DisCo Specification Language: Marriage of Actions and Objects

Hannu-Matti Järvinen and Reino Kurki-Suonio
hmj@tut.fi, rks@tut.fi
Tampere University of Technology

Abstract

The potential of the action-oriented paradigm has been explored in the development of a new specification language DisCo, which can be characterized as both action-oriented and object-oriented. Its possibilities are introduced by contrasting them to the more familiar process-oriented approaches. Its execution model is state-based and leads to direct application of temporal logic in formal reasoning. Action-orientation allows a natural support for such forms of modularity that cut across process boundaries. At the same time, process-oriented abstractions are retained by object-orientation and the use of hierarchical statechart structures. The novel aspects of modularity are illustrated by a protocol example. The language is semi-executable, with properties that prevent automatic code generation in the general case. An experimental environment is available for simulation and animation of specifications.

Keywords: executable specifications, inheritance, joint action systems, modularity, object-oriented specifications, reactive systems, superposition

1. Introduction

Executable specifications of reactive systems are conventionally based on some notion of concurrent processes, like communicating automata in SDL and Estelle, or process algebraic processes in LOTOS. A more recent alternative is an action-oriented approach, as put forward in joint action systems by Back and Kurki-Suonio [3, 6] and in Unity by Chandy and Misra [10]. While all specifications are concerned with what is desired rather than how it is to be implemented, action-oriented specifications aim at describing the desired actions without indicating who is responsible for them, i.e., without allocating the associated responsibilities to some particular entities in the specification. Especially with distributed systems this decreases the bias towards particular hardware and software architectures.

The background of this paper is in joint action systems, where actions are executed jointly by one or more participants. This operational model is appealingly simple and may at first seem too primitive for explicit use as a language basis. Case studies ranging from toy problems to more realistic examples from industry demonstrated, however, their suitability for formal reasoning and rigorous development methods [3, 4, 21, 1]. Application of the basic ideas in a real-life industrial project also gave promising results [29]. However, these experiences revealed that careful language design, special support for structuring and derivation of specifications, and associated tools were needed to manage the complexity of large specifications. Based on the insight so gained, the DisCo language (for Distributed Cooperation) was developed [16], and an experimental environment was implemented to get practical experience of its use [28]. The main structuring ideas of the language have been described in [20, 17]; a more rigorous presentation of DisCo specifications can be found in [19], together with the derivation of a nontrivial example.

The action-oriented paradigm has been combined in DisCo with an object-oriented view of objects. Deviating from conventional object-orientation, the responsibility for actions (or "methods" in the object-oriented jargon) is not, however, assigned to individual objects but collectively to all participants. The basic ideas of DisCo are presented in this paper by contrasting them to the more familiar process-oriented approach. As different varieties of the latter we discuss LOTOS [8] and IP [14], two advanced process-oriented languages with different aims and theoretical bases. LOTOS (Language of Temporal Ordering Specification) is a recent ISO standard, based on process algebra and abstract data types. IP (Interacting Processes) is a notation for distributed systems, developed at MCC (Microelectronics and Computer Technology Corporation), and based on synchronous multiparty interactions. It should be emphasized that our intention is not to give a comparative evaluation of the three languages. Instead, we wish to provide insight to how action-orientation can be effectively utilized. The paper should be readable without detailed knowledge of any of these languages. A major part of it is devoted to a protocol example demonstrating some of the novel ideas for modularity.

Perhaps the most significant conclusions are the following. Firstly, it is possible to introduce effective structuring facilities without breaking the simplicity of the action-oriented execution model. Such forms of modularity then become natural that cut across process (or object) boundaries. Their use requires, however, a different way of thinking than traditional forms of modularity that have their roots in sequential programming. The reason is that control-oriented module interfaces are replaced by actions that are executed jointly by objects in different modules. Secondly, effective utilization of such interfaces leads to treating all systems as closed systems in which the environment is included. Although this is a widely accepted idea in the design of reactive systems [8, 30, 13], its application in DisCo seems to be novel. Thirdly, unlike in process-oriented approaches, a single notion of objects is sufficient to cover both active agents (processes) and passive components (data...
structures). This leads to uniform treatment of these entities, without an early partitioning of system state into control and data components.

The structure of the paper is as follows. The basic properties of DisCo are introduced in Section 2. Section 3 is devoted to issues of modularity. An example of protocol specification is given in Section 4, demonstrating some of the novel aspects in modular derivation of specifications. Summary and concluding remarks are given in Section 5.

