Refining Reactive Systems in HOL using Action Systems

Thomas Långbacka

Department of Computer Science, University of Helsinki

Joakim von Wright

Department of Computer Science, Åbo Akademi University

Turku Centre for Computer Science
TUCS Technical Report No 102
March 1997

ISBN 951-650-982-7
ISSN 1239-1891

Abstract

This report discusses how to refine reactive systems using the HOL theorem prover. We show how Action Systems – the formal framework supported – can be formalised in HOL. We describe a simple refinement example. We also discuss how the work presented here can be used in connection with the Refinement Calculator, a tool supporting program refinement.

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Reactive systems have been studied with great interest in the formal methods community during recent years. Many different formal frameworks have been suggested for dealing with such systems.

A reactive system is a program that interacts with an *environment*. Such a system observes the activities in the environment and reacts to certain types of events. Typically reactive systems are modelled as non-terminating programs, and also the environment is often described in the programming notation used. An operating system could be seen as an example of a reactive system. Its job is just to react to commands from its environment – its users. Although an operating system might be terminated and the computer shut down, it still has a state that lives on, e.g. the file system.

In this paper we will document work on formalising *action systems* in HOL. Action systems can be used to model parallel and reactive programs. In [5] a *trace semantics* and a corresponding refinement relation (trace refinement) are defined for action systems and it is shown that a simulation method can be used to prove trace refinement. Furthermore, simulation reduces reasoning about action systems to reasoning about sequential programs using the framework of the *Refinement Calculus* [2,3]. Thus, in order to formalise action systems in HOL we can build directly on the existing work on formalising the refinement calculus in HOL [4, 11].

Reactive systems can (and have) be treated in HOL formalisations of such formal frameworks as CSP [7], UNITY [1] or TLA [12]. We will not argue for or against the respective merits of any of the above systems (or their HOL incarnations) as compared to action systems. It is well known that working with HOL formalisations of programming logics is usually very difficult. Since reactive systems are often complex, this does not make the task easier.

One main motivation for our work is that it will be integrated into the *Refinement Calculator* [6], a tool for systematic development of sequential programs within the Refinement Calculus framework. By extending the Refinement Calculator with facilities for handling action systems, we get a tool that allows a user to specify and refine reactive programs using a graphical user interface and with every development step verified by a proof in the HOL theorem prover.

2 Background

In this section we give a brief description of those parts of the Refinement Calculus and its formalisation in HOL that are relevant to this paper.

2.1 The Refinement Calculus

The Refinement Calculus [2, 3] extends Dijkstra's weakest precondition semantics for programs [8] by introducing a refinement relation between programs and by adding specification statements that allow specifications to be expressed within the programming notation.

The semantics of a statement S is defined by the function wp which given a statement S and a predicate Q (the postcondition) returns $wp(S, Q)$, the weakest predicate P such that executing S in an initial state where P holds guarantees that S terminates and Q holds upon termination.

We will not assume a specific program notation; in the examples we use (multiple) assignments of the form $x := E$. Other constructs will be described as they are needed.

Refinement The refinement relation is defined in terms of the weakest preconditions of the related programs. Program S_0 is refined by S_1 iff

$$\forall P.\ wp(S_0, P) \Rightarrow wp(S_1, P)$$

Essentially, this means that program S_1 must satisfy all total correctness properties that program S_0 satisfies. The refinement relation (denoted $\leq$) is a preorder. Thus programs can be developed in a *stepwise*, linear fashion as in the following sequence

$$S_0 \ \leq \ S_1 \ \leq \ \ldots \ \leq \ S_n$$

which establishes the refinement $S_0 \leq S_n$, because of the transitivity of the relation.

An important property is that one can refine subcomponents of programs without affecting the total correctness of the whole program. Formally this means that if we have proved that $T \leq T'$ then we have in fact established the refinement $S[T] \leq S[T']$ (provided the context $S[\cdot]$ is monotonic with respect to refinement – however, all contexts that arise using ordinary specification and programming constructs are monotonic).

Data refinement In program development, it is often useful to specify a program using *abstract* (high-level and often unimplementable) data structures and then refine such specifications into executable programs where the abstract data structures has been replaced by *concrete* data structures.

In practice this is achieved by providing a relation (say R) that relates the unimplementable abstract data structure to the one used in an executable program and show that when this relation holds also refinement is achieved. In the Refinement Calculus, this is achieved using an *abstraction statement* of the form $E = \{+a - c.\ R\}$. This statement adds abstract variable(s) a to the state so that the condition R holds and then removes concrete variable(s) c from the state. Statement S over the abstract state is then *data refined* by concrete statement S' if the following refinement holds:

$$E; S \leq S'; E$$

We write $S_1 \leq_E S_2$ for this data refinement.

