been made available.

$SupervisedCardReader$

Observation Predicates

$currentCard : Account \times PIN$

Components

$SCR : (S : CardReaderSupervisor \text{ META TO } (CR : CardReader_0))$

Constraints

$alias \stackrel{dfn}{=} \forall a : Account, p : PIN \cdot currentCard(a, p) \Leftrightarrow CR.currentCard(a, p)$

Actions

$cardIn(acc : Account, pin : PIN) \stackrel{dfn}{=} SCR.\langle S.idle, CR.cardIn(n)\rangle$

$validateCard(userPin : PIN) \stackrel{dfn}{=} |_{X \in 2^{CriterionType}} SCR.\langle S.observeAccept(X), CR.validateCard(userPin)\rangle$

$cardOut \stackrel{dfn}{=} SCR.\langle S.idle, CR.cardOut\rangle$

$swallowCard \stackrel{dfn}{=} |_{X \in 2^{CriterionType}} SCR.\langle S.observeReject(X), CR.swallowCard\rangle$

$internal \stackrel{dfn}{=} SCR.\langle S.upgradeSecurityChecking(1)\rangle \mid$
$\quad \ldots \mid$
$\quad SCR.\langle S.upgradeSecurityChecking(S.MaxKnownSecurityLevel)\rangle$

The schema $SupervisedCardReader$ can now be used in place of the $CardReader$ schema in the previous ATM schema presentation.

ATM

Observation Predicates

Components

$CR : SupervisedCardReader$

$ND : SupervisedNoteDispenser$

$KP : KeyPad$

$DS : DisplayScreen$

Actions

$checkNotesAvailable(n : Int)$

pre	$\{ND.noteStore(m), m \geq n\}$
add	$\{\}$
del	$\{\}$

$checkBalance(acc : Account, bal : Int)$

pre	$\{\exists pin : PIN \cdot CR.currentCard(acc, pin)\}$
add	$\{\}$
del	$\{\}$

$debitAccount(acc : Account, n : Int)$

pre	$\{\exists pin : PIN \cdot CR.currentCard(acc, pin)\}$
add	$\{\}$
del	$\{\}$

$internal \overset{dfn}{=} CR.internal \mid ND.internal \mid \ldots$

The original banking system was presented as a component consisting of a bank records subcomponent and two ATM subcomponents. Given the evolvable nature of the ATM's subcomponents, it is now appropriate to restructure the two ATM subcomponents as a network of ATMs. The network component may then be endowed with its own supervisor that can oversee the evolution of the individual ATMs and potentially take pre-emptive action by enforcing security checking updates on one ATM that may be in a similar situation to another ATM, for which an update has already been determined necessary.

$ATM\_Network$

Components

$atm1 : ATM$

$atm2 : ATM$

Actions

$checkBalance(acc : Account, bal : Int)$

$\overset{dfn}{=} atm1.checkBalance(acc, bal) \mid atm2.checkBalance(acc, bal)$

$debitAccount(acc : Account, n : Int)$

$\overset{dfn}{=} atm1.debitAccount(acc, n) \mid atm2.debitAccount(acc, n)$

$internal \overset{dfn}{=} atm1.internal \mid atm2.internal$

The ATM network supervisor schema is presented in a similar way to the card reader supervisor. We assume local predicate *criterion* to record the results of time series statistical analyses that are performed when the supervisor observes that an ATM subcomponent has undergone an evolution (via $observeCR$). The local *updateCriterion* function is over the ATM identifier as well as the updated security level. The network supervisor has the wherewithal not only to enforce an update in security even though locally it may not appear to be deemed necessary, but also to update the card reader supervisor, for example, with additional security transformations. To keep the level of detail low, we include just the former supervisor action (as $upgradeCR\_Security$).