2. Basic Properties

The general state-based approach of Manna and Pnueli [25] can be directly taken as an operational model for reactive systems. With this model in mind the basic properties of DisCo will be introduced by contrasting them to corresponding facilities in process-oriented languages, with special reference to LOTOS and IP.

2.1. Execution Model

Operational models for these languages are basically very similar. All assume an interleaved execution of multiparty actions in which several processes - or objects in DisCo - may participate. (This applies also to LOTOS, where actions are syntactically two-party interactions. Further participants can, however, be added in nested parallel composition, and one-party actions can also be expressed with the aid of auxiliary internal processes.) Some differences are, however, worthwhile to notice already at this general level.

All events that can take place in a system are explicitly defined atomic actions in DisCo. In particular, there are no implicit state changes within single objects. In contrast, control structures like guarded constructs in LOTOS introduce local actions for processes. The semantics of such implicit actions need not be straightforward, as can be seen from the discussion of team enrolment in [14].

Another area where basic differences can be noticed is liveness specification. Both LOTOS and IP assume implicitly the fundamental liveness property that "if something can happen, then something will happen". No facilities are available to express further fairness requirements, although it is assumed in IP that specific fairness notions (like weak or strong fairness with respect to actions) be applied to achieve the desired liveliness properties. In DisCo, all liveness requirements have to be expressed explicitly in terms of strong fairness with respect to actions. Even the fundamental liveness property is not automatically assumed, since DisCo specifications describe closed systems, and it is not reasonable to make implicit liveness assumptions about environment behavior. Notice that, in contrast to programming languages, a specification language need not be restricted to fairness notions that can be automatically enforced.

The simple interleaved execution model is obviously too simplistic for implementation. There is, however, a closely related but more realistic operational model where the duration of actions is modeled as follows. Instead of treating each action as a single atomic event, \( n+1 \) separate events are taken for each \( n \)-party action: one for its synchronized beginning, and one for its (non-synchronized) ending for each participant. All state changes associated with an action are assumed to take place in the begin event, but each participant is prevented from participating in further actions before its end event. Fortunately, without fairness requirements the two models are equivalent, and the simpler model is also sufficient for proving system properties under which this equivalence holds for a given system. Therefore, reasoning on DisCo specifications can always assume the simpler model. This idea of two related operational models, developed in [5, 6], has also been adopted in IP.

2.2. Global State

In process-oriented languages the global state of a system consists of the currently operating processes and their local states. The local state of a process consists of its control state and of its local variables. Control state is implicit and is not accessed or updated similarly to local variables.

DisCo partitions the global state into local states of objects, each object being an instantiation of a class. No implicit control state is included, i.e., all state changes are explicitly given in actions. Objects may, but need not, model active agents of execution. Independently of whether viewed as active agents or passive data structures, objects can be structured as hierarchical statecharts [15] with variables associated with states. (We only take the structuring of statecharts and omit their communication facilities. When interpreted as passive data structures, our statecharts can be understood simply as records of tagged union types.) As discussed by Bolognesi and Brinksma in [8], there is usually a trade-off between process and type definitions. The aim with DisCo classes and statechart structures is to postpone such considerations from specification to later stages of design.

No global names are used of DisCo objects. Actions are the only units of execution, and within each action the local states of its participants can be referenced with names that are local to the action, as it will be explained in the sequel.

As far as data types are concerned, IP leaves them to an underlying expression language, whereas LOTOS uses abstract data types. The approach of DisCo is relatively straightforward but flexible. In addition to simple basic types, references to objects are available, as well as sequence and set types. These facilities are needed for expressing relations between objects in a natural way. On the other hand, conventional structured data types are omitted. Arrays are omitted as low-level structures, and the need for record structures is satisfied by object classes and by special stuff types, which will be discussed below.

Instead of instantiating new objects during execution, as processes are created in LOTOS and IP, all DisCo objects are created once and for all before execution. This simplifies actions, which then need no facilities for generating or destroying objects. This means no restriction, either, since an infinite number of objects can be assumed to be initially in a state that corresponds to "not yet generated". The initial state (including the number of objects) need not be specified explicitly; only assertions are given that must be satisfied. A DisCo system therefore stands for a whole class of concrete systems, which allows
to express and prove properties that are common to this class. Obviously, simulation requires the generation of a particular system of this class.