2.2 Formalising The Refinement Calculus in HOL

Program state When formalising a (state-based) programming logic in HOL an important decision one has to make is how to represent the state of a program. Since we are working with a shallow embedding, one natural way to deal with states is to represent them as tuples. On the general level, states are defined using the polymorphic type[1]. In individual programs, each program variable is represented as one component in the tuple.

Predicates The semantics used is weakest precondition semantics where the meaning of programs are defined using predicates over the program state. Thus, predicates (pred) in refcalc are functions of type $\mathsf{state}\rightarrow\mathsf{bool}$, where the state is represented as described above. For predicates of this type a set of connectives (as well as truth values) have to be defined (lifted HOL connectives). For example, we define

$\vdash_{\mathsf{def}} \mathsf{true} = \lambda \mathsf{v}.\ \mathsf{T}$
$\vdash_{\mathsf{inf}} \mathsf{and\ p\ q} = \lambda \mathsf{v}.\ \mathsf{p\ v} \wedge \mathsf{q\ v}$
$\vdash_{\mathsf{inf}} \mathsf{implies\ p\ q} = \forall \mathsf{v}.\ \mathsf{p\ v} \Rightarrow \mathsf{q\ v}$

Here the symbol "$\vdash_{\mathsf{inf}}$" indicates that an operator is defined as an infix. Ordinary prefix operators are preceded by the symbol "$\vdash_{\mathsf{def}}$". Theorems that have been proved in HOL will be preceded by the symbol "$\vdash_{\mathsf{HOL}}$". Note that $\mathsf{implies}$ is the relation of strength on predicates.

[1] We use the convention from HOL90 for writing type variables, i.e. they are written in the form 'a etc.

Predicate transformers Program statements are defined directly as *predicate transformers*, i.e. functions of type pred→pred (this means that we have a *shallow embedding* of programs; we do not define a new type representing the syntax of program statements). Assume that s represents a program state and f represents a state function (meaning it has type state→state). We then define assignment and sequential composition by the constants assign and seq, as follows:

$\vdash_{\mathsf{def}}$ assign f q = λs. q(f s)
$\vdash_{\mathsf{inf}}$ seq S1 S2 q = S1(S2 q)

For example,

assign (λs:num#num. (FST s,FST s + 1))

represents the assignment $y := x + 1$ in a two-component state space (x, y). In fact, the initial and final state need not be the same, so a state function can be of the form f:state1→state2 (this feature is used when abstraction statements are defined in HOL).

Refinement and data refinement The refinement relation is simply implication lifted from predicates to predicate transformers:

$\vdash_{\mathsf{inf}}$ ref S1 S2 = ∀q. (S1 q) implies (S2 q)

To use data refinement in HOL we assume state spaces of the form 'x#'s where the first component represents the local state and the second component represents the global state. The abstraction statement is then defined as a predicate transformer in HOL:

$\vdash_{\mathsf{def}}$ abst r q = (λw. ∃a. r(a,FST w,SND w) ∧ q(a,SND w))

where r:('a#'c#'s)→bool is the abstraction relation and q:('a#'s)→bool is a predicate over the abstract state space.

Now the data refinement relation can be defined directly:

$\vdash_{\mathsf{def}}$ dataref r c c' = ((abst r) seq c) ref (c' seq (abst r))

Here c is a program operating on the abstract data space ('a#'s) whereas c' operates on the state space ('c#'s).

3 Action Systems – theory

Action systems [5] can be used to model reactive systems. In this paper, we use following (ad hoc) syntax for action systems:

$$\begin{array}{ll} \mathit{var} & v \\ \mathit{ini} & A_0 \\ \mathit{act} & A_1 [\!] A_2 [\!] \ldots [\!] A_m \\ \mathit{fin} & A_F \end{array}$$

where v is the list of *local variables*, A_0 is the *initialisation* predicate, A_F is the *finalisation* predicate and $A_1, \ldots, A_m$ are *actions*. Each action is a *guarded command* of the form $g \to S$ where S is a statement (the *body*) and g is an enabling predicate (the *guard*). The semantics for the action is then give by $wp(g \to S, Q) = g \Rightarrow wp(S, Q)$. The global variables are not specified; All program variables that are not explicitly listed as local variables are considered to be global.[2]

Informally an action system behaves as follows

1. The local variables are assigned initial values that make the initialisation predicate hold.

2. Actions are chosen for execution non-deterministically over and over (an action can be chosen only if its guard evaluates to true).