META *ATM_NetworkSupervisor*

TYPES

$CriterionType \stackrel{dfn}{=} \ldots$

$CriterionValue \stackrel{dfn}{=} \ldots$

$ConfigName$

FUNCTIONS

$s : ConfigName \rightarrow ConfigName$

$c_0 : ConfigName$

$updateCriterion : CriterionType \times CriterionValue \times CID \times Int \times Time$

OBSERVATION PREDICATES

$clock : Time$

$criterion : CriterionType \times CriterionValue$

$holds : \text{FORMULA} \times ConfigName$

$current : ConfigName$

$evolve : \text{STATETRANSFORMER} \times \text{COMPONENTMAP} \times \text{SCHEMADEFS} \times ConfigName$

ACTIONS

idle

pre	$\{\}$
add	$\{\}$
del	$\{\}$

$observeCR\_Evolve(atm : CID, X : 2^{CriterionType})$

pre	$\{current(c), atm.CR.SCR.S.current(c_1), clock(t), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$
add	$\{holds(atm.CR.SCR.S.current(s(c_1)), s(c)), current(s(c)), holds(atm.CR.SCR.S.evolve(n, \_, \_, \_, s(c_1)), s(c)) \bigwedge_{ct \in X} criterion(ct, updateCriterion(ct, cv_{ct}, atm, n, t))\}$
del	$\{current(c), \bigwedge_{ct \in X} criterion(ct, cv_{ct})\}$

$upgradeCR\_Security(atm : CID, level : int)$

pre	$\{current(c), securityUpdate(level, st, ct, cs), component(thisComp \text{ as } [C \mapsto \langle \ldots, [atm \mapsto \langle \ldots, [CR \mapsto \langle CardReader_{SL}, \ldots \rangle, \ldots], \ldots \rangle], \ldots \rangle])\}$
add	$\{current(s(c)), component(ct(thisComp)), evolve(st, ct(thisComp), cs(CardReader_{SL}), s(c))\}$
del	$\{current(c), component(thisComp)\}$

The ATM network component and its associated supervisor are then composed and finally included in a revised banking system component.

SupervisedATM_Network

COMPONENTS

$SNet : (S : ATM\_NetworkSupervisor \text{ META TO } (ATM\_Net : ATM\_Network))$

ACTIONS

$checkBalance(acc : Account, bal : Int) \stackrel{dfn}{=}$
$\quad SNet.\langle S.idle, ATM\_Net.checkBalance(acc, bal)\rangle$

$debitAccount(acc : Account, n : Int) \stackrel{dfn}{=}$
$\quad SNet.\langle S.idle, ATM\_Net.debitAccount(acc, n)\rangle$

$internal \stackrel{dfn}{=}$
$\quad |_{atm \in CID, X \in 2^{CriterionType}} SNet.\langle S.observeCR\_Evolve(atm, X), ATM\_Net.internal\rangle \mid$
$\quad SNet.\langle S.updateCR\_Security\rangle$

Finally, the overall banking system is given below.

$BankingSystem$

COMPONENTS

$bank : BankRecords$

$atmNet : SupervisedATM\_Network$

ACTIONS

$$checkBalance(acc : Account, bal : Int) \stackrel{dfn}{=}$$
$$\langle bank.checkBalance(acc, bal) \,||\, atmNet.checkBalance(acc, bal)\rangle$$

$$debitAccount(acc : Account, n : Int) \stackrel{dfn}{=}$$
$$\langle bank.debitAccount(acc, n) \,||\, atmNet.debitAccount(acc, n)\rangle$$

$$internal \stackrel{dfn}{=} atmNet.internal$$

To summarise, we have outlined, using a highly simplified model of a banking ATM system, how decisions about evolutionary behaviour can be incorporated in a design in the proposed logical framework. The structuring of the design and the elements of the components are fairly natural for this application – communication between components is described by joint actions and evolution through meta-level components. The examples of this paper suggest that this revision-based logical framework provides a useful setting for the description of fairly complex evolvable systems.

5 Conclusions

We have shown how, starting with a simple revision-based logical description of a system, we can incorporate notions of evolution using meta-level logics. By introducing tree-structured logical descriptions and associated revision operations, we can extend the framework to evolvable systems built from hierarchies of evolvable components.

Although the examples we have developed within this framework are preliminary and application to large-scale design is yet to be undertaken, it is clear that the framework allows us to use evolutionary structuring as an integral part of the architecture of software systems, enabling evolution to be incorporated at various levels of a system design. This therefore is another dimension for software design in which design decisions about the appropriate evolutionary structure are required. Even for the small examples of this paper, there is a choice of where supervisory processes should be placed and what their evolutionary powers should be. This appears to open a range of interesting issues in software engineering.