2.3. Actions

As was already mentioned, process-oriented approaches tend to divide actions into two fundamentally different classes: explicit process interactions, and those actions that are implicit in the control structures of processes. The latter are identified in LOTOS and IP by interaction names. Their execution arises dynamically from matching action denotations in the participating processes, and the effects are determined by these denotations collectively.

The effect of an interaction is basically a sequence of interprocess assignments. In LOTOS these assignments are based on a message passing paradigm, where each participant either offers a value, or requests one, whereas IP uses explicit assignments. Non-deterministic assignments can be expressed in LOTOS by letting each participant request a value, in which case a suitable value is generated.

In both LOTOS and IP an interaction denotation may have conditions that determine whether a process will accept the interaction or not. The guard of an interaction is the conjunction of these conditions, together with implicit conditions on the program counters of the participants. For efficient implementability, the guarding conditions are restricted in IP to be local to the processes, while LOTOS allows more general conditions that may depend on the messages that would be received if the interaction would take place. If several alternative interactions are possible at the same time, the selection between them is non-deterministic.

In DisCo all actions are defined as separate syntactic entities with the following format:

```
action <action name> <optional parameter list> by
  <list of participant roles> is
  when <guard> do
    <body>;
  end;
```

When executed, an action has some objects as participants. In an action definition the participants are represented by formal names called roles. The required object class is specified for each role, and one object cannot simultaneously assume more than one role.

The optional parameter list introduces formal names and types for action parameters. These are non-deterministic values that are generated in action execution similarly to value generation in LOTOS interactions. As in LOTOS, this non-determinism may also be unbounded.

The guard is an enabling condition that determines whether an action is enabled for a collection of potential participants and parameter values. The selection of the next action to be executed is non-deterministic between different actions and between all collections of actual participants and parameter values for which they are enabled.

The scope of parameter and role names does not extend beyond an action. In addition to parameters and local states of participants, the guard may refer to other objects by quantification over object classes. Therefore, although an action always has a fixed number of participant roles, any number of objects may participate in the evaluation of its enabling condition. For instance, a guard may state the condition that the value of a local variable in a participant is the maximum of these variables in all objects of this class.

The body is essentially a sequence of (possibly conditional) assignments within the combined local states of the participants. (Special syntax is used for "assignments" that correspond to state transitions in the statechart representation.) No non-deterministic constructs are available in the body, since the parameter mechanism gives the same effect. In addition to allowing alternative effects for actions, parameters provide an effective mechanism to reduce this non-determinism in action refinement, which will be discussed below.

Comparing to process-oriented approaches the most important differences are that all actions are treated in a uniform way, and that action guards and bodies are not distributed to (possibly several places) the participants. All actions, including their guards, are explicitly given syntactic entities, which is an essential advantage for reasoning. With IP we share the avoidance of explicit communication, and with LOTOS the parameter mechanism and the possibility for implicit guards for which algorithmic implementations cannot be automatically derived.

Since actions are syntactic units, their names have no significance in DisCo execution. They are needed, however, in action refinement, which will be discussed below.

For complex specifications objects and actions also provide a natural basis for graphical visualization. While objects can be understood as hierarchical statechart structures, actions provide the transitions for them. DisCo language is textual, but graphical visualizations can be synthesized from the text, and their level of detail is easy to adjust. During execution the notions of objects and actions also lend themselves to animation. For instance, the selection of actions and their participants can be displayed, together with the most interesting parts of object states, including dynamic relations between them.

3. Reasoning and Modularity

3.1. General

The backgrounds of the three languages are widely different. The roots of IP are in distributed programming languages, but its design has been heavily influenced by research on associated proof methodologies. LOTOS, on the other hand, is based on CCS process algebra and its equivalence relations. However, with data structures and value communication this formal foundation cannot be effectively utilized in full LOTOS.

The semantics of all three languages are defined in terms of state transition systems. Based on joint action systems, DisCo was explicitly designed for the state-based approach of temporal logic [25, 26], where the identity of actions and the values communicated have no significance. (For a general presentation and discussion of this approach, see Lamport [23].) Recently, Lamport has put forward a temporal logic of actions [24], which turns out to be a natural vehicle for expressing
properties of DisCo systems and reasoning about them. The close relationship makes it possible to understand DisCo as a language to construct specifications in this logic, with an object-oriented structuring of state, and with syntactic support for modularity [16].

As already mentioned, DisCo specifications are understood as closed systems, i.e., systems that include their environments. In other words, the environment is described in the same language, and this description is available for both simulation and formal proofs. When environment behavior is unrestricted, a “dummy” environment needs no extra effort from the specifier, as will be seen below.