3. If the finalisation predicate evaluates to true (at any point in the computation) then the action system *may* terminate.

If the finalisation and all action guards are false at the same time, then the system has *deadlocked*.[3]

Semantics and refinement of action systems Since reactive programs frequently are supposed to run forever, a predicate transformer semantics is not adequate. Instead, action systems are given *trace* semantics. Thus, the meaning of a given action system is defined as the set of *traces* (a trace is a state sequence with the local state components filtered away and all finite stuttering removed) that its execution can generate. Thus, a trace is a (finite or infinite) sequence of global states.

[2]the traditional notation for action systems has been $|[\mathsf{var}\ v = v_0; \mathsf{do}\ g_1 \to S_1 [\!] \ldots [\!] g_m \to S_m\ \mathsf{od}]|$. This corresponds to the notation of this paper but with a deterministic initialisation and an implicit finalisation predicate $\neg(g_1 \vee \ldots \vee g_m)$.

[3]The original theory of action systems does not include finalisation predicates – they have been introduced by Back and von Wright in (as yet) unpublished work.

An action system $\mathcal{A}$ is *trace refined* by another action system $\mathcal{C}$ (written $\mathcal{A} \sqsubseteq \mathcal{C}$ if every trace of $\mathcal{C}$ is also a possible trace of $\mathcal{A}$. Thus, trace refinement is a suitable relation for talking about refinement of action systems as closed systems. However, when considering action systems as reactive components that interact with other action systems, then we are more interested in *simulation refinement.*

Assume that the following two action systems are given

$$\begin{array}{rcll} \mathcal{A} & = & \mathit{var} & a \\ & & \mathit{ini} & A_0 \\ & & \mathit{act} & A_1 [\!] \ldots [\!] A_m \\ & & \mathit{fin} & A_F \end{array}$$

and

$$\begin{array}{rcll} \mathcal{C} & = & \mathit{var} & c \\ & & \mathit{ini} & C_0 \\ & & \mathit{act} & C_1 [\!] \ldots [\!] C_m \\ & & \mathit{fin} & C_F \end{array}$$

and that $E = \{+a - c.\ R\}$ is an abstraction statement. Then $\mathcal{A}$ is *simulation refined through* E *by* $\mathcal{C}$ (written $\mathcal{A} \leq_E \mathcal{C}$) if the following three conditions hold:

$$C_0 \ \Rightarrow\ wp(E, A_0) \tag{1}$$

$$E; (A_1 \wedge \ldots \wedge A_m) \ \leq\ (C_1 \wedge \ldots \wedge C_n); E \tag{2}$$

$$wp(E, \neg A_F) \ \Rightarrow\ \neg C_F \tag{3}$$

where $S_1 \wedge S_2$ is the *demonic choice* between statements S_1 and S_2; its semantics is $wp(S_1 \wedge S_2, Q) = wp(S_1, Q) \wedge wp(S_2, Q)$. The above conditions are referred to as *initialisation simulation* (1), *action simulation* (2) and *finalisation simulation* (3). Note that condition (2) is a data refinement – thus simulation refinement between action systems reduces to data refinement between the actions involved. Since these actions are sequential program statements, the original Refinement Calculus for sequential programs can be used to finish the details of a proof of simulation.

An important special case of simulation refinement is when the abstraction statement is *local* (i.e., of the form $\{+a - c.\ R\}$ where R does not mention any global variables). We write $\mathcal{A} \preceq \mathcal{C}$ if there exists a local abstraction statement E such that $\mathcal{A} \leq_E \mathcal{C}$ holds.

Simulation is a stronger relation than trace refinement. Thus, the following rule is valid:

$$\frac{\vdash \mathcal{A} \preceq \mathcal{C}}{\vdash \mathcal{A} \sqsubseteq \mathcal{C}} \tag{4}$$

Parallel composition Reactive programs are executed in an *environment*. For action systems, this is modelled using parallel composition $\|$. Given an action system $\mathcal{A}$, executing it in the environment $\mathcal{E}$ means executing the action system $\mathcal{A}\|\mathcal{E}$.