It may be worthwhile to reflect briefly on the structure of the logical framework that we have proposed, not least because it may seem unduly complicated. One of the initial difficulties we found in formulating an account of evolvable systems was to get the level of abstraction correct. The actual mechanisms involved in internal monitoring (such as the 'insertion of probes') and in invoking evolutionary change (such as local termination and system decomposition), whilst possibly necessary for implementing such systems, are not at the right level of abstraction for software design and development methods. What we have managed to achieve with this framework is a logical description of individual components and the way they combine, and also of the relationship of supervisors to supervisees through the meta-view relations which provide a behavioral description of the effects of monitoring and of evolutionary intervention.

A key to the account is the introduction of 'configurations' which describe not only the current state of a system, but also its logical theory and the collection of its actions. What perhaps we haven't brought out in the above development is how the form of configurations is determined not only by the descriptive power of the specifications that we want but also by the capabilities of evolution. Each form of evolutionary step makes a change to the configuration. In order for this change to be described as a simple revision of the configuration we need to ensure that the relevant property of the system is suitably localised in the configuration. For example, a change to the capacity of the table in the blocks world example should not involve extensive 'editing' of action descriptions and parts of the supporting theory, but needs to be localised, in this case as a constraint in the object-level theory. Not only does this need to be localised but the way that these constraints are used in specifying the behaviour of a system needs to be through a naming of the formula – an 'abstraction' predicate – rather than use of the formula itself. By doing so, a revision which changes the formula but keeps its name allows the constraint to change but its use to persist.

Much of the intricacy in our account lies in needing to provide a meta-description of the (changing) logic of each object-level system. As is recognised in mathematical logic, and also in the provision of generic logical frameworks in automated reasoning, describing in detail the structure of a logical system is itself a non-trivial task. However, it is generic: that is, the the way that, say, formulae are built is, to a large extent, independent of the actual logic and its constants. Moreover, some of the actions of supervisors have the same genericity. For example, the 'observe' actions for monitoring system behaviour are of the same structure whatever system is being observed. A similar comment applies to 'evolve' actions invoked through *evolve* predicates defined in terms of object-level configurations. At the moment, this genericity is not exploited — it is inherent in the specification of one theory as 'meta' to another.

A further comment about the form of configurations: we use the same descriptive framework at the meta-level as we do at the object-level. That is, supervisors are described as systems in the same form as that of the object-level systems, with the same kind of states, revision actions and configurations. This enables us to build hierarchies of evolvable systems, with local supervisors able to monitor and evolve parts of systems, including other supervisors.

The introduction of component-based systems is a natural move in software engineering, and indeed the form of architecture for evolution that we propose lends itself readily to a component-based description. Moreover, introducing components has not altered the basic logical structure of evolution through meta-level processes. However, the overhead in structural and notational terms is quite heavy. The first change required was simply introducing a naming scheme for components alongside the tree hierarchy of components and of schema. Amongst several possible equivalent structures, it is not clear which is the best for describing systems built out of evolvable components. Each is fairly complex, but much of this complexity is unavoidable and arises from the very nature of component-based system construction.

It may be useful to summarise the new features which arise when components are introduced into the logical framework:

1. Revision is no longer the simple process of augmenting and/or restricting sets of formulae. Not only do states and configurations in the tree change, but the structure of the tree itself may change, and for evolutions involving general reconfigurations of a system, these tree manipulations can themselves be complex. Standard revision logics consider only simple revisions but, as we have shown, it is possible to extend the notion of revision to hierarchical systems.

2. Component-based systems are built from components which are either fixed (unevolvable) or consist of a pair of a supervisor and supervisee. In the latter case, what is any higher supervisor observing? The answer is the synchronous pairs of actions of the supervisor and supervisee. These *paired actions* consist either of (1) a normal computational step of the component together with a monitoring step, or (2) an evolutionary step invoked by the supervisor and the corresponding induced evolutionary action in the component.

3. Components may communicate *vertically* through supervisory processes, or *horizontally* with each other (including the possibility of supervisors communicating with each other) through *joint actions* which are defined as disjoint revisions on the two component states.