Processes are the units for abstraction and modularity in LOTOS. IP is richer in this respect, having teams for abstracting joint behaviors of groups of processes, and a superposition construct that supports layered design of systems. The facilities of DisCo resemble those in IP, but due to a simpler execution model their semantics is simpler.

Within objects, the hierarchical state structure of DisCo gives rise to levels of abstraction. These can be utilized in layered design of systems, as well as in proofs and graphical visualization. Units for modularity are, however, always complete systems, which consist of class and action definitions, together with assertions that restrict the initial state. Such modules can be combined with each other and enhanced with further properties, preserving the crucial properties that have been proved of each module.

The single mechanism for using previously given systems is to import them to new systems. Any number of systems can be imported, and these can also share definitions from common imported systems. An important property of this mechanism is that a system can be projected to each of the imported systems analogously to how projection was used by Lam and Shankar in [22].

To be more specific, if system A is imported to B, then the state of (each instance of) B is an extension of (some instance of) A. Therefore, as far as the state is concerned, projection simply removes these extensions. Similarly, when all effects on state extensions are removed, each action in B projects either to some action in A or to an implicit stuttering action (which is always enabled and, when executed, does not change the state). In this way computations can be projected to computations in each of the imported systems.

Associated with this mechanism, there is a facility to restrict the visibility of imported actions and classes so that not all of them can be modified. Several disjoint visibility lists can also be given. This is useful, for instance, in describing interfaces, of which different actions and classes are modifiable in different directions.

3.2. Superposition

Superposition (or superimposition) is a layered design method that is especially suited to distributed systems [11, 18, 10, 9]. In DisCo this is supported by allowing an imported system (or a combination of several imported systems) to be refined as follows. System state can be modified by extending previously given class definitions and by adding new classes. Old actions can be refined by strengthening their guards and by allowing them to have new parameters and participants, and to update the newly added state extensions. New actions can also be added that do not affect the old state. These possibilities give, in fact, a syntactically restricted form of a more general refinement relation [2, 7], and the language guarantees that all safety properties of the original systems are preserved.

Notice that superposition effectively removes the restriction that a class or action definition is given in a single place. Unlike process-oriented partitioning of actions, this supports modular development and proofs. It is also possible to perform independent superposition steps in parallel and combine the results later.

When an object is understood as record-structured data, superposition allows its easy extension with new fields. In order to allow similar possibilities for action parameters, special stuff types are included. When a stuff type is introduced, no definition is given for it, but when imported to another system it can be specialized into a record, which may again have components of further stuff types. As will appear below, this facility is useful in protocol specification, where message structures are specialized when going from lower protocol layers to higher ones.

Although superposition is a natural construction for action-oriented languages, it can also be included in process-oriented languages, as is shown by the corresponding facility in IP. The distribution of actions in processes means, however, that the modifications introduced by superposition are less local and essentially more complicated to express.

No corresponding constructs are available in LOTOS. It is interesting to notice, however, that the addition of data structures and value communication to basic LOTOS can be viewed as a special application of superposition. Similarly, the constraint-oriented specification style [8] can also be understood as superposition, where new processes are added to participate in interactions, with associated strengthening of enabling conditions.

3.3. Inheritance

While superposition is essentially a method for top-down development, a related bottom-up method works in DisCo as follows. When a system has been imported, new classes can be defined that inherit an imported class. An object of such a class then contains a subobject of the inherited class. Such a subobject is an object in its own right in the sense that it can assume roles in (imported) actions. Imported actions can also be specialized to take the context of inheritance into consideration.

A novel aspect of DisCo inheritance is associated with the fact that it is not restricted to the outmost state level of objects; inheritance to substates is also allowed. This is useful for situa-
ations where objects are expected to go through several stages during execution, and the inherited capabilities (for subobjects to participate in imported actions) should be active in particular stages only.

Inheritance to substates makes, however, projection into imported systems slightly more complex. When this facility is used appropriately, computations can be partitioned into phases within which projection to an imported system is possible and the safety properties of this system are preserved. However, an additional proof is needed that the conditions for appropriate use are satisfied. Such situations and proofs methods have been considered by Elrad and Francone in [12] and by Stomp and de Roever in [27] under the names "communication-closed layers" and "sequentially phased reasoning".