The parallel composition of $\mathcal{A}$ and $\mathcal{C}$ is the following action system:

$$\begin{array}{rcll} \mathcal{A}\|\mathcal{C} & = & \mathit{var} & a, c \\ & & \mathit{ini} & A_0 \wedge C_0 \\ & & \mathit{act} & A_1 \,[\!]\, \ldots \,[\!]\, A_m \,[\!]\, C_1 \,[\!]\, \ldots \,[\!]\, C_n \\ & & \mathit{fin} & A_F \wedge C_F \end{array}$$

Thus, the local state of the parallel composition is the combination of the two local states every action is also an action of the parallel composition.

The following compositionality rule can be used to refine individual action systems in a parallel context:

$$\frac{\vdash \mathcal{A} \preceq \mathcal{C}}{\vdash \mathcal{A}\|\mathcal{E} \preceq \mathcal{C}\|\mathcal{E}} \tag{5}$$

In the general case (when E is not local) there is an additional *noninterference condition* associated with this rule. This condition states that every action of the environment must commute with the abstraction statement E.

An example The following is an example of an action system.

$$\begin{array}{rcll} \mathcal{A} & = & \mathit{var} & g : \mathit{bool}, y : \mathit{num} \\ & & \mathit{ini} & \neg g \\ & & \mathit{act} & f \wedge \neg g \to f, g, y := \mathit{false}, \mathit{true}, x \ [\!] \\ & & & g \to g, \mathit{out} := \mathit{false}, \mathit{consume}(y) \\ & & \mathit{fin} & \neg f \wedge \neg g \end{array}$$

Here $f : \mathit{bool}$, $x : \mathit{num}$ and $\mathit{out} : \mathit{num}$ are global variables. The behaviour of $\mathcal{A}$ can be described as follows. Its first action moves a values from x to the local variable y and the second action transforms the value (the function *consume*) and moves it to the global variable *out*. The flags f and g make sure that no new value is taken in before the previous value is consumed (when both flags are *false* the system terminates).

In isolation, $\mathcal{A}$ cannot do very much. But in an environment where new values are put in x and the flag f is reset, $\mathcal{A}$ can go on consuming values indefinitely. The parallel composition $\mathcal{A} \parallel \mathcal{B}$ is a *producer–consumer system*, where

$$\begin{array}{rcll} \mathcal{B} & = & \mathit{var} & \\ & & \mathit{ini} & \mathit{true} \\ & & \mathit{act} & \neg f \to f, x := \mathit{true}, \mathit{produce}(\mathit{in}) \\ & & \mathit{fin} & f \end{array}$$

Here $\mathcal{B}$ has no local variables at all, and *in* is an additional global variable from which variables to be consumed are taken.

4 Action Systems – HOL formalisation

As described in section 3 an action system essentially consists of three separate parts: the initialisation and finalisation predicates and a list of actions. The two predicates are directly represented as boolean functions over a two-component state space, as described in Section 2.2. Actions are represented as pairs (g,c) where g is a predicate and c is a predicate transformer.

In order to model lists of guarded statement we use the HOL theory of lists. We define the constant aldch to represent the demonic choice over a list of actions (it is needed in the definition of simulation) as follows:

$$\begin{array}{l} \vdash_{\mathsf{def}} \mathsf{aldch}\ [\ ]\ \mathsf{q} = \mathsf{true} \wedge \\ \quad \mathsf{aldch}\ (\mathsf{CONS\ a\ al})\ \mathsf{q} = (\mathsf{not}\ (\mathsf{FST\ a}))\ \mathsf{or}\ ((\mathsf{SND\ a\ q})\ \mathsf{and}\ (\mathsf{aldch\ al\ q})) \end{array}$$

Here FST accesses the guard (the first component) of an action while SND accesses the body.

Many things that are self-evident in a pen-and-paper treatment have to be proved explicitly in order to be used. For example, to be able to reorder actions in a list of the type above, we have proved the following HOL theorem:

$$\vdash_{\mathsf{HOL}} \forall \mathsf{a\ b.\ aldch\ (APPEND\ a\ b) = aldch\ (APPEND\ b\ a)}$$

Example Consider the action system $\mathcal{A}$ from section 3. In HOL it looks as follows:

```
(not g,
 [(f and (not g), assign (λs. (T,x s),F,x s,in s,out s));
  (g, assign (λs. (F,y s),f s,x s,in s,consume (y s)))],
 (not f) and (not g))
```

Here the program variables have been represented as projection functions on the state. The local state is (g,y) while the global state is (f,x,in,out), so we have defined, e.g., $\mathsf{f} = \mathsf{FST} \circ \mathsf{SND}$ and $\mathsf{x} = \mathsf{FST} \circ \mathsf{SND} \circ \mathsf{SND}$.