In this paper, we have introduced a logical framework. What we have not considered here is *programming* in this logical framework. That is, we have not considered languages and mechanisms for the sequencing of revision actions, nor have we considered how to describe conditional actions. For example, we may wish the supervisor action of *Expand* in the blocks world to be invoked only when the table is observed to be full by the supervisor, or instead, we may use historical information, for example, invoking *Expand* when the supervisor observes that the table has been full three times in the last ten actions, say. More generally, we may expect to use formulae of various temporal logics as conditions under which evolutionary actions may be invoked.

In a follow-up paper (Part II, in preparation), we present an account of programming for evolvable systems using a simple Guarded-Command language of revision actions. Each component comes equipped with its program and for each way that components combine, there is a combination of programs to form a new program for the combined system which expresses how the individual programs are to be sequenced. The particular combination of a supervisor with a supervisee expresses the monitoring and evolutionary capabilities of this combination. We have developed an SOS-style operational semantics for this language and also a trace-based denotational semantics. There is much further work in this area, including the incorporation of temporal logics, developing the relationship with runtime monitoring, and investigating specialist monitoring languages.

Another area which this paper opens up is the automated logical monitoring of evolvable systems. The logical framework, because it is based on revision logics, provides a *logical abstract machine* whose computational steps are revision actions on sets of formulae and, for component-based systems, on trees of states. A description of a system within the framework may then be 'run' on this abstract machine. Notice that for its implementation, the logical abstract machine needs access to automated theorem-proving tools to implement validity checks for preconditions, consistency checks on states, and to test the validity of meta-view relations.

How are system descriptions (with programs) in this logical framework related to actual evolutionary programs in, say, Java, and how do we pass from system specifications to Java programs which implement them? A further question is: Is there a notion of 'Evolve Java' in which the structuring mechanisms for evolution that we have introduced here are present as an extension of Java? We are considering some of these issues at the moment and also the relationship between the logical abstract machine and runtime verification of evolutionary programs (e.g. in Java).

Finally, the proof-theory which we have begun in this paper requires considerable development to provide a basis for us to reason about the behaviour of systems described within the logical framework and to establish the requisite control and safety properties of evolutionary systems.

A Appendix: Models and satisfaction for revision-based logics

In this appendix, we present a brief account of the logic underlying the logical framework of the paper. This introduces a non-standard notion of satisfaction which we define in terms of an 'observational' order on models, and minimum models under this order.

A revision-based logic describes states of a system in terms of sets of formulae, which we consider to be 'observations' or 'facts' about states. Models of these state descriptions are not simply models of the formulae. The descriptions are intended to be 'full descriptions' in a sense which we make precise.

Mathematically, this account is straightforward, but is very general: The closed formulae that serve as 'facts that may be observed' are arbitrary formulae. The form of the logic and that of the axioms is unrestricted and models are general set-based models.

A.1 Models and satisfaction

We consider a typed, first-order (classical) logic $\mathcal{L}$ with formulae $\mathsf{Form}(\mathcal{L})$ built from a signature of function and predicate symbols.

Set-theoretic models are standard:

Definition A.1 *Let $\mathcal{L}$ be a first-order typed logic. A* model *α of $\mathcal{L}$ is an allocation of a set $\alpha(T)$ for each type T of $\mathcal{L}$, a function $\alpha(f) : \alpha(T_1) \times \cdots \times \alpha(T_n) \to \alpha(T)$ for each function symbol $f : T_1 \times \cdots \times T_n \to T$ of $\mathcal{L}$, and a relation $\alpha(r) \subseteq \alpha(T_1) \times \cdots \times \alpha(T_n)$ for each predicate symbol $r : T_1 \times \cdots \times T_n$ of $\mathcal{L}$.*

For logics with equality, the equality symbol is treated as equality of elements in models. Where the logic allows *enumeration types*, these are interpreted strictly: Each declared constant c of an enumeration type T denotes a distinct element of $\alpha(T)$ and this exhausts $\alpha(T)$ i.e. $\forall x \in \alpha(T).\ \exists$ constant $c : T.\ \alpha(c) = x$.

The definition of the interpretation of a formulae $\psi \in \mathsf{Form}(\mathcal{L})$ in a model is standard. A closed first-order formula ψ is satisfied in a model α, written

$$\alpha \approx\!\!\!\!\models \psi,$$

iff the interpretation of ψ in α is true. We use this special symbol $\approx\!\!\!\!\models$ here as we reserve $\models$ for another satisfaction relation which we introduce later and which is used in the logical framework above.