Superposition and inheritance are complementary to each other in DisCo, and they can also be used in combination. It is interesting to notice that the purpose of the team construct in IP is very close to that of DisCo inheritance, even though a similar dualism between superposition and teams is not apparent there. When an IP process enroles in a team, it commits itself to interactions that have been defined within that team. Analogously, when a DisCo object enters a state into which some class has been inherited from an imported system, the actions of that imported system become available for it. Notice that IP has essentially the procedure mechanism of sequential programming and generalized it for multiple participants: each participant calls a role in a team and is exclusively committed to the team until this role has been completed. Inherited actions impose no similar sequentiality in DisCo: when in a stage where inheritance allows the execution of imported actions, an object is not exclusively committed to these but may also participate in other actions in an interleaved manner.

3.4. Combination of Modules

While superposition and inheritance support certain methods of refinement, a facility is also needed to compose systems of components. LOTOS processes, for instance, can be combined to form new processes, and communication between component processes is achieved by equating some (non-hidden) interaction names in them.

As was mentioned above, the modules of DisCo are complete systems, which can be combined by importing them to a new system. Communication between subsystems is achieved by letting their objects participate in each other's actions, which can be expressed by the technique of superposition. For instance, if module $A$ has an action where a parameter denotes an arbitrary input value, this parameter can be bound to the value of some variable in module $B$, when $A$ and $B$ are combined. Actions in $A$ and $B$ can also be combined (similarly to how interactions are combined in LOTOS), in which case the combined action represents an event that affects both subsystems. Notice that each action can always be projected to all imported (sub)systems. The default is that such a projection is a non-stuttering action in just one subsystem, but in the case of combined actions there are non-stuttering projections in several subsystems.

4. Example of Protocol Specification

This section is devoted to an example that describes a simple protocol with layer structure. The focus is not on the protocol itself, but on the structure of the specification, which illustrates modularity achieved by closed systems, superposition, specialization of stuff types, and combination of both independent and parallel derivations.

The modules in this example are illustrated in Figure 1, the arrows indicating importation and superposition. Upper is a module that describes communication at a level of abstraction where reliable communication can be assumed. Lower is an otherwise similar module, but its channels may also loose messages. High is a simple application module that makes use of the reliable communication services provided by Upper. Mid is an implementation of Upper by using the unreliable communication facilities of Lower. Finally, High_and_Mid is a module formed by combining both High and Mid; it describes the combined behavior of the application and the communication system, including both the underlying unreliable channels and the abstraction provided by the reliable channel.

4.1. Upper Interface

Module Upper describes the behavior of Mid as seen from the environment, i.e., it describes error-free simplex point-to-point communication between two parties. In addition, since all DisCo systems are closed, it needs to describe unrestricted environment behavior.

Three object classes are introduced:

```
class channel_mid ls
  state idle, busy(data: message); Initially idle;
end;
class sender_mid(n: integer; to: channel_mid) ls
  assert $n > 0 \land$ to $\neq$ null;
  queue: sequence message; Initially queue $= <>$;
end;
class receiver_mid(n: integer; from: channel_mid) ls
  assert $n > 0 \land$ from $\neq$ null;
  queue: sequence message; Initially queue $= <>$;
end;
assert $\forall c: channel_mid$
size(s: sender_mid | s.to $= c) = size(r: receiver_mid | r.from $= c) = 1$;
```

A channel_mid object represents a reliable channel between sender and receiver objects. It has two exclusive states: either it is idle, which is the initial state, or it is busy with a message that is kept in the associated variable data. The parameter notation for data indicates that its value cannot be changed without exiting and re-entering this state.

![Figure 1. Derivation of a specification.](image-url)
Each `sender_mid` has two parameters whose values are set at initialization: `n` is the capacity of the associated buffer, and `t` is the associated channel. In addition, it has a variable containing the buffer itself, i.e., a sequence of items of type `message`. Correspondingly, each `receiver_mid` is associated with information on its buffer size, the associated channel, and the contents of the buffer. All buffers are initialized as empty.

The assertions state that each sender and receiver has a buffer and is associated with an existing channel, and that each channel is point-to-point between one sender and one receiver. No restrictions are given on their number. Figure 2 illustrates the case where only one channel exists.