Simulation In order to define simulation we start by defining three *projection functions* to access the different components in HOL action system triples (and also to improve readability):

$\vdash_{\mathsf{def}}$ ini = FST
$\vdash_{\mathsf{def}}$ act = FST $\circ$ SND
$\vdash_{\mathsf{def}}$ fin = SND $\circ$ SND

The conditions (1) and (3) are given the following formalisation in HOL:

$\vdash_{\mathsf{def}}$ inisim r AA CC = (ini CC) implies (abst r (ini AA))
$\vdash_{\mathsf{def}}$ finsim r AA CC = (abst r (not(fin AA))) implies (not(fin CC))

Here r:('a#'c#'s)$\rightarrow$bool is the abstraction relation, AA is an action system triple working on the state space 'a#'s and CC is an action system triple working on the state space 'c#'s

The action simulation (2) condition is defined as follows:

$\vdash_{\mathsf{def}}$ actsim r AA CC = dataref r (aldch (act AA)) (aldch (act CC))

The general simulation relation is now defined as follows:

$\vdash_{\mathsf{def}}$ rsim r AA CC = inisim r AA CC $\wedge$ actsim r AA CC $\wedge$ finsim r AA CC

We are (currently) interested primarily in local simulations, since the compositionality rule for local simulations has no side conditions. Thus we define a specific relation for that. First we define locality:

$\vdash_{\mathsf{def}}$ local r = $\exists$rr. $\forall$s. r s = rr(FST s,FST(SND s))

This definition says that locality of a relation r (of the same type as above) means always being able to find a relation rr:('a#'c)$\rightarrow$bool that has the same value as r regardless of what value the global state component has.

Now we define the local simulation relation:

$\vdash_{\mathsf{inf}}$ lsim AA CC = $\exists$r. local r $\wedge$ rsim r AA CC

Parallel composition defined in HOL Two action systems can be composed in parallel if they work on the same global state. Our idea is to formalise parallel composition of two action systems $\mathcal{A}$ (on state space 'a#'s) and $\mathcal{C}$ (on state space 'c#'s) so that $\mathcal{A}\|\mathcal{C}$ works on the state space ('a#'c)#'s. To do this, we *adapt* the two action systems involved to the larger state space using a special adaption operator.

Before we define adaption and parallel composition, we make the following definitions:

$\vdash_{\mathsf{inf}}$ ppar p q u = p (FST u) $\wedge$ q (SND u)
$\vdash_{\mathsf{inf}}$ par c c' q u = $\exists$ q1 q2. (q1 ppar q2 implies q) $\wedge$
c q1 (FST u) $\wedge$ c' q2 (SND u))

The ppar operator composes two predicates p and q that operate over different state spaces in "parallel" w.r.t. a combined state space u. Similarly par defines a "parallel" composition operator for statements (c and c') working on separate state spaces. Given these definitions we can define an adaption operator as follows:

$$\vdash_{\mathsf{def}} \mathsf{adapt\ f\ g\ c} = (\mathsf{assign\ f})\ \mathsf{seq}\ (\mathsf{c\ par\ skip})\ \mathsf{seq}\ (\mathsf{assign\ g})$$

Using state functions f:'s→'s1#'s2 and g:'s1#'s2→'s that reorder the components of the state space (with g begin the inverse function of g) the adaption operator has the effect that execution of adapt f g c corresponds to executing c over state space 's1 and skip over state space 's2. In what follows we write $\overline{\mathsf{c}}$ to denote an adapted command c.

Now we define parallel composition of two action systems as follows:

$$\vdash_{\mathsf{inf}} \mathsf{apar\ AA\ BB} = (\mathsf{inicomb\ AA\ BB},\mathsf{actcomb\ AA\ BB},\mathsf{fincomb\ AA\ BB})$$

where inicomb, actcomb and fincomb are defined to adapt and combine the corresponding parts of the two action systems AA and BB. We have

$$\begin{array}{l} \vdash_{\mathsf{def}} \mathsf{inicomb\ AA\ BB} = (\lambda \mathsf{s}.\ \mathsf{ini\ AA\ (FST(FST\ s),SND\ s)} \wedge \\ \qquad\qquad\qquad\qquad\qquad \mathsf{ini\ BB\ (SND(FST\ s),SND\ s))} \\ \vdash_{\mathsf{def}} \mathsf{fincomb\ AA\ BB} = (\lambda \mathsf{s}.\ \mathsf{fin\ AA\ (FST(FST\ s),SND\ s)} \wedge \\ \qquad\qquad\qquad\qquad\qquad \mathsf{fin\ BB\ (SND(FST\ s),SND\ s))} \end{array}$$

where the state is s:('a#'c)#'s, i.e., the space of the action system that is created using parallel composition

To combine the lists of actions we use adaption

$$\vdash_{\mathsf{def}} \mathsf{actcomb\ AA\ BB} = \mathsf{APPEND}\ \overline{\mathsf{act\ AA}}\ \overline{\mathsf{act\ BB}}$$

Here $\overline{\mathsf{act\ AA}}$ is the adaption of the list of action from AA. The definition of this adaption essentially states that we map suitable adaptions into every guard and body of the list. The full definition (including type information) is fairly long and complex and therefore not included in the text.