We extend this to sets of closed formulae Ψ: $\alpha \approx\!\!\!\!\models \Psi$ iff for all $\psi \in \Psi$, $\alpha \approx\!\!\!\!\models \psi$.

For a set of closed formulae Ψ and closed formula ψ, we write

$$\Psi \approx\!\!\!\!\models \psi$$

iff for all $\mathcal{L}$-models α, $\alpha \mathrel{|\!\!\approx} \Psi \implies \alpha \mathrel{|\!\!\approx} \psi$.

For first-order typed theory W, we say α is a model of W (or α is a W-model) when, for all $\psi \in W$, $\alpha \mathrel{|\!\!\approx} \psi$. We write $\Psi \mathrel{|\!\!\approx}_W \psi$ iff for all W-models α, if $\alpha \mathrel{|\!\!\approx} \Psi$ then $\alpha \mathrel{|\!\!\approx} \psi$. For typed theories with equality, the standard rules for equality (equivalence and substitution) are assumed. For types described as enumeration types, the distinctness and exhaustiveness of the enumerations are assumed in the theory.

A.2 Observations and minimum models

Let $\mathcal{O}$ be a set of closed $\mathcal{L}$-formulae which we interpret as a collection of 'possible observations or facts' about a model ($\mathcal{O}$ may be empty).

Definition A.2 *For theory W of $\mathcal{L}$ and $\Delta \subseteq \mathcal{O}$, a W-model α satisfies Δ iff $\alpha \mathrel{|\!\!\approx} \Delta$. Let $\mathsf{Mod}_W(\Delta)$ be the set of all W-models that satisfy Δ.*

We define a pre-order $\lesssim$ on $\mathsf{Mod}_W(\Delta)$ by

$$\alpha \lesssim \beta \text{ iff } \forall \varphi \in \mathcal{O}.\ \alpha \mathrel{|\!\!\approx} \varphi \implies \beta \mathrel{|\!\!\approx} \varphi.$$

We now consider minimum models *in $\mathsf{Mod}_W(\Delta)$ under this pre-order, i.e. models $\alpha \in \mathsf{Mod}_W(\Delta)$ such that for all models $\beta \in \mathsf{Mod}_W(\Delta)$, $\alpha \lesssim \beta$, that is:*

$$\forall \varphi \in \mathcal{O}.\ \alpha \mathrel{|\!\!\approx} \varphi \implies \forall \beta \in \mathsf{Mod}_W(\Delta).\ \beta \mathrel{|\!\!\approx} \varphi.$$

Minimum models are "observationally equivalent", i.e. for two minimum models α and β,

$$\forall \varphi \in \mathcal{O}.\ \alpha \mathrel{|\!\!\approx} \varphi \text{ iff } \beta \mathrel{|\!\!\approx} \varphi.$$

However, minimum models need not be isomorphic or logically equivalent.

Minimum models need not exist – it depends on the theory W, the choice of observations $\mathcal{O}$ and the set Δ (see later).

We introduce another satisfaction relation, $\models$, defined using minimum models:

Definition A.3 *For W an $\mathcal{L}$-theory, $\Delta \subseteq \mathcal{O}$, for a set of closed $\mathcal{L}$-formulae $\mathcal{O}$, and ψ a closed $\mathcal{L}$-formula, write*

$$\Delta \models_W \psi$$

iff for all α minimum *in $\mathsf{Mod}_W(\Delta)$, $\alpha \mathrel{|\!\!\approx} \psi$.*

Note that $\Delta \mathrel{|\!\!\approx}_W \psi \implies \Delta \models_W \psi$.

A.3 Characterising minimum models

We now consider the existence of minimum models, first beginning with a characterisation theorem.

Theorem A.1 (Characterising minimum models) *Consider a theory W of $\mathcal{L}$ and $\Delta \subseteq \mathcal{O}$, for a set of closed $\mathcal{L}$-formulae $\mathcal{O}$. Define $\mathcal{T}(\Delta) \subseteq \mathsf{Form}(\mathcal{L})$, by*

$$\mathcal{T}(\Delta) = \Delta \cup \{\neg\varphi \mid \varphi \in \mathcal{O} \text{ and } \neg(\forall \beta \in \mathsf{Mod}_W(\Delta).\ \beta \mathrel{|\!\!\approx} \varphi)\}.$$

Then $\alpha \in \mathsf{Mod}_W(\Delta)$ is minimum iff $\alpha \mathrel{|\!\!\approx} \mathcal{T}(\Delta)$.