Four actions are introduced; in Figure 2 they are shown as arrows indicating the direction of message flow between participants:

- **Action `enter_mid`**
  - Parameters: `m: message`
  - Body:
    ```
    action `enter_mid` (m: message) by sm: `sender_mid`
    when size(sm.queue) < n do
      sm.queue := sm.queue & m;
    end;
    
    action `send_mid` (m: message) by cm: `channel_mid`
    when sm.to = cm & size(sm.queue) > 0 & cm.idle & m = head(sm.queue) do
      cm.busy(m);
      sm.queue := tail(sm.queue);
    end;
    
    action `receive_mid` (m: message) by rm: `receiver_mid`
    when rm.from = cm & size(rm.queue) > 0 & cm.idle & m = head(rm.queue) do
      rm.queue := rm.queue & m;
      cm.idle;
    end;
    
    action `exit_mid` (m: message) by rm: `receiver_mid`
    when size(rm.queue) > 0 & m = head(rm.queue) do
      rm.queue := tail(rm.queue);
    end;
    
    Action `enter_mid` describes an interaction where a message is accepted by a `sender_mid` object. Parameter `m` denotes an arbitrary message that is generated in this action. The action is enabled if the buffer capacity has not been exhausted, and its effect is to append the message to this buffer. The type message is left arbitrary, but it can be specialized to concrete message types later, as will be shown below.

Whenever there are messages in a sender buffer and the associated channel is idle, action `send_mid` may put the next message into the channel. In order to allow easy reference to the message in later refinements of this action, it is given as an action parameter, even though its value is uniquely determined by the guard. The arrow in the action body denotes a state transition statement where the channel enters state busy, with the value of its state parameter set to the message. Correspondingly, whenever there is a message in a channel and the buffer of the associated receiver is not full, action `receive_mid` may append the message to this buffer and cause an associated state transition in the channel. The message is again given as an action parameter in order to allow easy reference to it later.

Finally, action `exit_mid` describes an interaction where a transmitted message is given to the environment in the receiving end. Again, the message is a parameter whose value is uniquely determined by the guard.

Liveness assumptions are also needed to guarantee that messages are, in fact, sent and received. They are given here by prefixing the sender and receiver roles in `send_mid` and `receive_mid` by a star. This indicates the requirement that these actions cannot be infinitely often enabled for the same sender (or receiver) without being infinitely often executed.

The whole DisCo system that specifies the upper interface can now be given as a module with two export lists, one for each direction:

```plaintext
module Upper
  export Up is
    stuff message; action `enter_mid`, `exit_mid`;
  end;
  export Down is
    class `channel_mid`, `receiver_mid`, `sender_mid`;
    action `send_mid`, `receive_mid`;
  end;
  -- class and action definitions to be inserted here
end;
```

The list `Up` indicates that `message` is a stuff type that can be specialized when this list is used, and that actions `enter_mid` and `exit_mid` can then also be modified. Similarly, the list `Down` gives those classes and actions that can be modified in implementing the requirements that `Upper` imposes on message transmission.

Module `Upper` is now a complete system that can be simulated or reasoned about as a closed system. In particular, actions `enter_mid` and `exit_mid` are well defined without any additional environment, but they have been designed so that communication with other modules can be achieved by utilization of action parameters.

### 4.2. Application Layer

To demonstrate how module `Upper` can be used by application modules, we consider a simple situation with one sender that generates consecutive integers, and one receiver that keeps computing their sum. In principle, we have two possibilities. Either we give the application as an independent module and combine it later with `Upper`, or we write it using `Upper` explicitly. Here we take the latter approach.

Module `Upper` is imported with export list `Up`, and the application is added to the system by superposition. Since the messages are to be integers, the stuff type `message` is specialized
requirements which (together with the fairness assumptions on
Figure 3 indicates the structure of
variable
hypens begin comments. The sender of this level is added,
ceiving of all messages.
also refined to update the sum. Both actions are given fairness
actions
are refined
to be inserted here
New class definitions are introduced to describe the sender and
objects of the application. Both need knowledge
about the object with which to communicate in
dition, the sender needs a variable for the integers to be gener-
ated is restricted to