Compositionality as a HOL theorem Before we can refine action systems containing parallel composition we have to prove the compositionality theorem (5). We have done this, i.e. we have proved the following HOL theorem:

$$\begin{array}{l} \vdash_{\mathsf{HOL}} \forall\ \mathsf{AA\ CC\ BB.\ AA\ lsim\ CC} \wedge \mathsf{EVERY\ mono\ (MAP\ SND\ (act\ AA))} \Rightarrow \\ \qquad\qquad\qquad (\forall \mathsf{b}.\ (\mathsf{AA\ apar\ BB})\ \mathsf{lsim}\ (\mathsf{CC\ apar\ BB})) \end{array}$$

In this theorem the term $\mathsf{EVERY\ mono\ (MAP\ SND\ (act\ AA))}$ means that every action body of the system AA must be monotonic.[4] The definition of monotonicity is as follows:

$$\vdash_{\mathsf{def}} \mathsf{mono\ c} = \forall \mathsf{p\ q.\ p\ implies\ q} \Rightarrow \mathsf{(c\ p)\ implies\ (c\ q)}$$

In order to prove simulations between action systems we also need a large collection of theorems that facilitate data refinement proofs between action lists (this is the most difficult part in simulation proofs). Typical theorems deal with breaking down lists into parts etc. An example is the following theorem:

$$\begin{array}{ll}\vdash_{\mathsf{HOL}} \forall\ \mathsf{r\ a\ b\ c\ d.} & \mathsf{dataref\ r\ (aldch\ a)\ (aldch\ b)} \wedge \\ & \mathsf{dataref\ r\ (aldch\ c)\ (aldch\ d)} \\ & \Rightarrow \mathsf{dataref\ r\ (aldch\ (APPEND\ a\ c))\ (aldch\ (APPEND\ b\ d))}\end{array}$$

which is useful in the simple case when actions are replaced one by one in a simulation.

The definition of simulation reduces refinement between action systems to data refinement between sequential program statements. This means that we can reuse existing HOL theories of program refinement [11] to finish off refinement proofs.

Trace refinement and HOL The trace semantics mentioned in section 3 has not been formalised in HOL. Formalising it would be a demanding exercise in theorem proving, but it would not add any advantage from a practical point of view, since all proof efforts when working with action systems are aimed at proving simulations. Defining the trace semantics would be interesting only in order to prove the rule in equation (4). Here we treat this rule is as a *meta-level axiom* that is outside the HOL theory.

5 An example of a simulation proof

Recall the two action systems $\mathcal{A}$ (consumer) and $\mathcal{B}$ (producer) from Section 3

$$\begin{array}{rcll}\mathcal{A} & = & \mathit{var} & g : \mathit{bool}, y : \mathit{num} \\ & & \mathit{ini} & \neg g \\ & & \mathit{act} & f \wedge \neg g \to f, g, y := \mathit{false}, \mathit{true}, x \ \| \\ & & & g \to g, \mathit{out} := \mathit{false}, \mathit{consume}(y) \\ & & \mathit{fin} & \neg f \wedge \neg g\end{array}$$

[4] Monotonicity – one of Dijkstra's original "healthiness conditions" for predicate transformers – cannot be taken for granted in the HOL formalisation of program statements; it has to be proved explicitly to hold for a statement. However, monotonicity for standard statements can be proved automatically in the HOL theory of refinement.

and

$$\begin{array}{rcll} \mathcal{B} & = & \mathit{var} & \\ & & \mathit{ini} & \mathit{true} \\ & & \mathit{act} & \neg f \rightarrow f, x := \mathit{true}, \mathit{produce}(\mathit{in}) \\ & & \mathit{fin} & f \end{array}$$

As an example, we describe a small but complete refinement of the action system $\mathcal{A}$ (the consumer) by another action system $\mathcal{C}$, where the local variables g and y have been replaced by a list z.