Proof. *If $\alpha \in \mathsf{Mod}_W(\Delta)$ is minimum, then for all $\varphi \in \mathcal{O}$,*

$$\alpha \mathrel{|\!\!\approx} \varphi \Leftrightarrow \Delta \mathrel{|\!\!\approx}_W \varphi$$

(by definition of minimum). Hence, for $\varphi \in \mathcal{O}$,

$$\Delta \not\mathrel{|\!\!\approx}_W \varphi \Rightarrow \alpha \mathrel{|\!\!\approx} \neg\varphi.$$

Thus $\alpha \mathrel{|\!\!\approx} \mathcal{T}(\Delta)$.

Conversely, suppose $\alpha \mathrel{|\!\!\approx} \mathcal{T}(\Delta)$. Then for all $\varphi \in \mathcal{O}$, if $\alpha \mathrel{|\!\!\approx} \varphi$ then $\Delta \mathrel{|\!\!\approx}_W \varphi$ (since, if not i.e. $\Delta \not\mathrel{|\!\!\approx}_W \varphi$ then $\alpha \mathrel{|\!\!\approx} \neg\varphi$ – contradiction). Hence, $\forall \beta \in \mathsf{Mod}_W(\Delta).\ \beta \mathrel{|\!\!\approx} \varphi$. Thus α is minimum.

Corollary A.1 (Existence of minimum models) *Consider a theory W of $\mathcal{L}$ and $\Delta \subseteq \mathcal{O}$, for a set of closed $\mathcal{L}$-formulae $\mathcal{O}$. A minimum model in $\mathsf{Mod}_W(\Delta)$ exists iff $\mathcal{T}(\Delta)$ is W-consistent, i.e. there is a model $\gamma \in \mathsf{Mod}_W(\Delta)$ with $\gamma \approx\!\!\!\!\!| \ \mathcal{T}(\Delta)$.*

We now characterise the satisfaction relation $\models$ in terms of $|\!\approx$. This provides a justification for the 'absence as negation' interpretation that may be imposed on observational descriptions.

Corollary A.2 (Characterising satisfaction for minimum models) *Consider a theory W of $\mathcal{L}$ and $\Delta \subseteq \mathcal{O}$, for a set of closed $\mathcal{L}$-formulae $\mathcal{O}$. Define $\mathcal{T}(\Delta) \subseteq \mathsf{Form}(\mathcal{L})$ as above (Theorem A.1). Then, for any closed $\psi \in \mathsf{Form}(\mathcal{L})$, we have*

$$\Delta \models_W \psi \ \textit{ iff } \ \mathcal{T}(\Delta) \mid\!\approx_W \psi.$$

Proof. Assume $\Delta \models_W \psi$, i.e. for all $\alpha \in \mathsf{Mod}_W(\Delta)$ minimum, $\alpha \mid\!\approx \psi$. But by Theorem A.1, α is minimum iff $\alpha \mid\!\approx \mathcal{T}(\Delta)$. Thus, for all $\alpha \in \mathsf{Mod}_W(\Delta)$ with $\alpha \mid\!\approx \mathcal{T}(\Delta)$, $\alpha \mid\!\approx \psi$, i.e. $\mathcal{T}(\Delta) \mid\!\approx_W \psi$.

Conversely, assume $\mathcal{T}(\Delta) \mid\!\approx_W \psi$, i.e. $\forall \alpha \in \mathsf{Mod}_W(\Delta)$. $\alpha \mid\!\approx \mathcal{T}(\Delta) \Rightarrow \alpha \mid\!\approx \psi$. But $\alpha \mid\!\approx \mathcal{T}(\Delta)$ iff α is minimum. Hence $\Delta \models_W \psi$ as required.

For application to evolvable systems, the sets $\Delta \subseteq \mathcal{O}$ are the 'states' of the system, where $\mathcal{O}$ is the set of ground atomic formulae built from the observation predicates of the theory. The form of satisfaction $\Delta \models \psi$ that we need is that defined above (Definition A.3) in terms of minimum models. The characterisation of this relation (Theorem A.2) in terms of standard satisfaction allows us to implement and automate the underlying logic of this logical framework for evolvable systems.