Ellipses are a syntactic notation of DisCo denoting particip-
ants, guards, and bodies taken from imported actions; double
hyphens begin comments. The sender of this level is added, as
another participant, to send_high. The value to be communi-
cated is restricted to be the value of i within send_high, and
variable i is also incremented. Correspondingly, the receiver of
this level is added to participate in receive_high, which is
also refined to update the sum. Both actions are given fairness
requirements which (together with the fairness assumptions on
send_mid and receive_mid) guarantee the sending and re-
ceiving of all messages.
Figure 3 indicates the structure of High, including the import-
ed upper interface. The non-specialized actions enter_mid and
exit_mid are still available for other specializations. This
means that the same channel could be used for other purposes
also. Since different versions of enter_mid and exit_mid only
deal with the associated specializations of the message type,
there is no danger of different kinds of messages being con-
fused. Actions send_mid and receive_mid, on the other
hand, do not distinguish between such specializations, and
they transmit all messages regardless of their structures.
It is interesting to compare module interfaces in DisCo, as illus-
trated by this simple example, to conventional procedure
interfaces. The services of Upper are not used by transfer of
control between High and Upper, but by sharing and modifi-
cation of actions that have been defined in Upper. The basic
control-less character of the execution model is therefore pre-
served.

4.3. Lower Interface
Module Lower, which is illustrated in Figure 4, is similar to
Upper, except that there is no buffering, and messages may be
lost arbitrarily by a drop action:

module Lower is
  export Up is
    stuff frame; action enter_low, exit_low;
  end;
export Down is
  class channel_low, sender_low, receiver_low;
  action send_low, receive_low, drop;
  end;
class channel_low is
  state idle, busy(data: frame); Initially idle;
  end;
class sender_low(to: channel_low) is
  state *to = null;
  state *no_data, sending(data: frame); Initially no_data;
  end;
class receiver_low(from: channel_low) is
  state *from = null;
  state *no_data, received(data: frame); Initially no_data;
  end;
assert ∀ c: channel_low:
  size(s: sender_low | s.to = c) = size(r: receiver_low | r.from = c) = 1;
  action enter_low(frame) by st: sender_low is
    when st.no_data do
      → st.sending(frame);
    end;
  action send_low(frame) by st: sender_low; cl: channel_low is
    when st.to = cl.idle ∧ I = st.sending.data do
      → cl.busy(frame) → st.no_data;
    end;
Figure 4. Objects and actions in the lower interface.
Receivers and senders of this level have been modeled to have two alternative states, depending on whether there is a message or not. The message type is given by the stuff type frame. The double bar (\(|\)\) is a combinator for parallel assignments or state transitions. Fairness requirements are given for send_low and receive_low in the same way as was done in Upper.

### 4.4. Mid-Layer

The most interesting part of the example is Mid, which demonstrates the combination of two independent modules. The idea is to implement the Down part of the upper interface by using services provided by the lower interface. Therefore, both Upper and Lower are imported. Communication between these two subsystems demonstrates the possibilities described in Section 3.4.

To implement reliable communication, Lower is used both for actual messages and for acknowledgments, and alternating bits are added to both to distinguish between consecutive messages. This leads to the following structure for Mid:

```plaintext
system Mid
import Upper(Down), Lower(Up);
use frame(bit: boolean; msg: message) as specialization for actual messages
rename enter_low as send_low, exit_low as receive_low;
use frame(bit: boolean) as specialization for acknowledgments
rename enter_low as send_ack, exit_low as receive_ack;
combined receive_thru_low of receive_mid, receive_msg;

class and action definitions to be inserted here

end;
```

When actions from different imported systems are independent, as is the case here, the default is that they are all kept as separate actions. Here we wish, however, an action receive_thru_low to possess the functionalities of both receive_mid in Upper and exit_low in Lower. Such a combined action has all the parameters and participants of its component actions, the guard is the conjunction of their guards, and the body has the effect of both bodies. Such combination is obtained by the `combined` clause.

No new classes are needed in Mid, but sender_mid and receiver_mid need extension. Both need a variable for their current values of the alternating bit, and knowledge about the objects with which to communicate in Lower. In addition, the former is extended with two internal states to indicate whether there is a message to be sent for which an acknowledgment has not been received:

```plaintext
extend sender_mid(... sl: sender_low; rt: receiver_low) by
assert sl = 0 \land rt = 0;
state idle, sending(msg: message); initially idle;
bit: boolean; initially bit = true;
end;
```

The assertions state the structural requirement that Lower is used to provide two-way communication between sender_mid and receiver_mid objects.