$$\begin{array}{rcll} \mathcal{C} & = & \mathit{var} & z : (\mathit{num})\mathit{list} \\ & & \mathit{ini} & f \wedge (z = [\,]) \\ & & \mathit{act} & f \wedge (z = [\,]) \rightarrow f, z := \mathit{false}, [x] \; [\!] \\ & & & g \rightarrow z, \mathit{out} := [\,], \mathit{consume}(\mathit{hd}\ z) \\ & & \mathit{fin} & \neg f \wedge (z = [\,]) \end{array}$$

The refinement as such is fairly trivial, as it does not change the structure or behaviour of the system. However, it illustrates the basic principles of action system refinement: the simulation refinement of a component in a parallel composition using techniques of data refinement. In HOL the theorem proving the above simulation looks as follows (note what $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ look like in HOL notation):

```
⊢HOL ((not g,
        [(f and (not g), assign (λs. ((T,x s),F,x s,inv s,out s)));
         (g, assign (λs. ((F,y s),f s,x s,inv s,consume (y s)))],
        (not f) and (not g))
      apar
        (true,
        [(not f, assign (λs. (dum s,T,produce (inv s),inv s,out s)))],
        f))
     lsim
      (((λs. z s = []),
        [((λs. f s ∧ (z s = [])), assign (λs. ([x s],F,x s,inv s,out s));
         (λs.¬(z s = [])),
             assign (λs. (TL (z s),f s,x s,inv s,consume (HD (z s))))],
        (λs. ¬(f s) ∧ (z s = [])))
      apar
        (true,
        [(not f, assign (λs. (dum s,T,produce (inv s),inv s,out s)))],
        f))
```

(here dum:one is a dummy local component, added to the action system $\mathcal{A}$ so that its state space has the required form, i.e., a product of a local and a global state).

In order to prove this theorem one first matches the whole term against the compositionality rule. This way we get rid of the environment (in this case $\mathcal{B}$). We then need to need to prove that all the actions of $\mathcal{A}$ are monotonic (this is done using an automatic monotonicity prover) and that the lsim relation holds between the two action systems $\mathcal{A}$ and $\mathcal{C}$. We use the definition of lsim to rewrite the current term and then supply the abstraction relation we want to use. It is $(g \wedge (z = [\ ])) \vee (\neg g \wedge (z = [y]))$, which is represented in HOL as follows:

$$\lambda\mathsf{s}.\ (\neg\mathsf{g}(\mathsf{old\ s}) \wedge (\mathsf{z}(\mathsf{new\ s}) = [\ ])) \vee (\mathsf{g}(\mathsf{old\ s}) \wedge (\mathsf{z}(\mathsf{new\ s}) = [\mathsf{y}(\mathsf{old\ s})]))$$

where old and new recover the state spaces of the old (abstract) action system and the new (concrete) action system from the state space of the abstraction relation (so old s = (FST(FST s),SND s) and new s = (SND(FST s),SND s)).

The proof is then continued by proving that the above relation is local (which is trivial) and that the rsim relation holds. Proving initialisation and finalisation simulation is straightforward. To prove action simulation we use the theorem for breaking down data refinement between action lists into data refinement of individual actions (see section 4) and then use the existing theories for proving data refinement.

6 Future Work

The HOL theory presented here shows that HOL can be used to prove refinement theorems for action systems. However, it is also clear from the presentation that the theory is not easy to use directly. We are well aware of this fact and the solution we propose is to integrate the theory into the Refinement Calculator [6], an extendable tool that supports the production of refinement calculus proofs in HOL. Working with the Refinement Calculator means selecting (using the mouse) parts of a program that one wants to refine, and executing suitable commands to carry out refinements and other transformation steps.

In order to make the action system theory usable within the Refinement Calculator there are a number of things that have to be done:

- Since the Refinement Calculator is based on the HOL window Library [9], the theory has to be adapted so that window inference can be used. This requires reflexivity and transitivity of the simulation relation to be proved. This has already been achieved, i.e. we have proved the theorems:

$\vdash_{\mathsf{HOL}} \forall\ \mathsf{AA}.\ \mathsf{AA}\ \mathsf{lsim}\ \mathsf{AA}$
$\vdash_{\mathsf{HOL}} \forall\ \mathsf{AA}\ \mathsf{BB}\ \mathsf{CC}.\ \mathsf{AA}\ \mathsf{lsim}\ \mathsf{BB} \wedge \mathsf{BB}\ \mathsf{lsim}\ \mathsf{CC} \Rightarrow \mathsf{AA}\ \mathsf{lsim}\ \mathsf{CC}$

Furthermore a set of so called *window* rules have to be proved. These rules justify transforming a term in a context containing other terms. In our case, the main window theorem is the compositionality theorem which states that an action system can be (local) simulation refined in any parallel context. Other window rules allow a user to select a subcomponent of an action system (e.g., a guard or a part of a body) and transform it.