A brief note on this account: Other authors, for example [6], [21], [23], [10] and [22], have considered orders on models and minimum, or minimal, models but the logical settings and definitions of order differ from that here. When restricted to Herbrand models, the minimum models here (when they exist) coincide with minimal models under inclusion. Special cases of the construction of $\mathcal{T}(\Delta)$ from Δ occur in various accounts of 'circumscription' and 'augmentation' in logics for Artificial Intelligence (see, for example, in [6], [21] and [10]).

References

[1] D. Balasubramaniam, R. Morrison, G. N. C. Kirby, K. Mickan, B. C. Warboys, I. Robertson, R. A. Snowdon, R. M. Greenwood, and W. Seet. A software architecture approach for structuring autonomic systems. In *Proceeding of ICSE 2005 Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005)*, St Louis, MO, USA, 2005. ACM Digital Library.

[2] H. Barringer, G. Gough, D. Brough, D. Gabbay, I. Hodkinson, A. Hunter, R. Owens, P. McBrien, M. Reynolds, and M. Fisher. Languages, meta-language and METATEM, a discussion paper. *Journal of the IGPL*, 4(2):255–272, 1996.

[3] H. Barringer and D. Rydeheard. Modelling evolvable systems: A temporal logic view. In Artemov, Barringer, d'Avila Garcez, Lamb, and Woods, editors, *We Will Show Them! Essays in honour of Dov Gabbay on his 60th Birthday*, volume 1, pages 195–228. College Publications, 2005.

[4] S. A. Cook and Yongmei Liu. A complete axiomatization for blocks world. *J. Logic and Computation*, 13(4), 2003.

[5] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 2005 International Conference on Software Engineering*, St. Louis, Missouri, 2005.

[6] D. W. Etherington, R. E. Mercer, and R. Reiter. On the adequacy of predicate circumscription for closed-world reasoning. *Computational Intelligence*, 1:11–15, 1985.

[7] K. Eurviriyanukul, A. A. A. Fernandes, and N. W. Paton. A foundation for the replacement of pipelined physical join operators in adaptive query processing. In *Current Trends in Database Technology (EDBT Workshops)*, pages 589–600. Springer, 2006.

[8] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[9] D. M. Gabbay. *Fibring Logics*. Oxford University Press, 1999.

[10] M. R. Geneseret and N. J. Nilson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, 1987.

[11] M. P. Georgeoff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682, Seattle, WA., July 1987.

[12] R. M. Greenwood, I. Robertson, B. C. Warboys, and B. S. Yeomans. An evolutionary approach to process system development. In *Proceedings of the International Process Technology Workshop*, Villard de Lans (Grenoble), 1999.

[13] R. M. Greenwood, B. C. Warboys, R. Harrison, and P. Henderson. An empirical study of the evolution of a software system. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering*, pages 293–296, Honolulu, 1998. IEEE Computer Society Press.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[15] X. D. Koutsoukos, P. J. Antsaklis, M. D. Lemmon, and J. A. Stiver. Supervisory control of hybrid systems. In *Proc. of the IEEE, Special Issue on Hybrid Systems*, volume 88, pages 1026–1049, 2000.

[16] M. M. Lehman and J. F. Ramil. Software evolution: Background, theory, practice. *Information Processing Letters*, 88(1–2):33–44, 2003.

[17] P. Maes and D. Nardi (Eds). *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[18] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[19] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[20] R. Morrison, D. Balasubramaniam, G. N. C. Kirby, K. Mickan, B. C. Warboys, R. M. Greenwood, I. Robertson, and R. A. Snowdon. A framework for supporting dynamic systems co-evolution. *Automated Software Engg.*, 14(3):261–292, 2007.

[21] D. Perlis and J. Minker. Completeness results for circumscription. *Artificial Intelligence*, 28(1):29–42, 1986.

[22] P. Rondogiannis and W. W. Wadge. Minimum model semantics for logic programs with negation-as-failure. *ACM Trans. Comput. Logic*, 6(2):441–467, 2005.

[23] J. C. Shepherdson. A sound and complete semantics for a version of negation as failure. *Theor. Comput. Sci.*, 65(3):343–371, 1989.

[24] T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.