The sending of a message in Upper is now refined to include a state transition that prevents repetition of this action until an acknowledgment has been received:

```plaintext
refined send_low is -- derived of send_mid(m) by *sm, cm
when sm.idle do

... sm.sending(m);
end;
```

This should enable send_msg with the correct message and alternating bit value. This is achieved by the following refinement that is repeatedly executed (modeling a timeout) until an acknowledgment is received:

```plaintext
refined send_msg by ...
-- derived of enter_low([bit, msg]) by sl where f = [bit, msg]
when sm.sl = s \land f.bit = sm.bit \land f.msg = sm.sending.msg do

... sm.idle do

... sm.sending(m);
end;
```

Even though a combined action has been formed of receive_mid and receive_msg, these two still exist as individual actions also. The former has to be removed by strengthening its guard to be identically false. The combined action, which corresponds to the receipt of a message through both channels, is refined as follows:

```plaintext
refined receive_thru_low is
-- derived of receive_mid(m) by *rm, cm
-- and of exit_low([bit, msg]) by rt where f = [bit, msg]
when rm.rt = r \land r.bit \land m = f[msg do

... rm.bit \land ~ rm.bit;
end;
```

The receipt of a message through the lower-level channel only remains to describe receive of redundant copies of a message:

```plaintext
refined receive_msg by ...
-- derived of [bit, msg] by rt where f = [bit, msg]
when rm.rt = r \land r.bit = rm.bit do

... sm.idle do

end;
```

Finally, correct sending and receiving of acknowledgments is obtained by refining send_ack and receive_ack:
The former are based on dication of concurrency, whereas the latter are based on semantic definition of both process-oriented and action-oriented paradigm and of statechart structures. Combining and unnecessary indexing, led to the inclusion of the object-oriented languages, their approaches to concurrency are opposite. Although similar interleaved execution models are used for the semantic definition of both process-oriented and action-oriented languages, their approaches to concurrency are opposite. The former are based on sequential processes, with explicit indication of concurrency, whereas the latter are based on full concurrency (i.e., arbitrary interleaving of actions) that is restricted explicitly. The fundamental question in the design of DisCo was, whether the latter approach could be made manageable with structuring facilities that would not impose non-essential sequentiality. Such facilities have been described in this paper, showing that action-orientation gives rise to different forms of modularity than what are natural for either process-oriented or conventional object-oriented approaches.

Earlier applications of action-orientation [3, 10] have developed modularity at the cost of losing process-oriented abstractions, which also seem useful in the modeling of real-life problems. IP tries to avoid this drawback by incorporating similar ideas in a process-oriented language. For specifications, as opposed to programs, this seems like treating symptoms instead of the disease. With DisCo we claim to have demonstrated that (for specifications) the advantages of processes are not in their implicit control states. The same modeling capabilities can be achieved without control states by objects that are described separately from actions, using statechart structuring to obtain process-like abstractions with states and transitions. At the same time, such objects provide a uniform treatment of data structures and processes, with the object-oriented ideas of encapsulation, classes and inheritance.

Another novel characteristic in DisCo is its closed system principle, which fits well with the new constructs for modularity. Action parameters are an essential mechanism in this connection, as their nondeterminism provides a convenient way to describe unconstrained environments, and they also provide a handle to restrict this nondeterminism.

The DisCo language has been designed so that the associated modular development methods are effectively supported by its syntax and static semantics. Enhancement by superposition, reuse by inheritance, and combination of both independent systems and of parallel derivations can therefore be considered more as language mechanisms than development methods. This simplifies the task of formal verification essentially.

Designed to be a specification language, DisCo has possibilities that exclude automatic code generation in the general case. Implicit conditions in action guards, unbounded nondeterminism, and fairness specification clearly belong to this category. Even without them, the decision not to specify how responsibilities in guard evaluation should be distributed is an obstacle for efficient general implementation. Notice, however, that the development methods associated with the language can be used to refine a specification into a design that does allow efficient implementation on a given architecture by automatic means. An example of such a derivation and associated hand-translation into CSP with output guards was given in [3].

A prototype of an early language version, with a visualization and animation environment is currently in experimental use, and a new version of the language is being implemented. Directions of continued work include formal definition of the language, its experimental use in industry, addition of capabilities for real-time modeling, and further research into formally based specification methodologies.
Acknowledgments

The joint action approach was developed together with Ralph Back from Åbo Akademi. Research on its use as a basis for a practical specification language has been carried out in project DisCo at Tampere University of Technology. This ongoing project is part of the FINSOFT programme of the Technology Development Centre of Finland (TEKE5), and is supported by four industrial partners. This paper would not have been possible without the work of other project members, Tauno Kankaanpää, with whom the early language ideas were developed, Olavi Eerola, who did much of the implementation work, and Kari Systä, who developed and implemented the ideas for visualization and animation.

References