- Translation and pretty printing between the action system notation presented in this paper and a suitable surface syntax must be handled. The Refinement Calculator supports using a surface syntax. This is very convenient; it allows working with named program variables (hiding the state variable), and it allows a traditional programming notation to be used. Adding parsing and pretty-printing will not be very difficult since action systems add very few new syntactic elements compared with what is alredy supported by the Refinement Calculator.

- A collection of theorems and functions that are used in almost every action system refinement have to be proved and made available to the tool, so that they can be applied automatically whenever needed. A typical example is the theorem saying that the parallel operator is commutative up to "simulation equivalence":

  $\vdash_{\mathsf{HOL}} \forall\ \mathsf{AA}\ \mathsf{CC}.\ (\mathsf{AA}\ \mathsf{apar}\ \mathsf{CC})\ \mathsf{lsim}\ (\mathsf{CC}\ \mathsf{apar}\ \mathsf{AA})$

  Another example is a function that automatically proves the a given abstraction relation is local.

Once these additions above have been done, working with reactive refinement in the Refinement Calculator will be more or less equivalent to working with ordinary refinement. In fact the most difficult part in proving a simulation is the action simulation part, and proving action simulation amounts to proving an ordinary data refinement.

The Refinement Calculator is currently being extended with a tool for data refinement. This tool can compute the data refinement of a given program, given the abstract program and the abstraction relation [10]. It can also be used to prove that the data refinement holds between two explicitly given statements. Since the main step in a simulation proof is data refinement, this tool can be directly used by the tool that we propose for action systems.

Finally, it should be noted that more advanced forms of action system refinement can also be handled by extending the work presented here. These include refinements that change the number of actions in an action system, e.g., by introducing stuttering actions.

7 Conclusions

Our formalisation of action systems shows that it is possible to represent a fairly complex theory of reactive systems in HOL, in a way that makes it possible to prove refinement also between nontrivial systems. Since we have used a shallow embedding and represented the state as a tuple, program variables can have any HOL type and a program can contain variables of different types.

The standard interface of the HOL system makes it hard to work with action system refinement directly. However, the aim of our formalisation of action systems is to provide a foundation for a tool that manipulates HOL terms (representing action systems) through a graphical user interface that allows easy selection and choice of transformation and that hides details that the user is not interested in at the moment (through the focusing mechanism of the HOL window Library).

Since such a tool already exists for sequential programs (the Refinement Calculator) and since the theory of action systems reduces reasoning about reactive systems to reasoning about sequential programs, the steps remaining until we have a prototype tool are few.

References

[1] F. Andersen, K.D. Petersen, and J.S. Petterson. A Graphical Tool for Proving UNITY Progress. In T.F. Melham and J. Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications – 7th International Workshop. Valletta, Malta, September 1994*, volume 859 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[2] R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[3] R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[4] R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2:247–272, 1990.

[5] R. Back and J. von Wright. Trace refinement of action systems. Reports on computer science and mathematics 153, Åbo Akademi, 1994.

[6] M. Butler and T. Långbacka. Program derivation using the refinement calculator. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference*, volume 1125 of *Lecture Notes in Computer Science*, pages 93–108. Springer Verlag, August 1996.

[7] A. Camillieri. Mechanizing CSP trace theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.

[8] E. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.

[9] J. Grundy. Window inference in the HOL system. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Proceedings of the International Tutorial and Workshop on the HOL Theorem Proving System and its Applications*, pages 177–189, University of California at Davis, August 1991. ACM-SIGDA, IEEE Computer Society Press.

[10] R. Ruksenas and J. von Wright. A tool for data refinement. TUCS Technical Report ??, Turku Centre for Computer Science, Lemminkäisenkatu 14A, 20520 Turku, Finland, 1997.

[11] J. von Wright. Program refinement by theorem prover. In *BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development. 5th – 7th January, City University, London, UK.*, 1994.

[12] J. von Wright and T. Långbacka. Using a theorem prover for reasoning about concurrent algorithms. In G. von Bochmann and D.K. Probst, editors, *Computer Aided Verification – Fourth International Workshop. CAV '92. Montreal. Canada. June 29 – July 1. 1992*, volume 663 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**